Towards meta-evolution via embodiment in artificial chemistries

Adam Max Nellis

Submitted for the Degree of Doctor of Philosophy

University of York

Department of Computer Science

September 2012

Abstract

Meta-evolution, the ability to generate novel ways of generating novelty, is one of the major goals of Artificial Life research. Biological systems can be seen to perform open-ended meta-evolution, but unfortunately this has proved very difficult to replicate within computer programs. This thesis defines a theoretical framework for thinking about the issues in this area and charting out possible solutions. We use this framework to take some tentative steps towards meta-evolutionary algorithms.

We begin with a review of how meta-evolution has developed historically, through different novelty-generation algorithms. We describe previous algorithms in terms of their *biological model* and their *computational model*, to distinguish their biological inspiration from their computational implementation details. We conclude that the route to meta-evolution lies in enriching the biological models of current algorithms, rather than their computational ones.

We use the theoretical idea of *embodiment* to analyse both biological and computational systems, and decompose the problem of achieving meta-evolution. We present a new definition of embodiment, allowing the concept to be applied to biological systems. We conclude that biological systems achieve meta-evolution by their embodiment within the physical world. We notice that current computational systems have poor embodiment within their virtual worlds. This leads to the hypothesis that improving the embodiment of computational systems, within virtual worlds, is a route to improving their meta-evolution.

In discussing how embodiment can be realised in virtual worlds, we use *Artificial Chemistries* as a language in which to program embodiment. We present a new definition of Artificial Chemistries, mapping their components onto our definition of embodiment. We end by presenting a new Artificial Chemistry, *GraphMol*, designed with embodiment in mind. We use GraphMol to embody the process of *copying a string*, and analyse its meta-evolution via a range of different experiments.

Contents

List of Figures

List of Algorithms

Acknowledgements

Many thanks to my supervisor, Susan Stepney, for invaluable help and guidance before, during and beyond my PhD.

Thanks to Jon Timmis, for incisive feedback at the different milestones of my PhD, and for giving me a job after my funding ran out.

I would like to thank the other members of the Plazzmid project: Ed Clark, Simon Hickinbotham, Peter Young, Tim Clarke, and Mungo Pay, for useful and entertaining discussions and feedback on my work.

I have benefitted immensely from the stimulating research environment of YCCSA (the York Centre for Complex Systems Analysis), whose members have provided interesting and enlightening discussions.

Finally, thanks to Becky Naylor and all my family, for their support throughout the years.

Declaration

The contents of this thesis are my own work, except where explicitly stated.

My PhD was part of the Plazzmid project (EPSRC grant EP/F031033/1). The *Stringmol* artificial chemistry [41, 42, 43], that I reference in this thesis, was produced as part of this project. I helped with the design and analysis of Stringmol, but the majority of the work was done by Simon Hickinbotham and the other members of the Plazzmid project. The *GraphMol* artificial chemistry that I present in this thesis (chapters 6, 7 and 8) is my own work.

Chapter 6: GraphMol and *Chapter 7: Junk changes GraphMol copier* are an expanded version of my conference paper: *Embodied copying for richer evolution*, published at the ECAL conference in 2011 [74].

The definition of artificial chemistries given in section 5.1 was first introduced in the context of run-time metaprogramming (self-modifying code), in the Plazzmid paper: *Embodied genomes and metaprogramming*, published at the ECAL conference in 2011 [45].

Appendix B is based on my conference paper: *Automatically moving between levels in artificial chemistries*, published at the ALife conference in 2010 [73].

Chapter 1

Introduction

Evolution is really good. It is an open-ended creation process that continually generates novelty by creating new life-forms and new ways for life-forms to interact. This amazing process that happens in nature, is clearly doing a lot of computation (however you choose to define *computation*). It would be useful if we could capture this computational power and use it in our own computers, to solve our problems, rather than solving the problem of *living* that natural evolution does. Many people have tried to capture the power of natural evolution within a computer. While there have been interesting algorithms developed, no-one has yet written a computer program that displays true open-ended evolution and novelty generation. It is not clear whether such a program could theoretically exist, but until we know definitively one way or the other, it makes sense to try.

While I have not succeeded in producing this dreamed-of computer program, I highlight some of the deficiencies in current attempts. I outline some principles that I believe are necessary for capturing the power of biological evolution within a computer program, chief among these being embodiment.

In chapter 2, I introduce the computer science fields of evolutionary algorithms and artificial life. These are where other researchers have tried to build artificial systems that display open-ended evolution and novelty generation. I describe these artificial systems in terms of a biological model and a computational model, to show how the early systems have been extended to become more open-ended and better able to generate novelty.

But ultimately, all of these systems have failed to display true open-endedness. The reason for this is that they have gone down a route which I label *meta evolution*. They have tried to write down a set of static, crisp rules that define how evolution works. They have evolved objects using these rules and found a limit to the complexity of their evolved objects. Facing this limit, they have tried to overcome it by evolving their rules using a different set of static, crisp (meta-)rules. This approach has problems, which I outline towards the end of chapter 2. I describe

the different possible types of meta-evolution, and highlight a strategy that I think has a chance of displaying true open-endedness, because it is much closer to how biology works than other approaches.

My strategy is to have evolution emerge from a lower-level process, rather than writing evolution as a computer program in a traditional programming language. In this situation, evolution is not a piece of computer code, but is a consequence of the organisms that are evolving. So evolution acting on those organisms can change itself indirectly, by changing the substrate out of which it emerges.

The theoretical principle underlying my approach is *embodiment*, which I describe in detail in chapter 3. Embodiment means implementing something within a rich environment, in order to gain benefit from interacting with the environment. In terms of programming open-ended evolution, this means using a static, crisp programming language to create a rich, interesting environment that defines a sloppy, uncertain programming language. In this sloppy programming language, we implement the mechanisms that evolution can emerge from.

I emphasise the importance of embodiment by highlighting its ubiquitous exploitation by biological evolution in chapter 4. I show in chapter 5 how the computer science field of artificial chemistries can provide some good metaphors for embodiment, and a starting point for how to implement embodied computer programs.

In the following chapters I investigate my artificial chemistry, GraphMol, that is designed to explore the ideas of embodiment. In chapter 6 I define GraphMol and describe how it is implemented, including how to implement an embodied copying mechanism in GraphMol. Chapter 7 is a proof-of-concept experiment showing the GraphMol's embodied copying mechanism has the potential to change itself. Chapter 8 shows the mechanism copying itself freely in an open environment, to investigate how it goes about changing itself.

I end with a conclusions and future work chapter (chapter 9). This critiques GraphMol, suggesting ideas about how it can be improved. I then reflect on the process of designing embodied systems, ending with some speculations on the future of meta-evolution and open-ended novelty generation.

1.1 What this thesis is about

My message is very simple, but might be easy to misunderstand. I am trying to explore ideas rather than present a tool or an algorithm. My main ideas are:

1. Meta-evolution is more complicated than might be initially thought, and is better considered as a component of *novelty generation*.

2. *Embodiment* is a possible way of achieving open-ended novelty generation within a computer program.

3. *Artificial Chemistries* provide a useful metaphor and pattern for implementing embodiment.

The methods I discuss (evolutionary algorithms and models of biology) are used by other researchers to do things that are different from my work. It is worth making explicit how my work is different from theirs, to prevent the reader misunderstanding my purpose.

1.1.1 Not optimisation

Evolutionary algorithms have been used to solve engineering problems by optimising functions. This has been successful in many areas, but is not what I am using evolution for. I am using evolution to generate novelty rather than to optimise a function.

It is interesting to note that the evolutionary algorithms that have been successful at function optimisation, have not been successful at open-ended evolution. Thus we need a different type of evolutionary algorithm to the ones that are good at optimising functions.

Another important point is that biological evolution is very good at open-endedness, but does not optimise a function. So when we are seeking inspiration to help us create computer programs that display open-ended evolution, the world of biology seems a more useful place to look than the world of function optimisation.

1.1.2 Not simulation

Systems biology researchers create models of biological systems in order to simulate aspects of biology. They care about the real biology, and their computer programs are simulations of this real biology, that they use as a tool to help them understand more about the real system.

In this situation, the physical world is *real* and the world inside the computer is a *simulation*: it is not the real thing, but an approximation of it. This is a fair

point of view when you are not interested in the computer-generated world on its own, but it is not the point of view that I take here.

I am not making simulations. I am interested in creating computer programs for their own sake (to generate novelty), rather than to learn about the physical world. I acknowledge that current computer programs are less capable of generating novelty than the biological world is, and so I look to the biological world for inspiration on how to design novelty-generating computer programs. But my computer-generated worlds are real, because I am interested in them on their own; they are not a simulation of anything.

Therefore I will not use the word *simulation* to describe my work, nor will I use the word *real* to describe biology (or chemistry or physics). I will talk about my computer-generated worlds as *virtual*, and compare them to the *physical* world. This terminology lessens the psychological temptation to think of computer-generated objects as somehow less real than physically-generated objects.

1.2 My contribution

The novel contribution of this thesis comprises three parts:

1. A review of how meta-evolution has developed historically, through different novelty-generation algorithms, by changes to their biological and computational models. This includes a classification of the different types of meta evolution (chapter 2).

2. Applying the ideas of embodiment to the problem of meta-evolution. This includes:

   (a) A new definition of embodiment (chapter 3).

   (b) A detailed evaluation of how biological systems benefit from embodiment, discussing how they use it to implement meta-evolution and novelty-generation (chapter 4).

   (c) A discussion of how embodiment can be realised in artificial systems, taking inspiration from the biology, using artificial chemistries as a language in which to program embodiment (chapter 5).

3. A new artificial chemistry, GraphMol, designed with embodiment in mind. GraphMol is used to implement an example embodied mechanism (copying), and experiments are performed to evaluate the novelty-generation capabilities of the system (chapters 6, 7 and 8).

Chapter 2

From evolution to meta-evolution

This chapter describes the computer science fields of *evolutionary algorithms* and *artificial life*, from the viewpoint of *open-ended evolution*. This thesis takes the stance that one of the goals of both evolutionary algorithms and artificial life is to produce computer algorithms capable of generating novelty, ideally without bounds. I use the phrase *novelty-generation algorithms* to refer to evolutionary algorithms or artificial life models built for the purpose of generating open-ended novelty, or algorithms that can be viewed in this way. This phrase also allows the inclusion of open-ended novelty-generation algorithms from other fields, although I am not aware of any such algorithms.

The idea of open-ended evolution is inspired by our universe, in particular biological evolution, which has produced a staggering level of complexity (life) from very simple components (the fundamental units/laws of physics). It is unclear, and probably unknowable, whether our universe is truly open-ended or not. But it is abundantly clear that our universe has generated (and is currently generating) complexity on a scale that far surpasses that generated by our most sophisticated computer algorithms. Of course, any such algorithms are also part of our universe, and so the complexity generated by our universe includes the complexity generated by computer algorithms. But despite this philosophical thought, it is clear that our current computer algorithms are not very good at generating complexity.

The words *complexity* and *novelty* are difficult to define precisely. I have not seen an author give a convincing definition of either of these words, and I do not propose any technical definitions of them. I describe in section 2.1 what I mean by *complexity*, and explain why I choose to use the phrase *novelty-generation algorithms* to describe the types of system I am interested in.

Novelty generation contrasts with the view of evolutionary algorithms as tools for optimising fitness functions. The idea of evolution as an optimisation tool will be ignored in this thesis, because it is a different topic from the one explored here (as explained in section 1.1.1).

This chapter's description of novelty-generation algorithms starts with the earliest evolutionary algorithms and artificial life models, describing the ways in which they have been modified. A key theme across all of the specific algorithms is that in order to make them more able to generate novelty, they have been modified to be more *biologically realistic*. But the converse is not true! Augmenting an algorithm with arbitrary inspirations from biology is not guaranteed to improve the algorithm in any way. The additional biological inspirations must represent some underlying process that contributes to biology's ability to generate novelty, but is missing from current algorithms.

The purpose of this chapter is not to comprehensively review of all evolutionary algorithms and artificial life. Instead, this chapter describes the types of thinking that have been used currently to extend these algorithms. This is explained through the use of two different ways of looking at novelty-generation algorithms: through their computational models and through their biological models. I define and explain these two models in section 2.2.

I put forward the idea that for any novelty-generation algorithm, extending its computational model can increase its novelty-generation capabilities only so far. However, extending the biological models of these algorithms (making the algorithms more biologically realistic) can increase the novelty-generation capabilities of the algorithms arbitrarily. In principle, this could allow these algorithms to (eventually) match the novelty-generation capabilities of biological evolution. It allows extensions to be added on top of extensions in a principled manner, that is in keeping with the original motivation for novelty-generation algorithms: achieving, in a computer, the complexity-generating properties displayed by the natural world, in particular biological evolution.

The ultimate goal of this trend towards increasing novelty-generation is meta-evolution: the ability to generate novel ways of generating novelty. While meta-evolution has not yet been achieved, there has been progress down this road. This chapter ends with my classification of the different ways of approaching meta-evolution, and a summary of the current progress along each of these routes. I describe how previous authors have tried to implement meta-evolution by extending the computational models of current novelty-generation algorithms, and detail some of the problems with this approach.

I highlight a particular type of meta-evolution, that can be seen as a modification of the biological model of current novelty-generation algorithms. This is the type of meta-evolution that I investigate in the remainder of the thesis. I call it *embodied evolution*, because the process of evolution is embodied within the same world as the organisms that are being evolved, rather than being implemented as an external system. I describe what *embodiment* means in chapter 3, and I describe my modified biological model in detail in chapter 5.

2.1 Complexity and novelty-generation

We make computers, and computer programs, to do work for us. Computers are very good at performing repetitive and mundane tasks that we can program into them. This has been clearly demonstrated over almost 200 years of digital computers. But it begs an interesting question: Do computers have to always operate in this way? Can computers perform tasks that have not been programmed into them? Can they invent their own, novel, behaviours, overriding their original programming and performing tasks of their own devising?

This may seem a strange thing to ask of computers, and was thought impossible when computers were first invented. A famous quote by Ada Lovelace cautions against expecting computers to generate new behaviours that have not been explicitly programmed into them [66]:

> It is desirable to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine. ... The Analytical Engine has no pretensions whatever to *originate* any thing. It can do whatever we *know how to order it* to perform. ... Its province is to assist us in making *available* what we are already acquainted with. [Italics in original.]

But recent work on evolutionary computation (that I describe in this chapter) is attempting to do precisely what Ada Lovelace stated as impossible. Evolutionary computation explicitly tries to develop computer programs that *can originate* their own things, and use them to solve problems that we *do not know how to solve*. Why should we think this is possible? Why would we imagine it could be possible to develop such computer programs? The answer is because such a computer system already exists: it is called *biological evolution*.

The evolution of life on Earth has produced a staggering variety of different lifeforms. From very simple beginnings, a wide array of different types of life have evolved, with different behaviours and different degrees of complexity. A famous quote by Charles Darwin (from the final paragraph of *The origin of species*) encapsulates the idea of evolution producing novel things that are different from those it started with [22, p.429]:

> It is interesting to contemplate a tangled bank, clothed with many plants of many kinds, with birds singing on the bushes, with various insects flitting about, and with worms crawling through the damp earth, and to reflect that these elaborately constructed forms, so different from each other, and dependent on each other in so complex a manner, have all been produced by laws acting around us. ... There is grandeur in this view of life, with its several powers, having been originally breathed

> by the Creator into a few forms or into one; and that, whilst this planet has gone cycling on according to the fixed law of gravity, from so simple a beginning endless forms most beautiful and most wonderful have been, and are being evolved.

A more modern view of the *origin* of life would not invoke a *Creator* as Darwin does, but Darwin's description of the *evolution* of life is still valid today: starting from a small number of *forms* and over time producing many different forms of many different kinds.

It can be argued whether or not biological evolution is the same thing as a computer program, but let us assume for the purposes of this argument that they are the same. In any case, we can act *as though* they were the same, and see how far this takes us in terms of inventing new computer programs. Now, we can make a computer simulation of biological evolution, by *ordering our computer to perform* the rules of evolution laid out in Darwin's seminal work. By analogy with our observations of the natural world, we would expect this computer program to display *endless forms most beautiful and most wonderful*, as does biological evolution. Note that this is (for now) only a thought experiment! It is possible to write such a computer program *in principle*, but the *practice* of actually implementing it is very difficult. This is the field of *evolutionary computation*, that I describe in this chapter. This thought experiment raises two important problems/questions:

1. How can we resolve the seeming conflict between Lovelace's (true) statement that computer programs can only do what we know how to order them to do, and Darwin's (true) observation of evolution as producing many different forms from a simple starting point?

2. Assuming that such a computer program could produce *endless forms most beautiful and most wonderful*, precisely what are these *forms*? How are they different to each other and how do we know that the computer program has genuinely produced something *original*?

The answers to both of these questions involve the notion of *complexity*, but each answer uses complexity in a different way.

2.1.1 Lovelace vs. Darwin

It is true (as Lovelace said) that computer programs can only do what we know how to order them to do. Yet we still hope to implement a computer program that can produce novel things that we did not program into it. The resolution of this conflict is *complex systems*.

Complex systems are systems with simple rules but complicated behaviours. There are some computer programs (for example, fractals [10]) that are very easy

to program (iterating equations and colouring pixels) but for which it is impossible to predict their behaviour (the resulting colours of each pixel). The most efficient way to know the results of some complex computer programs is to run them and observe the results. Thus we can *order our computer programs to perform* tasks that we do not know (and *cannot know*) the outcomes of beforehand.

Biological evolution is a complex system. We can know the simple rules by which evolution operates on a day-to-day basis, but this knowledge does not allow us to predict the *endless forms* that will be produced far into the future. The challenge for evolutionary computation is working out which simple rules to order our computer programs to perform, so that we can observe them generating endless *virtual* forms.

2.1.2 Computer-generated novelty

How do we know when a computer program has produced something original or novel? What exactly does this mean?

When I talk about *novelty*, I definitely *do not* mean merely previously-unseen re-arrangements of fixed components in a template. For example, it is trivial to implement a computer program that shuffles and deals out decks of 52 cards. Such a program could be run continuously, dealing out a newly-shuffled deck every second. If this program had been running for as long as the universe has currently existed, it would be unlikely that the program had every dealt out the same deck twice[1]. This program continually produces novel arrangements of its 52 cards, but this is *not* what I mean by *novelty*.

From a higher-level viewpoint, the card-shuffling program is not producing novelty because it is always performing the same *behaviour* each time. It always deals out a permutation of the 52 cards it was given to start with. It will never deal out 49 cards, or 57 cards. It will never deal out 32 cards and one banana. It will never start building a house out of the cards.

However, biological evolution *does* do the equivalent of the facetious suggestions above. This is what makes biological evolution so interesting from a computational viewpoint: it produces novelty on a scale far in advance of our current computer programs. One way of summarising biological evolution's novelty-generation is to consider Maynard-Smith's *major transitions* that life has gone through [67]. These are the evolutionary transitions:

- From replicating molecules to populations of molecules in compartments.
- From RNA acting as both genes and enzymes, to DNA as genes and proteins as enzymes.

[1]There are $52! \approx 8 \times 10^{67}$ different decks of cards, and 13.77 billion years $\approx 4 \times 10^{17}$ seconds in the history of the universe.

- From prokaryotes to eukaryotes.
- From asexual cloning to sexual reproduction.
- From single-celled organisms to multicellular organisms.
- From solitary individuals to social colonies.
- From primate societies with simple language, to human societies with sophisticated language enabling memes.

Each of these transitions involves life-forms becoming more complex because of evolution. *Complex* in this sense means that after the transition, the life-forms are able to perform more complicated behaviours. They can now do things that were beyond their capabilities before. Complexity in this sense is notoriously difficult to define precisely. John von Neumann articulates this when talking about evolutionary computation [101]:

> There is a concept which will be quite useful here, of which we have a certain intuitive idea, but which is vague, unscientific, and imperfect. This concept clearly belongs to the subject of information, and quasi-thermodynamical considerations are relevant to it. I know no adequate name for it, but it is best described by calling it "complication." It is effectivity in complication, or the potentiality to do things. I am not thinking about how involved the object is, but how involved its purposive operations are. In this sense, an object is of the highest degree of complexity if it can do very difficult and involved things.

This formulation captures the fact that people have an intuitive notion of what it means for one thing to be *more complex* than another, and that this notion is difficult to pin down rigorously. But is has to do with creating objects that can perform behaviours. And objects that can perform more involved behaviours are more complex. For example, the major transitions listed above clearly involve increases in complexity.

One point we must be very careful with when talking about evolution and complexity is the fact that evolution is not *progressive* in any meaningful sense. Biological evolution does not *improve* organisms over time, it does not make them *fitter* over time and it does not increase their complexity over time [4]. The major transitions above have happened during the history of life on Earth, and they have happened one after the other, over time increasing the complexity of the most complex life-forms on Earth. But this does not mean that evolution increases the complexity of life in general, in any meaningful sense. It does not mean that evolution selects for increased complexity. It merely means that different life-forms

grow from each other. Since more complex life-forms evolve from simpler ones, the *most complex life-form that has ever existed* must increase in complexity over time. But there are many examples in biology of complexity decreasing due to evolution. This has been noted by biologists since Darwin [22, pp.108-112].

Traditional computer programs *do* operate progressively, in contrast to biological evolution. Computer programs generally solve a problem by taking progressive *steps towards a goal*, or making successive approximations to a *correct answer*. In trying to implement computer programs based on biological evolution, we must be very careful to avoid the trap of treating biological evolution like a traditional computer program: because it does not operate progressively and does not have a concept of a *goal* or a *correct answer*. I believe that the first generations of evolutionary algorithms did fall into this trap (as evidenced by the use of explicit fitness functions), which I discuss in this chapter.

Because complexity is difficult to define, and because biological evolution does not increase complexity, I think that complexity is not the best feature to try and obtain in computational evolution. This is why I focus on *novelty-generation*. Complexity is definitely an important aspect of evolution, and we need to evolve virtual organisms that can *change* in complexity, because changing complexity is hugely useful and is a major way of generating novelty. But complexity is not the major characteristic of biological evolution. The power of biological evolution is not to improve its organisms over time, or to increase their complexity, but to generate *novel* organisms that perform *different* behaviours. Not *better*, or *more complex*, but *different*.

I fear that later generations of evolutionary algorithms, having avoided the trap of progressing towards goals, may be falling into the trap of progressing towards increasing novelty. This is a subtle issue, because complexity-progressing algorithms would be a useful thing to develop (just as goal-progressing algorithms are useful when applied to appropriate problems). But if we are wanting to develop algorithms that harness the power of biological evolution, then we should focus on that which biological evolution does powerfully. That is generating novelty.

2.2 My approach: Models and the CoSMoS

The general computer science field of bio-inspired algorithms involves observing a biological system that has some interesting property (such as novelty generation) and trying to capture (computationally) the essential features of the biological system that give rise to the property. These algorithms do not attempt to copy the whole of biology inside a computer. This would be impossible, not least because biologists themselves do not understand the whole of biology. Rather, these algorithms take a simplified model of the current understanding of biology, and

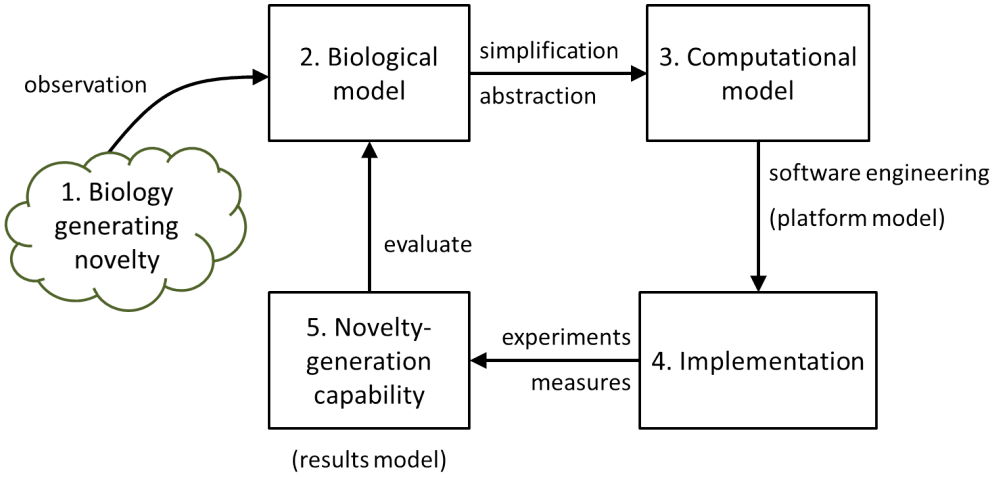


Figure 2.1: The CoSMoS process as I am using it in this thesis. Different models document different stages in the process of building bio-inspired algorithms.

abstract out a computational algorithm from this simplified model.

This is clearly a difficult task, and one that needs approaching carefully. Therefore, we need a principled way of looking at biological systems and producing computer programs from them. The CoSMoS process [94] is a principled way of developing bio-inspired algorithms. Different people will no doubt have their own preferred ways of looking at bio-inspired algorithms, but in this thesis I look at each different algorithm from the viewpoint of the CoSMoS process. This allows me to look at very different algorithms and describe them using the same language, thus giving a coherent description of a complicated research area.

The idea behind the CoSMoS process is to take the imprecise notion of *being inspired by biology*, and make this more rigorous. The process of biological inspiration is documented by the creation of different models to describe different parts of the process. There are subtly different version of the CoSMoS process, so here I describe the version I use (inspired by [3] [94]), which is shown in figure 2.1. It is possible to apply the CoSMoS process using different levels of detail and formality. I use it in an informal (but structured) way, taking the ideas of biological and computational models, and describing these using natural language rather than a formal notation. The CoSMoS process, as illustrated in figure 2.1, is:

1. A part of biology displays some interesting property, which in this case is novelty-generation.

2. The biological system is observed and a *biological model* made. This model describes everything relevant to the property of interest that is known about

the biological system. This model is written in biological language, and can be checked (and argued over) by biologists. This can be thought of as a *requirements document* from software engineering.

3. A description of a biological system is not suitable for turning directly into an algorithm. So a second model is made, this time from the point of view of writing a computer program. This *computational model* describes the biological concepts in computer science terms, and can be thought of as a *design document* from software engineering.

4. Traditional software engineering can be used to turn the computational model into a concrete *implementation*: a piece of code that runs the algorithm. I ignore this stage in the thesis: I assume that software engineers are capable of turning a computational model into a bug-free computer program that can run in reasonable time on some (unspecified) hardware.

5. Given a running computer program, we can measure it and run experiments on it, to see how good it is at generating novelty (taking *novelty generation* to be defined as explained above). We can then compare different biological (and computational) models to see how good each is at generating novelty.

And once the process has been completed, we have a piece of code (step 4) that has had its novelty-generation capabilities measured (step 5). Now we can return to the biological model (step 2) and go around the loop again, with a different biological model, to see if we can change the biological model to make the resulting algorithm more able to generate novelty.

In the remainder of this chapter, I show how novelty-generation algorithms have (implicitly) gone around this loop. Major advances in the abilities of evolutionary algorithms to generate novelty have been gained by changing the biological models (step 2) of evolutionary algorithms to make them more biologically realistic. Changing the computational models (step 3) of different algorithms has led to some improvements, but the most major improvements have come about by changes to the biological models (described later in this chapter).

This is because, for bio-inspired algorithms, every computational model is based on a biological model (whether this is explicitly stated or not). So changes to the computational model are limited to finding different abstractions and computational methods for implementing the same biological ideas. It may be that a particular algorithm could be improved by modifying its computational model in a way that is not consistent with its biological model. But then we would not be able to make further improvements by looking to biology. We would have moved away from bio-inspired computation and towards traditional algorithm design. The reason for looking to biology for inspiration is that we are addressing problems that

traditional ways of designing algorithms have been unable to solve (such as novelty-generation). In order to do this, we choose to base each computational model on a biological model.

The biological model, on the other hand, is based on biological knowledge that is being constantly improved by biology research. As biologists discover more about how biological systems generate novelty, the biological models of bio-inspired algorithms can be improved accordingly. And since the biological models of novelty-generation algorithms are currently lagging behind cutting-edge biology research, we do not need to wait for more biological research before extending our biological models: we can use existing biological understanding to do this.

In the following chapters, I look at extending the biological models of current novelty-generation algorithms to include the concept of meta-evolution. Extending the biological models of our algorithms gives us a principled and robust method for extending the resulting computational models (and hence the implemented computer programs). Using this approach, we are guaranteed to always have ways in which we can extend the novelty-generating capabilities of algorithms (assuming that biologists continue to improve their understanding of the underlying biology, and that the underlying biology actually generates novelty).

2.3 Evolutionary Algorithms

2.3.1 Classical Evolutionary Algorithms

The field of Evolutionary Algorithms started separately in four different places: Rechenberg [86] introduced *Evolutionary Strategies*, which was further developed by Schwefel [12]; Fogel, Owens and Walsh introduced *Evolutionary Programming* [34]; Cramer introduced *Genetic Programming* [19], which was further developed by Koza [58]; and Holland introduced *Genetic Algorithms* [47], which was further developed by Goldberg [38]. All these approaches use the same ideas, of mimicking the biological process of evolution within a computer program, to evolve solutions to computational and engineering problems.

The purpose of this chapter is to talk about biological models and how to extend them, rather than perform a historical review of evolutionary algorithms, which has already been done by others. For a review and further references, see one of the many books on the subject, for example [72].

2.3.1.1 The biological and computational models

While the original algorithms of the field were inspired by biology, their authors did not explicitly state the models of biology that they were using to produce their algorithms. With hindsight we know that explicitly stating the biological model

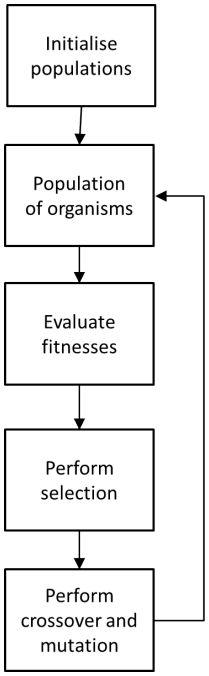


Figure 2.2: Flow diagram for the standard evolutionary algorithm. This loop is iterated as many times as required. Compare with a co-evolutionary algorithm (figure 2.4) and an evo-devo algorithm (figure 2.6).

of an algorithm can be very useful, because it allows us to add more biology into evolutionary algorithms in a principled way (by extending their biological model, see section 2.2). It would also allow us to look at the biological models of different algorithms and compare them with different (including more realistic) models of biology. This would allow us to find ways in which we can extend the simple models to give the resulting computational algorithms more novelty-generating capability.

So in the spirit of this approach, I attempt to reverse-engineer the biological models used in the early algorithms by reference to their computational models, which are explicitly stated in the literature. I concentrate on Holland's Genetic Algorithm as an example of the early evolutionary algorithms, because Holland stated explicitly that he was trying to build an abstraction of biological processes (i.e. a biological model), rather than just an algorithm [47]. We will see that early evolutionary algorithms used very simple biological models. And later algorithms, that are better able to generate novelty, used more biologically-realistic biological models.

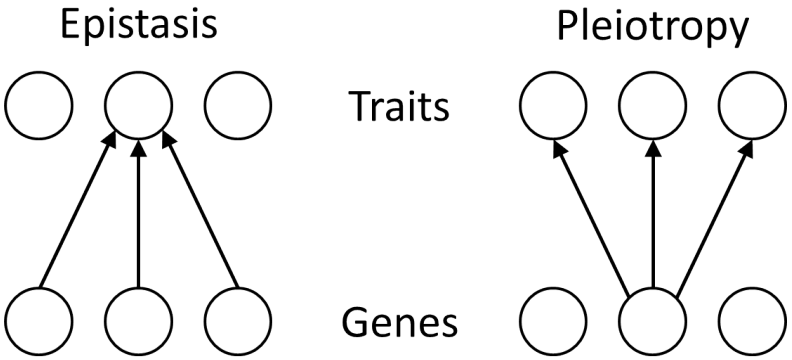


Figure 2.3: Epistasis refers to one trait of a biological organism being influenced by more than one gene in its genome. Pleiotropy refers to multiple traits being influenced by the same gene. The combination of these two effects makes the search landscape of biological evolution non-linear.

The biological model of Genetic Algorithms consists of the following components:

1. Organisms and fitness
2. Chromosomes and alleles
3. Mutation and sexual reproduction
4. Survival of the fittest

which are linked together as shown in figure 2.2. These are explained in detail below, along with the reasons for including them in the biological model.

Organisms and fitness The key idea behind Genetic Algorithms is that evolution via Darwinian selection is a search algorithm [47, p.1]. Traditional computer search algorithms search over a *utility* or *payoff* landscape, which assigns a measure of *usefulness* to each potential solution to a problem. And evolution searches over a *fitness* landscape, which assigns a *fitness value* (reproductive success) to each biological individual, affecting how well it survives to reproduce in its environment.

The benefit of evolution is that where traditional computer algorithms struggle with complicated search landscapes containing many local optima, biological evolution appears to succeed on these landscapes. The non-linear interaction of genes in a biological organism is termed *epistasis* and *pleiotropy*. Epistasis means multiple genes influencing one observable trait in the individual; pleiotropy means one gene influencing multiple traits, as shown in figure 2.3. Holland equates epistasis and pleiotropy with complicated traditional payoff landscapes [47, p.10].

Chromosomes, genes and alleles; Mutation and sexual reproduction We have the concept of evolution as a search algorithm, choosing between different organisms based on their fitness. In order to implement this in an algorithm, we need a way of representing the organisms, and a way of moving between them. The biological inspiration behind this is that the theory of genetics.

Genetics tells us that evolution searches over different *genes*, grouped together on *chromosomes*, that determine an individual organism. The biological model is that an organism is determined by a fixed set of genes, which can take one of several different forms, or *alleles*. Mutations to the DNA of the organism cause genes to change from one allele to another. Also, sexual reproduction between organisms causes combinations of alleles to be rearranged in future organisms. These two mechanisms allow evolution to move between different organisms in the search space, by mutating genes and crossing over chromosomes.

Holland parallels this with traditional search techniques [47, p.4]. For example: in control engineering, the chromosomes correspond to control policies and the ways of moving between policies involve applications of Bayes' rule and successive approximation.

Survival of the fittest Now that we have organisms with fitnesses, and a method of making new organisms from old, we almost have a search algorithm. The only thing missing is the way of deciding which organisms to keep and which to throw away. This is a *search heuristic* from traditional search techniques. In evolutionary algorithms, the search heuristic is the Darwinian idea of *survival of the fittest*. Populations of organisms reproduce exponentially, but with limited resources, so most organisms must die; only the fittest survive.

Since unbounded exponential growth of populations does not happen (for very long!) in biology, and would certainly not be practical within computational algorithms, some decision must be made as to which organisms to kill. Holland's approach is to fix a constant population size [47]. Some individuals die each generation, and some survive to reproduce. Those that reproduce have their reproduction rates normalised to keep the overall population size constant.

2.3.1.2 Comments on the biological model

There are two subtle points on which the biological model described above is not biologically accurate.

Fitness The first point concerns whether a biological organism's fitness *determines* how well it survives to reproduce in its environment, or whether the fitness of an organism is a *description* of how well it survives. The idea of *fitness* is a human construct that we use to describe and classify parts of the world. Biological

organisms do not actually possess a single attribute called fitness. Each organism possesses many varied and nuanced attributes that make it an individual. These multiple, low-level attributes interact with each other and the environment in a very complex way that determines how well a given organism survives. As humans, we are unable to make sense of this complexity by staring at its lowest-level detail, so we make abstractions and simplifications of the detail, producing statements such as: *Giraffes with longer necks were fitter than those with shorter necks.* But as well as helping us to understand a complex world, abstractions such as fitness can help us to design algorithms. We cannot hope (with our current technology) to have a computer simulating all the complex attributes of a biological organism.

We can summarise this complexity in a single attribute called *fitness*. So although biological organisms do not actually have a fitness, we can choose to give our computational organisms a fitness, to avoid simulating all the complex attributes possessed by biological organisms. This saves simulation time, but introduces the problem of deciding how to assign fitnesses to computational organisms. So it is trading off run-time complexity against needing to make more design decisions.

This artificial fitness is similar in some ways to humans selectively breeding biological organisms such as pigeons and dogs. The difference is that in selective breeding, biological evolution is an already-established process for generating novelty, that would carry on without human intervention. Humans just need to observe the novelty and choose breeding stock that seem interesting or useful. But in computational evolution, there is no evolutionary process happening that is separate from the fitness function — the purpose of computational evolution is to implement this process. This is why the fitness function is a key component of these systems, because it must capture all the nuances of biology's low-level detail in a single number.

Optimisation The second point is: the fact that evolution *searches* over a complicated landscape does not mean that evolution *optimises* over this landscape. And in particular, it is not necessarily the case that biological evolution optimises the fitness of organisms, when *fitness* is a human-created measure that we use to understand evolution, rather than an active driver of evolution [80]. This does not stop us from trying to use computational analogues of evolution to optimise computational fitness functions. But it is worth noting that this is not what happens in biology. Because the creators of the early evolutionary algorithms did not explicitly state their biological models, we do not know whether this difference is: (1) a misunderstanding of how biological evolution works, and they mistakenly thought that evolution does optimise fitness; or (2) an application of biological processes to a different problem — evolution does generate and select variation, so maybe this can be used to optimise, even though biology does not use evolution in this way.

2.3.1.3 Variations on the computational model

The above biological model leads to the canonical Genetic Algorithm in which organisms are binary strings with each bit representing two possible alleles of each gene. Some organisms are selected to reproduce each generation, depending on their fitness. Crossover and mutation are used to produce the next generation of organisms, and the process repeats (as shown in figure 2.2).

Once we have a computational model (such as that described above), we can examine it from a computational perspective and ask why different parts must be the way they are. This leads to re-design and optimisation of the computational model, without needing to be constrained by following a biological model. There are many different ways in which the computational model of Genetic Algorithms can be changed. It would not be constructive to list every permutation here, so the following examples highlight some of the types of change that can be made.

Representation There are many different ways to represent a real-world problem in terms of a chromosome of genes. Using a binary string is an obvious approach, as any problem can be represented with a binary string, given a suitable encoding. However, not all binary strings are equal. For example, encoding an integer as a binary string has two obvious encoding methods: (1) direct translation into a binary integer and (2) Gray coding. One can see that these two methods will cause the algorithm to perform differently by considering mutations. A Gray-coded integer can move smoothly between any two numbers, by a sequence of single-bit mutations (this is what Gray codes were designed for). But a directly-coded integer would have to make multiple coordinated mutations to move smoothly between most pairs of numbers, which would be less likely to happen by random chance.

This example also demonstrates that the choice of representation affects how the mutation and crossover operations work. There are different types of crossover (1-point, 2-point, n-point, uniform) and many different types of mutation, all of which affect how the algorithm works. In addition to different ways to encode a problem as a binary string, problems can be encoded as other types of string as well. Evolutionary Strategies began by evolving vectors of real numbers [12], Evolutionary Programming began by evolving finite state machines [34], Genetic Programming began by evolving trees representing computer programs [58], and has been extended to evolve graphs instead of trees [71].

Selection There are different methods of selecting which organisms will survive, and how fecund they will be. The method of assigning fecundity proportionate to how fit an organism is (roulette wheel selection) can make the algorithm suffer from premature convergence [39]. One way around this is to rank the organisms

by their fitness and assign fecundities proportionate to this ranking (ranked selection). A development of this technique is Stochastic Universal Sampling [7], which improves the mathematical properties of this selection method. An alternative selection method is to have subsets of organisms compete against each other, with the fittest organism surviving to reproduce (tournament selection). This method allows flexibility in the type of fitness function used, as an absolute number does not need to be assigned to each organism: all that is needed is a ranking of the organisms in each tournament.

Fitness In many problems, the computational analogue of fitness is not one-dimensional. This makes it difficult to assign a single number to an organism as a measure of its ability to solve the problem. Many problems possess multiple objectives that must be traded off against each other, for example the time complexity and space complexity of an algorithm. Whilst it is possible to combine multiple objectives into a single number (e.g. by a weighted sum) it is often more useful to consider them separately. A variation on fitness is to have a multi-dimensional fitness function (one dimension for each objective) and to consider the Pareto front of generated solutions [48]. Often, examining the solutions found within different regions of this Pareto front can give insight into the problem that is being solved [24]. The idea of using an evolutionary algorithm to gain insight into the real-world problem being solved is not restricted to Pareto fronts: any evolutionary algorithm has the potential to provide insight into the problem being solved, if it is applied towards this end.

2.3.1.4 Computational models are not enough

Changing the computational model of a novelty-generation algorithm is a double-edged sword. It provides a lot of freedom, because potential changes to the algorithm are not constrained by the methods that biology uses. But it relies on those making the changes being able to predict how their decisions will affect a highly non-linear system (or being lucky). While this is certainly possible to a degree, it is difficult to make huge qualitative changes to the algorithm by changing the computational model. This means that varying the computational model will improve the performance of the algorithm in certain situations, particularly where domain-specific knowledge is available about the problem being solved. But it will struggle to improve the novelty-generation capabilities of the algorithm in general.

Stepping back a level and moving to the biological model, however, makes this possible. Major increases in novelty-generation capability may be gained more easily by changing the underlying biological model. This is because the original algorithm was based on a certain model of biology. If we assume that the underlying biology is doing a form of computation that we have so far been unable to

mimic computationally, then it is reasonable to think that changing the biological model to make it more biologically accurate may increase the computational power of the resulting computational model. Two examples of this type of change to the biological model are co-evolutionary algorithms (section 2.3.2) and evo-devo algorithms (section 2.3.3).

2.3.2 Co-evolutionary algorithms

Co-evolutionary algorithms are an example of how changing the biological model of novelty-generating algorithms can make them more able to generate novelty. Co-evolution is an extension to the biological model of classical evolutionary algorithms. There are many examples of co-evolution, both in biology [97] and in evolutionary algorithms [23], but the purpose of this chapter is to talk about biological models rather than to review the literature on co-evolution. So to explain the concept I will use the case-study of sorting networks, which were an early example of a co-evolutionary algorithm.

2.3.2.1 Example: Sorting networks

In some problems, specifying the fitness function is a difficult task. For example, a sorting network is a sorting algorithm that sorts lists of a pre-defined length. It can be represented as a sequence of comparisons to be made between elements of the list. Since there are only a finite number of lists of a set length (as far as sorting is concerned), an intuitive definition of fitness for a sorting network would be the number of these lists that are sorted correctly. However, there is a problem here. The number of possible lists of length n increases exponentially with n, and so it would not be tractable to exhaustively test each evolved sorting network against each possible list for even moderately-sized n. Therefore, a sample of possible lists must be chosen to test against. But this introduces a further problem. How difficult to sort should the sample lists be?

This is the common problem of choosing a benchmark to test against evolved solutions. If the benchmark is set too high, then the search will never be able to get started, as all of the initial solutions will perform very poorly against the benchmark. This will give no information about which solutions are better than which, as they all perform the same against the benchmark (very poorly). On the other hand, if the benchmark is set too low then the search can get started but it will reach a plateau when every population member in the search performs brilliantly against the benchmark. Since the benchmark is set low, none of the solutions are acceptably fit but again there is no information about which solutions are better, since they all perform the same against the benchmark (brilliantly).

A possible way of modifying the computational model to avoid the benchmark problem is to occasionally change the benchmark. However, this creates many problems, not least of which is that a range of benchmark programs are needed, of varying difficulty. A process is needed that can reliably generate benchmarks of a pre-determined difficulty. (Or generate benchmarks of varying difficulties and classify them.) Generating benchmarks at random is unlikely to be sufficient, because they will not have the range of difficulties needed. For example, in the sorting-network problem, randomly-generated benchmarks are too easy to sort [46].

2.3.2.2 Biological model

Stepping back to the biological model, we can change the notion of fitness by realising that a biological organism's fitness is not an isolated number depending only on the organism. It depends on the organism's environment, and in particular on the specific details of other species interacting with the organism. For example, in a predator-prey system it does not matter how efficiently rabbits eat grass if they cannot run away from foxes. And it does not matter how fast foxes can run to catch rabbits if they do not have sharp enough teeth with which to eat them. The foxes and rabbits are caught in a perpetual arms-race or *red queen effect* [100].

2.3.2.3 Computational model

We can use the metaphor of different species interacting and consider sorting networks to be one species and the benchmark data they are tested on as another species. This computational coupling of problem solution and benchmark is analogous to the biological coupling of a predator species and a prey species, or alternatively a host and a parasite. The fitness of a particular sorting network is a function of how well it sorts each data item from the benchmark population. The fitness of a particular data item from the benchmark population is, similarly, a function of how badly each sorting network sorts it. So the sorting networks evolve to do well at sorting the benchmark data, and the benchmark data evolve to be hard for the sorting networks to sort. Figure 2.4 shows how this affects the control flow of the evolutionary algorithm.

Hillis managed to evolve a 16-input sorting network that required only 61 swaps by using co-evolution, but could only manage 65 swaps without. The best known network at the time used 60 swaps [46].

While co-evolution can lead to populations that either oscillate or behave chaotically [52], the hoped-for result can also occur. The two populations synchronise their fitnesses so that population A is just a little bit fitter than population B. Then population B improves a little bit, which causes population A to respond, creating the desired evolutionary arms-race [75] [76].

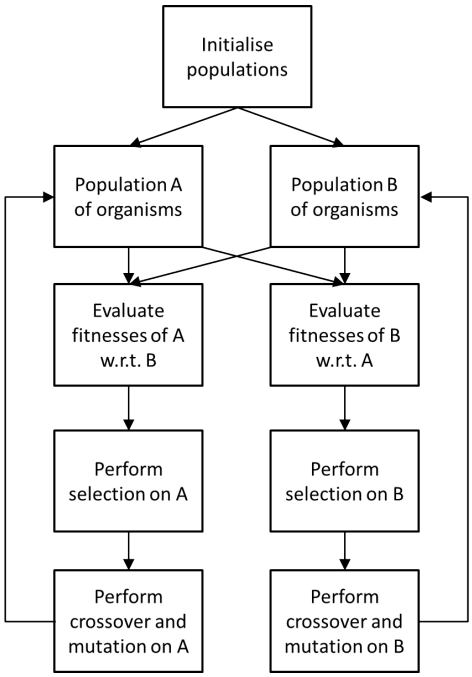


Figure 2.4: Flow diagram for a co-evolutionary EA. This loop is iterated as many times as required. Compare with a standard EA (figure 2.2) and an evo-devo EA (figure 2.6). The key difference between this algorithm and a standard EA is that in co-evolution, two populations of organisms evolve. And their fitness functions depend on the organisms in the other population.

2.3.2.4 Variations on the computational model

Co-evolution is typically split into two different types: competitive and cooperative. Competitive co-evolution is described above. Two populations co-exist, with fitness functions that are the reverse of each other: one population becomes fitter by making the other population less fit. The hope is that this situation will lead to an evolutionary arms race where both populations become *better*, by some external measure. Cooperative co-evolution, on the other hand, splits a large problem into two separate problems. This is to get around the fact that evolutionary algorithms tend to struggle with high-dimensional problems. Each population solves half of the problem, independently of the other half. However, to evaluate the fitness of an individual from one population, a *cooperating* individual from the other population is needed in order to have a full solution to the problem. This is where the fitness functions of the two populations are linked.

Clearly, the above types of co-evolution are not limited to two populations. They can be easily extended to more than two populations, and also to just one population. An example of competitive co-evolution within a single population would be evolving computer programs to play a game such as chess, and using tournament selection to choose which population members should play against each other. Figure 2.5 shows some examples of different ways in which competitors or cooperators could be chosen from a single population or from two populations.

2.3.2.5 Novelty generation

Co-evolution changes the biological model of classical evolutionary algorithms to alter the concept of fitness. Fitness changes from a property of a single organism, to a measurement made of a single organism in the context of other organisms, which are themselves evolving.

This has the effect of taking a design decision of the algorithm (the choice of benchmarks) and having the algorithm make this decision itself. This makes the algorithm better able to generate novelty in two different (but related) ways:

1. It can generate novel benchmarks for itself. This complexity previously had to be given to the algorithm by a human designer, but now the algorithm can generate it for itself.

2. It can generate better solutions to its problem. Having the algorithm generate its own benchmarks makes it better able to solve its application problem.

Note that point 1 does not imply point 2. Just because an algorithm does something for itself, this does not mean that it will be better at doing it than a human would. But in some cases, such as the sorting network example above, algorithms can be better than humans. With a human-designed benchmark, the best evolved

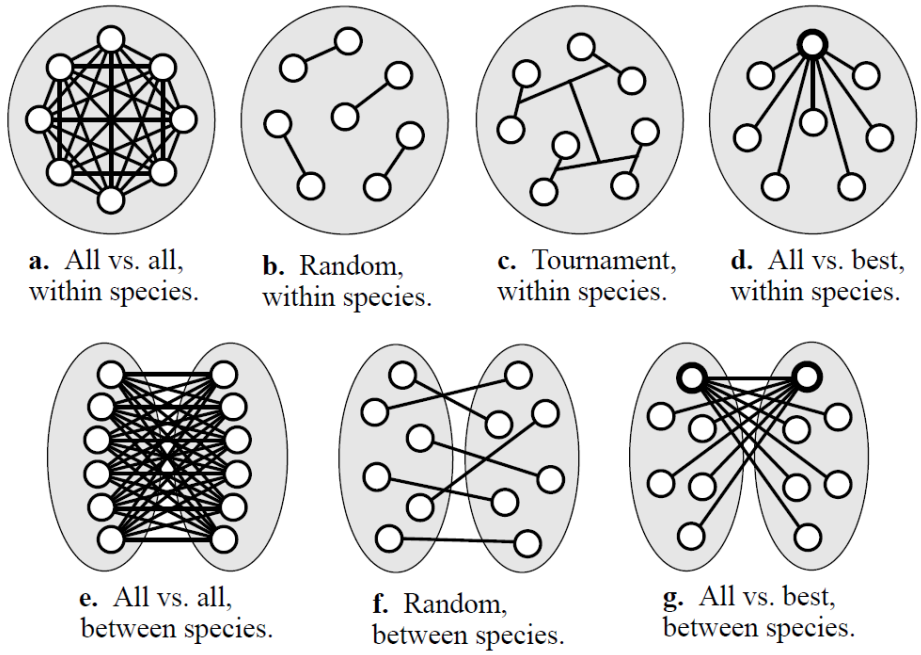


Figure 2.5: Different ways in which competitors or cooperators can be chosen. Two interesting points are: using every individual with every other is expensive; using every individual with the best (rather than with random individuals) gives a standard benchmark. Adapted from Figure 2 of [89].

sorting network contained 65 swaps. But with a co-evolving benchmark, this was reduced to 61. And the best human-designed sorting network needed only 60 swaps. So the novelty-generation algorithm was able to evolve its own benchmark, and was better at determining its benchmark than humans, but was not quite as good as humans at solving its application problem. This example illustrates that when making statements such as: *the computer program is better than humans* (or even: *program X is better than program Y*), we must be very precise about *what* the program is doing better.

The aim of novelty-generation algorithms is to take aspects of algorithms that were previously thought of as design decisions (such as the choice of benchmark problem) and allow the algorithm to vary them for itself. The way in which we are going about this in this thesis is by changing the algorithm to make the design decision an emergent property of a process happening in the algorithm. And the way we decide how to change the algorithm is by finding a biological system that displays this emergent property and using this as inspiration for changing the biological model of our algorithm. The next section shows another example of this.

2.3.3 Evo-devo Algorithms

Evo-devo algorithms are an example of how changing the biological model of novelty-generating algorithms can make them more able to generate novelty. Evo-devo is an extension to the biological model of classical evolutionary algorithms. There are many examples of evo-devo, both in biology and in evolutionary algorithms, but the purpose of this chapter is to talk about biological models rather than to review the literature on evo-devo. So to explain the concept I will describe in detail an example of an L-system, and make reference to other evo-devo algorithms.

2.3.3.1 The problem: Representations

One problem with classical evolutionary algorithms is that the representations they use make it hard to evolve organisms possessing complex structures. In particular, structures involving repetition of parts and symmetry are difficult to evolve [18] [90]. For example, in evolving a controller for a quadruped robot, the control signals sent to each leg should probably contain some symmetry. An example of a representation in which each leg is controlled separately could be a neural network that senses the angles of each leg joint and outputs the change in direction to apply to each joint (with, say, one hidden layer). The representation of the chromosome would be a vector of real numbers specifying the weights of each synapse in the neural network. In order to evolve a symmetric gait (which presumably would help

walking/running), the evolutionary algorithm would have to produce symmetries in the numbers contained in the chromosome, corresponding to symmetries in the weights of the neural network. While this may be possible for small problems, having the algorithm construct and preserve symmetries will become harder as the scale of the problem increases.

2.3.3.2 Biological model

A way in which symmetrical structures can be evolved is by changing the biological model of classical evolutionary algorithms, to modify the ideas of representation and fitness. In classical evolutionary algorithms, each organism is represented by a *chromosome*, which is evaluated to determine its fitness. But biological organisms are more complicated than this. Biologists talk about the *genome* of an organism, which is the information contained on its chromosome, and the *phenome* of an organism, which is the organism's physical presence in the world. The genome of an organism does not act directly in the world. Rather, the genome controls the development of the organism as it grows. The fully-developed organism then acts in the world, and it is only from this action that the fitness of the organism can be determined, and hence the fitness of its genome.

2.3.3.3 Computational model

The computational model of evo-devo algorithms contains a development step, in which the organism's genome controls the development of a phenome. This phenome is then evaluated to determine its fitness. Figure 2.6 shows how this affects the control flow of the algorithm.

The genome can be processed in one go to produce a phenome, in which case there is a simple function mapping genomes to phenomes. Or the development process can be iterated, in which case there is a simple function that is iterated to gradually develop a phenome through some intermediary stages. Iterated development re-uses genes from the genome and enables evo-devo algorithms to reliably construct and preserve symmetries in the phenomes evolved.

An example of development in one go is Genetic Programming [58]. The genome of the organism is the string of code representing the program. This develops into an actual program that is the phenome of the organism. The function that performs this development is the interpreter for the language in which the DNA is written. It is debatable whether *development* in one go, with one application of a simple function, should actually be equated with biological development, or whether it is simply a complicated representation mechanism. What is certain is that development by the use of an iterated function can produce complex organisms more easily than using one application of a simple function.

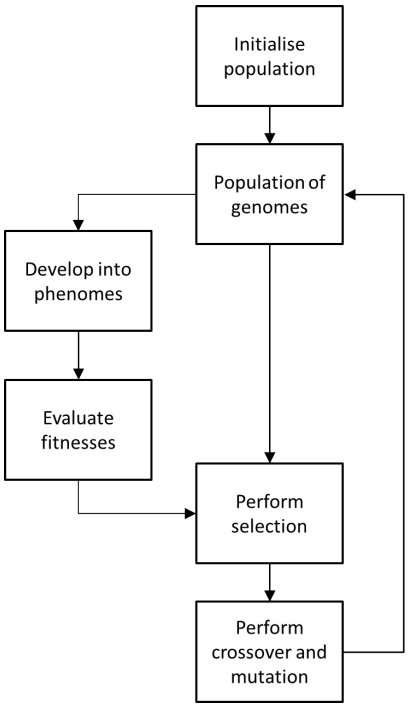


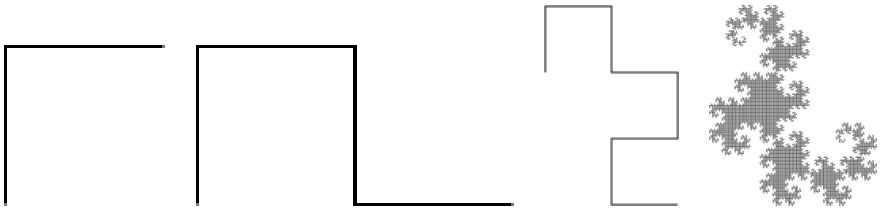
Figure 2.6: Flow diagram for an evo-devo EA. This loop is iterated as many times as required. Compare with a standard EA (figure 2.2) and a co-evolutionary EA (figure 2.4). The key difference between this algorithm and a standard EA is that in evo-devo, the genomes go forward to the next generation, but it is their phenomes that are evaluated to determine fitness.

Initial seed: F_l

Rewriting rules: $F_l \mapsto F_l + F_r +$
$F_r \mapsto -F_l - F_r$

1 step: $F_l + F_r +$
2 steps: $F_l + F_r + + - F_l - F_r +$
3 steps: $F_l + F_r + + - F_l - F_r + + - F_l + F_r + - - F_l - F_r +$

Figure 2.7: The definition of an L-system that generates the dragon curve (from [81, p.11]). The genome of this organism is the initial seed and the rewriting rules. The phenome is the developing string of F, $+$ and $-$ characters.



(a) 1 step (b) 2 steps (c) 3 steps (d) 15 steps

Figure 2.8: A turtle graphics visualisation of the developing dragon curve defined in figure 2.7, interpreting F as *draw a straight line*, $+$ as *turn right* 90° and $-$ as *turn left* 90°

An example of a type of iterated function that can develop a genome into a phenome is L-systems [81]. Figure 2.7 shows an example of an L-system. The genome of this organism is the initial seed and the rewriting rules. The phenome is a string of F, $+$ and $-$ characters that starts out as the initial seed. At each development step, the rewriting rules change the phenome of the organism into a more complicated string of characters (they are a parallel, generative grammar).

The potential for systems of this type to generate symmetries in phenomes may be clear from looking at the developing strings of figure 2.7. But it should be even clearer from looking at figure 2.8. This shows the string produced after fifteen development steps of this system, interpreted as commands to a turtle graphics package that turns the string into an image. It is the famous fractal known as the *dragon curve*. Following this method, the complex strings produced by L-systems can be transformed into complex structures in other domains (such as images). There are many examples of evo-devo algorithms that use L-systems for their development step, for example [50] develops moving creatures and [49] develops coffee tables.

2.3.3.4 Variations on the computational model

An iterated development function can use different types of input to determine how to develop the phenome. Taking L-systems as an example, there are different types of L-system that take into account different inputs at each development step. For a full description of these different types, see [81].

- Deterministic 0L-systems: Each component part of the phenome develops independently of the others, according to a fixed function (e.g. figure 2.7).
- Parametric L-systems: The parameters to the development function can change over time
- Context-sensitive L-systems: Different parts of the developing phenome can interact to determine how each part will develop.
- Probabilistic L-systems: Different parts of the phenome develop in different ways, according to a probability distribution.
- Environmental L-systems: The development process can take into account an environment (that can also contain developing phenomes).

There are many development functions other than L-systems. For example, HyperNEAT [18] is a method of evolving the structure of neural networks, and Karl Sims' work uses a representation based on directed graphs [90, 89]. But the purpose of this section is not to review all the development functions. This section shows that a small change to the biological model of evolutionary algorithms (adding in a development step) can open up a whole range of different computational models, and hence different algorithms, that are more able to generate novelty than previous algorithms.

2.3.3.5 Novelty generation

Evo-devo changes the biological model of classical evolutionary algorithms by splitting organisms into a static genome and a developing phenome. This changes the concept of fitness from a measurement taken of a static organism, to a measurement taken of an organism's dynamic phenome.

Since developed individuals typically contain a lot of internal structure that changes over their development, their fitness functions typically need to be more complex. For example when evolving a robotic controller, there is no mathematical function that can be evaluated to determine the fitness of a given controller. Rather, the controller must be tested on the vehicle that it is to control (either in simulation or in hardware). Its performance (fitness) must be determined after observing the controller operating for a period of time.

One way in which fitness tests can be designed uses competitive co-evolution. Carrying on the example of robot controllers, they are being designed to fulfil some particular task (such as locomotion, for example). Two controllers can then be set to work on the same task competitively. The controller that completes the task first, or does best at the task, has the higher fitness. Combining this with tournament selection gives a full fitness calculation and selection algorithm for robot controllers. For example, Karl Sims used this technique to evolve vehicles and controllers to swim, walk, jump [90] and control an area of space [89].

From the point of view of novelty-generation, the switch from organisms (and fitness) being static to being dynamic represents an increase in the novelty-generation capabilities of the algorithms. Throughout this chapter, we have seen novelty-generating algorithms go through three stages in the complexity of the organisms that they are able to evolve:

1. Static structures that exist in an isolated, mathematical space (classical evolutionary algorithms).

2. Structures that are able to interact with each other, and compete or cooperate (co-evolution).

3. Dynamic structures that can grow and interact with each other and with an environment (evo-devo).

The next section (artificial life) carries on this theme of using different biological models to explore different notions of *fitness*, and evolve different types of organism. Artificial life is not an extension of evolutionary algorithms; it is a different field of research. But it is closely related to evo-devo algorithms. It extends the concept of organisms developing in an environment.

2.4 Artificial life

Artificial life systems are fundamentally different from evolutionary algorithms in that they do not attempt to solve or optimise an explicit problem. Rather, they attempt to replicate the open-ended evolution seen in biology, by setting up a system and simply letting it run, without imposing an external fitness function.

Artificial life systems can be seen as evolutionary algorithms that are able to change their fitness function. Co-evolution allows evolutionary algorithms to change the benchmarks used by their fitness function (see section 2.3.2), by encoding the benchmarks as organisms in the algorithm. Artificial life takes this idea further, and encodes *the entire fitness function* within the organisms and the environment in which they exist. The fitness function changes from being a design decision to being an emergent property of the algorithm. This changes the

problem of *designing a good fitness function* into the problem of *designing a good environment* in which a good fitness function can emerge.

There are many different artificial life systems. The purpose of this section is not to review them all, but to explore the consequences of implicitly-defined fitness functions. To this end, this chapter focuses on a few key examples of artificial life systems.

2.4.1 Biological motivation

The biological metaphors for artificial life come mostly from ecology, in contrast to the biological metaphors for evolutionary algorithms which come mostly from genetics. The main ideas of ecology that artificial life uses are the notions of organisms living, reproducing and dying in an environment.

Living organisms exist in an environment, which contains other living organisms as well as non-living objects. Organisms act on their environment and are in turn acted on by their environment (including living organisms acting on each other). Organisms consume resources from the environment and deposit waste into the environment (and waste from one type of organism may be a useful resource for another). When organisms have enough resources and are in the correct situation, they can reproduce. But if they do not acquire enough resources or they enter the wrong situations, they may die.

This implies that the fitness of a genome is its ability to produce organisms that can survive and reproduce in an environment. The idea of fitness as survivability is in sharp contrast to evolutionary algorithms, that view fitness as an external function to be optimised. As discussed previously (in section 2.3.1.2), fitness as survivability is more biologically accurate than fitness as an external function. This implicit notion of fitness allows biological evolution to continually produce novelty. Because there is no fixed notion of a *fit* individual, biological organisms can always find a new way of surviving in any given environment. And when one species evolves to survive in a new way, this changes the environment and enables other species to find yet more ways of surviving. This continual proliferation of novelty is what artificial life systems aim to achieve. None have yet managed unbounded novelty generation. But some have taken the first steps along this road, creating systems that are seeded with some pre-defined ways of surviving, and watching a limited number of new ways evolve in the system.

2.4.2 Computational models

In contrast to evolutionary algorithms, where populations of individuals are reproduced synchronously, artificial life models tend to work asynchronously at the level

of reproduction. Organisms develop and replicate at different speeds in an environment containing limited resources. Fitter organisms can replicate more quickly, and so consume more resources than less fit organisms. This provides an implicit fitness function (reproducing quickly and consuming resources effectively) rather than having to define an explicit fitness function that is external to the world in which the organisms live. Although the high-level behaviours of the organisms happen asynchronously (reproduction and gathering/using resources), the low-level details of the organisms can be implemented in a synchronous world. For example, if the organisms are self-replicating emergent structures within a cellular automaton [88] then the underlying cellular automaton can be updated synchronously, but emergent structures of different sizes and compositions can replicate at different speeds.

Because of asynchronous reproduction and implicit fitness, co-evolution plays a huge role in artificial life. The selection pressures that drive evolution in artificial life systems come from the particular makeup of species that are present within the system at any given time, rather than from an externally-imposed fitness function.

There are many different artificial life systems, and many reviews that enumerate them (for example: [27] [61]). The purpose of this section is not to review the entire field of artificial life, but to present the biological model underlying the philosophy of artificial life, illustrate this with a few selected examples and link these to the biological models of evolutionary algorithms, with a view to developing a biological model of meta-evolution. To this end, the following sections describe two implementations of the artificial life biological model: *automata systems* and *artificial chemistries*.

The class of *Automata systems* contains *Tierra* and *Avida*, which are the most famous artificial life systems, and are (to date) the artificial life systems best able to generate novelty. These are part of a more general class of systems called *artificial chemistries*. I describe artificial chemistries because they encompass a very wide variety of artificial life systems, and they are the computational model I use to investigate meta evolution in later chapters.

2.4.2.1 Automata systems

A family of artificial life systems has been developed over time, in which the organisms are computer programs and their environment is the memory and processor of a computer. Multiple organisms/programs compete for the resources of processor time and memory space. The theoretical treatment of these ideas goes back to John von Neumann [69], although he was not able to run his programs on fast computer hardware. In this section, I describe the different modern-day instantiations of these ideas.

Core Wars The original member of this family of systems is a programming game called Core Wars [26]. (There was an earlier game, Darwin [29], that partly inspired Core Wars but did not show very complex behaviours.) To play the game, human competitors write programs in a simple assembly language called Redcode. Two programs are then run in parallel on a simulated microprocessor. The programs move through the memory of the computer, writing and executing instructions. The aim of the game is to force the other program to execute an invalid instruction, thus terminating its execution (i.e. killing it).

Human programmers playing the Core Wars game produced a number of different *species* of program that play the game using very different strategies. But no-one managed to produce a *best* program that *wins the game* and beats every other program. Within one species, some programs can consistently beat others, but this does not hold across species. The fitnesses of the best programs from each species are not transitive: each species can consistently beat at least one other species, but will in turn be beaten by at least one species. This is interesting from an evolutionary perspective (particularly with respect to co-evolution) because it shows that a given program is only effective against certain opponents, and can always be beaten. This is in stark contrast to the earlier game, Darwin, in which a best program was discovered [29].

Coreworld Core Wars was transformed from a programming game into a research tool by Rasmussen et al. They produced an artificial life system known as Coreworld [84]. Instead of human programmers creating new species of program, random fluctuations were introduced into the copying of instructions and the starting of new processes, to enable evolution to happen as the system was running. Also, each memory location was provided with an amount of computational resource (a number) that is used up by executing programs and is refilled by a constant influx. This is an example of a *limited resource* from the biological model described above. It means programs that execute *too much* can *starve* and die, thus creating an implicit fitness function that selects for programs that can copy themselves efficiently.

The novelty-generation consequences of these changes are that qualitatively different program behaviours are produced depending on the influx rate of resources. Low resource influx evolves only simple, cyclic programs; but high resource influx promotes more diversity and more complex, cooperating networks of programs. This shows that the environment can have an impact on the novelty-generation capabilities of a system. Also, the types of complex structures evolved under the high resource conditions are not the same on each run of the program. The particular structures evolved depend on chance events, or *frozen accidents*, that happen early in the run (as is the case in biological evolution [21]). An interesting property

of Coreworld is the idea of contingency playing an important role in the evolution of the system.

Tierra Coreworld was modified by Ray to create Tierra [85]. In Coreworld, because instructions can be written by any process to any memory location, processes become entangled. It is difficult to distinguish small processes from each other and from the environment. Whilst this may not always be a bad thing, Ray avoided it in Tierra by introducing memory management. Processes can allocate blocks of memory to themselves, that other processes cannot write to. A global memory manager protects this memory. This memory management makes distinguishing separate organisms easier.

Another major contribution of Tierra is addressing memory locations by templates rather than by relative addresses. This is inspired by the way in which biological enzymes bind to each other using complementary shapes. Templates are defined by sequences of two different *no-op* instructions (that do nothing when executed). Template addressing removes some of the brittleness of the Coreworld programs, enabling the program's instructions and operands to mutate separately. This is an example of where changing the biological model of the system (rather than the computational one) can introduce more novelty-generation capability into a system. It removes one of the brittle design decisions of the computational system (relative addressing) and replaces it with a softer biological analogy (template addressing).

One consequence of these changes is that self-replicating organisms no longer appear spontaneously in Tierra, as they did in Coreworld (i.e. when randomly generating programs, the chance of generating a self-replicating program is negligible). Self-replicators can be forced to appear by reducing the instruction set [77], so that the probability of randomly generating a self-replicating program is increased. However, this is a classic case of *programming the answer into the algorithm*. Reducing the instruction set just makes the random generation of self-replicating programs more probable, it does not make the idea of self-replication more of a concept that the system can evolve. It is a change to the computational model of the algorithm rather than a change to its biological model. Because self-replicating organisms do not appear spontaneously in Tierra, an initial (hand-designed) self-replicating organism must be used to bootstrap an evolutionary run. This initial Tierran organism can evolve by reproduction with mutation, into different species. However, these different *species* are very similar to the bootstrap program, differing from it by only a few point mutations [27, p.35]. One particular type of behaviour that can be evolved is parasitism. Arms races have been observed between parasites and hosts [85].

Avida Tierra was also extended, into Avida [2], by Adami. One of the main drawbacks of Tierra is that there are two global queues, which are necessary to implement the instructions of the Tierran assembly language. This feature of Tierra is a consequence of the fact that it is emulating a conventional microprocessor. However, it means that any program can potentially interact with any other program in only one timestep. This global connectedness of the programs limits their evolvability by allowing parasites to spread rapidly through the population of organisms.

In an attempt to evolve more novelty than the parasites of Tierra, Avida extends Tierra's biological model to include the concept of space, and remove the global connections between all organisms. Programs in Avida live on a two-dimensional grid, and can only interact directly with their eight neighbours. This imposition of locality into the world enables different species of program to evolve separately in different spatial locations.

2.4.2.2 Artificial chemistries

All the algorithms described above use metaphors from biology and ecology, talking about *organisms* living in an *environment*. But the computational organisms are vastly simplified versions of their biological counterparts, and the environments they exist in contain nowhere near the level of detail of biological environments. Effectively, the computational organisms are data structures with a simple definition (e.g. strings over an alphabet, or programs from a language), that interact with each other via a small number of well-defined means (e.g. cross over two strings, read/write instructions). Described in this way, these algorithms seem more like models of chemistry than of biology: chemicals are structures built from atoms, which interact with each other via the making and breaking of bonds. The chemical world is still much more complicated than these computational worlds, but is simpler than the biological world. So the language of chemistry can provide us with another set of metaphors that we can use to develop novelty-generation algorithms. *Artificial chemistries* use these metaphors explicitly, to build computational models inspired by chemical processes. The phrase *biological model* is not really appropriate for models of chemistry, so I will use the more general phrase *physical-world model* to refer to biological or chemical models, or any model of (a subset of) the physical world that has properties we want to emulate computationally.

The basic units of an artificial chemistry are analogues of chemicals, and the actions they can perform are analogues of chemical reactions. The environment in which the chemicals reside is an analogy of a well-mixed test tube (aspatial environment) or a petri dish (spatial environment). A bootstrap collection of artificial chemicals (which can be randomly generated) is put into an artificial test tube and allowed to react. From a novelty-generation point of view, the aim is to

design an appropriate system of chemicals and reactions so that as the contents of the test tube react, the collection of chemicals becomes more complex than the initial bootstrap. There are many different implementations of artificial chemistries (see [27] for a review); as with the other systems described in this chapter, I focus on a few examples to illustrate the underlying physical-world model.

There is no shortage of metaphors from chemistry for reactions that would not remain contained within a real-world test tube (i.e. explosions). The computational result of simulating such reactions would be filling up all the available memory or overflowing whatever data type had been used. In order to avoid these problems, a *chemostat* environment can be used, in which the volume of the mixture is artificially held constant. This can be roughly implemented by keeping the number of chemicals in the system constant. This imposes a selection pressure in favour of chemicals that reproduce quickly, where *quickly* is relative to the rate of removal of material from the chemostat, which is in turn relative to the rate at which other chemicals are reproducing. So there is a *competitive co-evolutionary process* occurring, that selects for quick reproduction. The result of running such systems, therefore, will be to evolve the most efficient replicators that are representable by the system. Note that *replicators* need not be just individual chemicals, they can be collections of chemicals that mutually reproduce each other: for example, autocatalytic sets [56].

A common feature of selection pressure for efficient replicators is the evolution of *parasites*. These are chemicals that exploit *host* chemicals which replicate the parasites but are not replicated by the parasites in return. There are two different types of parasite.

1. *Facultative parasites*: chemicals that can be replicated by a host, but can also replicate themselves (for example: [70]).

2. *Obligate parasites*: chemicals that are able to be replicated by their host, but are not able to replicate themselves (for example: [25]).

Explicit definitions An important classification of artificial chemistries is whether the chemicals and reactions are explicitly or implicitly defined. For example, we could define an artificial chemistry explicitly as follows:

$$\text{Chemicals} = \{A, B, C, D\},$$

and reactions defined as follows:

$$\begin{aligned}\text{Reactions} = \{&A + B \mapsto C,\\ &C + D \mapsto A,\\ &B + D \mapsto B\}.\end{aligned}$$

Whilst this is a useful tool for modelling systems where the chemicals and reactions are few in number and are all known beforehand, it is very limited from a novelty-generation viewpoint.

Implicit definitions A more appropriate way of designing an artificial chemistry for novelty-generation is to have implicit definitions of the chemicals and reactions. In an implicit definition, the chemicals contain some internal structure that can be rearranged to make new chemicals from existing ones. The reaction rules are defined in terms of how reactions alter and rearrange this internal structure. An example of this is the number-division chemistry [9]. The structure given to the chemicals is that they are positive integers greater than 1:

$$\text{Chemicals} = \{2, 3, 4, 5, 6, 7, 8, 9, 10, \dots\}. \tag{2.1}$$

The reaction rules are defined implicitly by taking two numbers a and b, and replacing b by the quotient b/a if this quotient is itself a positive integer greater than 1 (i.e. a divides b without remainder):

$$\text{Reactions} = \{[(a, b) \mapsto (a, b/a)] \text{ where } a, b \text{ and } b/a \in \text{Chemicals}\} \tag{2.2}$$

This implicit definition defines an infinite number of chemicals and an infinite number of reaction rules. Some examples of valid reaction rules are:

$$\begin{aligned}\text{Reactions} = \{&[(2,4) \mapsto (2,2)], [(2,6) \mapsto (2,3)], [(2,8) \mapsto (2,4)], \dots \\ &[(3,6) \mapsto (3,2)], [(3,9) \mapsto (3,3)], [(3,12) \mapsto (3,4)] \dots \\ &[(4,8) \mapsto (4,2)], [(4,12) \mapsto (4,3)], [(4,16) \mapsto (4,4)] \dots\}.\end{aligned}$$

The result of running this artificial chemistry on an initial population of randomly-chosen chemicals is that after a while, the population of chemicals will stabilise and no reaction rules will be applicable to any remaining chemicals. At this point, the population will contain only prime numbers (the prime factors of the initial population). Whilst this is not the most efficient implementation of a prime number generator, it demonstrates that repeated application of simple rules can uncover and exploit structure that is embedded in the definitions of the artificial chemicals and reaction rules.

The population of prime numbers left behind by this artificial chemistry is interesting from a novelty-generation point of view, because it contains a structure not present in the initial population. But, as with the early evolutionary algorithms (section 2.3.1), this is a static structure that does not have the ability to *do* anything. The next example of an artificial chemistry is the equivalent of evo-devo algorithms (section 2.3.3) in that it evolves a collection of chemicals that are complex in the way in which they *act* on each other, rather than the way in which they are composed.

Example: AlChemy Another example of an artificial chemistry with an implicit definition is AlChemy [35], developed by Fontana. This represents chemicals as programs in the very simplified functional programming language of the lambda calculus. Since the lambda calculus is an interpreted language, a chemical can be considered either a string of code or as a program that can take a string of code as input. This allows a reaction to be defined between chemicals A and B as applying the function represented by A to the data represented by B (as long as this function execution terminates in a reasonable amount of time). For full details of this definition, see [35]. As an example, some possible AlChemy chemicals are [35, p.187]:

$$A = (*(> (-a))(> (*(*('(> a))(+(+('a))))a))) \tag{2.3}$$

$$B = (> (*(*('(> a))(+(+('a))))a)) \tag{2.4}$$

$$C = ((a)(> (*(*('(> a))(+(+('a))))a))) \tag{2.5}$$

and an example reaction (using the chemicals above) is:

$$A(B) = C \tag{2.6}$$

This example is intended to give a flavour of the AlChemy language, rather than a deep understanding of its operation. For further details, see [35].

AlChemy systems are initialised with random chemicals and allowed to react, to see what happens to the collection of chemicals. Invariably, what happens is (in Fontana's terminology) the emergence of a *level-0-organisation*. This is a collection of a small number of closely-coupled replicator chemicals that are either universal replicators ($U(X) = X$ for any chemical X) or almost-universal replicators. The evolutionary reason for the emergence of these structures is that there is selection for them. In a reaction vessel of limited size, that is running a chemostat (as described above), there will be a selection pressure for chemicals or sets of chemicals that can reproduce quickly.

In order to make AlChemy better able to generate novelty, Fontana alleviated some of this selection pressure by changing the computational model of AlChemy. He changed the definition of a reaction to exclude those reactions that directly replicate one of the reactants (banning universal replicators). This produced *level-1-organisations*, which are very stable sets of huge numbers of molecules that are robust against the addition of new chemicals to the system. Once *level-1-organisations* had been created, *level-2-organisations* could be created by taking two *level-1-organisations* and putting them into the same chemostat, hoping that one did not out-compete the other. A *level-2-organisation* is the stable co-existence of two *level-1-organisations*, joined together with some *glue* chemicals. Note that *level-2-organisations* do not evolve spontaneously in AlChemy: they have to be

hand-crafted by separately evolving two *level-1-organisations* and manually putting them together in a new chemostat. And no more complex structures have been evolved in AlChemy.

Interestingly, these levels of organisation are not a feature just of AlChemy. It can be shown that many different implicitly-defined artificial chemistries share the same properties, even to the point where chemicals from one system can be equated with chemicals from other systems, with precisely the same networks of reactions seen between them [36]. No structures more complex than *level-2-organisations* have been evolved in an implicitly-defined chemistry simulated in a chemostat. The reason for this is the selection pressure towards efficient replicators that a chemostat naturally implies.

Fontana et al. noticed a limit in the ability of their AlChemy system to generate novelty, and they tried to change the system to overcome this limit. But their efforts were geared towards changing the computational model of their system (by excluding certain reactions and artificially bringing together separately-evolved sets of chemicals). I believe that the novelty-generation capabilities of artificial chemistries might be increased by changing their physical-world model, as demonstrated by co-evolutionary and evo-devo algorithms increasing the novelty-generation capabilities of evolutionary algorithms by changing their biological model. I explore this idea in the following chapters.

2.4.3 Novelty generation

Artificial Life systems can be seen as evolutionary algorithms with a different physical-world model. This model includes the idea of organisms (or chemicals) living, reproducing and dying asynchronously in an environment. The consequence of this is another change in how novelty-generation algorithms interpret the concept of *fitness*:

1. Classical evolutionary algorithms have a static fitness function that is external to the organisms and their world.

2. Co-evolutionary algorithms enhance this fitness function, allowing it to take other organisms into account, so organisms can change the fitness functions of other organisms.

3. Evo-devo algorithms change the fitness function again, from a single measurement of a static structure, to an assessment of how well a dynamic organism performs over time.

4. Artificial life makes a further change, transforming fitness from an external function into an emergent property of the world in which the organisms live.

In artificial life systems, fitness is no longer a design decision that must be made correctly; the system can change its own concept of fitness. But now, the design decision has shifted up a meta-level. Instead of designing good fitness functions, artificial life researchers have to design appropriate virtual worlds in which good fitness functions can (and will) emerge.

The types of novelty that artificial life systems can generate is focused on replication. Artificial life systems are initialised with organisms that can replicate themselves. They are either bootstrapped with hand-designed replicators, or they are designed such that randomly-generated organisms have a good chance of being self-replicators. From these simple self-replicators, more complicated systems of mutual replication can emerge, such as hypercycles [60] and autocatalytic sets [55]. A very common feature of artificial life systems is the emergence of parasites. These are organisms that can be copied by others, but do no copying in return.

The *artificial chemistries* described above serve as a useful, simplified model of novelty-generation, but they do not give the impression of organisms that are *alive*. It is debatable precisely what constitutes an *organism* in the artificial chemistries described above. Is it an individual chemical that floats in a chemostat? Or is it a collection of chemicals that mutually sustain each other in a chemostat? The precise answers to these questions do not concern us here. It is just interesting that artificial chemistries have rich enough behaviours for these questions to arise.

Artificial life systems try to make evolutionary algorithms better able to generate novelty by implementing part of the evolutionary algorithm (the fitness function) in a lower-level world. The next two sections describe two different methods of trying to achieve the same thing. Firstly by making novelty the specific focus of evolution; and secondly by implementing evolutionary algorithms that evolve evolutionary algorithms, i.e. meta-evolution.

2.5 Novelty search

What is the best way of changing the fitness function of evolutionary algorithms to make them better at generating novelty? One radical answer to this question, given by Stanley et. al., is to completely replace the fitness function of evolutionary algorithms with a direct reward of novelty. Stanley et. al. term this idea *novelty search*. They introduce it in [62] and describe it comprehensively in [63].

2.5.1 Biological motivation

Stanley et. al. describe the biological model of novelty search in [63]. Some Biologists claim that biological evolution imposes no selection pressure towards increasing complexity (see [63] for references). Moreover, some go as far as saying

that when species are under strong selective pressure to survive, they tend not to increase in complexity. And when there is weak selective pressure to survive, complexity increases by drift in complexity space.

Biological evolution searches passively for novelty. If organisms find new ways of making a living, then they do not need to compete as fiercely with other organisms in their environment. Thus they have a weaker selection pressure for survival and are able to drift through their evolutionary space, finding further new ways of making a living. This positive feedback is what makes biological evolution generate continual novelty.

Novelty-generation leads to increasing complexity, if it is coupled with gradual growth of the organisms. Biological organisms evolve gradually. They cannot suddenly take massive jumps in phenotype space; they must accumulate many small changes over time. Starting from a low level of complexity, there are only so many ways of making a living, because low-complexity organisms are limited in their possible behaviours. Thus any novelty-generation mechanism will exhaust the possible ways of making a living at low complexity. Then, the only new ways of making a living will require increasing complexity. So over time, generating continual novelty will also generate increasing complexity.

It is important to note that these views are contentious and are currently being debated by biologists.

2.5.2 Computational model

Stanley et. al. take the above biological model to its ultimate conclusion: if what we are interested in from evolutionary algorithms is novelty, then we should remove the fitness function because it hinders the evolution of novelty. Artificial life systems follow a similar argument, removing the explicit fitness function of evolutionary algorithms and replacing it with an implicit need to survive and reproduce in a virtual world. Novelty search differs in that it replaces the fitness function with an explicit *novelty function*, rewarding *novel* solutions rather than *fit* ones.

The use of an explicit function to reward novelty stands in stark contrast to artificial life systems, and is not the same as the biological model above. Where biology searches *implicitly* for novelty, the novelty search algorithm does this *explicitly* via a user-defined novelty function. Stanley et. al. are aware of this difference. They state [63, p.9] that they consciously choose not to follow how biology evolves novelty, because they want their system to be easily-applicable to engineering problems.

This is an interesting research direction, but it is not what I am doing in this thesis. I have stated that I *do* want my systems to have realistic biological models (section 2.2), because I believe this to be a sustainable way of improving bio-inspired algorithms over time (although it is slower to achieve immediate results).

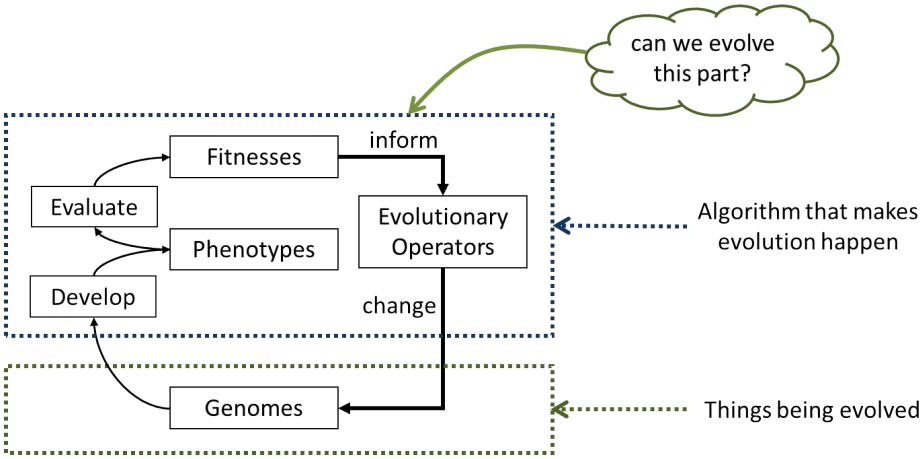


Figure 2.9: Evolutionary algorithms use a fixed algorithm to evolve genomes. Meta-evolution asks whether it is possible to evolve the evolutionary algorithm, as well as the genomes.

For this reason, my work follows the artificial life route of implicitly searching for novelty, rather than the novelty search route of explicitly searching for novelty. The following section describes an extreme version of novelty-generation: meta-evolution.

2.6 Meta-evolution

Meta-evolution is a branch of evolutionary algorithms that takes evolution to an extreme. It realises that evolution can be used to evolve organisms, and wonders: *can evolution can be used to evolve evolutionary algorithms as well?* From a novelty-generation point of view, evolutionary algorithms have a fixed algorithm that makes evolution happen. As shown in figure 2.9, evolutionary algorithms develop genomes into phenomes, evaluate their fitnesses and use the fitnesses to inform evolutionary operators that change the genomes. They use this algorithm to generate novelty in the genomes of their organisms. Meta-evolution asks whether an evolutionary algorithm can also generate novelty in its evolutionary algorithm.

The problem with meta-evolution is that it has not been very successful. There have been no landmark algorithms conclusively showing the benefits of meta-evolution over non-meta evolution, and I do not present such an algorithm in this thesis. In this section, I detail some attempts at meta-evolution from the literature. I say why I think they have not been very successful: because they were changing the computational model of evolutionary algorithms, rather than the biological one. In an attempt to address this, I present a biological model of meta-evolution.

I say what this means for computational models of meta-evolution, and explore these computational models in the remainder of the thesis.

Another problem with meta-evolution is that it is a rather vague concept. *Evolving an evolutionary algorithm* can mean many different things, which makes meta-evolution easy to misunderstand or dismiss as being impossible (by thinking only of one type of meta-evolution). So towards the end of this section, I present my classification of the different types of meta-evolution. I choose one type (that fits with my biological model) to concentrate on for the remainder of the thesis.

The purpose of this thesis is not to present a landmark algorithm that demonstrates meta-evolution. Rather, my purpose is to present a clear picture of what meta-evolution means, and how (one version of) it can be seen as a development of novelty-generation algorithms from the first evolutionary algorithms, through co-evolution, evo-devo and artificial life, to meta-evolution.

2.6.1 Changing the computational model

Previous attempts at meta-evolution have tried to change the *computational model* of evolutionary algorithms in order to allow parts of the algorithm to evolve. I describe three such changes:

1. Evolving the parameters of an evolutionary algorithm while it is running.
2. Evolving populations of parameters for evolutionary algorithms.
3. Co-evolving the evolutionary operators of an algorithm, along with the organisms.

These approaches did not always have the explicit aim of increasing *novelty-generation*. Sometimes they were just trying to better solve their application problem.

2.6.1.1 Evolving parameters

Some of the early Evolutionary Strategies evolved the mutation rates of their algorithms [12]. Mutation rate is a floating-point number that is a parameter of all evolutionary algorithms. In Evolutionary Strategies, the organisms are vectors of floating-point numbers. So another number can be added to the end of this vector, giving each organism its own mutation rate. This is shown in figure 2.10. Mutation rate has changed from being a parameter of the algorithm to a property of each individual. This means that the mutation rate can be different for each individual, and these mutation rates can mutate and evolve (along with their individuals) during an evolutionary run.

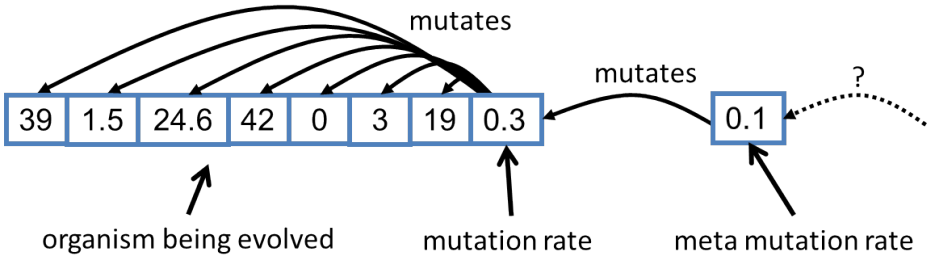


Figure 2.10: Evolutionary Strategies can incorporate a mutation rate into the genome of each individual. But this requires a meta mutation rate to specify the rate with which to mutate the mutation rate. And this introduces questions of whether to mutate the meta mutation rate, and what rate to use for this.

Mutating an organism's mutation rate raises the natural question of: *what mutation rate should be used to mutate the mutation rate?* Is it sufficient to simply have an organism's mutation rate mutate itself? Or would it be better to have a meta mutation rate, allowing the mutation rate to change at a different rate to the rest of the organism? It was discovered that having a meta mutation rate improves the performance of the algorithm [6]. The rate at which the mutation rate mutates should be different from the rate at which the rest of the organism mutates. This is because the meta mutation rate is solving a different problem to the mutation rate. The purpose of the mutation rate is to change the rest of the organism to find good solutions to the application problem. The purpose of the meta mutation rate is to change the mutation rate to find good ways of searching the space of solutions to the application problem. There is no reason why a strategy that is good at evolving solutions to an application problem should also be good at evolving mutation rates.

This version of meta-evolution introduces meta-parameters into the system. The need to specify the mutation rate of the organisms has been removed, but now there is a need to specify the mutation rate of the mutation rates of the organisms. This introduces the question of whether this meta mutation rate should be a fixed parameter of the algorithm, or whether it should also be allowed to evolve. In principle, we could have an indefinitely tall tower of meta-parameters and meta-meta-parameters all evolving each other. To prevent this happening in practice, it is necessary to choose a level at which to stop. The parameters at this level are not evolved and must be fixed (or chosen by another method, such as using the meta-mutation rate as the meta-meta-mutation rate). So a human must make a design decision at some point, choosing the value of some meta-parameter(s) (or the algorithm for choosing them). However, it is hoped that the problem of choosing the values of the meta-parameters will be easier than that of choosing the original parameters. So allowing one (or more) levels of parameter to evolve makes

the parameter-choosing problem easier to solve. This has proved to be the case in practice: evolving mutation rates can allow the algorithm to adapt to different environments in which the organisms might find themselves [11] [52].

2.6.1.2 Evolving populations of parameters

The Evolutionary Strategies described above allow some parts of an evolutionary algorithm to evolve. They do this by taking parameters of the algorithm and making them part of the organisms being evolved. This effectively allows the algorithm to choose its own parameters by changing them as it runs. An alternative approach is to have *a different evolutionary algorithm* change the parameters of an evolutionary algorithm.

Figure 2.11 shows a hierarchical evolutionary algorithm. This consists of an evolutionary algorithm that evolves solutions to an application problem, and a meta-evolutionary algorithm that evolves parameter values for the evolutionary algorithm. Each organism of the meta-evolutionary algorithm is itself an evolutionary algorithm. In an evo-devo process, the genome of one of these organisms is a set of parameter values for an evolutionary algorithm. The phenome of the organism is a running evolutionary algorithm.

This technique has proved very successful at solving well-defined application problems [30, p.26]. It can be seen as a way of automatically optimising the computational model of an evolutionary algorithm. But it is not appropriate for novelty-generation algorithms. Hierarchical evolutionary algorithms can only change the numerical parameters of an evolutionary algorithm. More involved design decisions (such as the crossover function and fitness function) must remain fixed (or chosen from a pre-defined set of fixed choices). Researchers have not managed to incorporate these more involved design decisions into hierarchical evolutionary algorithms [30, pp.3-6].

From the point of view of improving novelty-generation algorithms by changing their biological model, hierarchical evolutionary algorithms are not at all biological. Biology cannot have multiple versions of evolution competing with each other, because there is only one instance of the physical world. But nevertheless, the parameters of biological evolution do change over time. So the biological world must be using a different form of meta-evolution.

2.6.1.3 Co-evolving evolutionary operators

From the point of view of novelty-generation, the main limitation of the two above methods of meta-evolution is that they only evolve the *numerical parameters* of an evolutionary algorithm. There are many design decisions involved in setting up an evolutionary algorithm, that could in principle be evolved (and are in biology), such

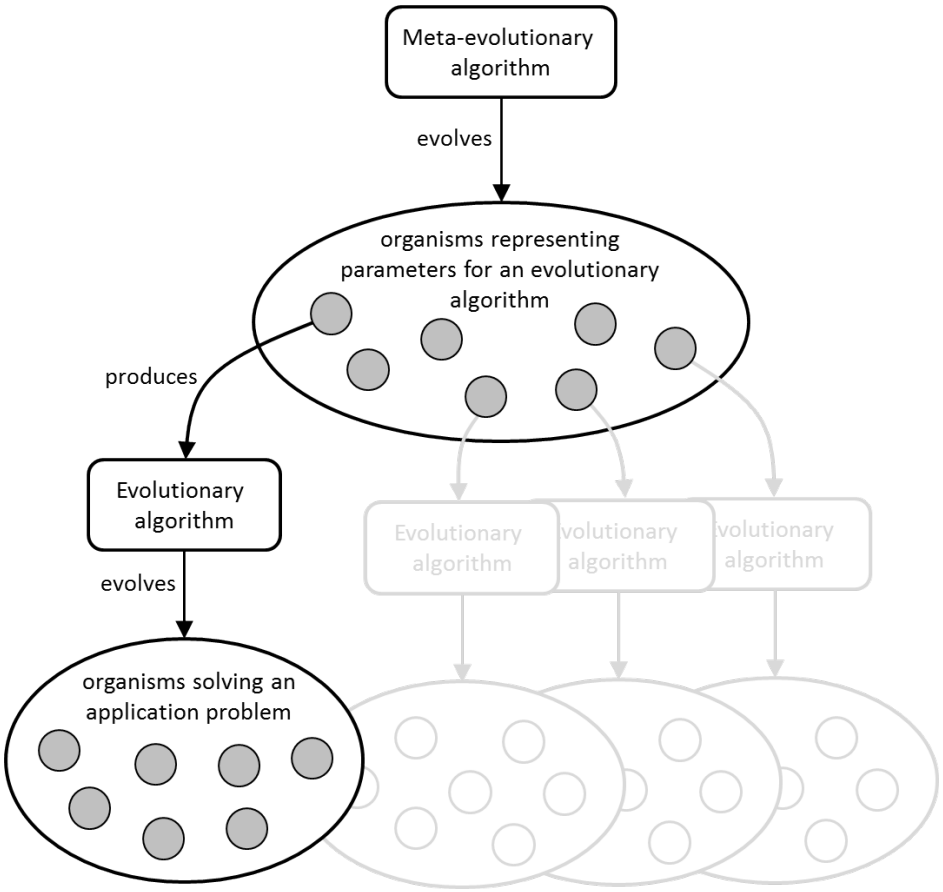


Figure 2.11: A hierarchical evolutionary algorithm evolves organisms that are parameter choices for an evolutionary algorithm. The lower-level evolutionary algorithm searches for good solutions to the application problem, while the meta-evolutionary algorithm searches for good parameter choices for the lower-level algorithm.

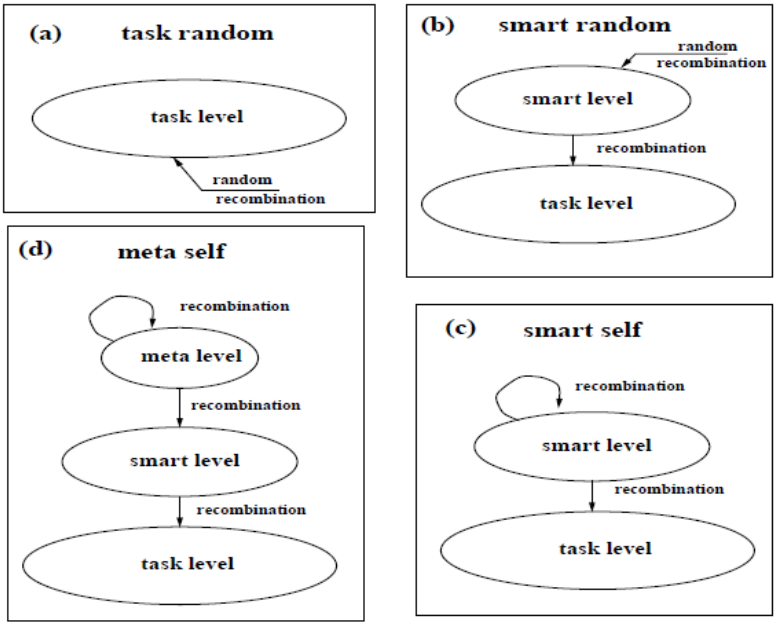


Figure 2.12: Organisms on the *task level* are solving the application problem. The *smart level* contains the crossover operators, that can be randomly crossed over (b) or can perform crossover on themselves, making them meta-operators (c). The *meta level* is a separate population of crossover operators that can specialise to be good at crossing-over crossover operators. Thus they work well when used as meta-meta operators (d). Kantschik found that further meta-levels were not any better than *meta self* [53], and neither was *meta random*. This image is figure 3 from [54]

as the: (1) problem representation, (2) fitness function, (3) mutation algorithm, (4) crossover method, and (5) selection method.

The work of Kantschik et al. looks at evolving one of the evolutionary operators: the crossover method [54]. Rather than having a fixed crossover operator, this approach co-evolves a population of crossover operators alongside a population of organisms solving an application problem. This is an example of cooperative co-evolution. The fitness of an organism is related to how well it can solve the application problem. The fitness of a crossover operator is related to its ability to produce children that are fitter than their parents. Kantschik et al. found that using this meta-evolution improved their algorithm's ability to solve their speech-recognition problem. And there is other empirical and theoretical work suggesting that this co-evolutionary form of meta-evolution can be useful [87].

As with Evolutionary Strategies above, Kantschik et al. came across a tower of meta-levels. Figure 2.12 shows some different implementations of this tower.

The organisms solving the application problem (the *task level*) are crossed-over by a population of crossover operators (the *smart level*). But then, which crossover method should be used to crossover the crossover operators? Kantschik et al. tried (1) crossing them over randomly, (2) having them cross over themselves, as well as crossing-over the organisms, and (3) having a meta-population of crossover operators (the *meta level*) that crossed over the crossover operators. They found the same result as Evolutionary Strategies: at least two levels of crossover operator are needed, and more levels seem to not add anything further (i.e. *meta self* from figure 2.12 performed best). This shows that the types of crossover operator that are good at crossing-over solutions to a speech-recognition problem (smart level) are not good at crossing-over themselves. But the crossover operators that are good at crossing-over crossover-operators (meta level) are also good at crossing over themselves. This is intuitive logically: the smart level and the meta level are both crossover operators, so it makes sense that the same strategy could work for both of them. But the smart level and the task level are very different (crossover operators versus solutions to a speech-recognition problem), so it makes sense that they might need different strategies.

2.6.2 Problems with current methods

The algorithms described above represent a good start at tackling the hugely difficult problem of implementing meta-evolution. But naturally, they have limitations. The problems with current algorithms fall into three broad categories:

1. tower of meta-levels;
2. brittle programming languages;
3. ignoring the biological model.

I describe these problems in more detail below, before suggesting some ideas for addressing them.

2.6.2.1 Tower of meta-levels

When part of an algorithm is allowed to evolve, this makes the algorithm easier to design by removing the need to design the evolving part. But it also makes the algorithm harder to design, because it introduces a meta-part that must be designed. For example, in the case of Evolutionary Strategies (section 2.6.1.1), the mutation rate of the organism is allowed to evolve (does not need to be designed), but the mutation rate of the mutation rate must be designed. And it must be decided whether to allow this meta-mutation rate to mutate, and whether to include a meta-meta-mutation rate.

Kantschik et. al. found that for their application problem, the tower of meta levels need only be two stories high. When evolving crossover operators, they obtained the best results using a population of crossover operators and a population of meta-crossover operators. But their system was very simple (evolving only the crossover operator) and limited (tested on only one application problem). It is not clear whether this result is general, or whether the best height of the tower will vary depending on the application problem and the part of the algorithm being evolved. If different parts of the algorithm prefer different heights of tower, then will this be possible in a given implementation of meta-evolution? And is it possible to have a system evolve the height of its tower(s)?

Biological systems evolve the heights of their towers, and do not have crisp distinctions between the levels. I describe this in my biological model of meta-evolution below, and suggest a way of implementing this in a novelty-generation algorithm.

2.6.2.2 Brittle programming languages

It is well-known in the evolutionary algorithms community that an important part in the design of an evolutionary algorithm is the representation of the organisms. The organisms must be represented in an *evolvable* language. We need a language where most small mutations to the *syntax* of the organisms correspond to small mutations in the *semantics* of the organisms, but some small syntactic mutations can have large semantic effects. So if we are to evolve the algorithm as well as the organisms, then clearly we must represent the algorithm in an evolvable language.

A major problem is that evolutionary algorithms are often written in high-level computer programming languages. It is well known that high-level programming languages are brittle and difficult to evolve [64] [65], but this does not imply that evolutionary algorithms must be brittle and difficult to evolve. It implies that high-level programming languages are not the best choice of representation for evolving evolutionary algorithms.

The problem of brittle programming languages is exemplified by Kantschik et. al.'s evolution of a crossover operator (section 2.6.1.3). They encoded their crossover operator as computer program represented by a Genetic Programming tree. However, this program did not perform crossover on two trees, as may have been expected. Instead, it merely traversed the trees, labelling nodes for crossover. A hardcoded algorithm written in a high-level language then performed the actual crossover, using the labelled nodes. While this method is capable of evolving the particular nodes that will be crossed over in a given crossover event, it is incapable of evolving the algorithm that performs the crossover. For example, it will never invent a type of crossover involving three parents (as can happen in biology: section 4.3.2). While this type of meta-evolution helped to better solve

their application problem, it is a serious limitation from the viewpoint of novelty-generation.

In order to evolve the actual algorithms, rather than just their parameters, we need more evolvable representations of the algorithms. We need new languages in which we can implement these algorithms. And we need languages that have been designed to be evolvable, rather than high-level programming languages which were designed to be the opposite (compact and efficient). In my biological model, I suggest that biology implements its algorithms in the evolvable language of chemical networks. I explain that artificial chemistries are a computational analogue of this, and could be one example of the evolvable language we need.

2.6.2.3 Ignoring the biological model

The algorithms above are ways of putting meta-evolution into the computational model, rather than the biological model. They consider a part of the computational model and ask:

> *What can we add to the computational model, in order that we need not design this part explicitly?*

A different way of approaching the problem would be to ask:

> *How can we change the biological model, to prevent this part appearing explicitly in the computational model?*

An example of successfully applying the second method is evolving the population size of a novelty-generation algorithm. This method was used in an Artificial Immune System called RLAIS (Resource Limited Artificial Immune System) [98], which is functionally identical to an evolutionary algorithm, just using different biological analogies. Biological systems do not explicitly maintain a fixed population size, as many evolutionary algorithms do. Rather, the appearance of stable population sizes in biology is a consequence of the environment containing limited resources. In RLAIS, there is a population of organisms but no fixed population size. The population size increases as the organisms reproduce, but the organisms must gather enough *resource* or be killed. Since the amount of resource is limited, the population cannot grow unboundedly, but can vary. In the case of RLAIS, this allows the algorithm to optimise a function while remembering the best local optima. RLAIS ends up with precisely one organism per local optimum. If the function has many local optima, then RLAIS can enlarge its population to discover them all. But if the function has few local optima, then RLAIS can shrink its population to run more efficiently.

It could be argued that RLAIS simply re-phrases the problem of choosing a population size as choosing a resource level, but the two are not the same. The resource level is a meta-parameter: one level higher up the tower than the population size. Choosing a resource level is an easier problem than choosing a population size, as a given resource level allows many different population sizes, which the algorithm is free to choose between. Also, the idea of resources allows further metaphors, such as providing more resources in certain areas of the search space. These types of metaphor are not possible (or at least not intuitive) when working in terms of population sizes.

We want to design biologically-inspired versions of meta-evolution that can be built upon in a principled way. To do this, we can modify the biological model of novelty-generation algorithms. Instead of staring at a computational model and wondering how to evolve parts of it, we can look to biology and wonder how biological systems allow their evolutionary algorithms to vary. The next section describes how I see biology doing meta-evolution, and the following section turns this into a new computational model.

2.6.3 A biological model of meta-evolution

In evolutionary algorithms, there is a clear distinction between the organisms being evolved and the algorithm that is evolving them. This distinction comes from an (incorrect) view of biological evolution as a tool for producing fit organisms. This trap is easy to fall into, especially for someone looking for a tool to solve their application problem. However, the biological reality is that the 'algorithm' of evolution cannot be separated from the organisms it is evolving. The biological 'evolutionary algorithm' is an emergent property of organisms living, reproducing and dying in an environment.

Looking at biology, there is no analogue of computer programs that *implement evolution*, let alone programs written in a brittle high-level language. Evolution is an *emergent property* of a system of interacting parts (organisms in an environment). When evolution changes individual parts of the system (via natural selection), it changes the system as a whole. This in turn changes the emergent properties of the system, one of which is evolution. Figure 2.13 shows this process. When viewed as an emergent property, it is difficult to see how evolution can change the organisms of a system and *not* change its own 'algorithm'.

The reason biology is able to achieve this is that the different parts comprising biology (organisms, reproduction, death, etc.) are not separate, isolated units, like functions in a computer program. They are all 'implemented in' the same language: that of *chemistry*. Biology can be thought of as an emergent property of chemistry, and evolution can be thought of as an emergent property of biology. So when evolution selects biological organisms, it is actually selecting collections of

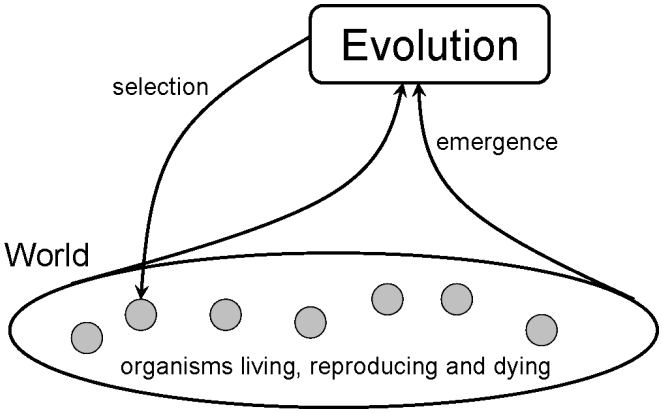


Figure 2.13: Organisms create evolution, as well as the other way around. If evolution is an emergent property of a system of organisms, then evolution acting on those organisms cannot help but change itself.

chemicals. Since the whole of biology emerges from these chemicals, selection has the power to change what types of biology are seen. Since evolution emerges from biology, selection also has the power to change what types of evolution are seen. It is this closure of the loop that gives evolution the power of meta-evolution: implementing the processes that lead to evolution in the same language as the organisms that are being evolved. This makes evolution an emergent property of the organisms that are being evolved, rather than a hardcoded, external algorithm.

The idea of meta-evolution naturally leads back to the origin of life: if evolution is continually evolving its 'algorithm', and it can be assumed that it always has been evolving this 'algorithm', then what 'algorithm' did it start with? There are debates raging among biologists about what came first — evolution, or the life-forms being evolved [67, chapter 2].

2.6.4 New computational models

In general, the field of meta-evolution is very broad. The idea of *evolving an evolutionary algorithm* can potentially encompass many different computational models. But, to date, no-one has managed to produce a landmark example of a successful, useful and widely applicable computational model of meta-evolution from this vast space of possibilities. Our approach to this problem is to use inspiration from biology to help guide our search through this space. The biological model described above specifies a sub-space of possible computational models. Rather than thinking about the general task of *evolving an evolutionary algorithm*, we are now thinking about the more specific task of *making evolution an emergent property of organisms interacting in an environment*. Perhaps focusing on this specific version

of meta-evolution might help the design of computational models and algorithms.

Hierarchical meta-evolution (section 2.6.1.2) does not fit with this biological model. Evolutionary Strategies' method of evolving parameters (section 2.6.1.1), and Kantschik's method of evolving crossover functions (section 2.6.1.3) do fit, but are very simple examples of emergent evolutionary algorithms. Most of the evolutionary algorithm is still hardcoded and not evolved. Only a small part is allowed to evolve (the mutation rate or the labelling of tree nodes for crossover), and this part is not an emergent property that evolves because its organisms evolve: it is explicitly implemented and explicitly evolved by the hardcoded evolutionary algorithm.

Banzhaf et. al. present a manifesto [8] in which they talk about the need to enrich the biological models of evolutionary algorithms, allowing these algorithms to address more complex problems in both engineering and biology. While this is not precisely emergent evolution, it is similar. Banzhaf et. al. talk about evolution in simulated physical and chemical systems giving rise to emergent, complex phenotypes. They present a research agenda rather than concrete algorithms implementing these ideas. The idea of emergent evolution, and the work presented in this thesis, can be seen as the start of an exploration of part of this agenda.

Spector et. al. use the term *autoconstructive evolution* to describe evolutionary algorithms evolving computer programs that both solve an application problem and create offspring computer programs [92]. They implement algorithms to test these ideas, using the Push programming language to create a system called *Pushpop*. This is a step towards emergent evolution, because Pushpop takes one part of the evolutionary algorithm (reproduction) and makes it a consequence of organisms (Push programs) interacting in an environment (executing in a virtual machine). But the problem with Pushpop is that it is a modification to the *computational* model of Genetic Programming, rather than the *biological* model. Thus it is difficult to determine whether Pushpop's strengths (and weaknesses) originate from its biological model or from its computational model. This makes it difficult to modify Pushpop to make more of its evolutionary algorithm emergent. Performing this modification would require a lot of effort making arbitrary changes to Pushpop's computational model, rather than taking inspiration from biology and making principled changes to its biological model. The abstract idea of autoconstructive evolution encapsulates one aspect of emergent evolution (organisms constructing their own offspring), but it does not go as far as making the entire evolutionary algorithm emergent. And unfortunately, the Pushpop implementation of this idea suffers from the problem of ignoring the biological model.

We can look at the progression of novelty-generation algorithms from early evolutionary algorithms, through co-evolution and evo-devo to artificial life (summarised in section 2.4.3) and see this as movement in the direction of emergent evo-

lution. In this story, we see the fitness function change, becoming more evolvable, until finally (in artificial life) it is an emergent property of a system of interacting organisms. Unfortunately, the fitness function is the only part the evolutionary algorithm that has gone through these transitions and become an emergent property. In the next section, I describe a system from the literature that evolves the problem representation of an evolutionary algorithm. It does not go all the way and make the problem representation an emergent property; it implements a system that changes the (explicit) problem representation. In terms of the fitness function's story, this system is the equivalent of co-evolution of the problem representation and the organisms.

Finally, I describe my computational model of emergent evolution, and summarise the different types of meta-evolution.

2.6.4.1 A meta-process changing an optimisation algorithm

The most general way of changing an evolutionary algorithm as it runs is to have a meta-algorithm monitoring the state of the evolutionary algorithm and making changes to the evolutionary algorithm based on its observations. This section describes such a system by Watson et. al. [102]. It is a simple neural-network-like optimisation algorithm (simpler than an evolutionary algorithm) that has a meta-process applied to it, to increase its ability to perform its optimisation task. I describe this system in detail because it is the closest approximation to emergent evolution that I have found in the literature.

The system The organisms in this system are very simple. They are agents that can choose between two different *strategies*, designated $+1$ and -1. We use the notation S_i to refer to the strategy chosen by organism i. The optimisation problem is represented as a set of constraints between pairs of agents, saying whether these agents benefit from choosing the same strategy or from choosing opposite strategies (and by how much). This is a matrix of floating point numbers, ω, where $\omega_{i,j}S_iS_j$ is a number that is positive if organisms i and j have chosen strategies that satisfy the constraint between them, and negative if they have not (and zero if there is no constraint between them). This connects the organisms together in a topology. The goal of each organism is to maximise its local utility. The local utility of organism i is the extent to which it satisfies all of its constraints with other organisms (j):

$$u_i = \sum_{j}^{N} \omega_{i,j} S_i S_j. \tag{2.7}$$

Each organism selfishly changes its strategy to maximise its local utility. The global optimisation problem is to choose a combination of strategies for the organisms,

to maximise the global utility, which is how well all of the organisms satisfy all of their constraints:

$$U = \sum_{i}^{N} u_i = \sum_{i}^{N} \sum_{j}^{N} \omega_{i,j} S_i S_j. \tag{2.8}$$

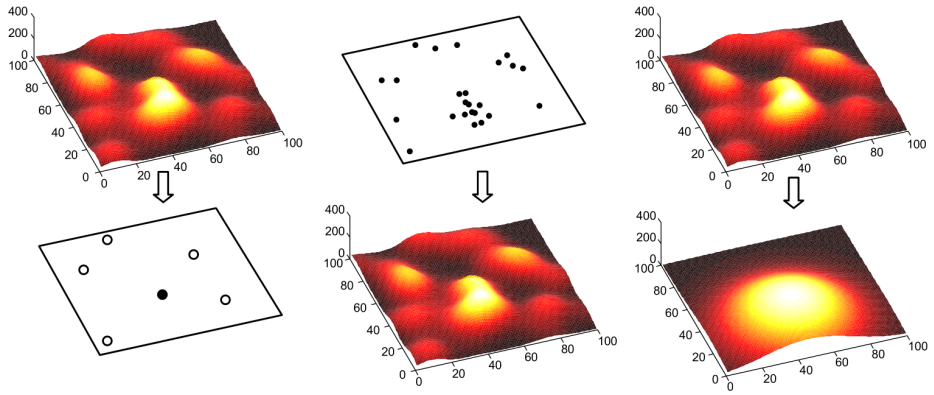
When one organism changes its strategy to increase its local utility, the other organisms can change their strategies in response, setting off a chain-reaction of organisms changing their strategies. The system will settle to a point-attractor if the weights are symmetric [102] (it might be chaotic or cyclic otherwise), so this algorithm only uses symmetric weights.

When running this system, the organisms all change their strategies to selfishly optimise their local utilities, until they settle down to a point attractor where no organism wants to change its strategy. In an ideal world, this attractor of the system would be the global optimum solution to the optimisation problem. But, as might be expected, this algorithm often does not find the global optimum. When there are local optima in the search space, this algorithm can get stuck in a local optimum rather than finding the global. And compared to other optimisation algorithms, this one performs very badly. But this is intentional. The optimisation proceeds poorly because the problem representation (the weights ω) creates a space that is difficult to search through. A meta-process is added to this system to allow the system to change this representation and make the optimisation problem easier to solve.

The meta-process The local utility of an organism (equation 2.7) depends on the organism's strategy (S_i), the strategies of the organisms it is connected to (S_j), and the constraints it shares with the other organisms ($\omega_{i,j}$). Rather than just allowing organisms to change their strategy (S_i), the meta-process allows agents to maximise their local utility by changing their constraints ($\omega_{i,j}$) as well.

At first glance, this may seem a nonsensical way to go about optimising a problem, because allowing the organisms to change their constraints is allowing them to change the problem they are solving. So they will surely just change the difficult optimisation problem into an easy problem that bears no relation to the original problem, and trivially solve this easy problem. But the system is set up to prevent this happening. Using a hardcoded, external algorithm, the original system is run (without the meta-process) for a time, until it has probably converged to a local optimum. Then one iteration of the meta-process is applied: the organisms change their weights by a small amount. The strategies of each organism are then randomised, and the original system is run again (and this process repeated).

This interleaving of the original system and the meta process (at the appropriate timescale) implements a form of Hebbian learning [102], where the system changes the problem it is solving, enlarging the basins of attraction of the local optima it has



a) Ideally an optimisation process finds a global optimum (solid point) in an (implicitly defined) adaptive surface. Local optimisation, as in the Hopfield network or a system of selfish agents, provides imperfect optimisation (open points).

b) Building an associative memory from a set of training samples in a Hopfield network is a form of model induction: a dynamical system is trained (i.e. by Hebbian learning) to represent a set of points by forming attractors that correspond to those state configurations.

c) The combination of local optimisation and associative memory demonstrated in this paper transforms an optimisation problem with multiple optima into a new optimisation problem that is easier to solve. But the easy-to-find solution to the new problem is, in fact, the hard-to-find solution of the original problem.

Figure 2.14: The meta-process changes the problem representation, making the optimisation problem easier to solve. This image is figure 3 from [102].

visited. Figure 2.14 shows this process. When used on appropriate problems [102], this allows the algorithm to generalise, enlarging the basin of attraction of the unseen, global optimum. This makes the global optimum easier to find, thus making the optimisation problem easier to solve. The use of the meta-process has allowed the algorithm to transform a difficult problem into an easy problem. And, under the right conditions, *"the easy-to-find solution to the new problem is, in fact, the hard-to-find solution of the original problem"* [102].

Problem representations The meta-process changes the problem that the system is optimising, but it does not change the location of the global optimum. So, if we define our problem to be finding the global optimum, then the meta-process does not change the *problem*: it just changes the *problem representation*. It changes the space through which we are searching for a solution to the problem. This is analogous to using a traditional Genetic Algorithm to minimise the function x^2, but choosing a fitness function of x^4 to make the algorithm converge faster. However, in the case of the algorithm described here, this choice does not need to be hard-coded by the algorithm designer. As the meta-process is running, the algorithm chooses for itself the best problem representation.

In terms of meta-evolution, the *problem representation* is usually part of the *algorithm implementing evolution*. This system re-phrases the problem representation, implementing it as part of the *organisms that are evolving*. It changes a

system of simple agents (selfishly choosing strategies) and a fixed problem representation (matrix of weights) into a different system with more complex agents (able to selfishly change their weights) and an evolving problem representation. Note that there is still the meta-problem-representation, which is the means by which the problem representation is changed (selfish changes to a matrix of weights, on a certain timescale). This system does not make the entire problem representation an emergent property, but is a step in this direction.

This system is one specific example of making a part of the evolutionary algorithm emergent. We require a general framework that we can apply to any and all parts of the evolutionary algorithm.

2.6.4.2 My computational model: Emergent evolution

The biological model described above (section 2.6.3) tells us that rather than implementing evolution explicitly, our meta-evolution algorithm should implement a collection of organisms interacting in an environment, in such a way that evolution can emerge as a property of this system.

Artificial life (section 2.4) provides a starting-point for this. Artificial life can be seen as an extension of the novelty-generation capabilities of evolutionary algorithms, that implements the fitness function as an emergent property. We can take this further, also implementing the other parts of evolutionary algorithms as emergent properties. Automata systems (section 2.4.2.1) are a step down this road. They implement organisms that can reproduce themselves, and evolve their reproduction algorithms, rather than having an external, hardcoded reproduction function. But the problem with automata systems is that they use brittle computer programming languages to define their individuals, so the evolvability of the individuals is limited. Template addressing is one example of a reduction in the brittleness of automata systems, which dramatically increased their novelty-generation capabilities. So we should look to reduce this brittleness further by using more evolvable, more biological programming languages.

Artificial chemistries (section 2.4.2.2) are an example of evolvable programming languages that have a relevance to biology. But the problem with artificial chemistries is that they are difficult to program in. The same properties that make them attractive for implementing emergent evolution, also make them difficult to program reliably and slow to execute (e.g. redundancy in their component parts, softness (not brittle), and emergent properties). However, artificial chemistries are the only programming languages (that I have been able to find) that satisfy all of the requirements (described above) for a language in which to implement emergent evolution.

When designing an emergent evolution algorithm, the effort of the algorithm designer shifts from working out *which evolutionary algorithm is best to use*, to

working out *which artificial chemistry is best to implement the organisms, so that evolution can evolve itself most easily.* This shows that if we want to make progress with emergent evolution, we need to understand how artificial chemistries work, and make progress designing artificial chemistries that are suited to emergent evolution. We need an artificial chemistry that can:

1. Implement organisms capable of solving an application problem (which might be just survival).

2. Implement the component parts of evolution: reproduction of the organisms, competition for resources, and death of organisms that fail to perform.

3. Encode the above things within the genomes of the organisms, so that they can be evolved by evolution.

4. Do all of the above in an evolvable, non-brittle way, so that evolution will be able to move successfully through the space of organisms and evolutionary algorithms.

The task of implementing an emergent evolution algorithm is clearly an immense challenge, that no-one has yet managed to achieve. And I do not manage it in this thesis. In the remainder of the thesis, I define embodiment and artificial chemistries in a way that is useful for emergent evolution, and I show how biological systems are embodied in physical-world chemistry. I then design an artificial chemistry that can implement the reproduction part of an evolutionary algorithm, using a non-brittle reproduction method that can evolve.

2.6.5 Different types of meta-evolution

In summary, I have identified four different categories of meta-evolutionary algorithm:

1. An evolutionary algorithm that has evolutionary algorithms as its individuals (section 2.6.1.2).

2. An evolutionary algorithm that is changed by an external process as it runs (section 2.6.4.1).

3. An evolutionary algorithm that changes itself as it runs, using an explicitly-implemented function that changes itself (sections 2.6.1.1 and 2.6.1.3).

4. An evolutionary algorithm that changes itself as it runs, because the algorithm is an emergent property of the system it is evolving (sections 2.6.3 and 2.6.4.2: *emergent evolution*).

Having an evolutionary algorithm evolve a population of evolutionary algorithms introduces massive computational overhead. This is why these algorithms are only able to optimise a small number of numerical parameters. The tower of meta-levels is not seen in these algorithms, because using more than one meta-level would make the computational requirements completely intractable.

In order to change large-scale properties of an evolutionary algorithm, such as the evolutionary operators or the problem representation, evolving a population of algorithms is not feasible. We need a single algorithm that changes as it runs. If this algorithm is changed by an external process (such as Hebbian learning), then the tower of meta-levels is seen. Each level must be explicitly designed, and there is the question of which type of external process to use on each level.

The biological solution is to have a single algorithm that changes itself as it runs. This can be achieved in the computational model by programming in explicit functions to change the algorithm (Evolutionary Strategies evolving mutation rates, and Kantschik et al. evolving crossover functions). But this again runs into the tower of meta-levels, and has the problem of brittle programming languages.

Meta-evolution can be achieved by changing the biological model and making the evolutionary algorithm an emergent property of the system of evolving organisms. This does not completely remove the tower of meta-levels, but it makes the tower implicit rather than explicit. The tower still exists, but the algorithm can now choose the tower's height itself. Also, different parts of the algorithm can evolve different towers of different heights. And different levels of the tower(s) can interact with each other, as they do in biology. Emergent evolution still faces the issue of brittle programming languages, but it forces this problem to be solved, rather than hiding it away as a difficult design decision. Current, brittle, computer programming languages are incapable of implementing emergent evolution. Thus, to make a start on emergent evolution, we must first design a programming language that is soft and biological (fulfilling the requirements of section 2.6.4.2). In this way, we know that if we manage to implement emergent evolution, we will have solved the problem of brittle programming languages (and can move on to the next problem of *evolvable* programming languages).

The theoretical idea underlying emergent evolution is:

> *By implementing a process (evolution) in terms of a suitable (evolvable) lower-level environment (artificial chemistry), we gain a benefit (meta-evolution).*

This idea is not new: it is generally called *embodiment*, and tends to refer to implementing processes in the environment of the physical world. An example of this is building physical robots rather than running simulations of them. But processes can also be embodied in virtual worlds. The next chapter describes embodiment, and explains how it relates to emergent evolution.

Chapter 3

Embodiment

This chapter describes the abstract concept of *embodiment*, which is the theoretical principle underlying embodied evolution. The purpose of this chapter is to set out the theoretical framework and terminology that we make use of throughout the remainder of the thesis.

The following chapter gives physical-world examples of how biological systems make use of embodiment to achieve meta-evolution. In later chapters, we use the ideas of embodiment and the examples from biology to build a detailed computational model of emergent evolution, and implement an instance of this model as a novel artificial chemistry.

3.1 Why do we want embodiment?

Our aim is to improve novelty-generation algorithms by making their biological models richer. But this opens up huge design problems of precisely *how* to make the biological models richer, and *how much* biology to include. At one extreme, we do not want to copy the whole of biology within a computer. Even if we were able to do this, we would just end up with another copy of biology, that would be as difficult to understand and use as physical-world biology. Some computer scientists believe that such a model would be useful [40], but in any case, it would be very difficult to achieve, and so looking at alternative approaches seems sensible.

The benefit of making a model of biology within a computer (whether to understand the underlying biology, or to exploit its properties computationally) is that the model can be made at a certain level of abstraction. We can simplify some of the biology to make the system easier to understand/control/program (whatever our aim is). This lets us separate out one property of biology (such as, for example, evolvability, robustness or novelty generation) from the multitude of properties that biology possesses. Biological systems contain many evolutionary artefacts and

frozen accidents [21] that are consequences of these systems having to survive and evolve in the physical world. By making models of biological systems, we can tease out which biological processes are necessary and sufficient to give the properties we want computationally, and which are consequences of biological systems having to survive in the physical world.

When choosing a level of abstraction at which to model, we face a problem. We know that biological systems display many different properties that we want to emulate within a computer program. And we know that biological systems are very complicated, containing many mechanisms that work together to give rise to the biological system as a whole, and copying them on their lowest level of abstraction is not feasible. But we do not know which (combinations of) low-level mechanisms give rise to which high-level properties. We do not know precisely how to change the biological model of a system to endow it with a given high-level property. In other words, we do not know how to engineer emergence.

The idea of *embodiment* provides us with a framework for deciding how to change biological models. It does not solve the problem of engineering emergence, but it gives us a set of different ways in which we can modify a given biological model and carry these changes through to the computational model. By having a language in which we can talk about changes to biological models, we obtain a clear way to analyse and compare different novelty-generation algorithms.

3.2 What is embodiment?

The word *embodiment* is used in the robotics community to refer to physical robots interacting with the physical world, in contrast to simulated robots interacting with a simulated model of the world. The physical world is always more complicated and heterogeneous than a simulated model. So running robots in the physical world gives them a rich environment to exist in, which has many benefits for robotics research [14]. While robotics is an example of embodiment (and an example of how useful embodiment can be), the general concept of embodiment extends to more than building robots [82] [93]: systems can be embodied within virtual environments as well as within physical ones.

The purpose of embodiment is the idea that by interacting closely with an environment, a system can more easily perform complex behaviours by using its environment as a source of complexity and computation. So a simple system interacting with a complex environment can perform complex behaviours.

In the robotics case this involves running physical robots in the physical world, to gain the advantage of not having to simulate the richness of the physical world. Simple robots can *use the world as its own model* [14] in order to perform behaviours that human observers consider to be complex and useful.

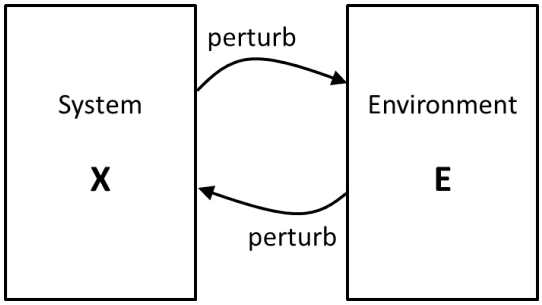


Figure 3.1: Quick's definition of embodiment considers two dynamical systems with the ability to perturb each other: the system being designed, **X**, and the environment it is embodied within, **E**.

With meta-evolution the situation looks slightly different, but the same underlying principles apply. For biological evolution, the *environment* is the constantly-changing, unpredictable nature of the physical world. This causes biological organisms to vary in their behaviour: rather than being rigid mechanisms that operate in precisely the same way all the time, they are *soft* mechanisms with behaviours that can be modified by the environment. This modification is not a problem: in fact, it is necessary for the existence of life! Constant modification of biological mechanisms is the source of novelty that causes mutations and drives evolution. But a given environment does not modify all the different biological mechanisms uniformly, in the same way and at the same time. Different parts of biology are affected in different ways by different environments. Evolution exploits this, by *changing biological mechanisms to change the effect that the environment has on them.* An example of this is given in the next section.

3.2.1 Previous definition: system and environment

The general definition of embodiment talks about a *system* being embodied in an *environment*. Quick defines embodiment in terms of dynamical systems [82]. He talks about a system, **X**, such as a robot or a biological organism, being embodied within an environment, **E**, such as the physical world. This allows him to give a formal definition of embodiment [82]:

> A system **X** is embodied in an environment **E** if perturbatory channels exist between the two. That is, **X** is embodied in **E** if for every time **t** at which both **X** and **E** exist, some subset of **E**'s possible states have the capacity to perturb **X**'s state, and some subset of **X**'s possible states have the capacity to perturb **E**'s state.

This is shown schematically in figure 3.1.

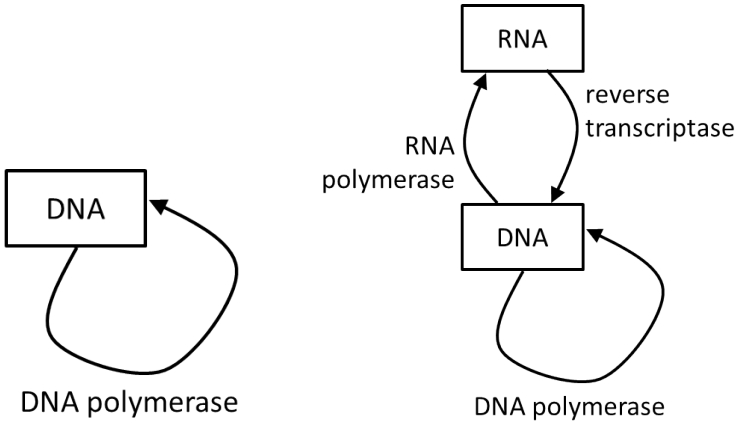
Now that we have a definition of embodiment where the environment is a general dynamical system, **E**, rather than the physical world (which is a particular example of a dynamical system), we have removed the reliance on the physical world. We can imagine systems embodied within dynamical systems that are not the physical world. In particular, we can imagine systems being embodied within virtual environments.

Embodying systems in environments other than the physical world raises the question of: given a system, **X**, and an environment, **E**, is the embodiment useful? We know that biological systems benefit from being embodied within the physical world, but that does not mean that an arbitrary system will benefit from being embodied within an arbitrary environment. Quick gives three ways of quantifying the degree of embodiment between a system and an environment [82]: (1) the size of the structural state spaces of **X** and **E**; (2) the structural plasticity of **X** and **E**; (3) the bandwidth of perturbatory channels between **X** and **E**. And Stepney gives nine design principles to help people create embodied systems [93].

An example of embodiment in biology is the process of copying the genome of an organism (which is described in detail in section 4.2). The genome is encoded on a chemical called *DNA*, which is copied by another chemical called *DNA polymerase*. Errors in the copying process happen because of its embodiment within the physical world. The DNA polymerase chemical does not always produce a perfect copy of the DNA. This is because of thermal noise, mutagenic chemicals and radiation in the environment. There is a system, **X**, comprising the DNA and DNA polymerase, that is embodied within the environment, **E**, of chemistry within the cell. This is a *copying system*, which in principle copies perfectly, but because it is embodied within this environment, it copies imperfectly. Thus embodiment provides the perfect copying system with mutations, that are the source of the variation necessary for evolution. So biology's embodiment of its copying process allows evolution to happen. Further to this, different versions of DNA polymerase can evolve, that are more or less susceptible to thermal noise. And different versions of DNA can evolve, that are more or less susceptible to radiation and mutagenic chemicals. The copying system can evolve the degree to which its environment is able to cause mutations. In other words, biology can evolve its own mutation function. So *biology's embodiment of its copying process allows meta-evolution to happen.*

3.2.2 Problem: Biology is not crisp

The above definition makes a crisp distinction between a system, **X**, and its environment, **E**. While this may be an appropriate stance to take when thinking about engineered systems such as robots, it is not as useful for thinking about biological systems. With the robot case, there is a clear system **X**: a robot that has been built



(a) Genome copying in most biological organisms

(b) Genome copying in RNA viruses

Figure 3.2: Biology has different systems that perform the same function, for example copying the genome of an organism.

by an engineer, and a clear environment **E**: the arena in which the robot is to act. But drawing crisp boxes around biological systems is not always straightforward or useful.

For example, consider the genome-copying case above. Most biological organisms encode their genome on DNA, and copy their DNA in the manner described above, but there are exceptions to this rule. Figure 3.2 shows one example of this. RNA viruses (described in detail in section 4.4.2) carry their information in chemicals of RNA, rather than DNA. RNA cannot be copied directly by a DNA polymerase chemical. These viruses encode a *reverse transcriptase* chemical that turns their RNA into DNA, which is copied by DNA polymerase, before being turned back into RNA by an RNA polymerase chemical.

The same abstract process is happening (copying an organism's genome), in the same environment, **E**, (chemistry within a cell). But two different systems, **X**, are implementing the process (one composed of DNA and DNA polymerase; the other composed of RNA, reverse transcriptase, DNA, DNA polymerase, and RNA polymerase). To further complicate matters, the RNA virus encodes (on its genome) the reverse transcriptase chemical, but not DNA polymerase or RNA polymerase. It usurps these from the host cell in which it replicates. So in drawing a box around the *RNA virus replication system*, **X**, we have to include some parts of the RNA virus, and some parts of the DNA replication system of another biological organism. To even further complicate matters, there exist *DNA viruses* that encode some of their replication system but also usurp some from a host cell. And there

are *satellite viruses* that require their host cell to be infected by a second, helper, virus before they can be copied. So, in biology, there are many different versions of a *genome-copying system*, **X**.

With all these different genome-copying systems, we can talk about the embodiment of copying by one system, and we can (separately) talk about the embodiment of copying by another system. But it is difficult to talk about the biological embodiment of copying in general, because copying is not implemented by just one system. This is because copying in biology is an abstract concept, rather than a concrete system. This situation is different from robot-building, where there is a concrete system, **X**, (the robot being built), that is separable from its environment, **E**, (the parts of the physical world that the robot engineers didn't build). When looking at biological systems, every box that humans want to draw around a part of biology, is really an abstract concept. While it is compelling (and necessary for the sanity of people trying to understand biology) to talk about sub-sets of biology as systems, it must be remembered that biological systems are not the same type of thing as engineered systems. Biological systems are actually abstract concepts. They are phenomena that can be observed to happen, and can be explained by processes occurring on a lower level (i.e. chemistry and physics). But the processes used to explain a biological phenomenon are not the same thing as the phenomenon itself. The same phenomenon may also be caused by a different set of processes, that may or may not overlap with the original processes.

3.2.3 My view: phenomena, mechanisms and worlds

Rather than splitting embodiment into a system and an environment, it may be more useful (from a biological point of view) to split embodiment into a *phenomenon*, some *mechanisms* and a *world*, as illustrated in figure 3.3. The phenomenon is an abstract concept, such as copying. This abstract phenomenon is embodied within a world if there exist some mechanisms within the world that enable the phenomenon to happen. A collection of mechanisms could be called a *system* (from Quick's definition), if this makes sense to the particular problem being investigated. But different collections of mechanisms (different systems) can give rise to the same phenomenon.

This version of embodiment is a special case of Quick's definition above. Rather than having a separate system and environment that can perturb each other, this definition of embodiment sees the system implemented *as part of the environment*. There is a single *world*, which comprises both Quick's *environment* and *system*. When observing this world, we identify sub-systems (*mechanisms*) that we find interesting, because their behaviour gives rise to an abstract *phenomenon* of the system. We can say that the abstract phenomenon is embodied within the world, via the mechanisms. And if the world contains some rich dynamics that are not part

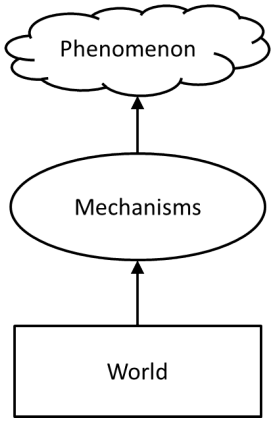


Figure 3.3: My definition of embodiment: Producing an abstract phenomenon by implementing mechanisms in a world. The world is the only thing that exists objectively. Mechanisms are sub-sets of the world, and the phenomenon is an abstract property that we can observe of the system.

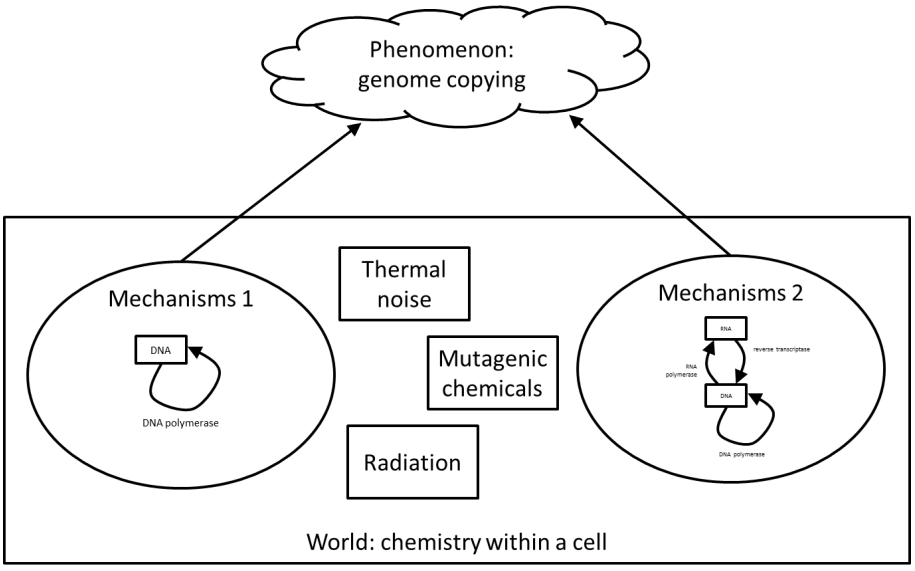


Figure 3.4: Different combinations of mechanisms can embody the same phenomenon within the same world. Some of these mechanisms have overlapping components in the world.

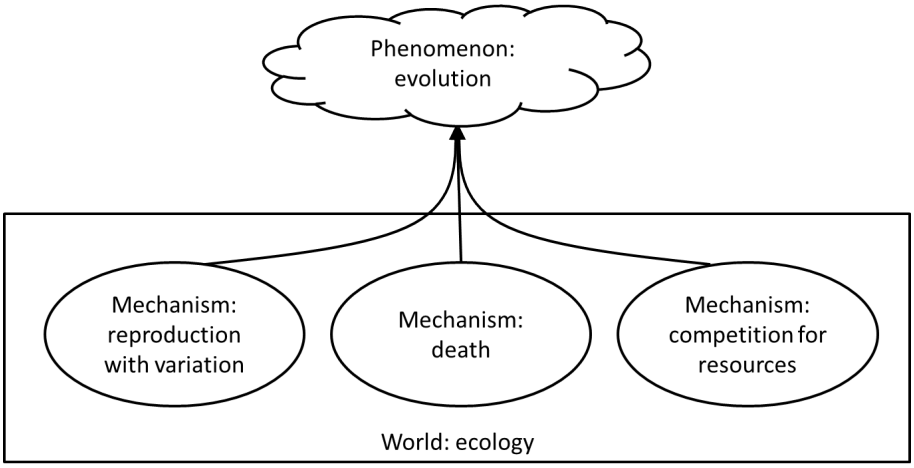


Figure 3.5: Emergent properties, such as evolution, can be viewed as phenomena embodied in a world by a collection of mechanisms.

of the mechanisms, then they can influence the mechanisms in the same way that Quick's environment influences his system. The benefit of this definition is that we can easily talk about the same phenomenon being embodied within the same world, but via different mechanisms. Or the same phenomenon being embodied within a different world. Figure 3.4 shows this applied to the genome-copying example described above. The phenomenon of genome-copying can be embodied within the world of chemistry by two different mechanisms.

Describing embodiment in terms of phenomena, mechanisms and worlds enables many different types of phenomenon to be considered. As well as clearly-defined phenomena, such as robots, that fit into Quick's definition, we can also consider more abstract concepts such as copying. In addition, we can think about the embodiment of phenomena that are impossible to describe as systems, such as emergent properties. For example, figure 3.5 shows how we can consider the emergent property of evolution to be embodied within the world of ecology via the mechanisms of: reproduction with variation, death, and competition for resources.

When considering the embodiment of very-high-level concepts such as evolution, we can see that embodiment is a hierarchical process, as shown in figure 3.6. We can think of evolution as the phenomenon and reproduction with variation as one of the mechanisms that embodies evolution within the biological world. But then we can consider reproduction with variation to be an abstract phenomenon. We can talk about genome-copying and cell division as some mechanisms by which reproduction with variation is embodied within the bio-chemical world. And we have already seen genome-copying expressed as a phenomenon, embodied within the chemical world.

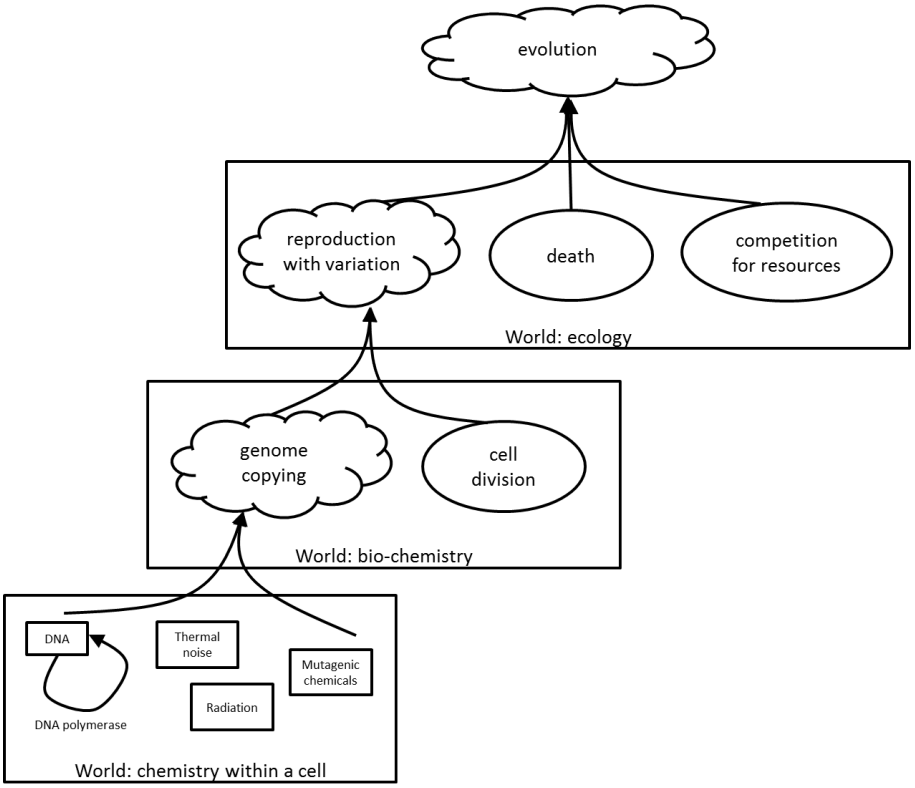


Figure 3.6: Embodiment can be a hierarchical process, with the mechanisms on one level being the phenomena on a lower level, embodied in a lower-level world.

The mechanisms of one level are the (emergent) phenomena of the level below. And each time we move down a level of embodiment, the world becomes more detailed. This highlights the fact that for any given phenomenon, there isn't just one world in which it is embodied. There are multiple worlds, at different levels of detail. A *world* in this sense is just a level of abstraction. In terms of the physical world, going down through different levels of embodiment gives us more and more information about how biological/chemical/physical processes happen. Physicists have not yet determined whether the physical world has a lowest-level description or not.

In terms of computer programs, there always has to be a lowest-level world. This is the world that is implemented by the computer programmer. It could be argued that programming languages are compiled to machine code, which is a language for controlling electrons in the physical world. But this all happens automatically, and is not under the control of the algorithm designer. We can use embodiment as a design tool, if we consider the implemented computer program to be a lowest-level world. The properties of this world determine the types of mechanism that can be implemented in it, and hence the types of phenomenon that can emerge from the computer program. Therefore, the correct design of this world is crucial for the emergent properties of the program. In order to implement emergent evolution, we need to have the mechanisms of evolution embodied within a novelty-generation algorithm. This will allow the algorithm to change the mechanisms of evolution, and hence achieve meta-evolution.

3.2.4 A measurement of novelty-generation

We are trying to design novelty-generation algorithms. That is, algorithms that can generate novel structures and behaviours. And we also want algorithms that can generate novel ways of generating novel structures and behaviours. Embodiment provides a way of measuring the extent to which an algorithm achieves this. I do not have a formal definition that gives a numerical measure of novelty-generation. I am not sure if this is possible, or even a sensible way to assess novelty-generation algorithms. I provide an informal measure of novelty-generation, and a framework for arguing that one novelty-generation algorithm is more able to generate novelty than another.

Thinking of embodiment as a hierarchical process, figure 3.6 shows how to assess the novelty-generation capability of an algorithm, or compare two algorithms. All novelty-generation algorithms can be viewed as attempting to embody the phenomenon of evolution (whether this was the intention of the algorithm designers or not). So any novelty-generation algorithm can be represented by a diagram similar to figure 3.6, with evolution at the top, and the different mechanisms implemented by the algorithm below. Any mechanism that is implemented in a

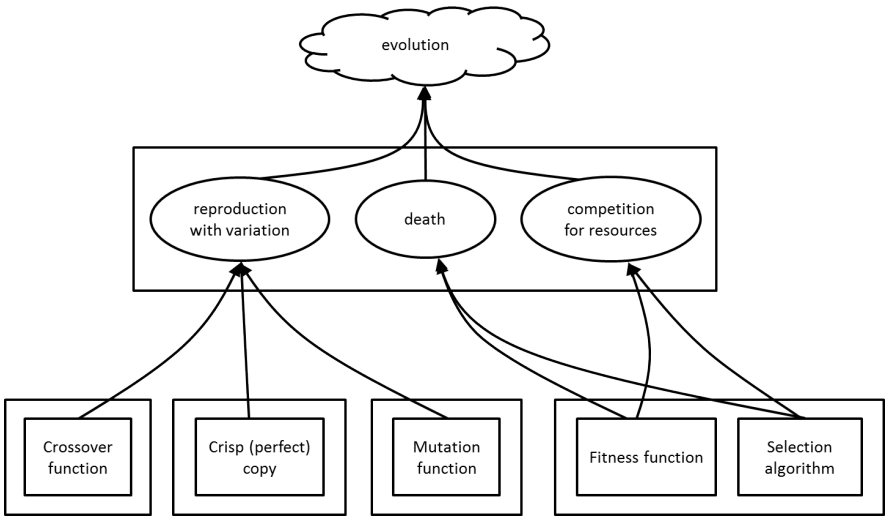
computer programming language and not able to evolve, is symbolised by an ellipse. Any mechanism that is embodied is symbolised by a cloud, and has a tree below it showing the mechanisms that embody it.

With a few caveats, one algorithm is more able to generate novelty than another if its diagram contains more clouds, i.e. its algorithm contains more embodied mechanisms. This is because the embodied mechanisms (clouds) are the parts of the algorithm that can evolve. This assessment must also take into account the importance of each embodied mechanism, since different mechanisms may contribute in different amounts to the running of the algorithm. Also, having more worlds (with ellipses in) is not necessarily a good thing. Two mechanisms can only interact with each other if they exist in the same world, and it is the interaction of mechanisms that embodies phenomena. So the greater the extent to which mechanisms are separated by existing in different worlds, the fewer phenomena can be embodied within the algorithm. Figure 3.7 shows an example of how these diagrams can be used to compare the embodiment structure of traditional Evolutionary Algorithms (figure 3.7(a)) with that of the Tierra Artificial Life system (figure 3.7(b)).

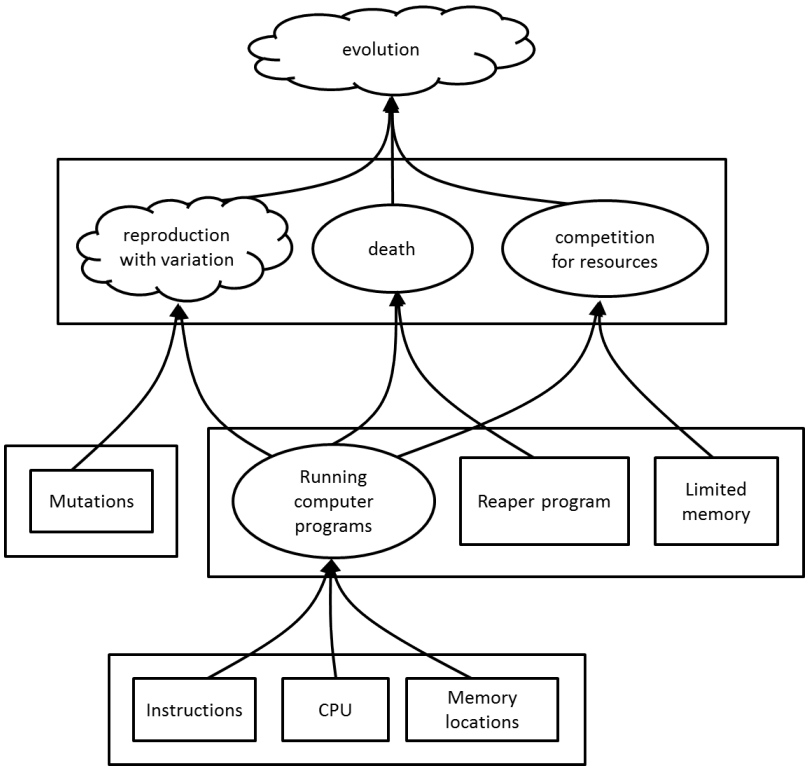
We know that traditional Evolutionary Algorithms are poor at generating novelty (section 2.3.1), and we can see this reflected in the diagram. The different stages of the Evolutionary Algorithm (selection, copying, mutation, crossover) happen in separate worlds, and are not able to interact with each other. The fitness function and selection algorithm can interact in a very primitive way, because the granularity of the fitness values influences how discriminating the selection algorithm can be. The novelty generated by the mutation function cannot go into the evolutionary algorithm, because this algorithm is partitioned into separate worlds.

In contrast, the Tierra system is able to generate a limited amount of novelty (section 2.4.2.1). Tierra is able to generate novel types of reproduction (parasites), because its *reproduction with variation* mechanism is embodied, in contrast to that of Evolutionary Algorithms. *Reproduction with variation* in Evolutionary Algorithms is implemented as a collection of different primitives in separate worlds (perfect copying, mutation, crossover). But in Tierra it is implemented by a (mutation process combined with a) collection of running computer programs, which are mechanisms existing in a world containing a CPU and many instructions residing in memory locations. Because of the level of indirection between the phenomenon and its implementation, the process of *reproduction with variation* can itself vary. The computer programs implementing the reproduction can change, giving rise to different methods of reproduction.

An ideal novelty-generation algorithm is shown schematically in figure 3.8. This algorithm contains only clouds in its first layer, showing that it embodies many different phenomena, that can all evolve. Below these clouds there is a single world containing all the ellipses. This means that all the mechanisms can interact with



(a) Embodiment structure of traditional Evolutionary Algorithms



(b) Embodiment structure of the Tierra Artificial Life system

Figure 3.7: The Tierra system shows more embodiment than traditional Evolutionary Algorithms.

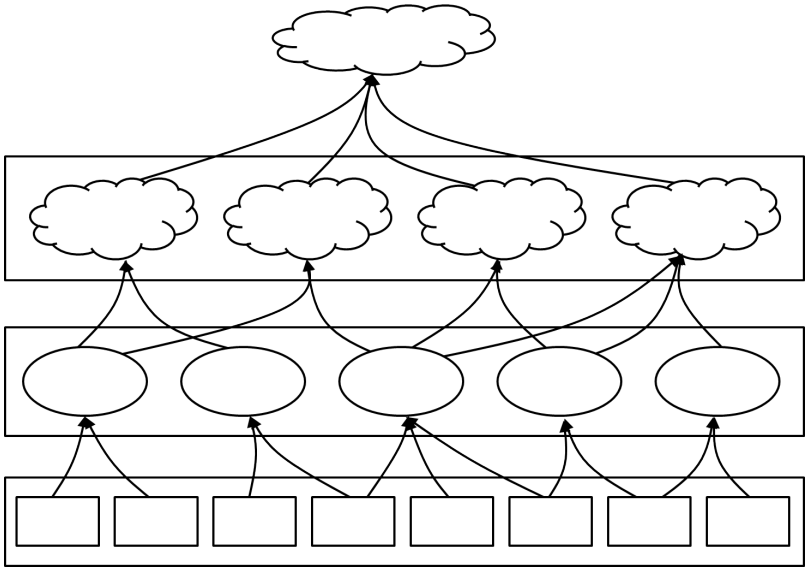


Figure 3.8: An ideal embodied algorithm has only one implemented world, from which an interesting collection of mechanisms and phenomena emerge.

each other, to embody the evolving phenomena. The lowest-level world (below the level of the mechanisms we are interested in) is likewise one single world, containing all the parts of the algorithm that are explicitly implemented by a computer programmer. This is an ideal and may never be realised in practice. It is useful to keep in mind and work towards, while realising that actual implementations of novelty-generation algorithms may well contain multiple worlds on different levels, thus embodying different phenomena to different degrees.

One issue with embodying many different phenomena within the same world is that this moves the implementation further down the tree. This means that the resulting computer program will take longer to run, because it is creating a complex world on a lower level of detail. This is the central tradeoff with embodied systems: trading off the degree of embodiment against the run-time of the resulting program.

3.3 Designing embodiment

In order to obtain the benefits of embodiment within a computer program, there are four things we need:

1. Some phenomena that we want our computer program to display (such as evolution, robustness, complexity, novelty generation, etc...). This is the objective of the computer program.

2. Some mechanisms that we believe give rise to the phenomena (such as reproduction with variation, death, and competition for resources). This is the biological model underlying the program.

3. A virtual world in which we can build the mechanisms out of simpler parts (such as an artificial chemistry). This is the computational model.

4. A traditional computer programming language. This is used to implement the virtual world, so we can run instances of the world on currently-available computer hardware.

For novelty-generation algorithms, we want the program to display the phenomenon of novelty generation. We are trying to achieve this by embodying the phenomenon of evolution. Our biological model is the sub-set of chemistry that gives rise to evolution as an emergent property, as explained in section 2.6.3. Artificial Chemistries provide a framework for describing the types of virtual world we are interested in.

3.3.1 Nested programming languages

These four requirements give us three programming languages, nested within each other, as shown in figure 3.9. We use a traditional programming language (e.g. Python, Java, C++) to implement a virtual world. This *implementation language* is an arbitrary choice that does not matter from an embodiment viewpoint. The virtual world can be thought of as a programming language in its own right (e.g. as an Artificial Chemistry). Using this *world language*, we implement the mechanisms of our biological model (e.g. self-replicating chemicals). A collection of mechanisms can also be thought of as a programming language, on a higher level of abstraction than the world language. Using this *mechanism language*, we implement the phenomena that we are interested in (e.g. reproduction with variation), possibly as emergent properties.

The primitives of the world language are basic building blocks, and all the properties that we want our mechanisms (or systems, in Quick's terminology) to gain *for free* from their environment (such as stochasticity, conservation laws, etc.). These properties are not free in terms of the system as a whole, because they must be calculated by the underlying implementation language, but they appear free from the viewpoint of the mechanisms because they are primitives in the world language.

The primitives of the mechanism language are ways of combining together the basic building blocks of the world language. The mechanism language builds structures in the world, and uses the 'free' properties provided by the world to help these structures act on each other. When the mechanism language runs, different

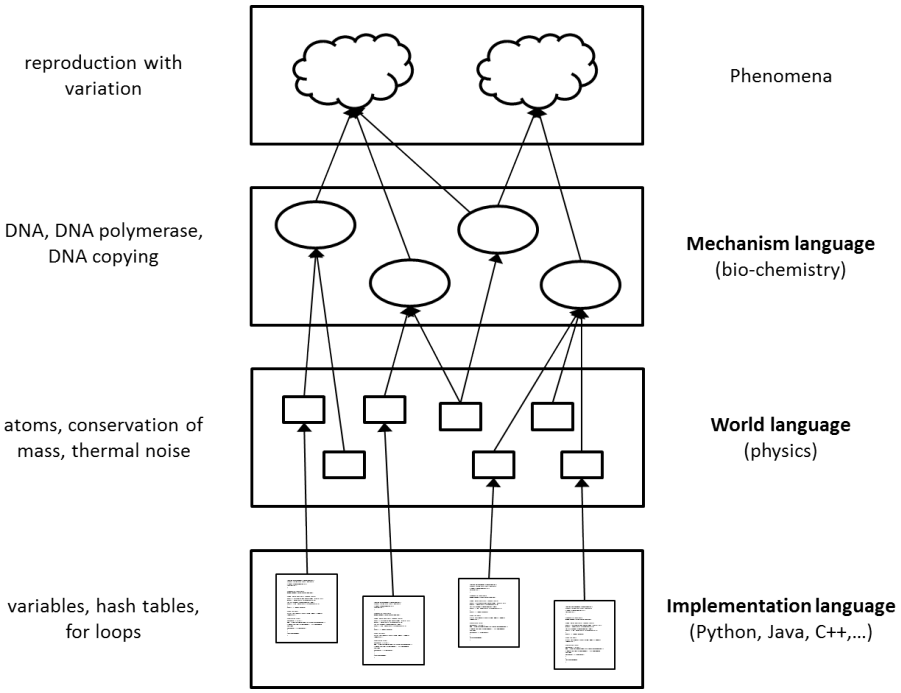


Figure 3.9: Embodying a phenomenon within a computer program requires three programming languages nested within each other. Everything can be thought of as a consequence of the implementation language, but explicitly noticing the world language and mechanism language allows us to study the system from the viewpoint of embodiment.

	Socio-economics	Self-reproduction	Fundamental physics
Phenomena	global recessions, businesses going bankrupt	reproduction with variation	quarks, electrons
Mechanisms	purchases, debts	DNA chemicals, DNA polymerase chemicals	strings splitting and recombining
World	people, commodities	atoms, conservation of mass, thermal noise	strings moving in 10 dimensions

Figure 3.10: The interpretations of what constitute *phenomena*, *mechanisms* and *worlds* can differ wildly between different systems.

mechanisms interact with each other and we observe the phenomena that we are interested in.

If our phenomena are biological, then we can think of the world language as physics and the mechanism language as bio-chemistry. Physics provides the basic building blocks of atoms and electrons, and the 'free' properties of mass, charge, attraction/repulsion, etc. From these building blocks we make bio-chemicals such as DNA and DNA polymerase, and using the properties we allow the bio-chemicals to interact with each other. This allows us to implement mechanisms such as DNA copying. When this mechanism runs, it displays the phenomenon of reproduction with variation.

Different applications will have different interpretations of what the world language and mechanism language correspond to. Figure 3.10 shows examples of this for three different applications on very different levels: a socio-economic system on a very high level; a system of self-reproducing chemicals on the biological level; and a fundamental physics system on a very low level. Also some applications will be hierarchical, with the world language on one level being implemented in terms of the phenomena of a lower level. But the same general theory of phenomena, mechanisms and worlds applies to (each level of) each of these examples.

This nesting of programming languages within each other is not a new thing. Whatever language it is written in, every computer program implicitly defines its own programming language [1, p.294]. This is because the purpose of programming is to build higher-level structures from lower-level components. Most programmers do not care about the implicit programming languages they are defining: they just care about building a working program. But in this case, we specifically care about the programming languages we are defining, because we are matching them to levels of complexity in biological systems. Our aim is to implement a low-level description of biology (chemicals and reactions) and have the higher levels emerge.

3.3.2 Crisp, stochastic and embodied languages

The world language and mechanism language are not like traditional computer programming languages. The purpose of embodiment is to gain variability from the environment. From the point of view of the mechanism language, this usually means that the mechanism does not operate perfectly. As the mechanism is operating, things go wrong because of the environment being inherently variable. With the physical world, this is easy to achieve. Everything in the physical world is noisy and not perfectly predictable. But with virtual worlds, we need to replicate this variability. The world language needs to contain variability, so that the mechanism language can implement mechanisms that do not operate perfectly. This section describes how we can achieve this, and gives some terminology for talking about computer programs embodied within virtual environments.

3.3.2.1 Crisp languages

In traditional computer programs, operations should happen *crisply*, without any errors. For example, if a programmer writes the assignment statement $a := b$ in their code, they expect the variable a to contain a precise copy of the variable b. Expecting the copying process to work perfectly is a very reasonable expectation to make of a computer programming language. The copying process here is treated as a black-box phenomenon. It does not matter *how* the programming language (or the underlying hardware) implements the copy, as long as it happens perfectly.

However, this is not necessarily what we want for novelty-generation algorithms. When biological life-forms (such as bacteria) clone themselves asexually, the clones are not exact copies of their parents (see any biology textbook, e.g. [15]). The biological *copy operation* does not work perfectly. But this is not a mistake. Biology would not be improved by a perfect copy operation. Imperfect copying in biology results in mutations that generate novelty, exploring the space of possible organisms that could exist and allowing evolution to happen.

The reason biological copying does not happen perfectly is that in biology, copying is not a black-box phenomenon: it is embodied within the physical world. In order for a biological organism to be copied, there must be a biological mechanism existing in the physical world and performing the copy (i.e. the DNA polymerase chemical). This biological mechanism (or collection of different mechanisms) embodies the phenomenon of copying within the physical world. Embodiment gives the biological copying process the benefits mentioned above. The variability of the environment causes errors in the copying process that allow evolution to happen. Because traditional computer programming languages are crisp and do not vary, a major task for novelty-generation programmers is replicating this variability in the world language of an embodied system.

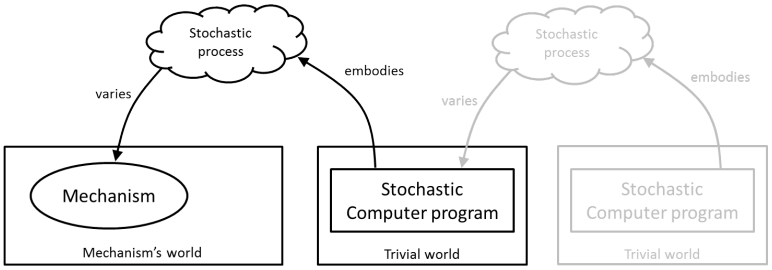
3.3.2.2 Stochastic languages

Stochasticity is a way of introducing variation into computer programs (or more generally, any systems). This variation can serve as a substitute for the unpredictability of the physical world. Stochastic programs are crisp programs with variation introduced via pseudo-random number generators. These do not represent true randomness, because all computer programs are strict on their lowest level. (Except ones that use stochastic hardware exploiting thermal noise or radioactive decay, but these are not yet widely-available.) Pseudo-random number generators use the mathematical idea of chaos to produce very unpredictable outcomes that are a good approximation of true randomness.

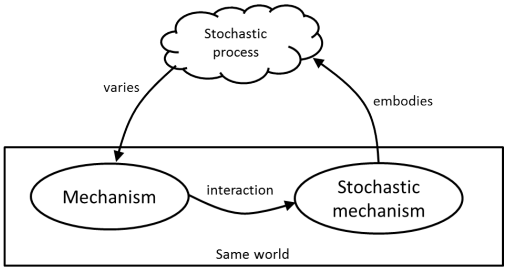
However, the variability of the physical world is not the same thing as mathematical randomness. The benefits gained by embodiment are more than having a good random number generator. This is because the randomness of the physical world has different effects depending on which mechanisms are implemented within it. For example, if you are drinking a cup of tea or coffee, then there will be random variations in the kinetic energy of each atom in the cup. If the energy of one of these atoms were to suddenly change by a small amount, it would technically change the temperature of the drink, but the change would be so small that you would not notice the difference. On the other hand, the same energy variations happen to the atoms of a bacterium's DNA as it is replicating. If the energy of (the appropriate) one of these atoms were to suddenly change by a small amount, then this could cause a mutation in the bacterium's DNA that allowed it to bypass your body's defences. If this bacterium was living on the biscuit you were dunking into your drink, then the result of this tiny change could be you getting food poisoning. This is a very different effect from not noticing a tiny change in the temperature of a drink, but the cause each time is the same. The same cause was able to have very different effects because it was a random change to different mechanisms.

3.3.2.3 Embodied languages

Every process is always embodied within some world, but the purpose of embodiment is to embody different processes within *the same world*. Suppose we have a mechanism that we want to vary. We could write some computer code to implement a stochastic process that varies the mechanism. This external stochastic process would be embodied in a trivial world, outside the world of the mechanism, as shown in figure 3.11(a). If we wanted the stochastic process to vary, we would have to implement a second stochastic process, in a separate trivial world, that varies the first process. This creates the tower of meta-levels that we have seen before (section 2.6.2.1). It can be broken by embodying the stochastic process within the system (figure 3.11(b)).



(a) An external stochastic process



(b) An embodied stochastic process

Figure 3.11: An external stochastic process can vary a mechanism, but the mechanism cannot vary the stochastic process. A higher-level stochastic process is needed for this. Embodying the stochastic process within the same world as the mechanism, allows the mechanism to interact with the stochastic process and vary it.

For example, a naive way to implement the phenomenon of *copying* would be to have an organism in a novelty-generation algorithm run a function called `copy`, that makes a perfect copy of the organism, and then stochastically introduces some errors. (This is how mutation happens in most evolutionary algorithms.) This is an example of an external stochastic process, because the organism has no control over the number or types of errors introduced. The organism is embodied in its world, and the stochastic copying process is embodied in a separate world: a trivial world that contains only the syntax of the organism that is being copied, and some virtual dice that can be rolled to make stochastic choices. As it stands, the only way to modify this external copying process would be to implement another external process, embodied in another separate world, creating a tower of meta-levels.

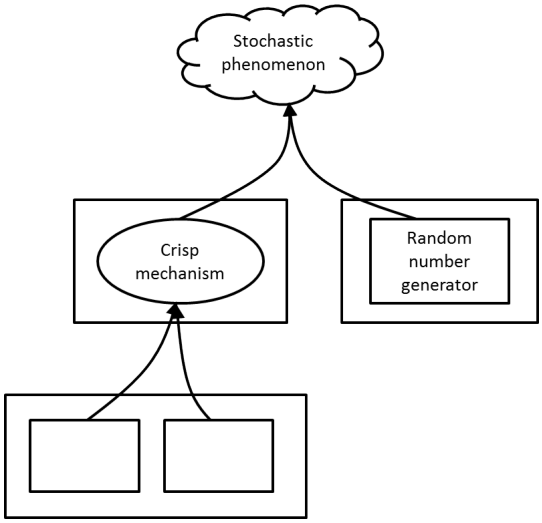
A better way to vary the phenomenon of copying would be to embody the copying phenomenon within the same world as the organisms that are being copied. Rather than creating a separate (trivial) world in which copying takes place, we can enrich the world in which the organisms are implemented, so that copying can take place there. This means implementing stochastic mechanisms that perform the copying process, within the world of the organisms. Now the organisms can change their copying process, because it is embodied in the same world as the organisms. In other words, the world language of the copying process is the same as the world language of the organisms. Figure 3.11(b) shows this.

3.3.2.4 Where does the randomness go?

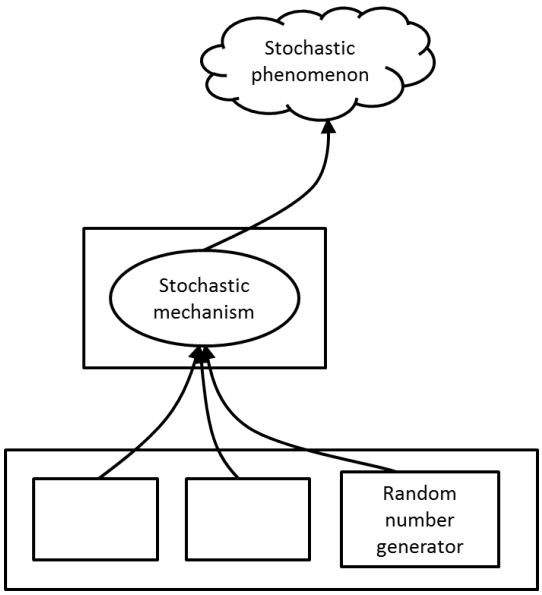
In an embodied system, we have three programming languages: the implementation language, the world language and the mechanism language. The implementation language must be completely crisp, because this is a traditional programming language (ignoring hierarchical systems, although these must have a lowest level that is crisp). So the choice is whether to put stochasticity into the world language or the mechanism language.

The stochasticity comes from a pseudo-random number generator that is part of the implementation language. Therefore, putting stochasticity directly into a mechanism, forces that mechanism into its own separate world (as in figure 3.11(a)). This is a very subtle point. The mechanism may be able to interact with other mechanisms, and may be able to operate in their world for some of its functions. But as far as its stochasticity is concerned, it is operating in its own world. This means that the stochastic mechanism may be able to change other mechanisms due to its stochasticity, but other mechanisms will not be able to affect the stochastic process (and in particular, it will not be able to alter itself).

Therefore, the best place to put the stochasticity is into the world language. Figure 3.12 shows the difference between putting the stochasticity into the world language versus the mechanism language. A major benefit of putting it into the



(a) Putting the stochasticity into the mechanism language



(b) Putting the stochasticity into the world language

Figure 3.12: Putting the stochasticity into the world language creates a more embodied and evolvable system.

world language is that the same source of randomness can be shared by many different mechanisms. The world language can contain a stochastic property that many different mechanisms can interact with. For example, the world language could contain the property of thermal noise. This property would randomly make changes to parts of the world, depending on their temperature. Because this randomness is a property of the world language, different mechanisms can control the effect it has on them. For example, if a copying process benefited from having a low error rate, then it could evolve a way to perform copies while remaining at a low temperature. Or alternatively, if a malicious mechanism wanted to disrupt the function of its rival, then it could raise its rival's temperature, and thus subject its rival to more thermal noise, and hence more randomness.

Technically, an embodied system does not require any randomness: every language could be crisp. But this is effectively just a (probably inefficient) implementation choice for a traditional, crisp algorithm. Doing this would have value if it was the first step in an iterative design process, where there was a desire to have a (working, crisp) world that could be examined to decide which parts to change to be stochastic or embodied.

3.3.2.5 The spectrum of embodiment

Embodiment is a spectrum: it is not all-or-nothing. Because every process is always embodied within *some* world, every system is always embodied *to some degree*. The purpose of embodiment as a concept is to identify the different worlds comprising each system: identifying the different worlds that different processes are embodied within. The more different processes are embodied within the same worlds, and thereby able to change each other, the more embodied the whole system is.

Because this use of embodiment is a new concept, we need new terminology to talk about it. My use of the words *crisp*, *stochastic* and *embodied* is a first attempt at identifying some landmarks on the landscape of embodiment. The terminology is not perfect, and is not complete by any means.

I use the word *crisp* to mean a composite of *deterministic* and *immutable*. Crisp processes are on one extreme of the embodiment spectrum. They always operate in the same way, and their operation cannot be changed.

Stochasticity is an obvious way of allowing systems to be changeable. Making a crisp process stochastic removes the *deterministic* part, allowing the process to be changed, but begs the question of *who changes the process?*

An *embodied* process is one that is stochastic and is changeable by *other entities within the same world*. Embodiment is a specific type of stochasticity, that answers the question of who changes the process. It is the complete opposite of crispness. I think that this particular type of stochasticity is a useful concept for evolutionary algorithms.

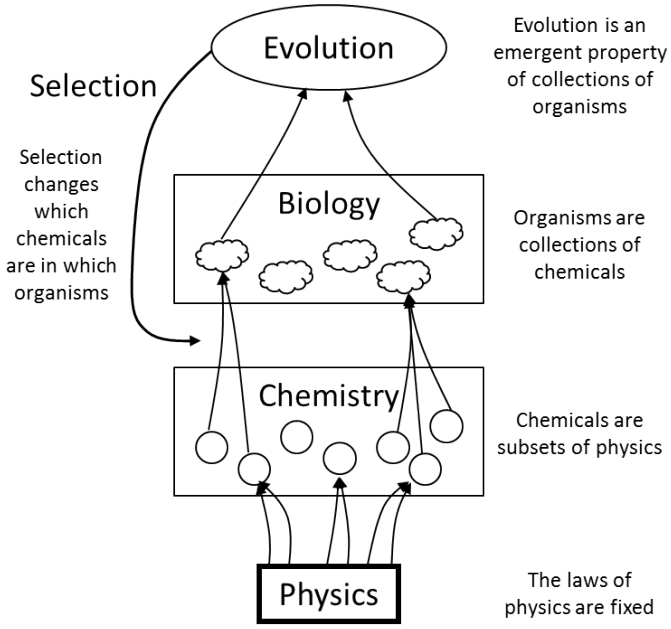


Figure 3.13: Emergent evolution

3.4 Emergent evolution: Summary

Figure 3.13 summarises this chapter, explaining how we can use embodiment to implement emergent evolution.

We start with the laws of physics, which are the things we implement explicitly within our computer program, and constitute the *world* of our embodied system. In principle, we could implement the laws of physics in terms of fundamental particles and fields, but this would lead to very slow programs by the time we got up to the level of evolution. So in practice we implement abstractions, simplifications and alternatives of the laws of physics, that run faster on our current hardware. The world of our simulations does not have to correspond directly to the physical world. The idea behind the world/physics is that it is a set of components that do not change or evolve.

By taking subsets of the components of physics and allowing them to interact, we get chemistry. This is the *mechanism* language of our system. A chemical mechanism is a collection of components from physics (e.g. atoms and bonds) that can interact with each other.

Biological organisms are the *phenomena* of our system. They are composed of collections of chemicals that act together to achieve a high-level result (e.g. self-reproduction). To close the loop, we need the biological organisms to be embodied within the chemistry language in such a way that they have evolution as

an emergent property (as described in section 2.6.4.2).

Evolution cannot change the laws of physics (the world language), nor can it directly change chemicals (the mechanism language). What evolution does is *select which mechanisms are present in which organisms.* The intrinsic variation (stochasticity) in the world language creates new mechanisms. Evolution selects which of these mechanisms will survive to be part of the evolving organisms. Because evolution is an emergent property of this collection of evolving organisms, evolution can alter the mechanisms of evolution.

This chapter has provided a very abstract, theoretical introduction to the idea of embodiment. Our aim from here is to implement embodied evolution within the world of computer programs: we want to write a computer program that embodies evolution. There are many different mechanisms we could use to do this, and so to help us choose which mechanisms to embody, we will look to biology for inspiration. The next chapter explores some of the ways in which biological systems benefit from embodiment, to illuminate the theory introduced in this chapter and provide inspiration for our computer programs (which we describe in later chapters).

Chapter 4

Biological embodiment

We are aiming to turn the theoretical concept of embodiment into a concrete computer program. In order to know which mechanisms to embody, we look to biology for inspiration. Biological organisms are embodied within the physical world, and display the novelty-generation properties we are interested in. This chapter describes some biological mechanisms that embody evolution, and explains how biological systems benefit from their embodiment within the physical world. Later, we take inspiration from some of these mechanisms in the design of our computational novelty-generation system.

Biology is a huge subject, so this chapter focuses on biological mechanisms that embody the phenomenon of *copying a genome*. This may seem a very narrow focus, especially from a computer programming viewpoint, but it is not narrow in terms of biology. Copying data is fundamental to computer programs, and copying operations appear as primitives in most programming languages (and often in the hardware of microprocessors). This makes it very easy to implement crisp copying mechanisms in computers. But this is not the case in biology. Biological 'copy programs' are embodied within the physical world as collections of mechanisms operating in parallel on different timescales. This chapter shows how biology's embodiment of its copying mechanisms contributes to the phenomena of *evolvability* and *meta-evolution*.

Most of the mechanisms described in this chapter are biological common knowledge, and can be found in any first-year undergraduate genetics textbook, for example [15]. Where I talk about things that are not common knowledge, I provide explicit references.

4.1 Looking at bacteria

Biology is complicated. There are huge numbers of mechanisms interacting within a single cell, let alone in a multi-cellular organism. If we wish to extend the biological models of existing novelty-generation algorithms, it makes sense to consider some of the *simplest* biological systems displaying the phenomena we are looking for. Since we are looking for evolution, with respect to novelty generation, bacteria provide a good model system. Bacteria are very good at organising and rearranging their genomes, which is a major design problem in novelty-generation algorithms (choosing appropriate problem representations and evolutionary operators).

On the very low level of copying their DNA, bacteria operate in the same way as the multi-cellular organisms that inspired previous novelty-generation algorithms, so much of the biological intuition from those algorithms carries over to bacteria-inspired algorithms. But because bacteria are unicellular, the process of copying their DNA chemicals takes up a significant proportion of their replication cycle. Because of this, bacteria have developed some ingenious mechanisms for working with DNA (both their own, and that of other organisms), that makes them particularly suited to our purposes.

For example, bacteria reproduce asexually. One cell divides into two daughter cells that each contain a copy of the parent's DNA, possibly with mutations. This is very different from the sexual reproduction that inspired genetic algorithms, where two parents combine their DNA to produce children. Sexual reproduction has evolutionary advantages over asexual reproduction, because it allows genes to be spread more rapidly through an evolving population. Bacteria have reproduced the phenomenon of *rapidly spreading genes*, but they have embodied it using very different mechanisms to multi-cellular organisms. Bacteria have evolved a mechanism (called *conjugation*) for transmitting pieces of DNA between each other, without having to reproduce.

So bacterial cells can change their own genomes, where multi-cellular organisms must have children in order to create organisms with different genomes. This illustrates the fact that different biological systems often show the same phenomenon, but use different mechanisms to embody that phenomenon. And different mechanisms come with different costs and benefits, and work better in different environments. By looking at a wide variety of different mechanisms used by biological systems to embody evolution, we can choose the ones most suited to implementing computationally, or choose the ones must suited to whatever application problem we are trying to solve. In this chapter, I describe the biology necessary to understand the conjugation mechanism, and see it as a marvelous example of meta-evolution.

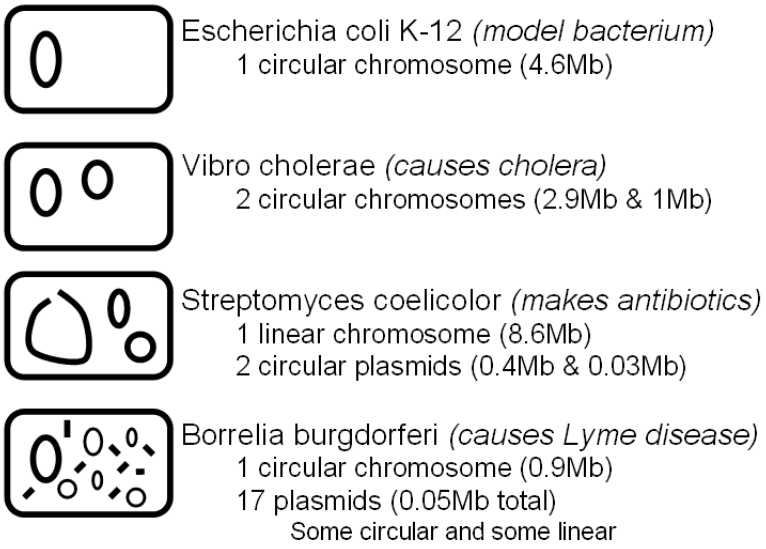


Figure 4.1: Some combinations of linear and circular plasmids and chromosomes in different bacteria. An example of the general rule that in biology, every rule has an exception. (*Mb* means *Mega bases* of DNA, or 10^6 bases.) Information from [15, p.229].

4.1.1 Genome organisation

Bacteria have very varied genetic organisations (some examples are shown in figure 4.1). They usually have one main chromosome of DNA, although some have two chromosomes (e.g. *vibro cholerae*). This chromosome is not linear, like human chromosomes, but circular. However, some bacteria do have a linear chromosome (e.g. *streptomyces coelicolor*). As well as having genes on chromosomes, they also have multiple copies of separate pieces of DNA called plasmids, which are usually circular. They contain genes that are useful in certain circumstances but are not part of the normal life-cycle of the bacterium. For example, some contain genes coding for antibiotic resistance, and some contain genes coding for toxins that attack the bacteria's host. A given bacterium may or may not contain a given plasmid, and may behave very differently if it acquires the plasmid. For example, the bacterium *escherichia coli* lives in the large intestine of all (healthy) humans, however, if this bacterium acquires a particular plasmid then it can cause food poisoning. Some plasmids provide bacteria with the ability to transfer plasmids to other bacteria (via conjugation). This mechanism allows genes to move between the plasmids and chromosomes of a bacterium, and also to other bacteria.

A bacterium's chromosome(s) and plasmid(s) are a collection of DNA chemicals that embody the bacterium's genome in the physical world. The genome is a

phenomenon, that can be described by a sequence of the letters A, C, G and T. The DNA chemicals embody this sequence of letters, but they are more than just the abstract sequence. They are chemicals that exist in the rich, physical world. They are acted on by mechanisms that exist in this complex, stochastic world. And so, the ways in which a bacterium's DNA can change are more than abstract functions applied to sequences of letters. Traditional novelty-generation algorithms try to work out which abstract functions are best to apply to a sequence of letters, and then wonder how to vary and evolve these abstract functions. We can see these abstract functions as biological phenomena. Embodied evolution asks which *mechanisms* biology uses to embody these phenomena. We can vary and evolve the phenomena by implementing these mechanisms in a stochastic world.

The sections that follow are a very simple introduction to some of the biology that makes bacteria evolvable. The purpose of this exercise is not to explain the biological processes in minute detail, but to look at the types of mechanism that exist in biological systems. We can then think about the phenomena that these mechanisms display. It is these phenomena that we wish to embody in computational worlds, rather than a realistic description of biology.

4.2 DNA replication and transcription

There are four main types of chemical present in a biological cell:

- *Proteins* are large chemicals that form the active machinery of a cell.
- *DNA* carries the information necessary to build proteins.
- *RNA* is an intermediary between DNA and proteins; each gene on the DNA is transcribed into multiple copies of RNA, before being translated into proteins.
- *Small chemicals* are taken into the cell from its immediate environment; these affect the way in which proteins and DNA work.

The interactions of these four different types of chemical implement the function of a cell and embody evolution, allowing populations of cells to evolve. This section describes the low-level mechanisms by which these four types of chemicals interact, which the higher-level mechanisms in the following sections build on.

4.2.1 Copying DNA

Copying DNA is an example of a mechanism that is hardcoded as a stochastic phenomenon in traditional novelty-generation algorithms. DNA is copied perfectly (by

a crisp program written in a high-level language) and is then explicitly mutated (by a stochastic program written in a high-level language). Biological cells, however, embody their copying process by using a protein called *DNA polymerase* to copy DNA (technically there are many proteins involved in the copying process, but DNA polymerase does the main work). It attaches to the DNA and moves incrementally along the strand of DNA, reading the current base (A, C, G or T) and adding the complementary base to a copy (A and T are complementary, as are C and G). The fact that DNA is copied by a machine moving incrementally along the DNA, has many computational consequences.

4.2.1.1 Bloat control

Because the copying machine moves incrementally along the DNA, it takes a certain time to copy the DNA. This time is proportional to the length of the DNA. Thus cells with a lot of DNA will take longer to copy their DNA, and so will take longer to reproduce. In environments where reproducing faster is beneficial, evolution will therefore work to control bloat in the DNA, so that reproduction time is not wasted copying unnecessary DNA. So embodying the copying process gives biological cells a bloat-control mechanism for free, solving a problem that affects traditional genetic programming.

4.2.1.2 Mutations

Since the copying mechanism is embodied in a stochastic world, it may make copying mistakes. Unlike explicit mutation (by a stochastic computer program), the frequency and type of these mistakes is not something hardcoded into a computer program. It is a complex interaction between the copying mechanism and the precise sequence of bases that are being copied. Because the copying mechanism is itself encoded on the DNA that it is copying, this provides a way for evolution to control its own mutation phenomenon. Note that this is not just a way of controlling an overall mutation rate, but a way of controlling what types of mutations are likely to occur at which places on the DNA.

4.2.2 Transcribing and translating DNA

As well as the copying of DNA, the translating of DNA is also performed by a collection of protein machines. *RNA polymerase* is a protein that incrementally reads the DNA and transcribes it into RNA. A schematic diagram of how it does this is shown in figure 4.2. A *ribosome* protein then incrementally reads the RNA and translates it into proteins. The difference between transcription and translation is the same as its use in linguistics: whether the resulting language is the same

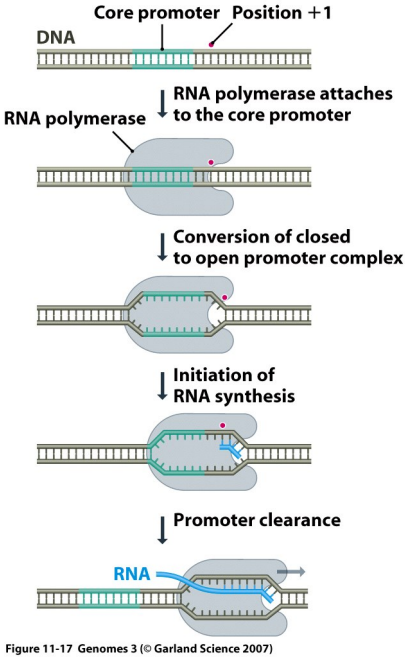


Figure 4.2: The RNA polymerase machine starting the process of reading DNA to produce RNA [15, figure 11.17].

or different. DNA stores information as a sequence of bases. In computational terms, this is a string over the alphabet {A, C, G, T}. RNA also stores information as a sequence of bases, as a string over the alphabet {A, C, G, U}. The chemical structure of DNA and RNA is slightly different, and the T base of DNA is replaced by the U base of RNA, but from an information viewpoint there is no difference; they are storing information in the same language. Proteins, however, store information in a different language. Although a protein is also a string over an alphabet, the protein alphabet is chemically very different from the DNA/RNA alphabet, and contains 20 characters. Each successive triplet of bases in the RNA string is translated into one character of the protein string, according to the table in figure 4.3.

We have seen that embodiment of the copying phenomenon gives biological cells extra properties, that are not available to computational systems that implement the phenomenon directly. In the same way, embodiment of DNA reading gives extra properties, which are described below.

4.2.2.1 Redundancy

There are $4^3 = 64$ different possible triplets of RNA, and only 20 different characters in the protein alphabet (plus one *start translating* character and one *stop*

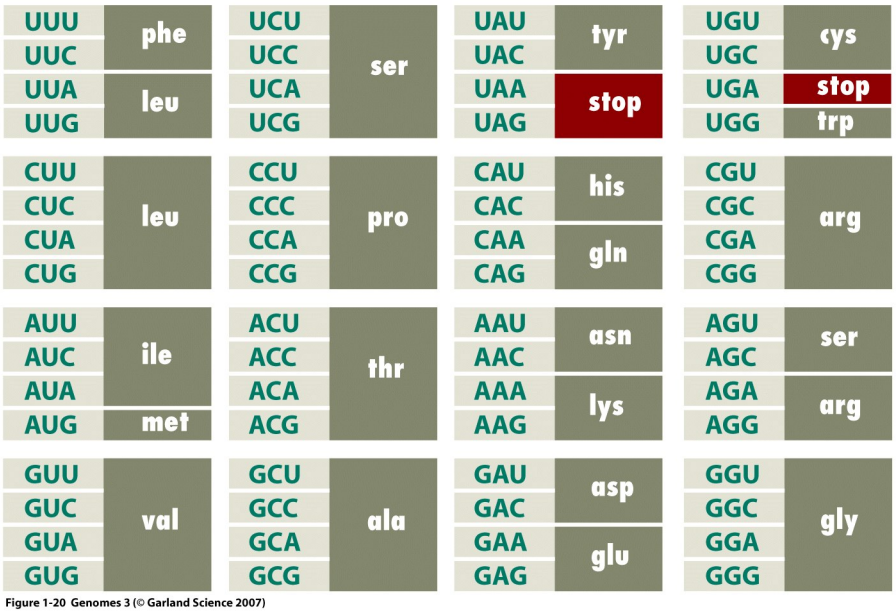


Figure 4.3: The genetic code, that translates triplets of RNA characters into protein characters [15, figure 1.20].

translating character). This implies a high level of redundancy in the translation process. There are many different strings of RNA that code for the same protein. Redundancy in the genotype-phenotype mapping exists already in evo-devo algorithms (section 2.3.3), and this was part of the biological basis for it.

As an example, the DNA double-string:

TAATGCTAGACGTGTTCTAGGA
ATTACGATCTGCACAAGATCCT

would be transcribed into the RNA single-string:

UAAUGCUAGACGUGUUCUAGGA,

which is a copy of the top DNA string (the complement of the bottom DNA string), with T replaced by U. This would then be translated into the five-character protein string:

met - leu - asp - val - phe.

The *start* triplet is AUG and one of the (redundant) *stop* triplets is UAG. (For the full copy of this mapping, see figure 4.3.) All RNA characters coming before the *start* triplet or after the *stop* triplet are ignored.

We can see from this mapping that many different DNA strings could have been used above, and would have mapped to the same sequence of amino acids. This

redundancy in the mapping between DNA and proteins helps biological cells to be evolvable. *Neutral mutations* can change the DNA sequence without changing the proteins that the sequence codes for.

4.2.2.2 Bursting

As with DNA copying, there are consequences to the fact that transcription is performed by machines moving along the DNA. For example, RNA polymerase can pause as it is moving along the DNA, which causes the genes to be transcribed in bursts rather than at a constant rate [83]. Moreover, the degree of bursting can be modulated by the actual sequence of bases on the DNA being transcribed.

This is an example of a property of the 'evolutionary algorithm' that can be evolved by the algorithm itself. The idea of having bursting in a novelty-generation algorithm might not even be imagined without the concept of machines moving along the DNA, without considering how difficult it would be to evolve. But having DNA-reading embodied as a mechanism within a world, allows the mechanism to operate differently, within the constraints imposed by the world. In the case of bursting, this is the transcription machine pausing as it moves incrementally along the DNA, rather than moving at a constant rate. Because the world allows mechanisms to operate in different ways, the phenomenon embodied by the mechanisms can be varied. An example of this is expressing proteins in bursts, with the degree of bursting modulated by the particular DNA sequence being expressed.

4.2.2.3 Parallelism

In multi-cellular organisms, transcription and translation happen in different spatial regions of the cell (transcription inside the nucleus, and translation outside). This ensures that transcription of one RNA chemical finishes completely before its translation begins. However, in bacteria this is not the case. Both transcription and translation happen in the same place in the cell (because bacterial cells do not have a nucleus), and they happen in parallel. Ribosomes begin translation of the RNA before the RNA polymerase has finished transcribing the RNA from the DNA. Also, multiple RNA polymerase proteins can be transcribing the same gene on the DNA in parallel, and multiple ribosomes can be translating the same piece of RNA in parallel. This is shown schematically in figure 4.4.

This parallelism is the reason that RNA polymerase pausing can cause bursts of translation, as described above. The *lead* RNA polymerase pauses, causing a *traffic jam* of RNA polymerases to build up behind it. When it unpauses, they all start moving again at the same time, and so all finish transcription at the same time. In multi-cellular organisms this causes a burst of RNA to be produced, but in bacteria the *traffic jam* carries over and jams the ribosomes as well, which causes

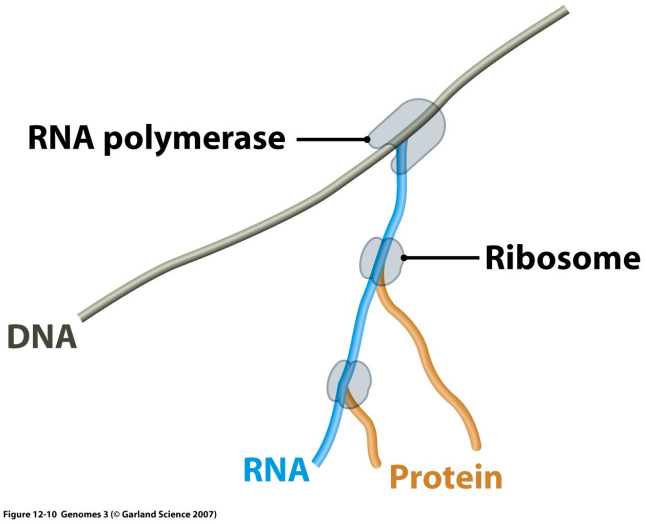


Figure 4.4: Multiple copies of biological machines (RNA polymerase and ribosomes) working in parallel [15, figure 12.10].

a burst of protein production.

The idea of multiple copies of machines working in parallel is ubiquitous in biology. For example, a bacterial cell can only divide once its chromosome has been copied by DNA polymerase. One copy of the chromosome is given to each daughter cell. But before the copying process has completed, each copy of the chromosome has already started its own copying cycle. When the daughter cells are produced, they are already well into their own copying cycles. This overlapping of stages stands in sharp contrast to evolutionary algorithms based on multi-cellular organisms, where each stage is fully completed before the next can begin. It is closer to the artificial life models of automata systems (section 2.4.2.1) in which individual instructions are executed in series, but programs of different length execute in parallel.

One computational property gained by overlapping different stages is the ability of different levels to influence each other. For example, RNA polymerase can pause due to the underlying sequence of DNA bases. This causes ribosomes to also pause, whether or not the ribosomes would have paused on their own. This influence across stages is difficult to achieve if the stages are explicitly split, but is easy to obtain if they are overlapped.

When abstract phenomena are implemented directly as computer programs, it is very difficult to overlap different stages of the algorithm. If we have a Java function called *copy_genome*, then we have to wait for this function to finish and return its result before we can think about copying the daughter genome. But bacterial copying systems do not have this limitation. They can start copying the

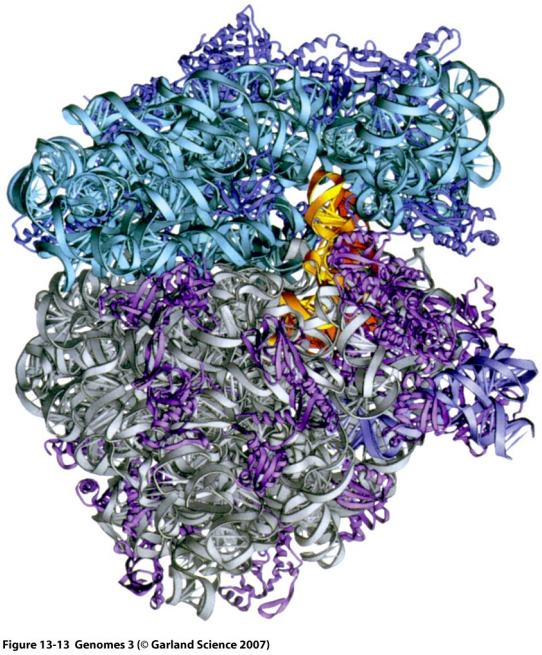


Figure 4.5: The (very complicated) protein structure of the ribosome from the bacterium *Thermus thermophilus* [15, figure 13.13].

daughter genome before the parent genome has finished copying, because copying is a phenomenon embodied by machines in a parallel world.

4.2.3 Biological complexity

In physical-world biology, the machines working on the DNA are very complicated. For example, figure 4.5 shows the structure of a bacterial ribosome. We are not looking to copy all of this complexity. Our purpose in studying biology is to pick out some of biology's mechanisms, and the phenomena they entail. For example, some of the mechanisms involved in DNA replication and transcription are: reading and writing DNA and RNA one unit at a time; encoding machines on the DNA; translation of the same information into a different language; working incrementally on different levels at the same time; and repeating the same process many times in parallel. These give rise to the phenomena of: bloat control; evolvable mutations; evolvable pausing and bursting; and interactions between different stages.

It is these mechanisms that we are wanting to copy in our computational models, rather than the specific ways in which biology has implemented them. We are trying to copy figure 4.4 rather than figure 4.5.

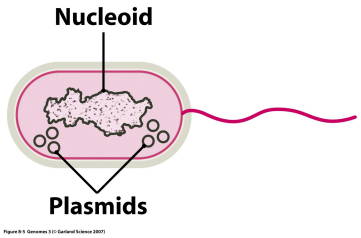


Figure 4.6: A bacterial cell, with a single chromosome (nucleoid) and multiple plasmids [15, figure 8.5].

4.3 Plasmids

Plasmids are a major way in which bacteria organise their genes. Figure 4.6 illustrates that they are individual pieces of (usually circular) DNA that can be thought of as separate to the bacteria's *core* genome, as they replicate separately from the main chromosome(s) and carry genes that are not part of the normal life-cycle of the bacterium [104]. Plasmids carry genes that are useful in certain special situations, such as:

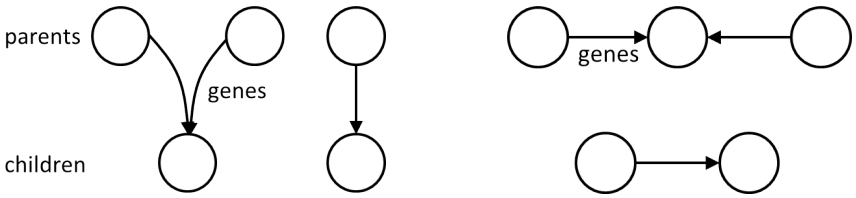
- Resistance to antibiotics.
- Toxins to kill other bacteria.
- Proteins that cause disease in hosts.
- Proteins to metabolise unusual food sources.
- Conjugation: transferring plasmids between bacteria.

Because plasmids are separate from the main chromosome (both physically and in the functions of their genes), bacteria can transfer plasmids between each other. This has many implications for meta-evolution, as explained in this section.

4.3.1 Horizontal gene transfer

In multi-cellular organisms, genes can only be transferred between individuals *vertically*, i.e. children inherit genes from their parents. But because bacteria can exchange plasmids, they can transfer genes *horizontally*: at any time during an organism's life, it can receive genes from another member of the population. Figure 4.7 shows these two mechanisms.

The fact that bacterial cells can transfer plasmids between each other makes them very evolvable. This is particularly so for features such as antibiotic resistance, production of toxins and the ability to metabolise unusual food sources.



(a) Vertical gene transfer (b) Horizontal gene transfer

Figure 4.7: Vertical gene transfer only allow genes to be passed from parents to children during reproduction. Horizontal gene transfer allows genes to be exchanged at any time, between arbitrary individuals of a population.

Only one bacterial cell needs to evolve one of these abilities, and (assuming it survives long enough) it can pass this ability on to other cells, which can then pass it on further. This allows beneficial mutations to spread through a population much more quickly than if they were spreading to children through reproduction. In order to amplify a beneficial trait in the population, it is only the beneficial plasmid that needs to be copied, rather than whole organisms.

Horizontal gene transfer is possible in biology because the genome is embodied within DNA chemicals. This enables different genes to reside on different DNA chemicals, and allows mechanisms to evolve that move these DNA chemicals between organisms. If the genome of a computational organism was implemented as a pre-defined, fixed data structure (such as a string of length 17), then it would not be possible to split this genome into separate pieces (and still have a valid string of length 17). This would be an implementation of the genome phenomenon, rather than embodying the genome within a DNA chemical existing in a world, where it can be split into pieces and moved around by different mechanisms.

4.3.2 Conjugation

Conjugation is a mechanism that implements horizontal gene transfer, and is a very good example of meta-evolution. The bacterium *E.coli* has a plasmid known as the *fertility factor* or *F-plasmid*. This plasmid contains genes that allow the bacterium to perform conjugation, which is a process that copies a plasmid from one bacterium to another, as shown in figure 4.8. The mechanism of conjugation operates as follows: the *donor* bacterium builds a structure (out of proteins) known as a *sex pilus*. This is a long appendage that it uses to attach itself to the *recipient* bacterium. The donor then draws the recipient towards itself, before opening a channel between the two cells. Down this channel is sent one strand of the double-stranded DNA composing the F-plasmid. After the plasmid has been sent, both cells complete their single-stranded plasmids into double-stranded ones (using DNA

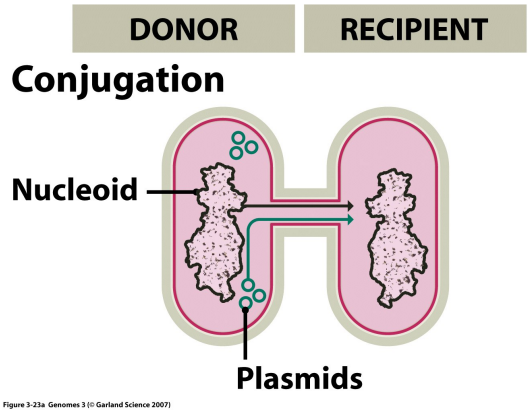


Figure 4.8: During bacterial conjugation, a donor cell transfers one of its plasmids (or its chromosome) to a recipient cell [15, figure 3.23(a)]

polymerase), and detach from each other. This has copied the plasmid from the donor to the recipient. This now enables the recipient to perform conjugation as well.

On its own, the F-plasmid would be little more than a virus: able to spread between bacteria and have them copy it, but not do anything more. However, when we consider that pieces of DNA can move between plasmids and the bacterium's chromosome, then the F-plasmid becomes an interesting mechanism. Figure 4.9 shows the different types of cell that can arise, along with their biological names.

- An F^- cell does not have the F-plasmid, and cannot be a conjugation donor. But it can receive conjugations from other cells.
- An F^+ cell has the F-plasmid, and so can donate this to other cells via conjugation. An F^+ cell conjugating with an F^- cell turns the F^- cell into an F^+.
- In an *Hfr* cell, the fertility genes have moved from the F-plasmid onto the chromosome. This can happen by random chance to any F^+ cell. If an *Hfr* cell performs conjugation, it will donate its whole chromosome to the recipient cell, since its fertility genes are on its chromosome rather than on a plasmid. Transfer of the whole *E.coli* chromosome in this way takes approximately 100 minutes, and if the transfer is interrupted at any point during this time then only part of the chromosome will be transferred.
- In an F' cell, the fertility genes have moved off the chromosome (of an *Hfr* cell) and back onto a plasmid, but they have taken part of the chromosome with them. An F' cell can conjugate with other cells to transfer its fertility plasmid, but this time some extra genes are sent as well.

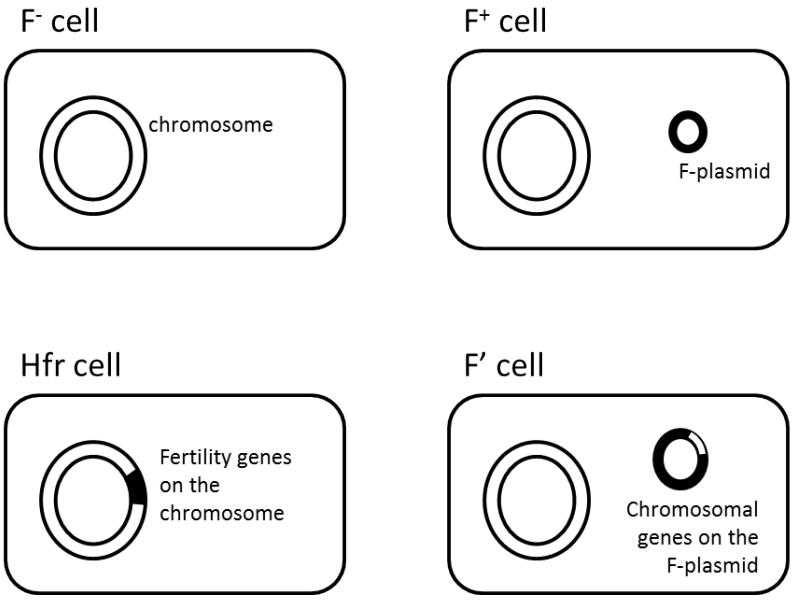


Figure 4.9: The different types of cell the can arise when fertility (conjugation) genes move between the F-plasmid and the chromosome.

Conjugation allows different bacterial cells to share genes. But more than this, it enables different *species* of bacteria to share genes, something that is not possible in higher organisms. Conjugation can even allow certain bacteria to transfer genes to certain plants [105]. And there is also the possibility of triparental mating, where where bacterium C helps bacterium A to conjugate with bacterium B.

The description of conjugation given here provides many computational opportunities, but is still a vast simplification of the underlying biology. For example, the process of moving genes between the chromosome and plasmids relies on the chromosome and plasmid sharing similar sequences of DNA. This means that different regions of the chromosome can evolve to be easier (or more difficult) for plasmids to remove. And different plasmids can evolve to integrate into different places on the chromosome. But even with this very simplified description of conjugation, we can see the benefits of embodiment for meta-evolution.

The meta-evolution implications of the conjugation mechanism are huge. Here is an evolutionary operator (conjugation), a piece of code that implements it (the F-plasmid) and a method for enabling other organisms to use the operator, after one organism has acquired it (after conjugation, both organisms have a copy of the F-plasmid). This operator is evolvable (the genes controlling conjugation can change) and can be used in other situations (the fertility genes can conjugate other plasmids and remove genes from the chromosome). And this is all possible

because the genomes of bacteria are embodied within DNA chemicals that exist in the physical world. Conjugation is one mechanism that implements horizontal gene transfer. The next section describes a different mechanism that implements the same phenomenon.

4.4 Bacteriophages

Bacteriophage is the name given to viruses that infect bacteria. Similarly to the conjugation mechanism by which plasmids transfer genes between bacterial cells, viruses can also transfer genes between bacterial cells, in a mechanism called *transduction*. This mechanism operates in two different ways, depending on whether the virus has a *lytic cycle* and a *lysogenic cycle*.

The lytic cycle is shown in figure 4.10. A virus attaches to a bacterial cell and injects its DNA into the cell. This viral DNA then hijacks the bacterial cell's transcription, translation and replication machinery, turning the cell into a virus factory. The proteins (and DNA, and sometimes RNA) produced by this factory are sub-components of the original virus particle. These sub-components self-assemble into full virus particles. When enough virus particles have been produced, the cell bursts from the pressure of the produced virus particles (and sometimes from a signal given by the virus). This scatters the virus particles into the environment (so they can go on to infect other bacterial cells and repeat the process).

The transduction mechanism begins when the virus particles are self-assembling. This assembly process packages the viral DNA into a protective protein coat (called a capsid). However, sometimes DNA from the bacterial cell can become packaged up along with the viral DNA. The second stage of transduction happens when this wrongly-packaged virus particle infects another bacterial cell. As it injects its viral DNA into the host cell, it also injects the bacterial DNA that was packaged up from the previous host. This embodies the same phenomenon as the conjugation mechanism: the transfer of bacterial DNA from one bacterium to another. (Note that viral infections do not always kill the host cell, so the bacterial cell can make use of the transferred DNA, without dying soon after from the viral infection.)

The lysogenic cycle is shown in figure 4.11. This is a slightly different mechanism that viruses can use to infect a host. When the viral DNA is injected into the host cell, rather than producing new viruses and destroying the cell immediately, the virus instead incorporates itself into the host DNA. This means that the host cell can go through many reproduction cycles as normal, each time copying the viral DNA along with its own. In this state, the virus is called a *retrovirus* or a *prophage*. At any time (possibly dependent on environmental conditions), the viral DNA can remove (excise) itself from the host DNA and switch to a lytic cycle.

With a lysogenic virus, transduction can happen by a slightly different mecha-

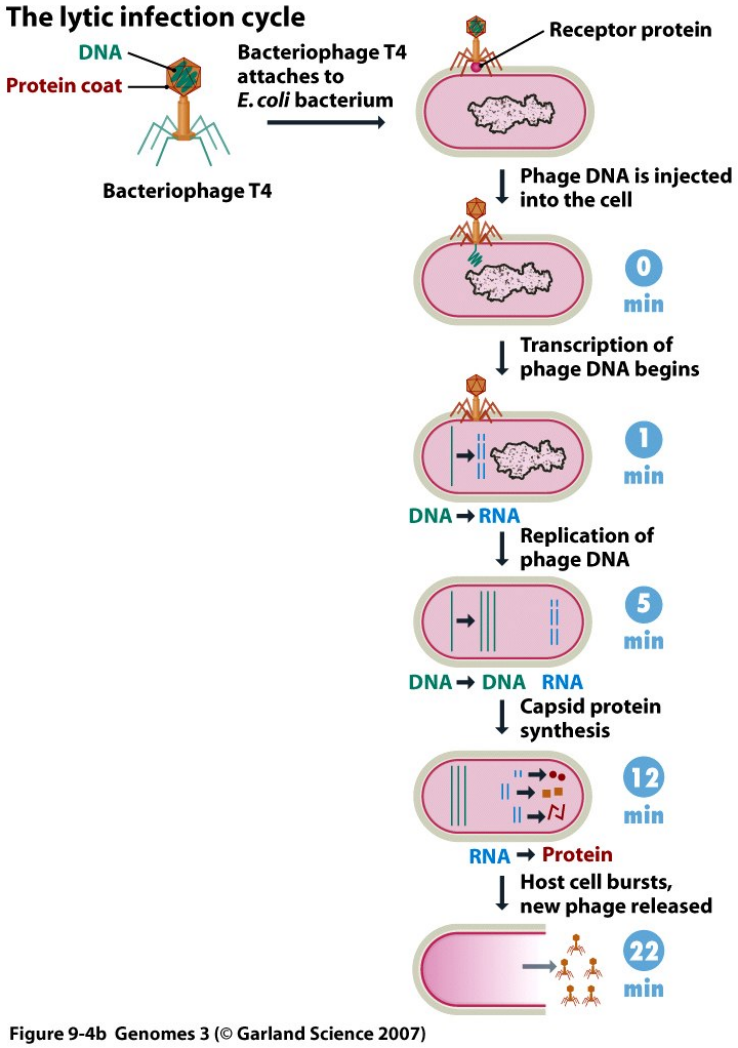


Figure 4.10: A virus (Bacteriophage T4) with a lytic cycle, infecting the bacterial cell *E.coli* [15, figure 9.4(b)].

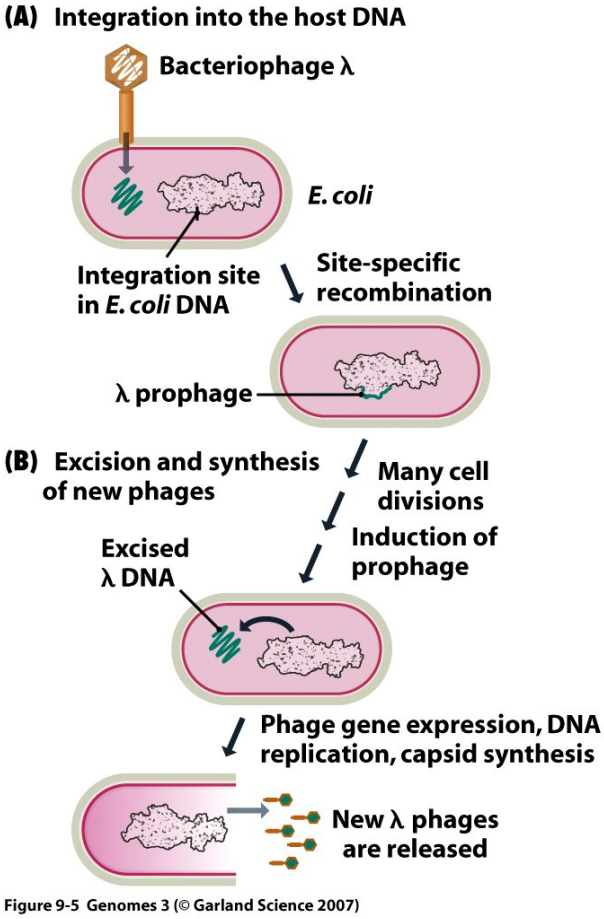


Figure 4.11: A virus (Bacteriophage λ) with a lysogenic cycle, infecting the bacterial cell *E.coli* [15, figure 9.5].

nism. When the virus DNA is removed from the host DNA, some host DNA might be removed along with the viral DNA, and be packaged up into newly-produced virus particles, to be transferred into the next infected bacterium.

This highlights the fact that biological systems are able to embody the same phenomenon via many different mechanisms. This is because they are embodied within a very rich world (the physical world). This degeneracy in the embodiment of biological phenomena provides biological phenomena with a form of robustness. If, for example, all the bacteria able to perform conjugation died out, then the phenomenon of horizontal gene transfer could still happen, via the mechanism of transduction rather than conjugation. And if all the lytic viruses died out, then the lysogenic ones would still be around.

4.4.1 Horizontal gene transfer, via a different mechanism

Plasmid conjugation and viral transduction are two mechanisms that embody the phenomenon of horizontal gene transfer. The computational difference between the two is that with conjugation, the process happens within a healthy bacterial cell. Thus the cell has some control over what is happening. But with transduction, the viral particles come from outside the bacterial cell. Since the rate and type of transduction depend on the precise DNA sequence of the virus, the bacterial cell has less control over what will happen. Also, when the bacterial DNA is packaged into an assembling virus, the cell has been taken over and turned into a virus factory. So the bacterial cell does not have the same control over what is happening within itself.

This is important for embodiment, because it shows that the two mechanisms are embodying the same process in very different ways. As well as using different parts of the world to embody the same phenomenon, they are using mechanisms that can be controlled in different ways (functions that have different parameters). As well as providing robustness against one type of part failing, this also provides a different type of robustness: giving different ways in which the same phenomenon can be controlled. Conjugation allows bacterial cells to control the phenomenon of horizontal gene transfer, whereas transduction allows the cell's external environment to control the same phenomenon (via the viruses it contains).

Biology does not work out the *best* way of embodying a phenomenon, and use only that. It develops different mechanisms by which the same phenomenon can be embodied using different parts and different methods of control. This degeneracy gives biology its robustness.

4.4.2 Mixing up levels

As explained previously (section 3.2.2), biological systems do not confine themselves to the crisp boxes we draw around them. But we need to draw boxes around parts to biology to understand anything.

4.4.2.1 Biology breaks the rules

As an example, biological organisms embody their genomes within DNA chemicals. Their DNA is transcribed into RNA and then translated into proteins. We can talk about the information flowing from DNA, through RNA, into proteins, and not in the opposite direction: the *central dogma of molecular biology* [20]. This crisp, logical description of biology helps us to understand the embodiment of genomes within DNA chemicals, and the cell cycles of biological organisms. But it is not correct. Or, rather, there are exceptions to it. The vast majority of biological

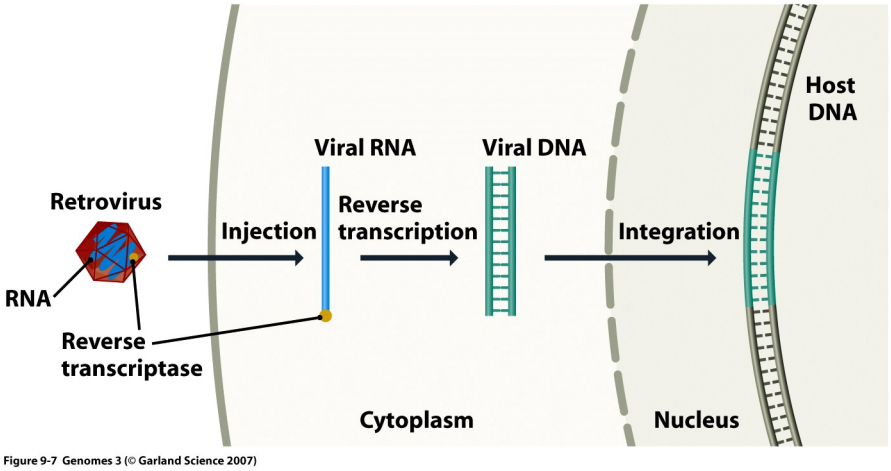


Figure 4.12: The steps by which an RNA retrovirus inserts its genetic material into the host's genome [15, figure 9.7].

organisms embody their genomes within DNA chemicals, and operate according to the above description, but there are a few exceptions that break the rules and mix up the crisp, logical levels.

Figure 4.12 shows an organism that breaks these rules. There exist *RNA viruses* that embody their genome within RNA chemicals rather than DNA. They can still integrate themselves into a host genome (made of DNA), by performing the reverse operation of transcription: turning their RNA into a strand of DNA that can be integrated into the host DNA. These viruses nicely close a loop because the mechanism performing the reverse transcription operation (a protein called *reverse transcriptase*) is encoded on the viral RNA itself. Thus the virus carries an encoding of the operator that integrates the virus into the host genome, but it needs the machinery of the host cell in order to decode (*forward* transcribe) and run (translate) this operator.

A further example of mixing up levels is that some huge viruses exist that contain more genes than some bacteria. These huge viruses can also be infected by smaller viruses [79].

4.4.2.2 Humans make the rules

The fact that biology escapes our boxes and mixes up levels is not a problem, but it is something we should be aware of. Humans need to chunk things into boxes and write down rules for what is in each box and how the boxes relate to each other. But it is important to remember that these boxes only exist within the minds of humans: they are not intrinsic to the physical world.

This is a particular issue when we make computational models of biology. Because of how computers have been built, computer programs are forced to be constructed out of rigid boxes[1]. So whenever we take a biological concept and put it into a computational box, we should be aware that are are losing some of the phenomena of the biological system. But if we *embody* a biological concept in a computer program, rather than hardcoding it, then we allow our computer program to *break the rules* as far as this concept is concerned. The design problem is working out which concepts to embody and which to hardcode.

We have said previously (section 3.1) that we do not want to copy the whole of biology inside a computer. We just want to capture some of the phenomena that biology displays, and so we need to model the biology at a certain level of abstraction. But we must make sure the level of abstraction is appropriate. If our level abstraction is too high then we lose the phenomena we want, because we have made rigid boxes (mechanisms) for them. We want our rigid boxes (mechanisms) to be below the level of the phenomena we are interested in, and have the phenomena emerge as a consequence of our mechanisms interacting. This is what *embodying a phenomenon* means.

4.5 Transposons

We have seen various ways in which bacteria can change their genome by acquiring genes from other bacteria. Transposons are a different take on changing genomes: they are a mechanism by which bacteria (and other organisms) can change their genomes, without requiring another organism to donate genes.

4.5.1 Example: colourful maize

Transposons are sections of DNA that can move from one position on an organism's genome to another position (possibly chosen at random). They are common to all organisms, not just bacteria, and are particularly prevalent in plants. For example, they can cause maize (sweetcorn) kernels to change colour (see figure 4.13) as they are developing, which is where transposons were originally discovered [68].

In maize, the colour of a kernel is determined by a chemical that moves through a pathway in which its colour is modified from light to dark (see figure 4.14(a)). It changes from white to yellow to orange to red to purple. Each step in this pathway is performed by a mechanism coded for in a gene on the DNA. If a transposon moves itself into the middle of one of these coding regions, it renders the gene useless and

[1] On a philosophical level, I do not believe this always has to be the case for computers in general (for example, because I believe that biological cells are computers), but for our current programming languages and models of computation, it definitely is the case.

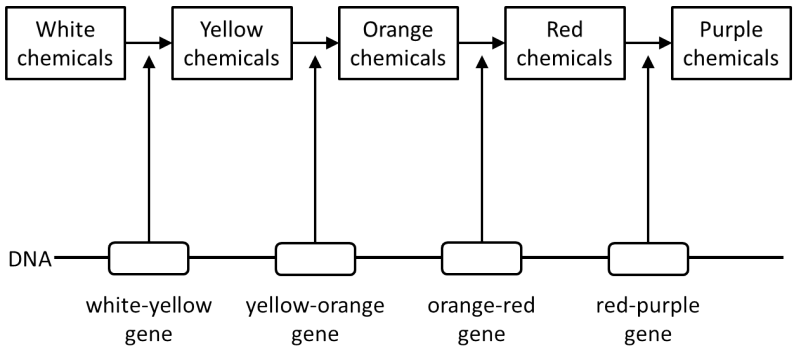
Figure 4.13: A colourful display of the extent to which transposons can affect developing maize kernels. Image from [13]

so the mechanism is not produced. This stops the pathway at this point, and determines the colour of the maize kernel, whether or not the mechanisms further down the pathway have been produced (as shown in figure 4.14(b)).

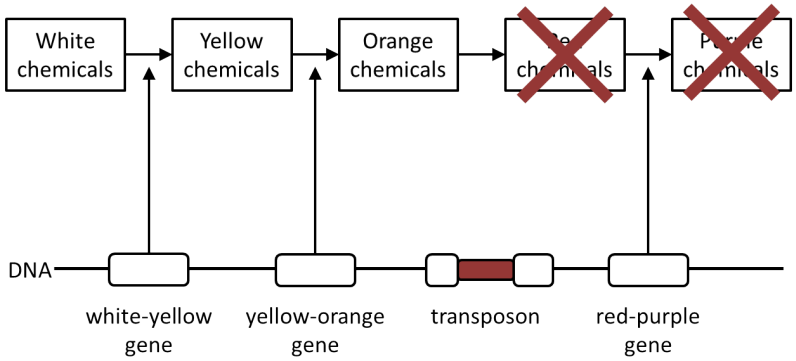
From an embodiment viewpoint, the power of transposons lies in their ability to change the genome of an organism as it is developing. Similarly to plasmids and viruses, transposons can change the genome of a living organism without having to wait for a child organism to be produced. But unlike plasmids and viruses, transposons can change the genome of an organism without the presence of another organism (or virus) donating some of its DNA. So they are a different mechanism that embodies the phenomenon of genome-change, with different requirements for how the mechanism can operate.

4.5.2 Types of transposon

There are two different ways in which a transposon can move from one place on the DNA to another: *conservatively* and *replicatively*, as shown in figure 4.15. Conservative transposition involves removing the transposon from its original place in the DNA and inserting it into a new place. In replicative transposition, the transposon remains in its original place and a copy is inserted into a new place. There are clear computational advantages to being able to both move and copy. Copying allows sections of the DNA to be duplicated, whereas moving allows sections of the DNA to be deleted (if the removal is successful but the insertion is not).



(a) Pathway determining the colour of maize kernels.



(b) The orange-red gene broken by a transposon.

Figure 4.14: If one of these genes is destroyed by a transposon, then all downstream reactions will not happen and the maize kernel will be a colour other than purple.

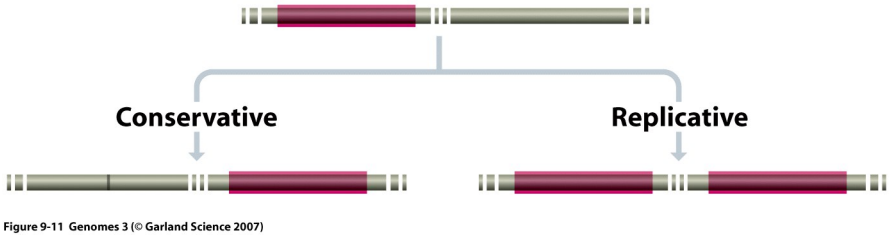


Figure 4.15: The difference between conservative and replicative transposons [15, figure 9.11].

Replicative transposons fall into two types: *DNA transposons* and *RNA transposons*. DNA transposons are transposed directly from DNA to DNA by a mechanism called a *transposase*. Some DNA transposons contain the gene that codes for their transposase, in which case the transposon can transpose itself whenever it feels like it. But some DNA transposons do not code for their transposase. These can only move if there is another transposon in the environment that does code for a transposase, since one transposase chemical can act on many transposons.

RNA transposons (or *retrotransposons*) are pieces of DNA that transpose via an intermediate RNA chemical, as shown in figure 4.16. Their DNA is transcribed (by the normal transcription mechanisms of the cell) into RNA. This RNA is then reverse-transcribed into DNA (not part of the normal mechanisms of the cell), before being integrated into the host DNA. Similarly to the transposase mechanism, the reverse-transcriptase mechanism can be coded for on the transposon or not. This reverse-transcription is the same mechanism by which retroviruses integrate themselves into host genomes (figure 4.12), and it has been shown that some RNA transposons are related to retroviruses. DNA transposons are common in bacteria, however, RNA transposons have so far only been discovered in higher organisms. But since we do not have to be bound by the specific implementation details of biology (or by what biologists have so far discovered), there is no reason why we cannot implement analogues of RNA transposons in our analogues of bacterial cells.

One use of transposons in bacterial cells is to move segments of DNA between plasmids and the main chromosome. In the same way that plasmids often carry non-essential but useful genes, such as genes for antibiotic resistance, transposons can similarly carry such genes. Also, some transposons carry genes that code for viruses, and are termed *transposable phages*. These viruses use transposition as a means to replicate themselves.

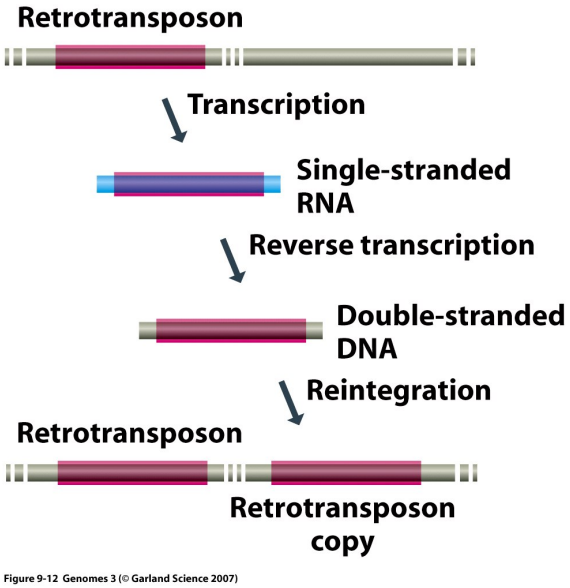


Figure 4.16: Details of how retrotransposons replicate [15, figure 9.12]. This is the same process by which retroviruses integrate themselves into host genomes (figure 4.12).

4.5.3 Common problems

We have seen that different mechanisms, performing different tasks, can face the same kinds of problems. In this case, we have a piece of DNA that is to be copied. The problem is whether or not this DNA should encode the mechanism that performs its copying:

- Some plasmids encode the genes necessary to make their host bacteria perform conjugation, whereas some plasmids need others to conjugate them.
- Some transposons encode their own transposase (and if necessary, reverse transcriptase), whereas some transposons need others to transpose them.
- All viruses require a host cell to replicate their DNA (or RNA), but RNA viruses encode their own reverse transcriptase, since the host cell does not have this.

In terms of embodied meta-evolution, we can think about programming mechanisms that contain all the necessary functionality to close their loops and sustain themselves. In terms of copying, this means we do not want to implement a copying function that sits in an external world and copies organisms. We want to implement a copying mechanism that exists in the same world as the organisms it

is copying. This copying mechanism must be able to copy organisms and also copy itself, using the same mechanism.

Even if a mechanism can close its loop and contain all the machinery necessary to sustain itself, it does not contain within itself all the *information* necessary to sustain itself. Every mechanism exists within a *world*, that provides an environment in which the mechanism can operate. The next section describes a feature of the biological world used by the three mechanisms described in this section (plasmids, viruses and transposons).

4.6 Recombination

Recombination is a low-level mechanism by which two double-strands of DNA can interact. It can be thought of as a feature of the world that allows the mechanisms of plasmids, viruses and transposons to operate. In computational terms, this means that if we wanted to embody these mechanisms within a computer program, then DNA chemicals and recombination is what we would hardcode using a crisp computer programming language. Given an implementation of DNA chemicals and recombination, the mechanisms of plasmids, viruses and transposons would be relatively straightforward to build.

4.6.1 Recombination is not crossover

Recombination is not the *crossover* operator used in genetic algorithms. The crossover operator was inspired by a process that occurs during cell meiosis (a type of cell division that produces sex cells). This was also the original biological meaning of the term *recombination*, however, during the 1960s biologists investigating it discovered that a variety of different processes occur within cells as a result of the same underlying mechanism. Thus the biological term *recombination* refined in meaning, and now refers to a general mechanism by which two double-strands of DNA come together and are broken and re-joined in such a way as to change the makeup and/or topology of the DNA strands. *Crossover* is therefore a special case of recombination, that occurs during cell meiosis. The general process of recombination happens much more frequently within cells. It is the process by which transposons and plasmids move onto and off the chromosome, and the process by which viral DNA integrates itself into the host chromosome.

There are three main types of recombination: homologous recombination, site-specific recombination and transposition.

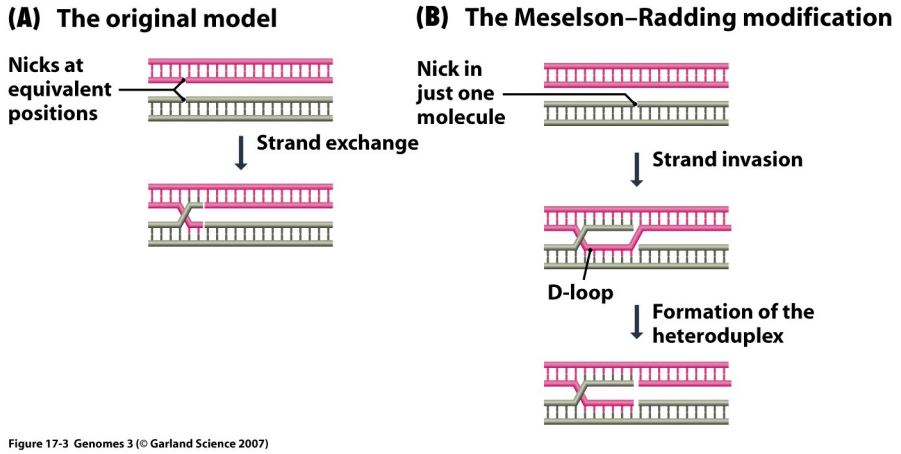


Figure 4.17: Showing how homologous recombination starts, by small breaks in either or both DNA molecules [15, figure 17.3]. (A) shows Holliday's original model. (B) shows a modification to this model that removes the need for both strands to be broken simultaneously. There is a further model (the double-strand break model) that explains a further problem that both (A) and (B) share.

4.6.2 Homologous recombination

Homologous recombination is a generalisation of the crossover operator from genetic algorithms. It is a mechanism in which two strands of DNA that are homologous (meaning *similar*[2]) come together. Because the two strands are homologous, they line up with each other (due to the electrostatic forces between the chemicals). Now, small breaks in either (or both) strands can cause the two strands to pair up wrongly (see figure 4.17), causing two strands to become crossed over. Resolution of this cross-over (by cutting the DNA) results in two strands of DNA that have exchanged some bases. There are different biological models of precisely how this mechanism operates, and figure 4.17 shows two of these models. Since we are not trying to emulate biology, but rather emulate its computational properties, we are free to choose which model of recombination we want to implement, to obtain the computational properties we want. However, if we cannot get the properties that we want from the model that we have chosen, then one place to look would be towards more biologically accurate models.

When homologous recombination happens between two different molecules of DNA, it can reproduce the one-point and two-point crossover operators from genetic algorithms, as shown in figure 4.18. This figure also shows that a single

[2] Technically, the word *homologous* means that the two objects being compared are similar because they share an ancestry, either in an evolutionary or a developmental sense. But for the purposes of this discussion, it is only important that the two strands of DNA are similar.

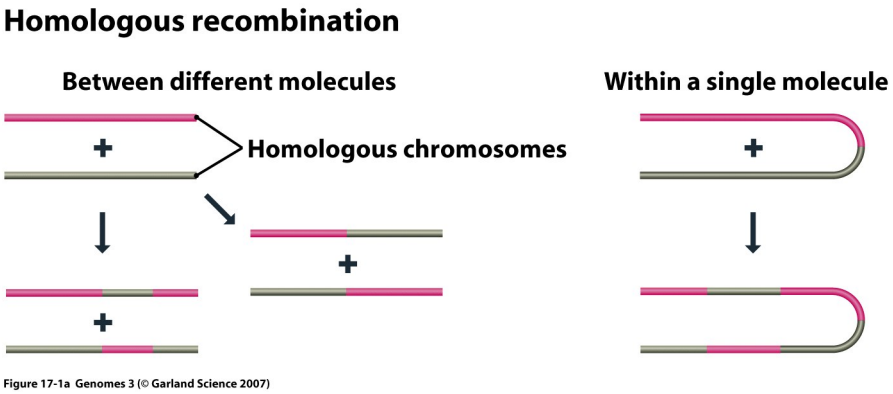


Figure 4.18: Homologous recombination can implement the genetic-algorithm one-point and two-point crossover operators, and also an operator not found in genetic algorithms [15, figure 17.1(a)].

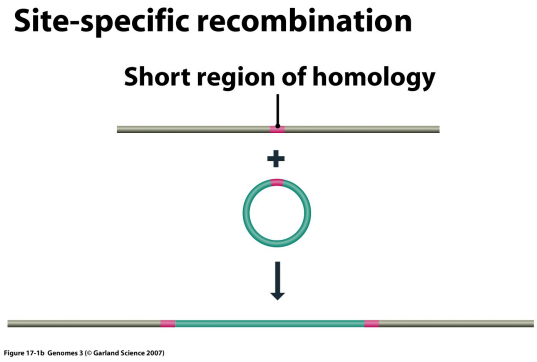


Figure 4.19: Site-specific recombination can integrate a plasmid into another piece of DNA (for example, the main chromosome) [15, figure 17.1(b)].

DNA molecule can undergo crossover with itself (if it possesses two regions of homology). This shows the novelty-generation capabilities of a general (embodied) model of recombination over a simple model of crossover. The idea of a strand of DNA performing crossover on itself would be very hard to imagine if the crossover mechanism was a hardcoded function taking two DNA strands as inputs. But if the crossover mechanism was a special case of a general recombination process that happens between homologous regions of DNA, then crossover within a single strand is easy to imagine.

4.6.3 Site-specific recombination

Site-specific recombination is the main way in which plasmids integrate into the main chromosome of their host bacterium (or more generally, integrate into any strand of DNA). As can be seen in figure 4.19, both the plasmid and the strand of DNA share a short region of homology. Essentially a one-point crossover happens within this region of homology but because the plasmid is circular, only one resulting strand of DNA is produced, rather than two. The has the effect of inserting the plasmid into the strand of DNA. Note that there is nothing stopping the strand of DNA itself being a plasmid, or any type of DNA chemical.

Once the plasmid has been integrated into the strand of DNA, the two regions of homology are still present (see figure 4.19). They flank the plasmid, providing information as to which part of the DNA came from the plasmid. Some retroviruses use this mechanism to insert their (circular) DNA into a host genome and remove it when they choose. The recombination process happens all the time, between homologous regions of DNA, but viruses code for mechanisms that speed up the process, giving them control over the process and making it more reliable. These viruses did not need to invent a whole new mechanism for integrating themselves into host genomes. The used the existing recombination mechanism, inventing machines to speed up the mechanism and invoke it when they choose.

4.6.4 Transposition

DNA transposons use recombination to implement their transposition mechanism. As explained above, DNA transposons can transpose replicatively (where the transposon is copied) or conservatively (where the transposon is moved). Figure 4.20 shows how these two types of transposition are implemented in terms of recombination.

4.6.4.1 DNA transposition mechanism

The transposase enzyme makes a series of cuts in the DNA, both around the transposon and at the site where the transposon will be moved to. It then joins cuts made at the target site to cuts made at the transposon site, to create a hybrid molecule in which the two DNA strands are joined together. This joining can be resolved in two different ways (replicatively or conservatively).

The structures at either end of the transposon in the hybrid molecule look very much like the structures that occur during DNA replication. This can cause a DNA replicase molecule to come in and replicate the transposon. After replication, the molecule is still a hybrid of both DNA strands, but it now contains a region of homology (namely, the transposon that has just been replicated). Homologous re-

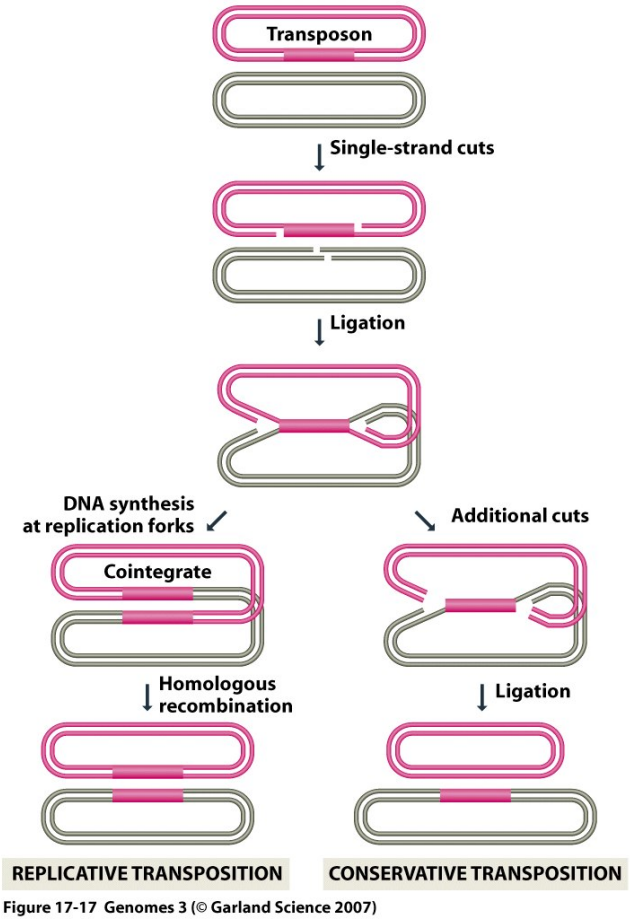


Figure 4.20: DNA transposition is implemented in terms of DNA cutting, ligation (joining), replication and recombination [15, figure 17.17].

combination now happens between the two copies of the transposon, to disentangle the two strands of DNA and complete the replicative transposition.

Alternatively, if the replicase molecule does not come in to replicate the transposon, then additional cuts can be made to disentangle the two strands of DNA. In this case the transposon does not replicate: it simply moves to the other strand.

4.6.4.2 Evolving transposition

DNA transposition is a marvelous example of biology creating mechanisms by reusing existing mechanisms and general properties of the world (lower-level mechanisms). DNA transposition requires a specialised machine, the transposase enzyme, to make cuts in the right places on the DNA. But this machine does not implement the complete transposition process. Lower-level mechanisms are needed: DNA

chemicals that can be cut; DNA sticking together at points that have been cut (ligation); and the mechanism of homologous recombination. Also, the mechanism for replicating DNA can be co-opted to help in replicative transposition. But it does not need to be co-opted, in which case transposition happens conservatively.

The evolutionary operator of transposition could not have evolved without these existing mechanisms being around and available for use in new situations. In an evolutionary algorithm in which all the evolutionary operators are hardcoded, the whole process of transposition would need to be hardcoded. To evolve the whole process of transposition from nothing would be a daunting task. However, in an embodied algorithm, the ideas of cutting, ligation, recombination and DNA replication are available as mechanisms in a common world. This makes the task of evolving transposition seem a lot easier.

4.7 Summary

Looking at some examples of how biological systems copy themselves, we have identified three different but related mechanisms: plasmid conjugation, viral replication, and transposition. These three mechanisms can each encode all the machinery necessary to close the loop and copy themselves within the physical world. But as well as sustaining themselves, their copying also confers evolutionary advantages to the bacteria they exploit. We have seen how the many advantages of embodiment can be gained from a relatively simple world (containing not much more than DNA chemicals and recombination).

Our task now is to take inspiration from these biological systems and use this to design computational worlds that are sufficiently rich. We do not need to implement the precise details of each biological mechanism in computer code. We need to implement the important properties of the physical world in our artificial worlds. Using a suitable artificial world, we can build mechanisms that embody interesting biological processes.

Chapter 5

Artificial embodiment

We have a general theory of embodiment (chapter 3) and some inspiration from how biology uses embodiment (chapter 4). We now use this to think about designing computer programs that are embodied.

When designing embodied computer programs, we must create the world and the mechanisms that exist within it. For this we need three nested programming languages, as described in section 3.3.1. We use a traditional programming language to implement our virtual world. The virtual world defines the second programming language, in which we can implement mechanisms that embody phenomena of interest (the third programming language). *Artificial chemistries* provide a good metaphor for the mechanism and world languages.

This chapter describes how we can think of the specification of an artificial chemistry as the world language, and the artificial chemicals themselves as the mechanism language. Artificial chemistries can serve as a metaphor for a wide range of (non-chemical) systems, and so using them as an example of world and mechanism language encompasses a wide range of novelty-generation algorithms. This chapter ends by describing how we can embody the phenomenon of *copying a genome* within a new artificial chemistry, *GraphMol*, designed to investigate embodiment.

5.1 Artificial Chemistries for embodiment

As an example of artificial chemistries being a metaphor for many different novelty-generation systems, consider the artificial life systems Tierra and Avida, described in section 2.4.2.1. These systems are inspired by ecology, with their agents (computer programs) being analogies of biological organisms competing for resources (processor time and memory). There is an obvious analogy here with (for example) thinking about the agents as grazing animals such as zebra or geese. But the

analogy with chemicals works just as well.

Throughout this section, it should be remembered that the *chemistry* metaphor is being used as an abstract concept to describe a broad class of systems. A lot of different systems can be thought of as artificial chemistries, even though they are not trying to model physical-world chemistry. And we are not trying to focus on physical-world chemistry: we are focusing on novelty-generation. There is nothing chemistry-specific about the abstract notion of an artificial chemistry, just as there is nothing tape-specific about the abstract notion of a Turing machine [99].

5.1.1 A physical-world model of artificial chemistries

Section 2.4.2.2 introduces artificial chemistries as they can be found in the literature. As with evolutionary algorithms, the people designing these systems often do not state the physical-world models that inspire their computational algorithms. Section 2.4.2.2 gives the historical definition of artificial chemistries (from [27]), which describes them in terms of three components: a set of *chemicals*, a set of *reactions* and a *mixer* algorithm that decides which chemicals react together when. This definition is useful for analysing some systems, but is not quite what we want for thinking about embodiment.

I define artificial chemistries in terms of:

1. A *world* (physics engine) containing primitive atoms and properties
2. A *chemistry* describing how atoms can connect together and how these connections can be modified.
3. A set of *constraints* (physical laws) that restrict how chemicals are allowed to act in the world (part of the world).

And we can refine the constraints, by thinking about:

4. An *energy model* that describes how likely different events are to happen (one of the constraints).
5. A *binding model* that describes which chemicals can interact with which others (part of the energy model).

This definition is more suited to looking at *open-ended artificial chemistries*, i.e. implicitly-defined artificial chemistries, built to cope with the introduction of novel chemicals. This is a requirement for artificial chemistries designed for novelty-generation. As stated earlier, the purpose of this work is not to exhaustively review every different artificial chemistry. So I will not specify in detail how every existing open-ended artificial chemistry fits into this definition.

This definition is a description of artificial chemistries in terms of concepts that exist in the physical world (atoms and conservation laws), rather than abstract computational concepts (sets and functions). This is useful, because we can use this physical-world model to compare artificial chemistries with real chemistry. For evolutionary algorithms, we have seen how enriching the physical-world model creates new algorithms with more novelty-generation capabilities. Hopefully, the same will be true for artificial chemistries: we will be able to make better novelty-generating artificial chemistries by enriching their physical-world models.

5.1.1.1 World (physics engine)

The *world* is the environment in which chemicals exist. For example, it could be a well-mixed aspatial bucket, a two-dimensional space mixed by arbitrary vector fields [59], or a cellular automaton.

The world contains all the primitive units (atoms) of the system, that cannot be broken down into smaller components. So if the chemicals are strings over an alphabet, then the world contains the letters of the alphabet. If the chemicals are random boolean networks [32], then the world contains the nodes of these networks.

The world also provides a set of properties, such as the distance between two chemicals, the neighbours of a chemical, mixing of chemicals, or stochasticity. These are the properties that our embodied mechanisms want to gain 'for free' from their environment.

5.1.1.2 Chemistry

The *chemistry* specifies how the units of the world can interact with each other. The world provides atoms and determines when atoms are close together, but the chemistry says which atoms can interact with each other in which ways. If the atoms are letters of the alphabet, then the chemistry says how these join together into strings. It says what the different letters mean, and specifies how two strings react with each other.

The chemistry specifies how the units and properties of the world can be used to perform basic computations (chemical reactions). By combining together multiple reactions and executing many reactions in parallel, we can build complex computer programs out of simple reactions.

5.1.1.3 Constraints

In order to perform computation, systems need to be *constrained* in some way. In the physical world, this corresponds to the conservation of physical quantities (mass, momentum, energy, ...). In our virtual worlds, we must choose which constraints to place on our systems to achieve the properties we are looking for. There

has not been much work done on this for artificial chemistries, but I believe that choosing the correct constraints will prove critical in designing artificial chemistries that evolve in interesting and useful ways.

Two constraints we have identified are *conservation of mass* and *conservation of energy*. It is not known what effect conservation of mass might have on a system. We simply note that the vast majority of artificial chemistries do not conserve mass in their reactions. Conservation of energy is something that has been looked at, in a limited way. In virtual systems, there are different ways to define *energy*, on different levels of abstraction. So rather than asking if a system conserves energy or not, a more incisive question is to ask what *model of energy* the system uses.

5.1.1.4 Energy model

In physical-world chemistry, energy plays a huge role in chemical reactions. The amount of energy within a system changes which reactions are possible, and the rates at which different reactions happen. Chemists talk about reactions in terms of *energy barriers*, meaning the amount of energy that must be added to the products to allow them to react.

Current artificial chemistries are not nearly this sophisticated. The closest thing to an energy model I have found (and it is not very close!) is the constraint placed on most aspatial worlds conserving the number of chemicals present in the world. Many aspatial worlds force a fixed number of chemicals to exist. Reactions that destroy existing chemicals are not possible, and if a reaction creates a new chemical, then this replaces one of the existing chemicals in the world. This is a purely computational conservation law, and does not correspond to any mechanisms from physical-world chemistry.

The Stringmol artificial chemistry [42] addresses this problem of a fixed number of chemicals in the same way the RLAIS artificial immune system addresses the problem of a fixed population size (section 2.6.2.3). Stringmol phrases this problem in terms of an energy model. Rather than having a fixed number of chemicals, Stringmol has a fixed *energy flux*. The world contains a number of *energy units* that build up over time because of the energy flux, and are consumed by chemicals reacting. Also, chemicals randomly decay over time (and so are removed from the system). Reactions can create new chemicals, and the number of chemicals in the world can increase. But the decay process, and the fact that reactions consume energy, balances the creation of chemicals, leading to stable numbers of chemicals being observed in the world (with stochastic noise).

Stringmol's energy model is clearly very simple compared to physical-world chemistry, since it treats energy as discrete tokens that are used up when reactions happen. This very high-level energy model embodies *the number of chemicals* within the world and provides a selection pressure for chemicals to replicate. I

believe that different energy models could embody different phenomena within artificial chemistries and provide different selection pressures. More sophisticated energy models would therefore give algorithm designers more control over how artificial chemistries evolve and behave.

5.1.1.5 Binding model

Binding is a constraint on which chemicals are allowed to react with which other chemicals. In physical-world chemistry, different chemicals have different shapes and different patterns of charge on their surfaces. Two chemicals can react if their shapes and patterns of charge are sufficiently complementary. This can all be understood in terms of energy, and so for physical-world chemistry, binding is a part of the energy model.

In artificial systems with impoverished energy models (or very high-level ones), binding can be hardcoded as a property of the world, rather than being embodied within the energy model (as it is in the physical world). A hardcoded *binding function* takes two chemicals as input and decides whether or not they can bind (and possibly, whereabouts on their surfaces they bind). In artificial systems, this function can be defined in any way.

Many artificial chemistries use a trivial binding function, allowing any two chemicals to bind. Some artificial chemistries run the reaction to see what the product would be, and decide based on this whether to allow the bind or not (e.g. AlChemy, section 2.4.2.2). A more biological way (that some artificial chemistries use) is to analyse the structure of the chemicals (without running their reaction) to see if they are *complementary*, for some definition of complementary (e.g. binary string matching (possibly with wildcards [47, ch.8]), r-contiguous bits, Smith-Waterman subsequence alignment [41], properties of random boolean networks [31]).

I believe that having a rich binding model is important for artificial chemistries. This means not letting every chemical bind to every other, and also having different chemicals with different promiscuity of binding. We have developed an algorithm for quantifying the richness of a binding model [17]. We have also shown that, for an artificial chemistry with a rich binding model and interesting properties (Stringmol), making the binding model trivial removes many of its interesting properties.

5.1.2 A metaphor for embodiment

The definition of artificial chemistries above allows us to see how artificial chemistries can be used to produce embodied computer programs. They provide us with the nested programming languages introduced in section 3.3.1.

- The *implementation language* is the programming language we use to implement the artificial chemistry so it can run on existing hardware. It is an arbitrary choice that does not concern us here.

- The *world language* is the specification of the artificial chemistry. It is the collection of all the components of the world, together with the behaviours of the chemistry and their constraints. It is a program written in the implementation language.

- The *mechanism language* is the collection of chemicals and reactions we can build within the artificial chemistry world. It is a program written in the world language. The richer the world is, the more complicated and interesting mechanisms we can build within it.

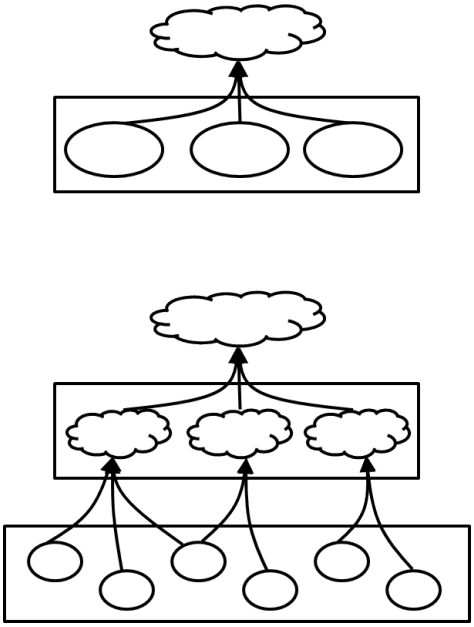
- The *phenomena* are properties we observe of the running artificial chemistry. These can be thought of as programs written in the mechanism language.

When the artificial chemistry runs, code from the implementation language executes to create the components of the world language. These interact with each other according to the mechanism language, which produces the phenomena as emergent properties.

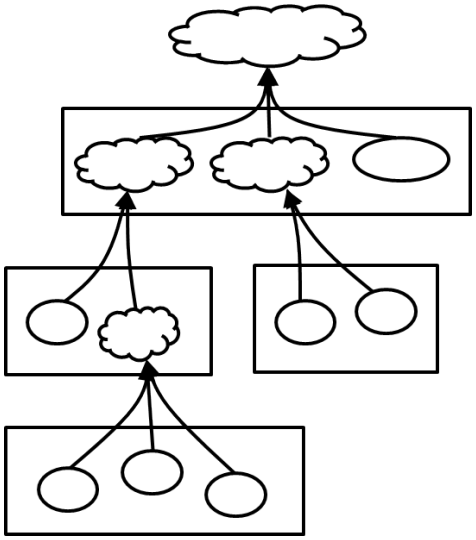
In our case, the world language corresponds to physics and chemistry, the mechanism language corresponds to biology and the phenomena are evolution and novelty-generation. These terms make sense in our context, because our focus is on mechanisms at the level of biology. If our mechanisms were at the level of fundamental physics or ant colonies, then these words would lose their metaphors. So this is not an absolute definition of what *physics*, *chemistry* and *biology* means to any embodied computer program: just to ones that focus on our level of abstraction.

It would be possible to only consider the world language, and build everything up from there. But this would be like trying to understand physical-world biology in terms of physics. In programming terms, this would be a nightmare to design and would be immensely slow to run. In practice, the chemistry defines the level of abstraction on which we are working, because it specifies the types of *behaviour* that are hardcoded into the system. By combining these world-level behaviours using the mechanism language, we create mechanism-level behaviours that are evolvable.

The benefit we can expect to gain from embodiment is limited by the richness of the world language (physics and chemistry). As we have seen from biology (chapter 4), the benefit of embodiment comes from mechanisms being able to exploit the world (and exploit existing mechanisms, that are implemented within the world). The ways in which our mechanisms are able to exploit their virtual world are limited by the richness of this world and the flexibility of the chemistry used to build



(a) Two different chemistries embodying the same phenomenon using mechanisms on different levels of abstraction.



(b) A single chemistry embodying a phenomenon using mechanisms on different levels of abstraction.

Figure 5.1: The two separate senses in which artificial chemistries can operate on different levels. Clouds represent embodied phenomena, ovals represent directly-implemented mechanisms, and boxes represent different worlds.

mechanisms. So to produce a system that benefits well from its embodiment, we need a rich physics/chemistry world in which we can build evolvable mechanisms.

5.1.3 Artificial chemistries on different levels

Embodiment is a hierarchical phenomenon (section 3.2.3) with systems being viewable on different levels, depending on which phenomena are of interest. Similarly, artificial chemistries can exist on different levels, allowing phenomena to be embodied to different levels of detail.

There are two separate ways in which artificial chemistries can be said to operate on different levels, as shown in figure 5.1:

1. One chemistry can embody a phenomenon using high-level mechanisms, whereas another chemistry can embody the same phenomenon using low-level mechanisms (figure 5.1(a)).

2. A single chemistry can embody a phenomenon using mechanisms at different levels of abstraction (figure 5.1(b)).

The best novelty-generation algorithm is a chemistry that embodies a high-level phenomenon using low-level mechanisms and many different levels of embodiment (figure 5.1(a), lower image). But the problem with these algorithms is that they take a long time to run. So in practice, it may be better to adopt the second approach (figure 5.1(b)), embodying the most crucial mechanisms on a low level, and implementing the others on higher levels. These types of system can always be extended to embody more of the higher-level mechanisms within the lower-level world(s) of the chemistry.

The highest level on which an artificial chemistry can be implemented is the symbolic level (section 2.4.2.2). This is not a suitable level on which to build novelty-generation algorithms, because every behaviour must be explicitly specified before running the algorithm. But it is still a useful level to consider. Whatever level(s) an artificial chemistry is implemented on, it is always possible to run the chemistry for a time and generate a symbolic description of that run. If the purpose of the chemistry is to generate novelty, then (hopefully) this symbolic description will not capture everything the chemistry is capable of. It will only capture a description of the events that happened during the run. Symbolic chemistries can be used during the process of designing an artificial chemistry, to describe scenarios the chemistry should be able to reproduce. For example, if chemicals in the chemistry are intended to replicate, then the following symbolic reactions can be considered, to illustrate different types of replication:

$$A \mapsto A + A \tag{5.1}$$

$$A + B \mapsto A + B + A \tag{5.2}$$

$$A + m \mapsto A + A \tag{5.3}$$

$$A + B + m \mapsto A + B + A \tag{5.4}$$

Equation 5.1 represents a chemical replicating itself, while equation 5.2 represents a copying machine, B, replicating a template, A. Equations 5.3 and 5.4 represent these two reactions with conservation of mass taken into account.

Symbolic chemistries can also be used to understand an artificial chemistry. For example, if an artificial chemistry was run and analysed symbolically, and one of the above equations (5.1—5.4) was contained within the symbolic description, then it could be concluded that the chemistry was doing replication.

There are some situations that involve crossing levels. Physical-world chemistry has gone through level-crossing events at different times during the evolution of life (the *major transitions in evolution* [67]), for example: naked replicating molecules becoming encased in compartments and replicating as populations; RNA acting as both genes and enzymes, changing to use DNA as genes and proteins as enzymes; and the evolution of multi-cellular organisms from single-celled organisms. These kinds of problem are interesting to systems biologists wanting to better understand

Algorithm 1 Deconstructing the string copy operation, as a prerequisite to embodying it.

```
result string_A := string_B

i := start(string_B)
while i not at-end(string_B) do
  string_A(i) := char-copy(string_B(i))
  i := next(i)
end while
```

what happened during the history of the physical world. They are also interesting for novelty-generation, as they are examples of natural systems increasing their own complexity, something that current artificial systems find difficult to achieve.

For more discussion about artificial chemistries at different levels, see [44]. The following section focuses on embodying a specific phenomenon (namely, *copying*) using mechanisms at different levels.

5.2 Embodying the copying process

We have introduced the theoretical concept of embodiment, seen how biological systems benefit from embodying their phenomena, and found a computational medium (artificial chemistries) for embodied computer programs. Our overall aim is to embody evolution within a computer program, to create a meta-evolutionary algorithm. Clearly, there is a lot to evolution in the physical world. It would take a lot of work to embody all of evolution's mechanisms within a computer program. I did not have time within one PhD to look at the whole of evolution. Therefore I have chosen one aspect of evolution: *copying*, to investigate in detail and use as a case-study for exploring the ideas of artificial embodiment.

Copying is very important to evolution because it is a major source of mutations, which are the novelty that drives evolution. The copying and mutation processes in most novelty-generation algorithms are stochastic, rather than embodied. This means that, although they can introduce novelty into the structures they copy, they cannot introduce novelty into the *process of copying*. Embodying the copying process is a way of allowing the copying process to change, and allowing different types of copying process to emerge.

5.2.1 Deconstructing the copy operation

In writing an embodied copying program, we must think about the *process* of copying a string, rather than the *result* of the copy. Thinking about the result

describes copying as a *phenomenon*, whereas we want to think about to process of copying, in order to design *mechanisms* that can implement this process. To say this in computer science jargon, a phenomenon is a *declarative* description of a process, whereas its mechanisms are an *operational* description.

Algorithm 1 breaks down the copying process into four parts, each involving a particular function: `start`, `at-end`, `char-copy`, and `next`. There are different ways to break down any process; the first design decision is how to break down the phenomena of interest into lower-level functions. Each of these four functions can be either *crisp*, *stochastic* or *embodied*.

5.2.1.1 Crisp

Making a function crisp means writing it in the implementation language, so the function appears as a primitive in the world language. The system's mechanisms can use this function but it will always work perfectly and cannot be changed in any way. If all four functions are crisp, then the overall copying process is crisp, and exact copies are always produced.

5.2.1.2 Stochastic

Making a function stochastic means writing it in the implementation language, but incorporating a (pseudo-)random number generator. This makes the function appear as a primitive in the world language, but it does not always give the same result (because of the random numbers). Although the function can now introduce variation into the system, this variation cannot be controlled by the system's mechanisms (without creating a tower of higher-level worlds).

If any of the four functions are stochastic, then the overall copying process will be stochastic. Making different combinations of these four functions stochastic introduces different kinds of variation into the copying process. For example:

- making `char-copy` stochastic could cause some characters to be copied incorrectly;
- making `at-end` stochastic could cause the copy to be truncated;
- making `start` stochastic could miss an initial substring off the copy;
- making `next` stochastic could cause some characters/substrings to be missed out of the copy, or repeated.

Different stochastic operations could be created by making different combinations

of the four functions stochastic or crisp, leading to 15 different types[1] of stochastic copy operation.

5.2.1.3 Embodied

Making a function embodied means writing it in the world language: combining primitives from the world language to build a mechanism. This makes the function appear in the mechanism language. If the function uses stochastic primitives from the world language, then it appears as a mechanism that can introduce variation into the system.

Rather than being a black box that cannot be changed, the function is a mechanism composed of primitives from the world language. This means that other mechanisms can interact with the function, changing how it operates and changing how it introduces variation into the system.

We can embody the copying process in different ways, and to different degrees, by embodying different combinations of the four functions. We do not need to embody all four of the functions: we can implement some of them as crisp or stochastic functions, as long as at least one of them is embodied. This leads to 65 different combinations[2]. But there are also multiple different worlds in which each of the functions could be embodied. So there are many different ways in which we can embody the copy operation. Each of these leads to different systems with different properties and different degrees of novelty generation. The following section describes one of these ways from the literature, as an example. The next chapter describes my implementation of a different way.

5.2.2 Example: Stringmol's copying chemical

The Stringmol artificial chemistry [41, 42, 43] has been used to implement an embodied copy operation. In terms of algorithm 1, it has embodied `start` and `at-end` functions, a stochastic `char-copy` function and a crisp `next` function. Figure 5.2 shows the structure of Stringmol's embodiment in detail.

Stringmol's copy operation is embodied within a *replicase* chemical that can be built as a string of characters within the Stringmol world. Two replicase chemicals can bind with each other, and undergo a reaction. The reaction takes multiple steps, and involves one of the replicase chemicals acting as a mechanism that copies the other replicase string, character-by-character (as biological DNA is copied: see section 4.2.1).

[1] Four different functions, with two choices each (stochastic or crisp), minus the single case where all the functions are crisp. $2^4 - 1 = 15$

[2] Four different functions, with three choices each. Then remove the 15 stochastic combinations, and the single crisp one. $3^4 - 15 - 1 = 81 - 16 = 65$

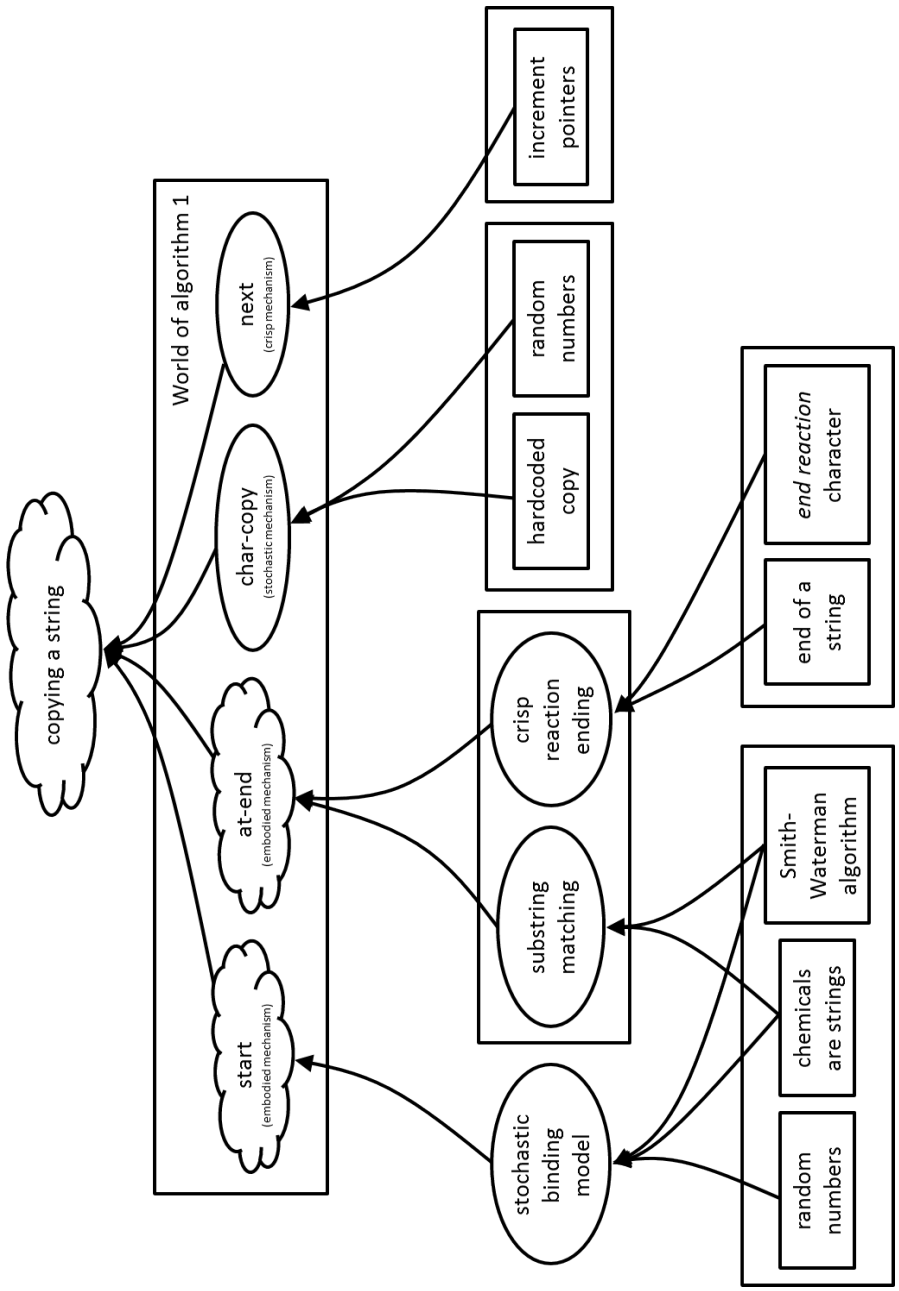


Figure 5.2: The mechanisms and worlds used by the Stringmol artificial chemistry to embody the phenomenon of copying a string.

Stringmol's `start` function is embodied within the binding model of Stringmol. Rather than always starting a copy at the beginning of the template string, the binding model uses a modified version of the *Smith-Waterman algorithm* [91] to find a pair of binding regions on the two chemicals. If two regions can be found that are approximately complementary, then the two chemicals can bind and react, with the copy beginning from the start of the binding region, rather than from the start of the string. If a string is to be copied from its start, then it must have a suitable binding region at its start, otherwise any copies will miss off the start of the string. This allows evolving Stringmol chemicals to alter where copies start from, and exploit the embodied `start` function.

The Smith-Waterman algorithm was developed in Biology to compare the genomes of organisms. It finds the longest common subsequence between two genomes (strings over the alphabet {A, C, G, T}), taking into account small insertions, deletions and point-mutations. Given two strings (e.g. genomes from completely different organisms), the Smith-Waterman algorithm searches the two strings, looking for regions that are approximately the same (e.g. a gene common to both organisms, that has mutated slightly).

Stringmol modifies the Smith-Waterman algorithm to search over strings from the Stringmol alphabet, treating matching subsequences as binding sites. Also, rather than matching subsequences common to both strings, Stringmol matches *complementary* subsequences. Stringmol does this by inverting one of the strings before searching for common subsequences. The details of the inversion function are important for evolution. For investigations into the inversion function, see [28]. For the implementation details of Stringmol's modified Smith-Waterman algorithm, see [43].

Stringmol uses the same Smith-Waterman substring-matching algorithm to embody its `at-end` function. Reactions end crisply when the copying mechanism reaches a special *end reaction* character, or when it reaches the end of the template string. But in Stringmol, the copy produced by the replicase is embodied in the system as a sequence of characters appended onto the end of the replicase chemical. When the copy is complete, the replicase mechanism cuts this copied chemical off itself. Thus the replicase mechanism needs to detect when the copy is complete and cut itself in the correct place before the reaction ends, otherwise the replicase would be left with the copied chemical attached to itself (which can happen in Stringmol runs, if one of these sub-mechanisms mutates). To perform these tasks, it uses the Smith-Waterman substring-matching algorithm to move its various pointers around the two chemical strings. This allows evolving Stringmol chemicals to change their matching sub-strings and alter the embodied `at-end` function.

Stringmol's `char-copy` function is stochastic. The replicase's *read pointer* iden-

tifies the next character to be copied, and its *write pointer* references the location it will be copied to. There is a *copy* instruction on the replicase chemical that copies the character from the read pointer to the write pointer, with a small chance of error. This copy instruction executes instantly, in a separate world to the rest of the reaction. So the `char-copy` function can introduce variation into the copied string, by copying some characters wrongly (single-character substitutions). But evolution cannot change the behaviour of the `char-copy` function.

Stringmol's `next` function is crisp. To operate the copying mechanism, different pointers need to be incremented at different times. The read pointer and write pointer are both incremented when the `char-copy` function is called. There is an *instruction pointer* that increments each timestep. These pointer increments are built into the world of Stringmol. They cannot vary, either by the influence of random numbers or other mechanisms.

5.2.2.1 Embodiment generates novelty

Stringmol's embodied copying process has been shown to generate novelty [42]. Because the phenomenon of copying is embodied within a population of replicase chemicals, evolution can change the copying phenomenon when the replicase copies (another instance of) itself. Copying errors produce different versions of the replicase chemical that embody the copying process using slightly different mechanisms. If one of these mechanisms is more efficient at copying than the others, then this will out-compete the others due to Stringmol's energy model (described in section 5.1.1.4), and the copying phenomenon will have changed.

One sequence of changes observed in Stringmol (described in detail in [42]) led to a completely different type of copying emerging from the initial replicase mechanism.

1. The stochastic `char-copy` function can make a certain single-character substitution in a copy of the replicase that stops it from implementing its `at-end` function properly.

2. When this mutated replicase tries to copy a non-mutated replicase, it does not detect when it has reached the end of the copy. The result is a mutated replicase chemical that has another copy of the replicase appended onto its end (so is twice as long as it should be).

3. When this double-length replicase attempts to copy a normal replicase, its embodied `start` function happens differently. The double-length replicase has two *start* binding regions, since it is two copies of the original replicase glued together.

4. This causes it to start copying the replicase in the wrong place: it misses out the first few characters.

5. The result of this chain of reactions is the creation of a replicase chemical that is missing its first few characters.

The only type of mutation programmed in to Stringmol is the stochastic `char-copy` function, that can make single-character substitutions. But because of its embodiment, the copying phenomenon has been able to evolve a different type of mutation: a macro-mutation that removes the first few characters of a chemical. This is meta-evolution: the evolution of one of the mechanisms of evolution (the mutation function). And this meta-evolution happened because of embodiment: the embodied `start` and `at-end` functions could be exploited by the system, and used in a novel way.

The macro-mutation on its own would be a strong enough argument for embodiment, but Stringmol went further than this with its novelty-generation. The short replicase produced by the macro-mutation can influence the evolution of the long replicase. There is a long period of evolution (described in detail in [42]), after which the two replicase mechanisms are no longer able to copy themselves, but can copy each other. This is a huge change to the copying phenomenon. Stringmol has changed from a single copying mechanism that copies itself, to two copying mechanisms that depend mutually upon each other.

The embodiment of Stringmol's copying mechanism produced something different from what would normally be expected of a copy operation in novelty-generation algorithms. It produced a novel type of mutation, which led to the emergence of a mutualism. The emergence of a new type of mutation is not possible using just a stochastic copy operation. Embodiment is needed to allow the intermediate stages of the copying process to be exploited and changed. This example shows the potential novelty-generation capabilities of embodying the copying process (or more generally, any process). I believe that by embodying different stages of the copying process, we will be able to observe different, unprogrammed types of mutation emerging from our novelty-generation algorithms. And these different types of mutation may lead to other different types of novelty, such as Stringmol's mutualism.

The following chapter describes my artificial chemistry, *GraphMol*, that embodies the `next` function of the copying process. I chose the `next` function in order to investigate a method of embodying the copying process that would be completely different from Stringmol's.

Chapter 6

GraphMol

GraphMol is an artificial chemistry I have developed to investigate how the phenomenon of copying can be embodied within low-level mechanisms in an artificial chemistry world. The world defined by GraphMol contains chemicals (represented as graphs) that bind to each other via multiple binding sites, and then run simple computer programs (encoded in the graphs) that modify the binding of these chemicals.

I have designed this world so that copying mechanisms can be embodied naturally within it. The current version of GraphMol is particularly suited to embodying the `next` function of the copy operation, described in section 5.2. The `next` function is what I concentrate on here, but the design of GraphMol is not limited to just embodying this aspect of copying. The design of GraphMol is intentionally flexible and modular so it could be extended and modified to allow the other functions to be embodied also. To illustrate this, some possible ideas for extending GraphMol to embody the `char-copy` function are discussed in the future work section (chapter 9).

This chapter describes the GraphMol artificial chemistry. I start by describing GraphMol in terms of my definition of artificial chemistries given in chapter 5. I then describe in detail the different components of the GraphMol world, before showing how these can be used to build a mechanism embodying the phenomenon of copying. The following two chapters describe experiments investigating embodied copying in GraphMol.

6.1 Overview of GraphMol

As explained in section 5.1.1, artificial chemistries can be defined by describing their *world*, *chemistry* and *constraints* (two important constraints being the *energy model* and the *binding model*).

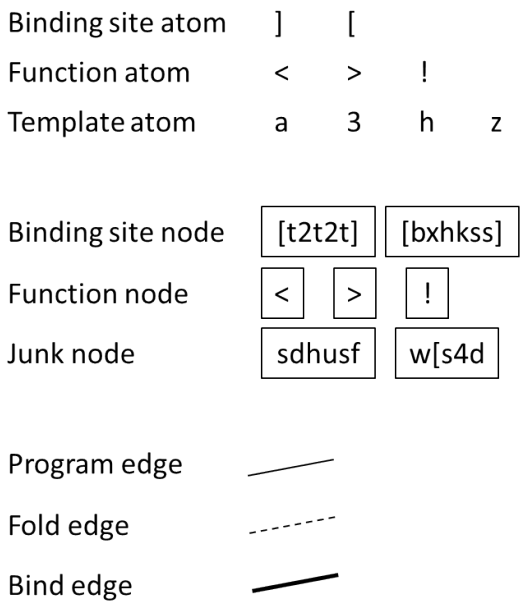


Figure 6.1: The components that make up the GraphMol world. There are three different types of *atom*, that combine to make three different types of *node* that are connected together by three different types of *edge*.

Figure 6.1 shows the different components that make up GraphMol's world. GraphMol chemicals begin their existence as a sequence of *atoms*. This sequence of atoms is *parsed* into a sequence of *nodes*, which are *folded* into a graph and connected together by *edges*. The parsing and folding processes are described in sections 6.2.2 and 6.2.3 below. Figure 6.2 shows an example of a GraphMol chemical graph, which exists in an abstract, aspatial world. We can visualise this graph in two dimensions, to help our intuitive understanding, but the GraphMol world does not have the geometry of our world. The GraphMol *distance* between two atoms is their distance through the edges of the graph. This graph distance is an important property of the world that is used heavily by the chemistry.

GraphMol's chemistry defines how two chemical graphs can react with each other and change their topology (i.e. add and remove edges). Each graph is both a data structure and a computer program. GraphMol's binding model stochastically searches for binding site nodes with matching templates. When it finds a matching pair of binding site nodes, it joins them together with a bind edge, changing the topology of their graphs. It also runs the computer programs encoded on each graph. The execution of a program is influenced by the topology of its graph and the data stored within it; and likewise, execution of the program changes the topology of the graph and the data stored within it. Data is stored in a graph in the

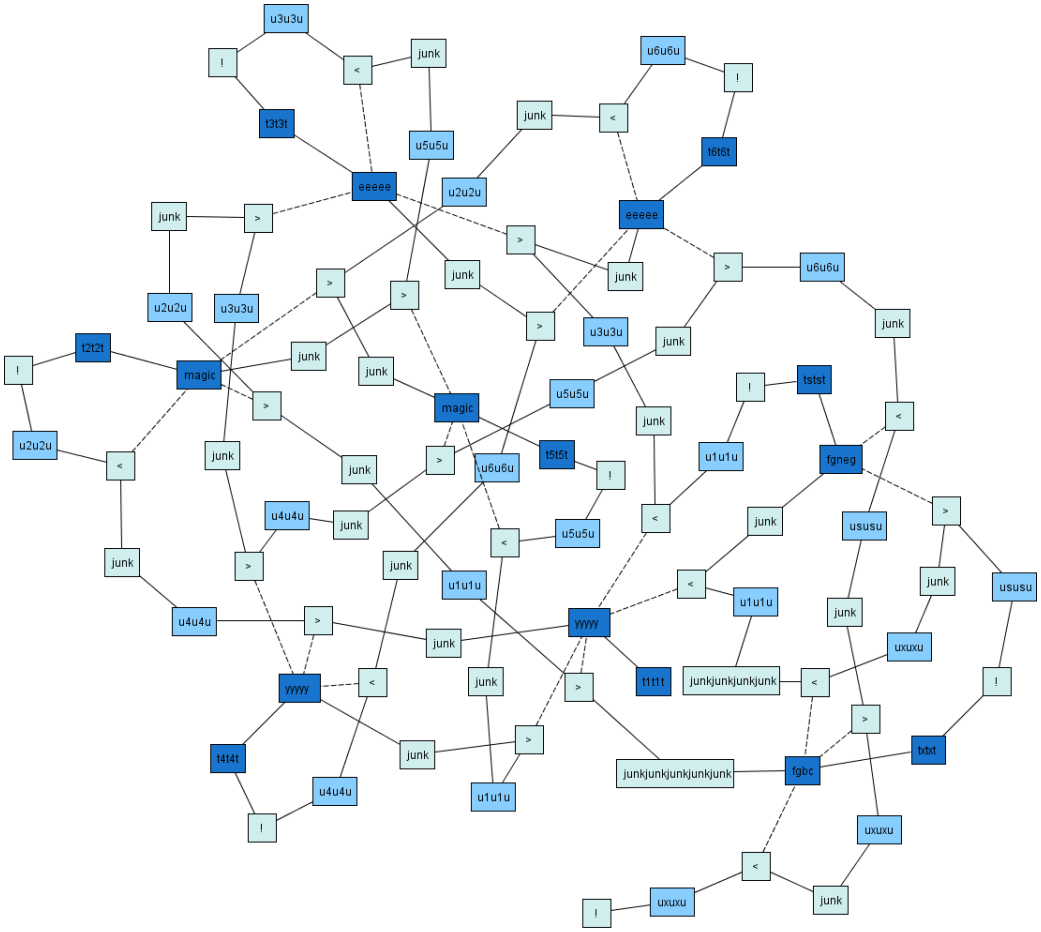


Figure 6.2: An example of a GraphMol chemical. Program edges are denoted by solid lines and fold edges by dashed lines. There are no bind edges in this image. `Shown` binding sites are dark blue, `hidden` binding sites are medium blue, junk and function nodes are light blue (see the text for details of these terms). The geometry of how this graph is visualised does not influence its function: it is the topology of the graph that matters.

form of its binding sites, which are each in one of three states: `shown`; `hidden`; or `bound`. `Shown` binding site nodes are the only sites available for the binding model to bind, `hidden` binding site nodes have been explicitly unbound and removed from the binding model by a graph's program, and `bound` binding site nodes are those that are already connected to a bind edge. GraphMol uses the same token-based energy model as Stringmol, to embody the number of chemicals within the world (described in section 5.1.1.4).

In *static* terms, a GraphMol system is a collection of nodes, connected together by edges. These form chemical graphs with binding sites that are `shown`, `hidden` or `bound`.

In *dynamic* terms, there are two different processes that work together to change GraphMol's chemical graphs:

1. a declarative, binding process (physics);

2. a collection of imperative, computer programs (chemistry).

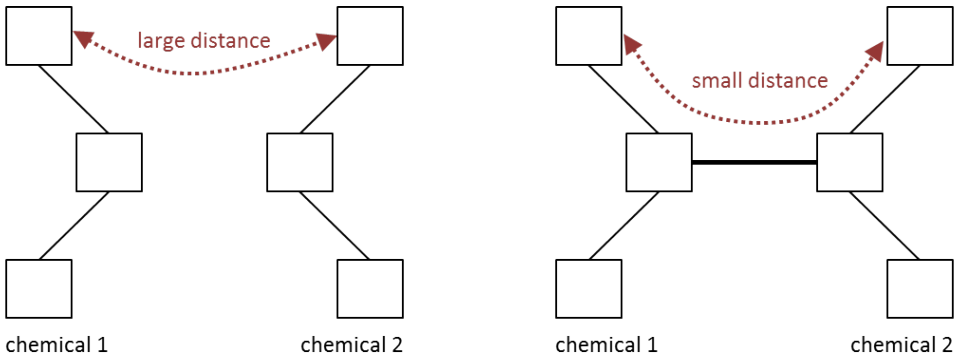
I describe these two processes below, and explain how they can be used to build mechanisms in the GraphMol world.

6.1.1 Declarative binding process (physics)

The binding process operates continually, as a declarative programming language. In other words, GraphMol chemicals specify *what* should bind (by defining binding sites with matching templates), and the GraphMol world (the binding process) specifies *how* binding happens. The binding process runs continuously, searching for matching binding sites and forming binds when it finds matches. This is analogous to the level of physics (processes happening continually to atoms).

The process is stochastic, and the chance of two sites binding depends on: (1) how closely their binding site patterns match; and (2) their distance apart through the graph, measured as the length of the shortest path between the two binding sites. It continually changes the graph topology by adding bind edges between `shown` binding sites.

Figure 6.3 shows how the binding process creates binds between different chemicals, thus joining together two different graphs via the addition of a bind edge between them. This new edge changes the topology of the graphs, and so changes the graph distances between binding sites. Sites that were on different chemicals were previously far away from each other, but are now close together because of the bind. As well as creating binds between different chemicals, the binding process can also create binds within a single chemical, if one chemical contains two binding sites with matching patterns.



(a) Before a bind has been formed

(b) After a bind has been formed

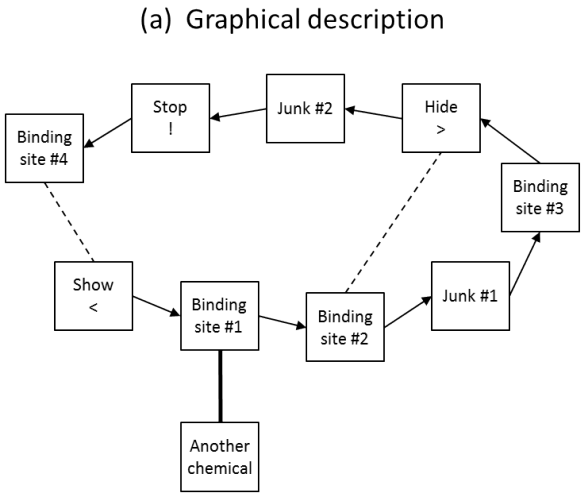
Figure 6.3: The binding process creates binds (stochastically) between matching binding site nodes by adding a bind edge between them. This changes the graph distances between other binding sites in the graph, changing the probabilities of future binds forming.

6.1.2 Imperative computer programs (chemistry)

The computer programs form an imperative programming language. Each graph contains a sequence of instructions (function nodes) that explicitly specify *how* to execute a program. Computer programs do not run continuously: they are explicitly started when a bind happens, then they run until they reach the end of the program, at which point they stop. This is analogous to the level of chemistry (reactions happening between chemicals, at a higher level than physics).

Figure 6.4 shows an example GraphMol computer program. The program is encoded in a chemical graph by the function nodes, program edges and fold edges. When the binding process forms a bind, it creates an instruction pointer at the binding site, that moves through the graph, following the program edges and executing the function nodes. The fold edges specify parameters to some of the function nodes (`show` and `hide`), in the form of a pointer to a binding site. When instruction pointers execute functions, they change the topology of the graph by `showing` and `hiding` binding sites. When binding sites are `shown`, new binds become possible; when binding sites are `hidden`, some binds become impossible.

In the binding process above, GraphMol chemicals are treated as *undirected* graphs. The computation of shortest-paths through the graphs treats equally each different type of edge (program edge, fold edge and bind edge). However, to the computer programs, GraphMol chemicals are *directed* graphs. The instruction pointers follow a linear sequence of directed program edges as the program executes. At `show` and `hide` function nodes, the program peeks down a fold edge to determine the parameter passed to the function.



(b) Sequential description

1. Another chemical binds to Binding site #1
   1. A program starts executing
   2. An instruction pointer (IP) is created at binding site #1
2. IP moves to Binding site #2
3. IP moves to Junk #1
4. IP moves to Binding site #3
5. IP moves to Hide
   1. The fold edge is followed, to find the Binding site to Hide
   2. The function "Hide Binding site #2" is executed
   3. Binding site #2 is hidden
6. IP moves to Junk #2
7. IP moves to Stop
   1. The instruction pointer is removed
   2. The program terminates

Figure 6.4: An example imperative computer program represented as: (a) its encoding within a GraphMol chemical; and (b) the sequence of steps happening as the program executes. The instruction pointer moves along program edges (solid lines) and uses the fold edges (dashed lines) to access parameters of function nodes.

6.1.3 Building mechanisms

The declarative binding process and the imperative computer programs interact with each other. When the binding process forms a bind between two binding sites, it starts two computer programs executing (one starting from each binding site). When a computer program runs, it can `show` and `hide` binding sites, making them available or unavailable to the binding process.

In terms of embodiment, the two processes can interact because they exist in the same world. The processes cannot change each other's implementations, but they can change the effect that the other process will have on a given GraphMol chemical. This interaction between the two processes can be used to implement mechanisms within GraphMol chemicals.

6.1.4 Running GraphMol

Algorithm 2 shows the processes involved in a complete GraphMol run. The world is initialised with a collection of chemicals, represented as sequences of atoms. In an initial setup phase, these atoms are parsed and folded to produce a collection of graphs. Then the main loop starts, where the binding process and computer programs run, modifying the collection of graphs. The main loop will typically run many times, until any desired stopping condition is met (e.g. novelty-generation observed, the chemicals have stopped reacting, we have run out of time, etc).

Algorithm 2 Pseudocode of a GraphMol run

```
Input: S          collection of sequences of atoms
C := ∅            collection of GraphMol chemicals

initial setup of the chemicals
for all sequence of atoms in S do
  Parse sequence of atoms into a sequence of nodes
  Connect nodes with program edges, creating a graph
  Fold graph, adding fold edges
  Add graph to C
end for

run the GraphMol world
repeat

  binding process
  for all binding site in all chemical in C do
    Stochastically search for a bind partner for this binding site
    if found a bind partner then
      Connect the two binding sites with a bind edge
      Create a new instruction pointer at each binding site
    end if
  end for

  computer programs
  for all instruction pointer in all chemical in C do
    Move instruction pointer along the next program edge
    if instruction pointer at a function node then
      Execute the function
    end if
  end for

until any desired stopping condition is met
```

6.1.5 Junk (controlling mechanisms)

GraphMol nodes are either binding sites (which interact directly with the binding process), function nodes (which interact directly with the computer programs), or junk (which interacts directly with neither process). Although it does not act directly in the world, junk is an important concept in GraphMol. It is a link between the binding process and computer programs, that can be used to tune the effect that each process has on the other.

Junk affects the GraphMol world in two different ways:

1. It affects the graph distance between nodes, used to calculate binding probabilities. Adding junk between two binding sites makes them further apart, and thus less likely to bind.

2. It acts as a no-op for instruction pointers moving through the graph, slowing down execution of programs with respect to the timescale of the binding process. Because the computer programs execute at the same time as new binds are being formed, adding junk into a computer program effectively makes the computer program run more slowly, compared to the rate at which new binds are being formed.

Because junk indirectly affects both processes in different ways, it can be seen as a parameter of a mechanism implemented in GraphMol. This parameter tunes the effect that the two processes exert on the mechanism. By changing a non-functional part of a mechanism, it is possible to change the effect that the world has on the functional parts of the mechanism.

Because junk is composed of atoms and nodes in the GraphMol world, a copying mechanism embodied within the GraphMol world will be able to change the amount of junk in its copies. Thus, GraphMol mechanisms can evolve their own junk parameter. This means that GraphMol mechanisms can evolve the effect that the world has on their functional parts. So GraphMol mechanisms can evolve their own functions, and hence achieve meta-evolution (at least in principle). This is an example of how embodied evolution can be used to implement meta-evolution.

We see later (in chapter 7) that junk can have a dramatic effect on how GraphMol mechanisms operate.

6.2 GraphMol chemicals in detail

The previous section gave a general overview of GraphMol. This section explains in detail how GraphMol chemicals are implemented. The next section describes in detail how GraphMol chemical reactions are implemented.

[	begin	Begin defining a binding site	binding
]	end	End definition of a binding site	binding
<	show	Show binding site	function
>	hide	Hide binding site	function
!	stop	Stop the execution of a program	function
a-z, 0-9	junk	Non-functional atoms	template

Figure 6.5: The alphabet of GraphMol atoms.

6.2.1 Biological inspiration

In biological cells, chemical mechanisms start their existence encoded in a sequence of DNA that is translated into a linear sequence of amino acids. These amino acids then fold into a 3-dimensional shape (a protein) that has both a structure and a function. The embodiment implications of this are described in more detail in section 4.2.

GraphMol chemicals start their existence as a sequence of *atoms*, corresponding to amino acids in biology. There are 20 different amino acids in biology, whereas in GraphMol there are 6 different types of atom, listed in figure 6.5 (although the junk atoms are different from each other, but not as different as they are from the functional atoms and binding atoms). This leaves plenty of room to add more atoms to GraphMol in the future, and still keep the biological analogy.

Protein folding in biology is known to be a very complex process, the precise details of which are not fully understood. But the result of protein folding is to create a chemical that has multiple *binding sites*, which are collections of amino acids that are physically close to each other and can recognise the binding sites of other chemicals by their physical shape and their pattern of electrical charge. It is the *properties* of biology that we care about, rather than biology's *implementations* of those properties. So we define an analogue of protein folding in GraphMol that achieves an analogous result, but using a different process that is much simpler to compute.

There are two steps to converting a sequence of GraphMol atoms into a chemical graph composed of nodes and edges. These are (1) parsing atoms into nodes and (2) folding: connecting function nodes to their corresponding binding site nodes. These processes are illustrated in figure 6.6.

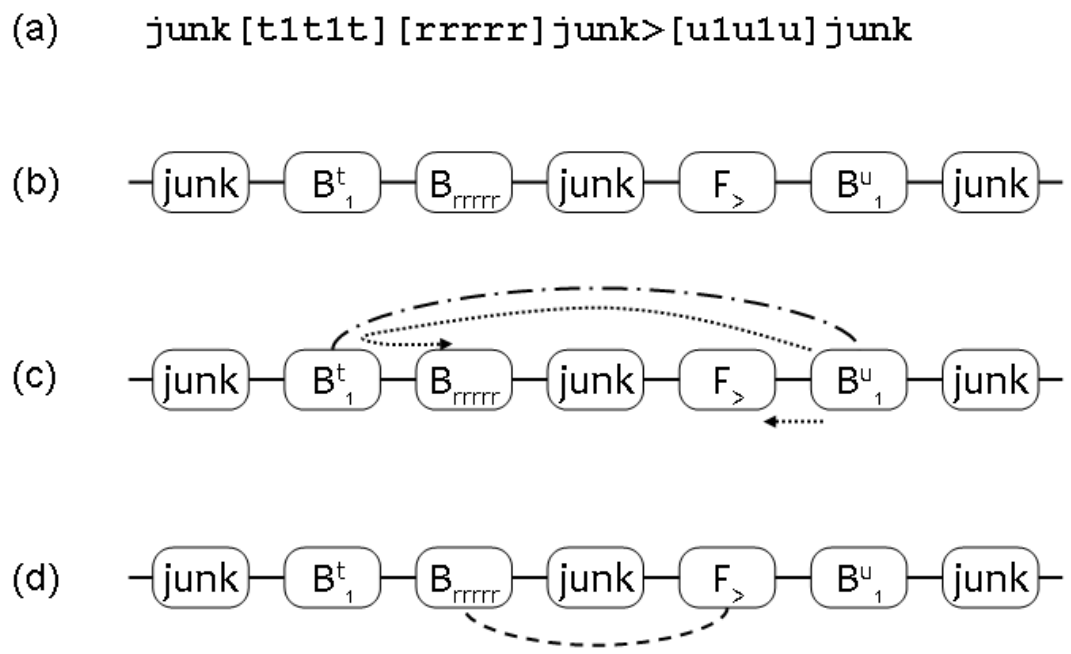


Figure 6.6: Parsing and folding: (a) a sequence of atoms; (b) the parsed graph of nodes connected by program edges (solid edges); (c) part way through folding: temporary edge between the `u1u1u` binding node and the `t1t1t` binding node (dash-dotted edge), used to find the closest function node and binding site (dotted arrows); (d) resulting folded graph, with fold edge between the function node and its binding site (dashed edge).

6.2.2 Parsing

Just as biological binding sites are composed of multiple amino acids that are close to each other, binding sites in GraphMol are composed of multiple atoms that are close to each other. In biology, amino acids can be close to each other in 3-dimensional space, but a long way from each other through the linear sequence of amino acids. In GraphMol, we simplify this by requiring that the atoms comprising a binding site must be close together (contiguous) in the linear sequence. This makes it easy to compute and design GraphMol binding sites.

We want to refer to binding sites as single units (graph nodes), rather than as collections of atoms, so we parse the string of atoms to identify binding sites. At this stage, we also group together contiguous junk regions, to make the GraphMol code run faster and to aid understanding of how GraphMol is working. The string of atoms (figure 6.6 (a)) is parsed into a linear graph (figure 6.6 (b)) of binding site nodes, junk regions, and function nodes (which contain only one atom).

6.2.2.1 Determining binding sites

A sequence of atoms enclosed in brackets `[aaaaa]` (with no internal brackets) defines a binding site. So the string:

```
[hdfggd[icsd]bdgd[dhdhd]ixr]ss[sda<dsa]djs>tf
```
(6.1)

defines three binding sites, with patterns: `icsd`, `dhdhd` and `sda<dsa`. Mismatched brackets are treated as junk, so the above sequence has five junk nodes: `[hdfggd`, `bdgd`, `ixr]ss` `djs` and `tf`. If there are any function atoms within a pair of matching brackets, then they are added to the pattern of their binding site and they do not turn into function nodes. So the above sequence has only one function node: `>`. The sequence above parses into the following nodes:

Node type	Pattern
Junk node	`[hdfggd`
Binding site node	`icsd`
Junk node	`bdgd`
Binding site node	`dhdhd`
Junk node	`ixr]ss`
Binding site node	`sda<dsa`
Junk node	`djs`
Function node	`>`
Junk node	`tf`

These nodes are connected together into a linear sequence with directed program edges. The *folding* process transforms this linear sequence into a general graph.

6.2.2.2 Why have explicit binding sites?

In biology, binding sites do not have special bracket symbols enclosing them. A biological *binding site* is an emergent property of the structure of a chemical, in relation to the other chemicals in its environment. But in GraphMol, binding sites are explicitly encoded in the string of atoms that defines a chemical. It is possible for artificial chemistries to have binding sites that are defined implicitly. For example, the Stringmol artificial chemistry [41] applies a subsequence alignment algorithm to two chemicals, to determine if they share a region of similarity (and hence determine if they bind).

My reason for using explicit binding sites in GraphMol is that I want the chemistry level (imperative computer programs) to explicitly manipulate binding sites, `showing` and `hiding` them. Stringmol's chemistry level is simpler than this, manipulating only four pointers, whereas GraphMol's chemistry level can manipulate an unbounded number of binding sites. There is an inherent tradeoff here between the complexity of the physics level and the complexity of the chemistry level. Stringmol has implicitly-defined binding sites in its physics, at the expense of explicitly-defined pointers in its chemistry. GraphMol has explicitly-defined binding sites in its physics, so that it can have implicitly-defined operations in its chemistry (binding sites as parameters to functions).

Clearly, biology's chemicals have both implicitly-defined structure and function. In principle, it would be possible to define both levels implicitly in an artificial chemistry, but (with our current understanding of artificial chemistries) this would create a world in which mechanisms were difficult to design and took a long time to run.

6.2.2.3 Why have two bracket symbols?

Having an opening bracket and a closing bracket is not the only way to define binding sites in a string of characters. I could have used a single symbol to toggle whether the next sequence of characters is a binding site or not, for example:

```
|hdfggd|icsd|bdgd|dhdhd|ixr|ss|sda<dsa|djs>tf
```
(6.2)

In contrast to the above example (equation 6.1), this method creates four binding sites: `hdfggd`, `bdgd`, `ixr` and `sda<dsa`. I decided against this method because it allows a mutation in one character to completely change all downstream binding sites and junk regions. For example, if the second `|` character above was mutated to a non-functional character, the above sequence would become:

```
|hdfggdzicsd|bdgd|dhdhd|ixr|ss|sda<dsa|djs>tf
```
(6.3)

The sequence now defines four completely different binding sites: `hdfggdzicsd`, `dhdhd`, `ss` and `djs>tf`. It contains none of its original binding sites or function

characters, and all the function characters it now contains were not present before the mutation.

If this same mutation were to happen in the representation using brackets, equation 6.1 would become:

```
[hdfggdzicsd]bdgd[dhdhd]ixr]ss[sda<dsa]djs>tf
```
(6.4)

Comparing equations 6.1 and 6.1, we can see that the mutation has only affected one binding site: changing it from `icsd` to `hdfggdzicsd`. The effects of this mutation are local to its binding site and adjacent junk region. None of the binding sites and function characters downstream of this change have been affected.

I considered large-scale mutational effects to be too drastic, but it would be interesting to see if evolving systems can survive when mutations like this are possible. Also, using brackets makes hand-designing chemicals easier, because looking at a chemical string, it is immediately obvious which sequences are binding sites.

6.2.3 Folding

Two of the functions, < and >, take parameters. When executed, the < function `shows` a particular binding site and the > function `hides` a particular binding site. These binding sites may be located anywhere in the GraphMol chemical. The folding process connects these function nodes to the binding sites they affect, by adding in fold edges between function nodes and the binding sites they operate on. Folding changes a linear sequence of nodes into a graph with (potentially) long-range connections and cycles. An example of a folded graph is shown in figure 6.2.

The folding process may seem convoluted at first glance, but (as explained below) it has been designed from a biological viewpoint, and it is useful computationally. It implements a form of indirect addressing, that should make GraphMol chemicals more evolvable than if direct addressing were used. The biological analogy is with how proteins fold in biology. The folding process could be embodied as a GraphMol *chaperone* chemical, rather than the crisp algorithm used here. Doing this could potentially allow the folding process itself to be under the control of evolution (although this is not investigated in this thesis).

6.2.3.1 Folding process

The folding process is described in algorithm 3 and illustrated in figure 6.6. To fold a chemical, the chemical is searched for pairs of binding sites with matching *folding patterns*. These are binding sites of the form `uxuxu` and `txtxt`, where `x` is any atom. For example, in figure 6.6, the matching patterns are `u1u1u` and `t1t1t`. Note that the rule used to determine matching *folding patterns* is different from

Algorithm 3 The GraphMol folding process

Input: a GraphMol chemical to be folded

for every possible fold in the chemical
for all binding sites **in** the chemical **do**
  if the binding site is of the form `uxuxu` **where** `x` is any atom **then**
    Search in the chemical for a matching binding site, of the form `txtxt`
    if found a matching binding site **then**

      perform the fold
      Add a temporary graph edge between `uxuxu` and `txtxt`
      Search the chemical graph breadth-first, ignoring `txtxt` and `uxuxu`
      Starting from `uxuxu`, search for the closest function node
      Starting from `uxuxu`, search for the closest binding site
      if found a function node **and** found a binding site **then**
        if the function node does not already have a fold edge **then**
          Add a fold edge between the function node and binding site
        end if
      end if
      Remove the temporary graph edge between `uxuxu` and `txtxt`

    end if
    Hide the binding site `uxuxu`
  end if
end for

the rule used during *binding*. These two binding sites do not *bind*, they are merely brought close together for the folding process. The implications of these design decisions are discussed in the following sections.

When two matching binding sites have been found, a temporary graph edge is added between them, bringing them close together in terms of graph distance. We then start from the `uxuxu` binding site and search for the closest function node (`show` or `hide`) and the closest binding site (ignoring the `uxuxu` and `txtxt` sites). This is shown in figure 6.6 (c). *Closest* here refers to distance along the edges of the graph, including the temporary edge (with all edges being the same length). This means that a function node and a binding site can be far apart in the chemical graph, but close together when the temporary edge is taken into account.

When the closest function node and binding site have been identified, they are joined by a fold edge, and the temporary edge is removed. The result is shown in figure 6.6 (d). Finally, the `uxuxu` binding site is `hidden`.

Every fold is controlled by a `uxuxu` binding site. One chemical might require many folds, and so contain many `uxuxu` binding sites. One fold is performed for each `uxuxu` binding site, until each such binding site has been `hidden`. This means that some function nodes may not get fold edges added to them (if there are more function nodes than `uxuxu`/`txtxt` binding site pairs). These function nodes do not have a parameter to act on, so they are treated as no-ops or template characters.

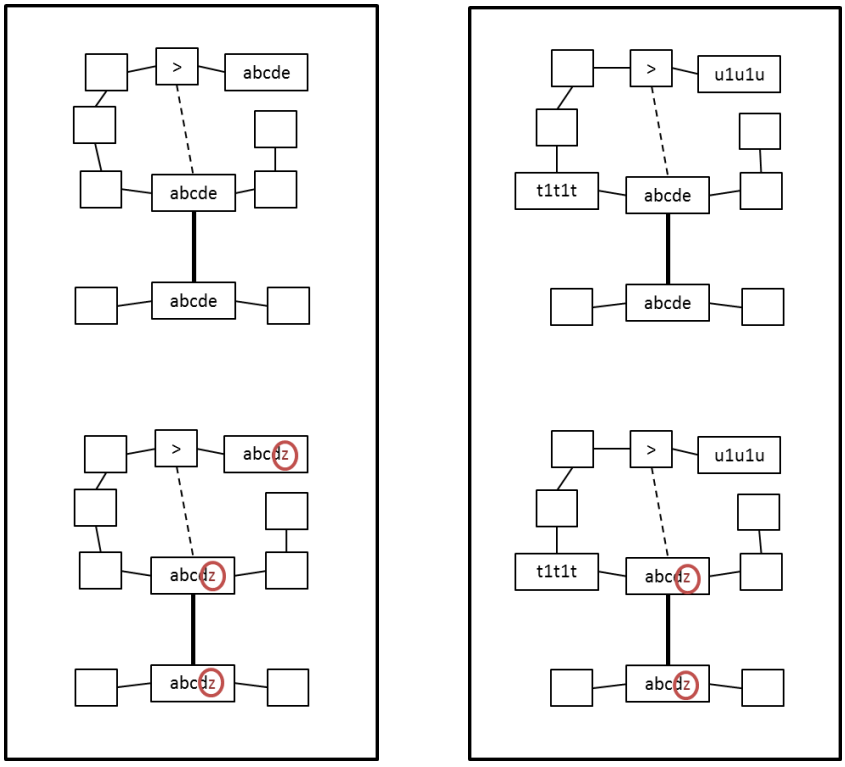
One useful design pattern is to place the `uxuxu` binding site immediately after the function node, and the `txtxt` binding site immediately before its target binding site, with junk used to provide buffers around these regions. This ensures that the function node and its target binding site will always fold correctly. It also means that multiple function nodes can fold to the same target binding site, because the target binding site is referenced by a `txtxt` binding site rather than a `uxuxu` binding site. Placing the `uxuxu` binding site *after* the function node and the `txtxt` binding site *before* the target binding site is a trick used to reduce implementation errors by having different standards for the two different types of folding binding site. This design pattern is used in the embodied copying mechanism described later.

6.2.3.2 Indirect addressing

The fold edge is a form of indirect addressing. Instead of the function node specifying explicitly which binding site it `shows` (or `hides`), it instead specifies a template: `uxuxu`. During folding, this template is *dereferenced* to locate the binding site: the closest binding site to the matching `txtxt`[1].

Indirect addressing should make the system more evolvable, because the templates can change independently of the pattern of the target binding site. Figure 6.7 illustrates this. For example, chemical A may have a binding site that binds to chemical B. But the pattern of the binding site on chemical B may change over evolutionary time. With indirect addressing (figure 6.7 (b)), the pattern on chemical A can change to match this, without affecting the folding of chemical A. But if direct addressing were used (figure 6.7 (a)), then for every mutation of chemical B's pattern, chemical A would have to make a mutation in its binding site, and an extra coordinated mutation in every folding template that references the binding site. This would be less likely to happen by chance, and could disrupt the folding of chemical A if it went wrong. Indirect addressing decouples the two tasks of folding and binding, allowing them to evolve independently and not disrupt each other.

[1]Technically, it is the closest binding site to `uxuxu`. But the purpose of the matching `uxuxu` and `txtxt` pair is to bring a different binding site close to `uxuxu`, via the temporary graph edge to `txtxt`.



(a) Direct addressing (b) Indirect addressing

Figure 6.7: Chemical A has a binding site matching chemical B. If the binding site on B mutates, then: with direct addressing (a), chemical A needs to make two coordinated mutations; but with indirect addressing (b), only one mutation is needed. Indirect addressing decouples folding and binding.

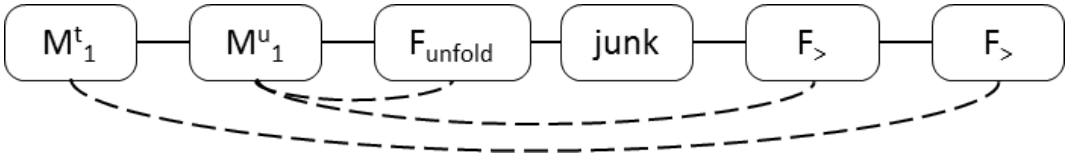


Figure 6.8: The (simplified) chaperone chemical that can implement the folding process, in conjunction with a modified GraphMol physics engine.

With the `uxuxu/txtxt` matching rule, there are a total of 41 different matching patterns, because there are 41 different atoms that could go in place of `x`. These are: `a-z 0-9 [ ] < > !` If very large chemicals were needed in a system, with more than 41 folds, then a different matching rule could be used, allowing for more different matching patterns. For example, if the two `x` characters in the above pattern were allowed to be different, and the matching patterns were `uxuyu` and `tytxt` (for any `x` and `y`), then there would be $41^2 = 1681$ different patterns available. And if three characters were allowed to vary in the pattern, then there would be $41^3 = 68,921$ different patterns available.

6.2.3.3 Crispness, stochasticity and embodiment

The `uxuxu/txtxt` matching rule used here is crisp (exact string matching), because folding is not the subject of this investigation. But the matching rule could easily be changed for one that is stochastic. Having a rich, stochastic matching rule (such as Stringmol's binding model based on inexact subsequence matching, section 5.1.1.5) would give evolution a finer level of control over the folding process, making GraphMol chemicals more evolvable.

The method of choosing the function node and target binding site is crisp (choose the closest ones to the `uxuxu` binding site). These choices could be made stochastic by choosing function nodes and binding sites randomly, weighted by their distance from the `uxuxu` binding site. This would mean that the 'correct' folding was most likely to happen, but it would also allow the possibility of 'mistakes' to happen during the folding process, which could help evolution.

The two ideas above would make the folding process stochastic, and so improve the evolvability of GraphMol chemicals. The following section describes an alternative approach, that makes the folding process itself evolvable by embodying it as a GraphMol chemical, within the GraphMol world.

6.2.3.4 Chaperone chemicals (embodied folding)

The folding process can be viewed as two different processes: a physics process and a chemistry process:

- The physics process connects function nodes to binding sites that they are close to.

- The chemistry process finds binding sites with matching folding patterns, and brings them close together.

This is analogous to how proteins fold in biology, although clearly GraphMol uses a much simpler physics engine than biology does. The laws of physics fold biological proteins into low-energy configurations. This is sufficient to fold some proteins, but others need help from *chaperone proteins* that bind to them to help with the fold, and unbind when their job is complete (see any molecular biology textbook for details, for example [15, p409-10]).

The GraphMol folding process has been designed so that it could be implemented using GraphMol chaperone proteins, rather than the crisp algorithm described above. This would embody the folding process within the GraphMol world. To do this, we implement the physics of the folding process within the GraphMol physics engine, and we implement the chemistry of the folding process as a GraphMol chemical.

The physics part of the folding process works in a similar way to the *binding process* described above. It runs continually, but rather than using bind edges to connect matching binding sites, it uses fold edges to connect function nodes to binding sites that they are close to. To keep the chemicals stable, function nodes only have one chance to make a connection: once they have connected to a binding site then they do not spontaneously connect to a different one in the future.

The chemistry part of the folding process is implemented as a GraphMol chaperone chemical, shown in figure 6.8. Figure 6.9 shows how this chemical (together with the physics part) can fold the example chemical from figure 6.6, described by the sequence of events below:

1. If we put a parsed chemical into the world, then the physics part will fold its function nodes to the binding sites they are close to in the linear graph (figure 6.9 (b)).

2. In order to make long-range folds, we need chaperone chemicals. A chaperone chemical contains two binding sites that match the folding sites on the target chemical (`uxuxu` and `txtxt`). Here we mean *match* in the sense of binding. So the chaperone chemical binds to a pair of `uxuxu` and `txtxt` binding sites on the target chemical (figure 6.9 (c)).

3. This brings the `uxuxu` and `txtxt` binding sites close together, because they are both bound to the same chemical (and the binding sites on the chaperone chemical are close together). So now, the target function node and its target

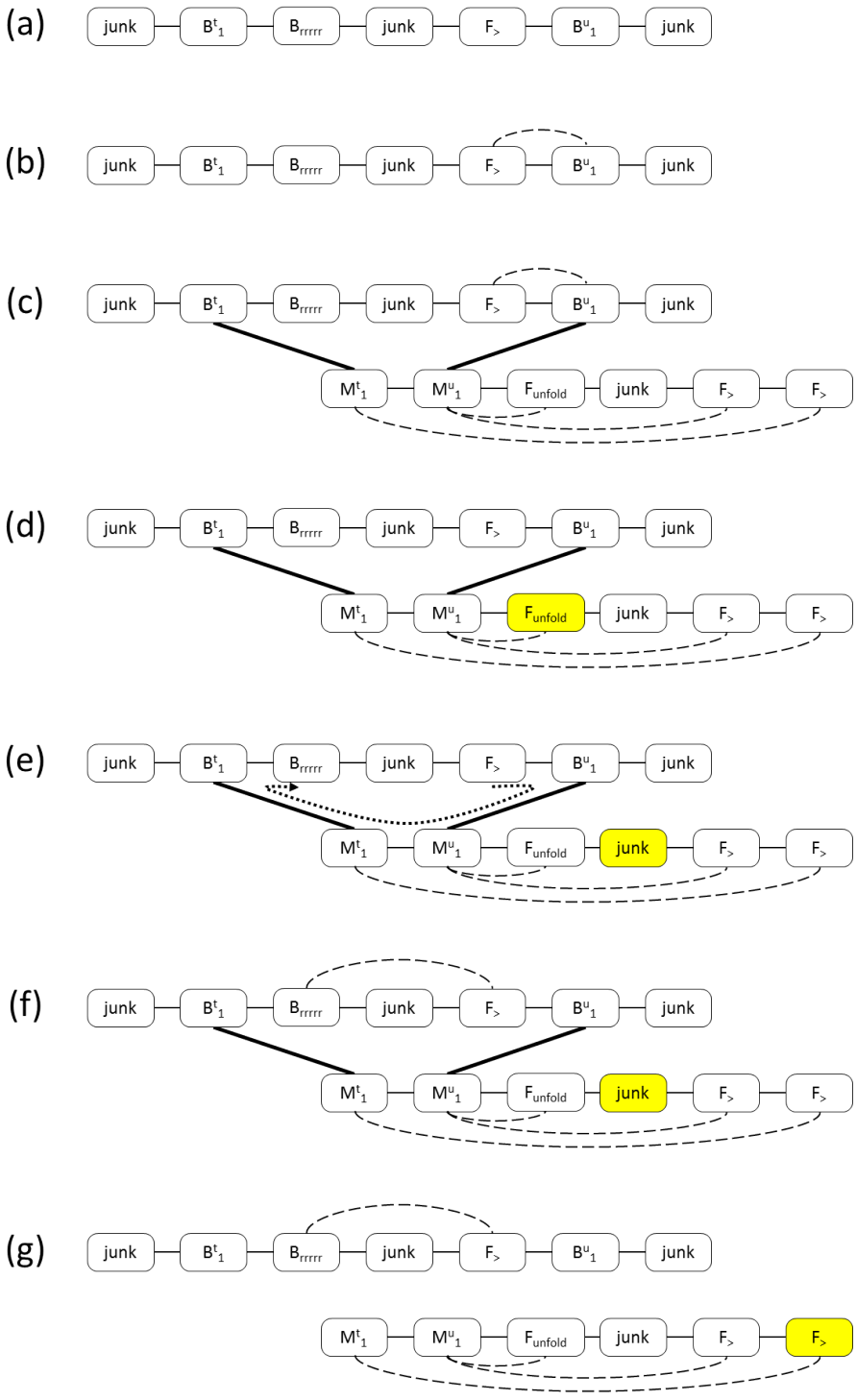


Figure 6.9: The chaperone chemical helping to fold a target chemical. See the text for details.

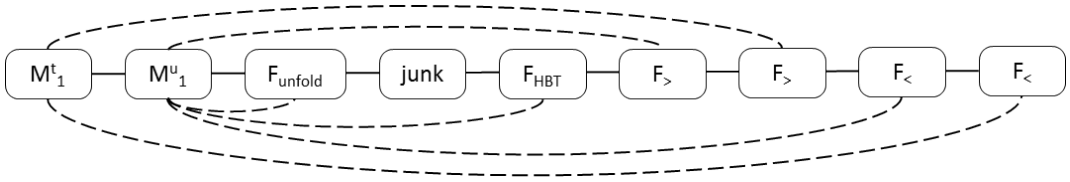


Figure 6.10: The full chaperone chemical, with all the functionality needed to fold multiple chemicals in the GraphMol world. The folding templates are omitted from this diagram, to more clearly show the functionality of the chaperone chemical, rather than the details of its folding. For the folding templates, see figure 6.11.

binding site are close together. But the fold cannot happen yet, because the target function node has already been folded: to whichever binding site was closest to it in the linear graph.

4. So GraphMol needs a new function node: `unfold`, with a new atom symbol: `%`. This unfolds a function node (the closest function node to the binding site that its target is bound to). When this function has been executed, the target function node will be unfolded (figure 6.9 (d)).

5. The physics part will fold the function node to its closest binding site, which will now be the target binding site (figure 6.9 (e) and (f)).

6. Depending on the parameters of the GraphMol physics engine, this process may take a little time to happen, so the chaperone chemical contains some junk to keep its program running and keep it bound to the target chemical.

7. After this junk region, the chaperone chemical contains function nodes that unbind it from the target chemical. When it executes these, it will unbind itself from the target chemical and leave the target chemical folded (figure 6.9 (g)).

Two technical details are needed before the chaperone chemical can operate successfully. Firstly, when the chaperone chemical unbinds itself from the target chemical, the function `> hides` its target binding site as well as unbinding it. So the chaperone chemical needs two extra `<` functions to `show` its binding sites after it has unbound itself. Secondly, we need to hide the `uxuxu` binding site on the target chemical after the refolding has happened. Otherwise, the chaperone chemical will keep binding to chemicals that are correctly folded and waste time. GraphMol chemicals cannot currently hide binding sites on other chemicals, but we can introduce another new function to do this: `HBT` (standing for *hide bound target*), with its corresponding new atom symbol: `&`. Figure 6.10 shows the full

```
[t1t1t] [mt1tm] ! junk
[t2t2t] [mu1um] junk
% [u2u2u] junk
junk junk junk junk
& [u2u2u] junk
> [u2u2u] junk > [u1u1u] junk
< [u2u2u] junk < [u1u1u] junk
```

Figure 6.11: The full chaperone chemical as a string of atoms, with all functionality and folding templates included. Junk is used to protect the folding templates and ensure correct folding. The `%` atom is the new *unfold* function; the `&` atom is the new *hide bound target* function. `mt1tm` is a pattern that matches (binds to) `t1t1t`. Whitespace is not meaningful — it is added here for readability only. For the folded version of this chemical, see figure 6.10.

chaperone chemical, with these additions. Figure 6.11 shows the full chaperone chemical in terms of atoms and folding templates.

This chaperone chemical introduces two new atoms into the design of GraphMol: *unfold*, `%`, and *hide bound target*, `&`. These atoms do not appear in the design of GraphMol above, because chaperone chemicals and embodied folding are not investigated experimentally in this thesis. The chaperone chemical is presented here to illustrate why GraphMol's parsing and folding process has been designed in the way it has. Also, the chaperone chemical shows the flexibility of the GraphMol language. New mechanisms can be obtained by modifying the chemistry slightly and exploiting the physics. We do not implement embodied folding by adding a complicated *fold* atom to the chemistry to fold chemicals with one function. Rather, we:

- add two simple, low-level functions to the chemistry (at a lower level than folding),
- re-use some existing functions of the chemistry (`show`, `hide`, `stop`),
- exploit some properties of GraphMol's physics (chemicals, graph distances, binding and unbinding).

This allows us to implement folding as an embodied process within the GraphMol world, with very minor modifications to the GraphMol chemistry and no modifications to the GraphMol physics. GraphMol is designed to be extended in this way

The chaperone chemical described here is able to fold *one* pair of folding templates, e.g. `u1u1u`/`t1t1t`, depending on which pair of templates its binding sites

match. But in general, GraphMol chemicals will need to contain more than one fold. This means that they will need more than one pair of folding templates. There are two solutions to this. One possibility would be to have different chaperone chemicals, with binding sites that matched different folding templates. Or alternatively, these different chaperone chemicals could be concatenated together into one (very) large chaperone. With an exact binding function (as used here), this would require a separate chaperone chemical (or a separate section of the large chaperone) for each pair of folding templates. For all practical purposes, this could be too much. Using an inexact binding function (such as Stringmol's, section 5.1.1.5), one chaperone chemical could match multiple folding templates. This redundancy would reduce the number of chaperone chemicals (or sections of the large chaperone) required.

An interesting consequence of embodying the folding process within chaperone chemicals is that chaperone chemicals also need to be folded. This raises the question of: *who folds the folding mechanism?* From a computer science viewpoint, this may seem like a circular reference that cannot be resolved. This is because traditional computer science thinks of algorithms existing in isolated worlds, where there is only one copy of each algorithm. But from a biological viewpoint, there is no problem. There is not just one single, isolated, chaperone chemical, but multiple chaperone chemicals existing in the same world. A given chaperone chemical does not need to fold itself; it folds a *copy of itself*. So a GraphMol system can be initialised with unfolded chaperone chemicals, as long as it is also bootstrapped with some pre-folded chaperone chemicals as well. This may seem like cheating, but it is not. Every biological system always starts its existence with a bootstrap. When a bacterial cell clones itself, the daughter cells do not just contain a copy of the DNA and re-create everything from scratch. Even though the DNA-reading machinery is coded for on the daughter cell's DNA, the daughter cell must be bootstrapped with some DNA-reading machines that have already been decoded, to start the process of reading the DNA. The same is true of GraphMol chaperone chemicals.

6.3 GraphMol reactions in detail

The previous section described in detail the *static* properties of GraphMol chemicals, including how they are parsed and folded. We have briefly introduced the *dynamic* processes involved in GraphMol chemical reactions: the binding process and the encoded computer programs. This section describes these in detail, explaining how these two processes allow GraphMol chemicals to change each other. The following section shows how we can use these processes to design GraphMol chemicals that embody the copying mechanism.

Once strings have been parsed and folded into chemical graphs, the graphs can start to react with each other. A GraphMol chemical can be described by its linear sequence of parsed nodes, and a collection of internal state information. The internal state of a chemical consists of the following different types of data:

- Which binding sites the chemical's `show` and `hide` functional atoms are connected to (as determined by the folding process). With a crisp folding process, this is fixed after the chemical has folded. With an embodied folding process, this can be changed by a chaperone chemical.
- Which of the chemical's binding sites are `shown` and which are `hidden`. This can be changed by the the execution of the computer programs encoded in GraphMol chemicals.
- Which of the chemical's binding sites are bound, and to whom. This can be changed by the binding process of the GraphMol physics engine.
- The number of instruction pointers moving along the chemical, and their current positions. New instruction pointers can be added by the binding process, and removed by the computer programs.

As GraphMol runs, the declarative binding process and the imperative computer programs change the internal states of the chemicals in the GraphMol world. The binding process uses binding sites to create instruction pointers, whereas the computer programs use instruction pointers to `show` and `hide` binding sites. Graphmol chemical reactions exploit the feedback between these two processes to implement mechanisms that embody phenomena.

6.3.1 Declarative binding process

The declarative physics process simulates the GraphMol chemicals moving in a physical space, and the binding sites being attracted to each other. In biology, chemical enzymes are physical objects that move through space and experience forces. The main forces that biological enzymes experience are: (1) impacts from their surrounding molecules (mostly water molecules), that cause the enzymes to move randomly; and (2) electrostatic attraction and repulsion between different (charged) parts of different chemicals.

In biology, any part of an enzyme can be positively or negatively charged. So to calculate the force on any given part of a chemical, many different charges (in the local area) need to be taken into account. Chemicals move around each other, reorienting their positions to minimise the energy implied by their (very complicated) electric fields. Any part of any chemical can potentially stick to any other part,

if their patterns of charges complement each other, and they manage to move together to align these complementary parts. Molecular Dynamics simulations capture these ideas computationally, but are very expensive to run.

GraphMol simplifies the above situation by using the abstraction of *binding sites*. GraphMol chemicals can only stick to each other (and to themselves) at certain, pre-defined positions: binding sites. This design decision has two consequences:

1. It simplifies the binding calculation, because every part of every chemical can no longer stick to every other. Only the binding sites of chemicals can stick together, so only the binding sites need to be considered when calculating binding.

2. It allows the functionality of a chemical (its imperative computer programs) to reference the binding sites as objects in their own right.

GraphMol's binding process is an analogue of electrostatics and the random movement of chemicals. It continuously monitors the binding sites that exist in a GraphMol world, creating binds between binding sites if their patterns match and the binding sites are close together. The process of creating a bind is described below, as is the process of determining if two patterns match. There are two different processes that stochastically search for binding sites that are *close together*:

1. The *mixing process* simulates the GraphMol chemicals moving randomly in space, and two binding sites happening to be close to each other. GraphMol uses an aspatial mixer, that stochastically determines which chemicals are close to each other at any given time.

2. The *graph distance process* takes account of the fact that binding sites that are close to each other in the same graph, would also be close to each other in physical space. This process creates binds stochastically between binding sites that are close together in terms of graph distance.

The mixing process brings together separate chemicals and allows them to bind (and hence react) in the same way as a traditional artificial chemistry; any artificial chemistry mixing algorithm could be used here. But in GraphMol, reactions are not atomic operations like they are in many artificial chemistries. The forming of a bind is atomic, as is the execution of one instruction of an imperative computer program. But after a bind has formed, and before the computer programs have finished executing, other binds can form and interact with the reaction. After a bind happens, the two chemicals are joined together by bind edges. This means that other binding sites from the two chemicals have now become close together. The graph distance process exploits this, allowing more binds to happen between

binding sites on the two chemicals, because they are close together through the graph.

Also note that the mixing process does not just bring together separate chemicals. It can bring together binding sites on the same chemical that are separated by a large graph distance.

6.3.1.1 Aspatial mixing process

As introduced in section 5.1.1.1, one thing an artificial chemistry world must specify is the type of space in which chemicals react. If an artificial chemistry has a non-trivial type of space, then its world can have a non-trivial mixing process as well. For example, if the chemicals were two-dimensional circles, then the physics engine's mixing process could simulate movement and collisions of two-dimensional circles in two-dimensional space. But equally, it could instead simulate these two-dimensional circles moving and colliding in a one-dimensional space, or a five-dimensional space.

Different definitions of space provide the artificial chemistry with different possibilities. Humans live in a three-dimensional (almost Euclidean) space and are familiar with its properties. But some properties that exist in three dimensions do not carry over into spaces with different numbers of dimensions. For example, in three dimensions you can tie knots in shoelaces (which are essentially one-dimensional lines), but in four dimensions these knots would fall apart and your shoes would slip off. In order to tie knots in four dimensions, you have to use two-dimensional planes (which is hard to visualise). But in two dimensions, you can't even get started tying a knot (using a one-dimensional line) because the space does not let you move the shoelace *over* itself. And in one dimension, you cannot even bend the shoelace to make it touch itself, let alone think about going further towards tying a knot. All you can do is slide the shoelace backwards and forwards along a straight line.

The number of dimensions is not the only property that space possesses. Another property is a definition of *distance*. This can have massive impacts on the computational properties of novelty-generation algorithms. For example, in Artificial Immune Systems using Euclidean distances[2] makes the algorithms perform increasingly poorly on higher-dimensional problems [95]. This issue can be resolved by using Manhattan distances[3]. The reason for this difference is that the volume of a hypersphere (Euclidean distance) does not grow as quickly as the volume of a

[2]Euclidean distance is the familiar notion of straight-line distance, calculated using Pythagoras' equation: $\mathrm{d}(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$, where n is the number of dimensions in the space.

[3]Manhattan, or *taxicab*, distance is named after the grid layout of most streets in Manhattan, where a taxi travelling between two points would drive on the edges of a rectangular grid, and cover a distance of: $\mathrm{d}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{n} |a_i - b_i|$.

hypercube (Manhattan distance) when the dimension of the space is increased.

The idea of an *aspatial* space is to remove as many of the complications of space as possible, while still allowing chemicals to react with each other. It is not clear how different spaces affect artificial chemistries, and complicated spaces take a long time to run. So aspatial spaces start from the minimum requirements on an artificial chemistry space, with the idea that further complications can be added later.

One major complication of space is *correlations* between chemicals. If two chemicals are close together at a certain time, then there is a high chance that they will still be close together after a short time has passed. Aspatial spaces remove this by not representing the space explicitly. Rather than keeping track of the positions of every chemical over time, aspatial spaces just store a list of all the chemicals in the world. At every timestep, the mixing process chooses stochastically which chemicals are *close to each other*. Literally, this does not make sense in terms of the physical world, but it makes sense mathematically and is easy to create within a computer. The physical-world analogy is of a well-mixed container that is being continually stirred to destroy the correlations.

Most aspatial artificial chemistries from the literature (described in section 2.4.2.2) use a very simple method for choosing which chemicals are close to each other each timestep. They stochastically pair up chemicals from the list, meaning that every chemical reacts with a (uniformly) randomly chosen partner every timestep. This works for chemistries with trivial binding models, where every chemical can bind with every other. But for chemistries with richer binding models, we can improve the mixing process by keeping track of which pairs of chemicals can bind, and only pairing up chemicals that are able to bind (and hence react).

We can extend the naive method of pairing up chemicals from the list, to include some spatial properties. In a well-mixed reactor in the physical world, chemicals are not paired up with each other every timestep. The chance of a particular reaction taking place at any time is a function of the *concentration* of the reactants in the world. The Stringmol artificial chemistry [43] replicates this by assigning an abstract concept of *volume* to the world and the chemicals within it. Each timestep, all the chemicals are randomly distributed throughout the world's volume. If the volumes of two chemicals intersect, then they are *close enough* to bind. (And if one chemical intersects with two others, then it binds to the closest one.) This can be implemented very efficiently. To test if a given chemical is close enough to bind with anything, we use the equation:

$$\text{probability of being close enough} = 1 - (1 - v_{\text{chem}}/v_{\text{world}})^n \tag{6.5}$$

where v_{world} is the volume of the world, v_{chem} is the volume of a single chemical, and n is the number of chemicals in the world that the target chemical can bind with.

Algorithm 4 Stringmol's aspatial mixing process (also used in GraphMol)

v_{world} = volume of the world
v_{chem} = volume of one chemical
W_t = the list of all chemicals in the world at timestep t
$W_{t+1} = \emptyset$

at timestep t, check which chemicals bind
for all chemicals, c, **in** W_t **do**
  remove c from W_t

  B = the list of chemicals in W_t that c can bind with
  $n = |B|$ *the number of chemicals in W_t that c can bind with*
  $p = 1 - (1 - v_{\text{chem}}/v_{\text{world}})^n$
  if random dice roll with probability p **then**
    chemical c binds with something

    d = random choice of chemical from B
    remove d from W_t
    form a bind between c and d
    add c and d to W_{t+1}

  else
    add c to W_{t+1}
  end if
end for

Algorithm 4 shows how this equation is used to implement Stringmol's aspatial mixing process. GraphMol uses this same method as its mixing process. GraphMol also optimises storage of the sets W, grouping together repeated instances of the same chemical (using multisets rather than lists). This speeds up the computation of B (and n), and speeds up most of the adding and removing.

Using this mixing process allows the binding process to take into account the *concentrations* of chemicals present in the world. This makes reactions happen more frequently when more of their reactants are present in the world, and less frequently when there are fewer. In combination with Stringmol's energy model (section 5.1.1.4) that allows variation in the number of chemicals in the world, this has been shown to produce interesting results in the Stringmol artificial chemistry [42]. This is why GraphMol uses Stringmol's mixing process and energy model.

Algorithm 5 Graph distance process testing for a single bind

```
B = set of all free binding sites in the world
filter B, keeping binding sites that are close to a matching, free target

if B is not empty then
  b = a random choice of binding site from B (uniform distribution)
  T = set of all free target binding sites that b is close to

  t = a random choice of binding site from T (uniform distribution)
  distance = the graph distance between b and t
  match = the strength of the match between b's pattern and t's pattern

  p = binding_probability_function(distance, match)
  if random dice roll with probability p then
    form a bind between b and t
  end if
end if
```

6.3.1.2 Graph distance process

Once the mixing process has brought together two chemicals, the graph distance process allows those chemicals to interact with each other by forming further binds between their binding sites. While the mixing process forms binds between arbitrary binding sites, the graph distance process forms binds between binding sites that are close together on the same graph. It is a stochastic process that is biased, with the probability of two sites binding based on their distance through the graph.

Algorithm 5 shows GraphMol's graph distance process, that chooses one binding site at random from the world and tries to find a nearby target binding site for it to bind with. During a GraphMol run, this process is continually iterated to form binds between binding sites that are close to each other on the same chemical. To optimise this process, the data structures B (filtered version) and T are cached, and updated when necessary, to avoid re-calculating them each time.

The probability of a given bind forming depends on two factors: (1) the distance between the two binding sites and (2) the affinity between the binding sites' patterns. In the current version of GraphMol, the matching function is crisp and trivial: two binding sites either match or they do not (exact string matching). It would be easy to add a rich, stochastic matching rule into GraphMol in the future (such as Stringmol's binding model based on inexact subsequence matching, section 5.1.1.5) . Because a GraphMol match is either true or false, it does not have a quantitative affect on the binding probability function (it only says whether a

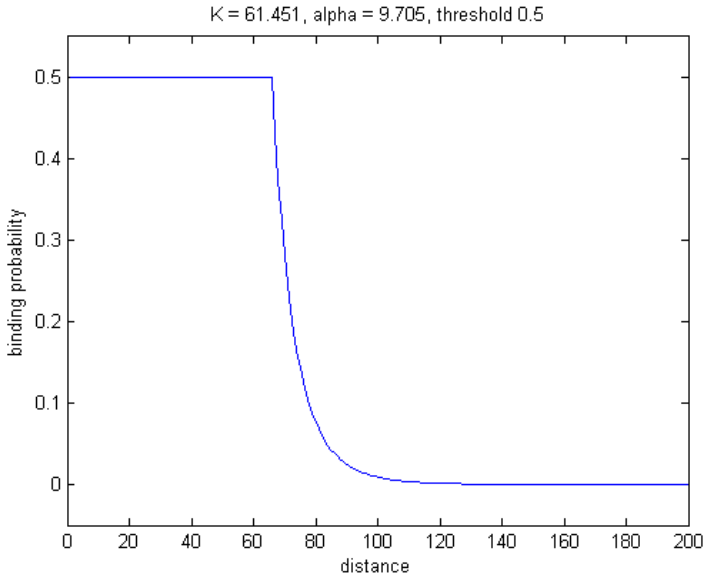


Figure 6.12: The binding probability function used in the current version of GraphMol.

bind can happen or not).

Any binding probability function could be used to convert distances into probabilities of binding. The current version of GraphMol uses the function:

$$\text{bind\_probability(distance)} = \begin{cases} \left(\frac{k}{\text{distance}}\right)^{\alpha} & \text{if this is } < 0.5 \\ 0.5 & \text{otherwise} \end{cases} \tag{6.6}$$

which is visualised in figure 6.12 for the parameters $k = 61.451$ and $\alpha = 9.705$. The particular form of this function was chosen to allow us to control the error rate of the embodied copying mechanism described later (see section 6.4.2.4 for details). Since all the GraphMol runs described in this thesis relate to the embodied copying mechanism, the above choice of binding probability function is appropriate. However, if different mechanisms were to be implemented within the GraphMol world, then a different function might be more appropriate here.

6.3.1.3 Calculating graph distances

In order to operate, the graph distance process must know the distances between each binding site in the graph. When binding sites create and break binds with each other, these distances will change. And if embodied folding were used, then these distances would change when chemicals were re-folded (this does not happen in the

Algorithm 6 Floyd-Warshall all-pairs shortest path algorithm

n = the number of nodes in the graph
D = an $n \times n$ table of distances between nodes in the graph
initialised with the distances between adjacent graph nodes

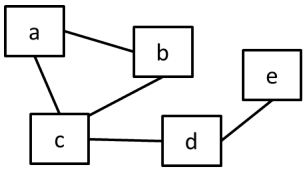
```
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      D[i][j] = min(D[i][j], D[i][k] + D[k][j])
    end for
  end for
end for
```

current version of GraphMol). One option would be to re-calculate the distances between binding sites on each iteration of the graph distance process. This would remove the problem of the graph structure changing between binds, but would be very inefficient to run. A more efficient method is to store the distances between binding sites and update them as binds change.

The standard algorithm for computing the shortest distances between all pairs of nodes in a graph is the Floyd-Warshall algorithm [33], described in algorithm 6. This looks at every triple of nodes (i, j, k) and asks the question: *Is it quicker to travel directly from i to j, or would I be better off going via k?* Iterating this question multiple times has the effect of calculating the lengths of the shortest paths between each pair of nodes in the graph. Using this algorithm to store the distances means we do not have to re-calculate them on every iteration of the binding process, but it does require us to re-calculate them all every time a bind is formed or removed. The problem with this algorithm (and similar ones, such as Dijkstra's algorithm) is that when a small change is made to the graph (such as adding or removing a bind edge), we must re-calculate all the distances between every pair of nodes, even if most of them have not changed.

To solve this problem, and only perform a small amount of re-calculation when a small change is made to the graph, GraphMol's distance process uses a modified version of the Floyd-Warshall algorithm, based on an algorithm used to route packets through the internet [37, pp.734-737][4]. Rather than having a global two-dimensional table, D, containing the distances between each pair of nodes, each node, x, has its own local one-dimensional table, D_x, containing the distances from node x to every other node in the graph. This breaks up the global distance table,

[4]Modern routers use a more sophisticated version of this algorithm, along with many other tricks, to route packets through the internet. The algorithm described here is an early routing algorithm based on the *distance vector routing protocol*.



D	a	b	c	d	e
a	0	1	1	2	3
b	1	0	1	2	3
c	1	1	0	1	2
d	2	2	1	0	1
e	3	3	2	1	0

(a) An example graph with its global distance table, D.

	a	b	c	d	e
D_a	0	1	1	2	3
N_a	-	b	c	c	c

D_b	1	0	1	2	3
N_b	a	-	c	c	c

D_c	1	1	0	1	2
N_c	a	b	-	d	d

D_d	2	2	1	0	1
N_d	c	c	c	-	e

D_e	3	3	2	1	0
N_e	d	d	d	d	-

(b) The corresponding local distance tables, D_x, with their *next node* tables, N_x.

Figure 6.13: An example showing the difference between global and local distance tables.

Algorithm 7 GraphMol routing algorithm for propagating a change (inspired by [37, pp.734-737]).

a small change is propagated throughout the whole graph

U is the set of nodes that have recently updated their distance tables
initialised to contain the nodes directly affected by the change

while U **is not** empty **do**
  U_c = a copy of U
  empty U
  shuffle U_c
  for all nodes, x, **in** U_c **do**
    for all neighbours, y, **of** node x **do**
      node x sends its distance table to node y (algorithm 8)
      if this caused node y to update its distance table **then**
        add node y to U
      end if
    end for
  end for
end while

distributing it between the nodes in the graph. Figure 6.13 shows this for a small example graph. As well as storing the distances to every other node, each node also stores the *next node* in the shortest path.

Because the distance table has been distributed, changes can be made to it only where they are needed. For example, in figure 6.13, if the edge between nodes a and b was removed, then this would affect the distance tables of nodes a and b, but not those of nodes c, d or e.

Algorithm 7 describes how to spread a change through the graph. We maintain a set, U, of graph nodes that have recently updated their distance tables. When a change happens (e.g. removing the edge between a and b), we add to U the nodes that are directly affected by the change (nodes a and b). In order to spread a change through the network, we iterate through U, removing each node and sending that node's distance table to all of its neighbours. Each neighbour integrates the distance table it receives into its own. If this integration causes the neighbour's distance table to change, then the neighbour adds itself to U, ensuring that the change will propagate to its neighbours as well. In the example above, node a (and also node b) will send its distance table to node c. Node c will examine node a's distance table and realise that the change does not affect node c. Thus, node c will not add itself to U, and hence nodes d and e will not be affected by the change.

Algorithm 8 GraphMol routing algorithm for updating a single node (called from algorithm 7).

node a sends its distance table to node b

D_x is the distance table for node x
$D_x[y]$ is the distance from node x to node y
N_x is the *next node* table for node x
$N_x[y]$ is the node that is next on the route from node x to node y

update every one of b's routes that goes via a first
for all nodes, c, **in** D_b **do**
  if $N_b[c] = a$ **then**
    calculate how long the route is, using a's new distance table
    new_$D_b[c] = D_b[a] + D_a[c]$
    if the distance has changed, then b updates its distance table
    if new_$D_b[c] \neq D_b[c]$ **then**
      $D_b[c] =$ new_$D_b[c]$
    end if
  end if
end for

check if b can make any of its routes shorter by going via a first
for all nodes, c, **in** D_a **do**
  check we are not making a circular route
  if $(c \neq b)$ and $(N_a[c] \neq b)$ and $(N_b[c] \neq a)$ **then**
    if a's route is shorter, then b uses it
    new_$D_b[c] = D_b[a] + D_a[c]$
    if new_$D_b[c] < D_b[c]$ **then**
      $D_b[c] =$ new_$D_b[c]$
      $N_b[c] = a$
    end if
  end if
end for

apply a cutoff distance
for all nodes, c, **in** D_b **do**
  if $D_b[c] >$ cutoff **then**
    remove node c from D_b and N_b
  end if
end for

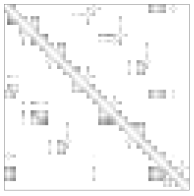
Algorithm 8 gives details about precisely *how* a node integrates a neighbour's distance table into its own. This is a distributed version of the Floyd-Warshall algorithm (algorithm 6). Figure 6.14 shows an example of the distributed algorithm running, gradually building up the distance table of a GraphMol chemical.

The benefit of distributing the distance calculation in this way is that it provides the GraphMol world with parameters that can be used to tune the distance calculation, and alter the effect it has on GraphMol chemicals. As can be seen in algorithm 8, it is straightforward to add a cutoff distance into this algorithm. The cutoff parameter stops the algorithm calculating distances between nodes that are very far apart. Since the binding process uses these distances to form binds with a greater probability between nodes that are close together, we can safely ignore distances over a high-enough cutoff. Technically this will prevent some binds happening that could have happened without the cutoff, but we can calculate which binds we will lose in this way. Using equation 6.6, we can calculate that using a cutoff of 250 units, we will lose all binds with a probability of less than 4.1×10^{-11}. Unless we were to run the GraphMol world for an inordinately long time, we would be extremely unlikely to see any of these binds happen, and we gain performance by not considering them.

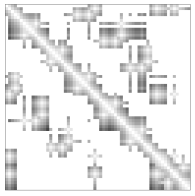
The set, U, of nodes that have to send their distance tables, provides many opportunities for variation and the creation of parameters. U is a rather complex data structure. It is ostensibly a straightforward set of nodes, but the way in which it is used makes it more involved than this. Nodes are removed from U, and they send their distance tables to their neighbours. But sending their distance table may cause further nodes to be added to U. So when removing a node from U, we may cause U to grow. And in removing further nodes from U, we may cause the addition of nodes that have been removed on previous iterations. Thus, statements such as *process every node in* U are not well defined.

In terms of ending up with correct distance tables, any method of iterating through U will converge on the same distance tables when U eventually becomes empty (and U will always become empty, provided the network is only changed once). But one benefit of this algorithm is that we do not need to wait for the distance tables to converge before doing other things in GraphMol. For example, looking at figure 6.14, after only 4 iterations, a good proportion of the nodes have most of their distances calculated. And after 6 iterations, almost all of the distances are correct. At this point, we could stop the distance-propagation process and start the binding process. The binding process would have a very high probability of forming the same binds as if we had let the distance-propagation process finish. This provides us with a parameter for the GraphMol world: the number of iterations we perform on U (the speed of the distance-propagation process) before performing one iteration of the binding process. This is a classical

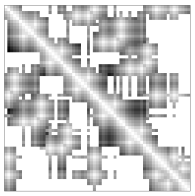
(a) Neighbour distances after initialisation



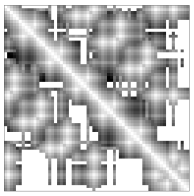
(b) After 1 iteration (63 nodes in U)



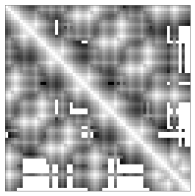
(c) After 2 iterations (61 nodes in U)



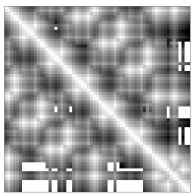
(d) After 3 iterations (63 nodes in U)



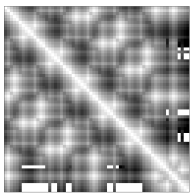
(e) After 4 iterations (52 nodes in U)



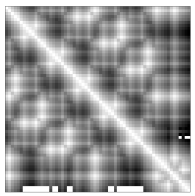
(f) After 5 iterations (51 nodes in U)



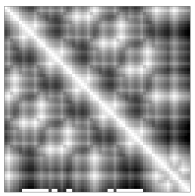
(g) After 6 iterations (34 nodes in U)



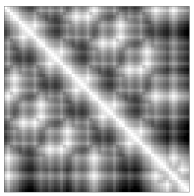
(h) After 7 iterations (31 nodes in U)



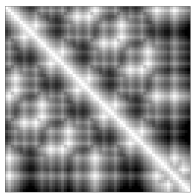
(i) After 8 iterations (20 nodes in U)



(j) After 9 iterations (6 nodes in U)



(k) After 10 iterations (1 node in U)



(l) Distances computed by Floyd-Warshall algorithm

Figure 6.14: An example of how the distributed algorithm gradually builds up its distance table and converges on that computed by the Floyd-Warshall algorithm. Darker shades equate to longer distances. For this graph (with 63 nodes), the algorithm takes between 8 and 14 iterations to converge.

speed-accuracy tradeoff. Running the distance-propagation process faster provides the binding process with a more accurate estimate of the distances, at the expense of processing time.

Algorithm 7 describes one method of iterating through U. During each iteration of the routing algorithm, a copy is made of U, and this copy is iterated through. Any nodes added to U during an iteration are not added to the copy being iterated. When repeated, this provides a *breadth-first* traversal of the data structure implied by U. Algorithm 7 iterates through the copy of U in a random order (by shuffling U_c). Iterating in the same order each time would cause artefacts to spread through the distance tables, making them fill out unevenly.

6.3.2 Imperative computer programs

The imperative computer programs are encoded within GraphMol chemicals. The binding sites and function nodes of a GraphMol chemical define crisp computer programs (one program for each binding site) and the junk nodes control the speed at which these computer programs execute. When a bind happens, two programs start executing: one starting from each binding site.

This programming language is completely crisp (it contains no stochastic elements). GraphMol's stochasticity is contained within its physics process; the crisp function nodes form a programming language that can exploit the stochasticity of the physics process to implement interesting computer programs capable of embodying phenomena of interest.

This section describes in detail how the computer programs execute, and discusses how different design decisions might impact the potential evolvability of GraphMol programs.

6.3.2.1 Instruction pointers

When the binding process forms a bind between two binding sites, it creates an instruction pointer at each binding site. These two instruction pointers start executing in parallel, moving along their respective chemical graphs' program edges (which have a fixed direction, determined when the chemical is created). Each time the instruction pointers are iterated, they move along one atom of the chemical. This means that when they reach nodes containing multiple atoms (junk regions and binding sites), it takes the instruction pointers multiple iterations to move over these nodes. This is how junk regions can be used to slow down execution of GraphMol programs.

When an instruction pointer reaches a junction in the graph, it moves along the graph's program edge, rather than along any fold or bind edges. Although the fold edges transform a linear string of nodes into a general graph, and the

bind edges connect together different graphs, the instruction pointers ignore all of this. To an instruction pointer, a GraphMol chemical is simply a linear sequence of atoms, as described by the graph's program edges. The fold edges could be used to implement *jump* instructions in GraphMol, and the bind edges could be used to allow one chemical to execute code from *another chemical* (as can happen in Stringmol [43]), but these have not been investigated in this thesis (see the future work section for a discussion of these). The fold and bind edges do, however, affect the binding process, because they alter distances through the graph (and hence the probabilities of binding).

When an instruction pointer moves to a function node, it executes this function. In the current version of GraphMol, there are just three functions: `show <`, `hide >` and `stop !`. These are described below. Clearly, some type of *copy* function is needed to implement a copying mechanism. This is added later, since the focus of GraphMol is on embodying the `next` function of the copying process, rather than `char-copy`. GraphMol has been designed so that adding new functions is easy. The folding process makes it possible to add any function that takes a single binding site as a parameter. Functions without parameters can be added also. For example, the description of the chaperone folding chemical introduces two new functions that both take a single parameter (section 6.2.3.4). The future work section shows how new instructions could allow GraphMol to embody the copying process in ways different from those investigated in this thesis. As with CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) processor architectures respectively, different versions of GraphMol could be designed that either (1) used many different functions covering all conceivable requirements, or (2) used a bare-minimum of functions, that could be combined to implement more complex operations.

When an instruction pointer reaches a stop instruction, `!`, or if it reaches the end of the chemical, the program stops executing and the instruction pointer is removed from the chemical's internal state.

6.3.2.2 Show and hide functions

As explained in the folding section above (section 6.2.3), the `show` and `hide` functions each take one binding site as a parameter: the particular binding site they `show` or `hide` when an instruction pointer executes them. This parameter is specified via a fold edge connecting the function node to the binding site. If, during the folding process, this fold edge is not created, then the function node essentially turns into a junk node, because it has no binding site to operate on.

The effects of `showing` and `hiding` binding sites are changes to the topology of the chemical graph, which affects the future behaviour of the chemical. `Showing` a binding site makes it possible for the `shown` binding site to bind and start an

imperative program running. `Hiding` a binding site makes it impossible for the `hidden` binding site to bind (until it is `shown` again). We have one process that makes certain things possible, and another process that makes certain things impossible. This allows us to perform computation. Later, we will see GraphMol's embodied copying mechanism use this to decide which characters to copy.

6.3.2.3 Timescales

GraphMol has four different processes operating on different timescales:

1. the distance-propagation process (quickest timescale),
2. the instruction pointer process,
3. the binding process,
4. the imperative programs `showing` and `hiding` binding sites (slowest timescale).

The fastest of these is the *distance-propagation process*. Every iteration of this process performs one routing-table update, taking some nodes from those that need updating and sending their distance tables to their neighbours. On a slower timescale, the *instruction pointer process* iterates, moving each instruction pointer one atom along its chemical. The *binding process* operates on a slower timescale again. One iteration of this process chooses a certain number of binding sites at random, and tests to see if they bind to anything.

The binding process must happen on a slower timescale than the distance-propagation process, so that information about distance-changes can spread through the graph before new binds are formed. The instruction pointer process should happen on a quicker timescale than the binding process, because the instruction pointer process changes what binds are possible (by `showing` binding sites). If the binding process happened on too quick a timescale, then it would waste time checking for new binds when none were possible.

Finally, there is a process that is not programmed into the GraphMol world, but rather programmed into the computer programs encoded on GraphMol chemicals. To implement a useful mechanism, a GraphMol chemical will have to *show* and *hide* its binding sites in some pattern, causing them to bind to other sites in a useful way. This *showing* and *hiding* of binding sites will have to happen on a timescale slower than that of the binding process, because the purpose of *showing* binding sites is to allow them to bind to something. But this process is implemented by instruction pointers, that operate on a quicker timescale than the binding process. This is where junk comes in. GraphMol programs contain junk so that they can effectively slow down their instruction pointers when they need to (when they are

waiting for binds to happen), and speed them up at other times (when they want to `show`/`hide` multiple binding sites before a bind happens).

For the experiments presented in this thesis, one *GraphMol iteration* corresponds to one iteration of the distance-propagation process, which processes 10% of the nodes currently in U. The instruction pointer process runs once every 31 iterations, moving each instruction pointer along one atom. The binding process runs once every 71 iterations, choosing one binding site on each chemical that could bind, and testing if it does bind. These parameters were chosen by testing each permutation of the three delays (from 1 to 91 in steps of 10) using the GraphMol copying mechanism (described later). Some parameter combinations prevent the mechanism from functioning, but there is a large region of parameter space in which the mechanism works, with different degrees of reliability (probability of making the correct binds) and speed (runtime of the GraphMol world). The chosen parameters trade off these two factors, to give a GraphMol world in which the copying mechanism can be implemented reliably, while avoiding obvious inefficiencies that would slow down execution of the GraphMol world. If other mechanisms were to be implemented in GraphMol, then these parameters may benefit from checking.

6.3.2.4 Uniqueness of programs

When a bind is formed between two binding sites, two instruction pointers are created. If the two binding sites are on the same chemical, then there will be two programs executing in parallel on the same chemical. These are different computer programs, since they each have their own instruction pointer, and each instruction pointer starts from a different place on the chemical (a different binding site). Also, these programs will always remain different from each other, i.e. their instruction pointers will never coincide. This is because all instruction pointers move along their chemicals at the same speed, and there is no jumping possible in the GraphMol language (although jumping could be added in the future, with a new instruction).

Two instruction pointers could coincide, if two separate binds happened with precisely the right timing. One of the instruction pointers created by the first bind would have to move along its chemical and reach another binding site. This binding site would have to bind to a suitable target at the same time as the instruction pointer reached it. Then, we would have two instruction pointers in precisely the same place. These two would move along their chemical at the same rate, and would always overlap. Because all current GraphMol functions (`show` and `hide`) are idempotent[5], the second instruction pointer would be completely redundant. It

[5]A function is idempotent if applying it twice has the same effect as applying it once. A function, f, is idempotent if $\mathrm{f}(\mathrm{f}(x)) = \mathrm{f}(x)$ As an example: stopping a video playing is idempotent, but turning up the volume is not.

would technically execute every instruction it reached, but this instruction would have already been executed by the first instruction pointer. Idempotency is not a requirement of GraphMol functions, we just note here that the current GraphMol functions are all idempotent. Non-idempotent functions could be added, for example, a function that toggled whether its binding site was `shown` of `hidden`.

As an optimisation of the GraphMol instruction pointer process, duplicate instruction pointers could be detected and removed. But running instruction pointers is very cheap computationally. So the cost of detecting duplicates may be greater than the time lost by running them. And the computational gains of removing duplicates are small.

Since the binding process is stochastic, the fact that a duplicate binding site appears at a given time is no guarantee that another duplicate will appear in the future. If the bind had (stochastically) happened a little earlier or a little later, then the duplicate would not have occurred. An *almost-duplicate* instruction pointer would be present in this case. This would closely follow its almost-duplicate, executing duplicate instructions very soon after they had first been executed. This would most likely have the same effect as an *exact* duplicate, because the state of the chemical is not likely to change between the duplicate executions in such a way as to make the duplicate instruction meaningful. But it might do. There is a small chance that another process could change the state of the chemical in the time between the almost-duplicate instruction pointers executing the same instruction. In this case, the duplicate execution of the instruction could have a major change on the chemical, disrupting other processes or allowing something else to happen.

This type of potential for change is what we are trying to allow our systems to do. We want processes that happen in the same way most of the time, but occasionally do different things. And the different things should come not from an external source perturbing the system, but from a process internal to the system. This is why we do not want to remove almost-duplicate instruction pointers. Indeed, we want to encourage them (and phenomena like them), and see what effects they have on the evolvability of systems containing them.

6.3.3 Types of reaction

The previous two sections described in detail that implementation of GraphMol's various processes. This section discusses how those processes fit with the traditional notion of artificial chemistries.

Traditional artificial chemistries define *chemicals*, that participate in *reactions*. GraphMol has a clearly-defined notion of *chemical*: a graph built from nodes, program edges and fold edges. But in GraphMol there are two different concepts of *reaction*, at two different scales:

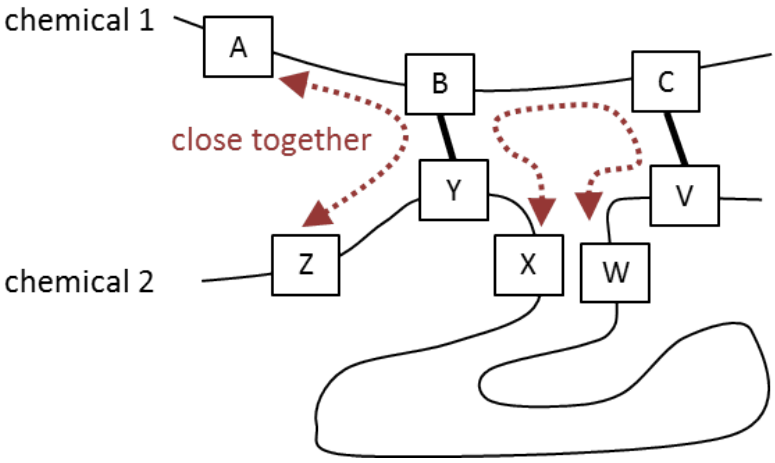


Figure 6.15: Two chemicals binding can bring together other binding sites from the two chemicals (A and Z), and also distant binding sites on the same chemical (X and W).

1. A *micro-scale reaction* is an interaction between two binding sites.
2. A *macro-scale reaction* is an interaction between two chemicals. This can either be designed into the chemicals, or be an emergent property of the system.

Micro-scale reactions are the mechanics of how GraphMol runs, and are programmed into the GraphMol world.

Macro-scale reactions are combinations of low-level events occurring in the GraphMol world, that have a well-defined outcome. They are *mechanisms* that can be built from components existing in the GraphMol world. And in principle, they could emerge from an evolving GraphMol system.

6.3.3.1 Micro-scale reactions

In the micro-scale case, a *reaction* is the same as a bind between two binding sites.

The immediate result of this type of reaction is a change in the topology of the graph, as shown in figure 6.15. The two chemicals are now connected together, so the distances between their binding sites have changed. Each binding site now has some new binding sites that it has become close to, namely those on the other chemical (in figure 6.15, sites A and Z have become close). Also, two binding sites on the same chemical may have become closer together. This could happen if the two chemicals were bound together in multiple places, and there was a shorter route between the two binding sites through the bound chemical than through the chemical they belong to (in figure 6.15, sites X and W have become close).

A longer-term result of this reaction is that when the bind formed, two computer programs started running, represented by the two instruction pointers that were created. If another bind happens before these programs finish running, then further programs start executing in parallel. Also, if the two binding sites are on the same chemical then there will be two programs running in parallel on the same chemical.

This definition of *reaction* views a GraphMol system as a collection of nodes in a graph. GraphMol updates these nodes by continually iterating the instruction pointers that exist (running the programs), and checking if any new binds happen (starting new programs). As the programs run, new binding sites can become visible and so new binds can happen. On the micro-level, GraphMol can be viewed as an artificial chemistry in which the chemicals are graph nodes and the reactions between them are binds. The artificial chemistry runs the physics of the world (the binding process), and there is also a higher-level chemistry running in parallel (the instruction pointers). We have a low-level artificial chemistry with a higher-level imperative language operating alongside. But this is not the only way of viewing GraphMol.

6.3.3.2 Macro-scale reactions

In the macro-scale case, a *reaction* is not defined explicitly as part of the GraphMol world: instead, it is a property of a running system. This can be an emergent property, produced by an evolutionary system. But in order to bootstrap evolutionary systems, we can design macro-scale reactions by hand-crafting GraphMol chemicals.

As an example, we can imagine designing two chemicals that interact with each other to perform a *reaction*. For example, one chemical copies the other, creating a third chemical as a product of the reaction. (We do not currently have enough function nodes in GraphMol to implement this, but for this example we can imagine that a future version of GraphMol does.) We can equip each chemical with a binding site whose pattern matches the other. We can set up the internal states of these chemicals so that only the two matching binding sites are `shown` (the rest being `hidden`). When we put just these two chemicals into a GraphMol world, they will bind and start executing their programs. The execution of their programs might cause other binding sites to become `shown` and other binds to happen, but eventually all the programs will stop and no more binds will be possible. The individual programs cannot go into an infinite loop, since they execute along the program edges of a finite, linear graph. The chemicals could go into an infinite loop of binding, un-binding and re-binding, but we assume not for this example.

We can think of this whole process as one *reaction*, and the system now looks like an artificial chemistry in which the chemicals are GraphMol chemicals and the reactions are complicated. The products of this type of reaction are whatever

is left in the GraphMol world when the chemicals finish interacting (and possibly unbind). In contrast to most artificial chemistries, the world in which GraphMol reactions happen is the same as the world in which the chemicals exist. In other words, GraphMol reactions are embodied within the same world as GraphMol chemicals. This gives GraphMol some advantages over other artificial chemistries:

1. Other chemicals can interfere with the process of a reaction.
2. Reactions involving more than two chemicals are possible.
3. Different reactions can take different lengths of time, without this needing any special treatment.
4. GraphMol can handle reactions that go into an infinite loop, without them needing any special treatment.

6.3.4 Summary of GraphMol's processes

GraphMol is more complicated than most artificial chemistries. Figure 6.16 shows a summary of the different processes that operate concurrently, on different timescales, to implement the GraphMol world. This section has described these processes in detail. The following section shows how these processes can be used to implement a mechanism that embodies the phenomenon of copying within the GraphMol world.

6.3.4.1 Time complexities

Since performance is an issue for GraphMol, it is useful to consider the time complexities of the different components of GraphMol. The are two separate stages to GraphMol: initialising a chemical (parsing and folding); and running the world (declarative binding and imperative programs).

When initialising a chemical the important variable is the length of the chemical (measured in atoms), which we denote by l. Parsing a chemical (section 6.2.2) is a straightforward linear process, requiring $\mathrm{O}(l)$ time. Folding (section 6.2.3) is lightly more complicated, involving a search through the chemical for matching fold sites. Searching the whole chemical for a matching folding site would require $\mathrm{O}(l)$ time, so the worst-case time complexity of the folding process is $\mathrm{O}(lf)$, where f is the number of folding sites in the chemical. In the worst case this is $\mathrm{O}(l^2)$, but in practical chemicals there will be fewer folds than there are binding sites in the chemical. In chemicals that make use of their folds, then searching for matching folding sites will be much faster than the worst case. So the time complexity of initialising a chemical is $\mathrm{O}(l^2)$ in the worst case, which is negligible compared to running the world.

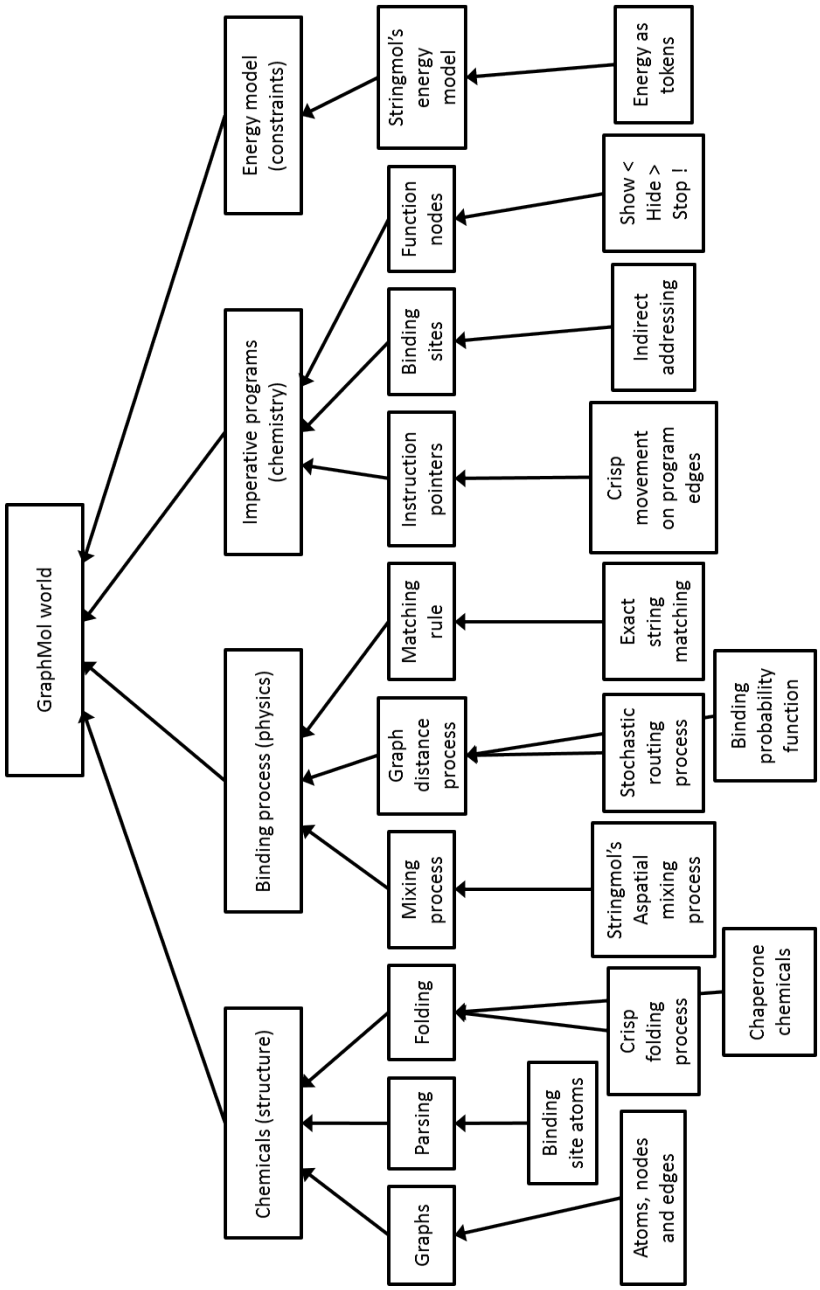


Figure 6.16: A summary of the different processes comprising the GraphMol world.

Running a GraphMol world requires four different processes operating on different timescales, as detailed in section 6.3.2.3. Since these four processes operate independently and in parallel, we can calculate their time complexities independently. Note that calculating the time complexity of a *phenomenon* embodied within a GraphMol world (such as copying) would be very difficult. The phenomenon relies on these four processes interacting appropriately over a period of time. Here, we are calculating the time complexities of the different components of the GraphMol *world*.

The slowest of these four processes is: *the imperative programs* `showing` *and* `hiding` *binding sites*. This process is implemented within the imperative programs of GraphMol chemicals, rather than implemented by the code that makes up the GraphMol world, so we cannot calculate its time complexity (because it can be changed by evolution).

The next process is the *binding process* (section 6.3.1), which is a combination of the *mixing process* and the *graph-distance process*. The important parameters here are the number of chemicals in the world, which we denote by n, and the number of binding sites in a chemical, which we denote by b.

The mixing process (algorithm 4) iterates through all the chemicals in the world, checking if any pairs of chemicals bind. Naively, one might expect this process to be $\mathrm{O}\big(n^2\big)$, but it is more efficient than this. Because GraphMol caches information about which pairs of chemical species can bind, every combination does not need to be tested every time. Thus the mixing process takes $\mathrm{O}\big(n\big)$ time.

The graph distance process (algorithm 5) iterates through all binding sites in a chemical (or in two chemicals, if they are bound together), checking if any pairs of binding sites bind. Similarly to the mixing process, the graph distance process does not need to re-compute the graph distances between each pair of binding sites each time. Thus, to test all binding sites in a chemical would require $\mathrm{O}\big(b\big)$ time. But in GraphMol, we do not test all binding sites on each iteration of the graph distance process: we pick one at random and test it, requiring $\mathrm{O}\big(1\big)$ time. This allows finer control of the interleaving of these processes (as explained in section 6.3.2.3).

The *instruction pointer process* (section 6.3.2) increments every instruction pointer on every chemical. Incrementing a single instruction pointer is very simple, requiring only $\mathrm{O}\big(1\big)$ time. For the GraphMol chemicals investigated in this thesis, each chemical creates a maximum of one instruction pointer per bound binding site, because each imperative program terminates before its binding site unbinds. This gives the instruction pointer process a time complexity of $\mathrm{O}\big(nb\big)$. In principle, GraphMol chemicals could bind and unbind rapidly, creating many instruction pointers that would need iterating. The worst case would be chemicals composed entirely of binding sites, that all bound to each other as often as possible. This would give the instruction pointer process a worst-case time complexity of $\mathrm{O}\big(nbl\big)$.

The final process, with the greatest time complexity, is the *distance-propagation process* (section 6.3.1.3). This updates the distance tables of each GraphMol node, successively updating the cache that approximates the distances between nodes in the graph. The important new parameter here is the number of binding sites that are within the cutoff distance of a binding site, which we denote by b_c.

One step of updating the distance table for a single node (algorithm 8) requires iterating through all the entries in both nodes' distance tables, which is $\mathrm{O}\left(b_c\right)$. Calculating the time complexity of the rest of this algorithm (algorithm 7) is more complicated. At any time during algorithm 7, the data structure U can contain at most b nodes (only considering one chemical). So the outer `for` loop in algorithm 7 executes at most b times. The inner `for` loop executes a constant number of times, since each node has a constant number of neighbours. The most complicated part is the `while` loop, since this iterates over a data structure that is changed by the loop body. The worst-case number of iterations of this `while` loop occurs for a completely linear graph, where each iteration of the `for` loop adds a new node to U. In this case, the `while` loop executes b times. But in this case the `for` loop only executes once, since U contains only one item. So the theoretical worst-case execution time of algorithm 7 is $\mathrm{O}\left(b^2 b_c\right)$. Considering multiple chemicals, this makes the theoretical worst-case execution time of the distance-propagation process $\mathrm{O}\left(nb^2 b_c\right)$. But the actual execution time will always be lower than this, since the worst-case scenarios for the `while` loop and the `for` loop cannot coincide.

Using a naive distance algorithm such as the Floyd-Warshall algorithm (algorithm 6) would give the distance-propagation process a time complexity of $\mathrm{O}\left(nb^3\right)$ *in every case*. The benefits of GraphMol's complex distance-propagation process are:

1. A lower worst-case time complexity.

2. A much lower average-case execution time.

3. The ability to interleave the distance calculation with other GraphMol processes, and always have an approximation to the 'true' distance.

The time-complexity of the all-pairs shortest path problem has (surprisingly) not been determined precisely, but no solutions with quadratic time complexity have been found [16]. GraphMol's distance-propagation process manages to attain a quadratic time complexity of $\mathrm{O}\left(b^2 b_c\right)$ because it uses a *cutoff* distance. The number of binding sites within the cutoff distance, b_c, is a parameter of GraphMol that we can use to change the performance of the distance-propagation process. Without a cutoff, b_c becomes b and we have a (theoretical worst-case) cubic time-complexity. With a cutoff, the time complexity becomes quadratic multiplied by a constant that we can control.

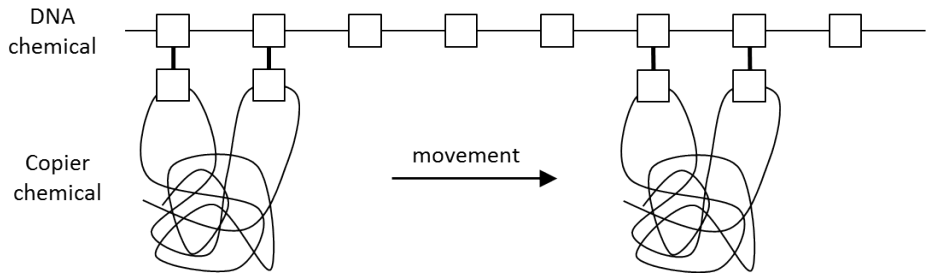


Figure 6.17: The copier chemical moves along the DNA by binding its binding site *feet* to a series of binding sites on the DNA.

This shows that the distance-propagation process is the most expensive part of GraphMol. And further, it shows that this process is not expensive because of an inefficient implementation, but because the problem is genuinely intensive to solve. The next section shows how GraphMol's embodiment of the copying process relies on this complex definition of *space* and *distance* within GraphMol reactions.

6.4 Embodying copying in GraphMol

This section describes a mechanism implemented in the GraphMol world: a chemical that can perform an embodied copy macro-reaction. The following two chapters describe experiments using this mechanism, investigating how this embodiment affects the mutations performed by the copying mechanism.

As introduced in section 5.2, this mechanism embodies the `next` function of the copying phenomenon: moving on to the next character of the template. The GraphMol world has been designed so that we can implement this mechanism within a GraphMol chemical. Binding and program execution change the topology of chemical graphs by `showing`, `hiding` and connecting binding sites. We can use this to make one chemical graph move, relative to another. Figure 6.17 shows a long, linear *'DNA'* chemical composed of binding sites separated by regions of junk. This chemical contains no function atoms, so will not change its own topology. A second, smaller, *copier* chemical *walks* along the DNA by alternately `showing` and `hiding` its six binding site *feet*.

The idea of a small chemical moving along a long, linear chemical is analogous to the way in which DNA is copied in biology. DNA is a long linear chemical. The chemical *DNA polymerase* moves along the DNA and copies it base-by-base (as described in section 4.2.1). The actual biology is much more complicated than this, and uses a slightly different mechanism (DNA polymerase wraps around the DNA in three dimensions, rather than walking with feet), but abstracting the process

of *moving base-by-base along the DNA* allows us to implement an embodied copy operation using different mechanisms.

Furthermore, many chemicals in biology move along DNA or RNA chemicals (for purposes other than copying). For example (see any biology textbook for details, for example [15]): helicases unwind the two strands of DNA, ligases glue together sections of DNA and ribosomes transcribe RNA into protein. All of these chemicals are *encoded in the DNA* on which they operate. For this reason, the DNA chemical is much longer than any individual chemical that operates on it (by many orders of magnitude). So these chemicals have to move along the DNA, rather than processing it in one go.

If we are interested in building computational analogies of biology, then movement of one chemical along another is a useful type of process to have in general, and has biological uses beyond copying.

In terms of the decomposition of the copying process described in section 5.2, the GraphMol copying mechanism comprises:

1. The *start* function: implemented as a program on the copier chemical, and as a pair of matching binding sites on the copier chemical and DNA chemical. Partially embodied.

2. The *at-end* function: implemented as a program on the copier chemical, and as a pair of matching binding sites on the copier chemical and DNA chemical. Partially embodied.

3. The *char-copy* function: implemented as a crisp process attached to the copier chemical. Not embodied at all.

4. The *next* function: embodied to a high degree within the interaction of multiple programs on the copier chemical and many binds between the copier chemical and DNA chemical.

6.4.1 GraphMol DNA chemical

Biological DNA stores information as a sequence of *bases* attached to a *backbone*. Figure 6.18 shows that the backbone is a linear structure composed of a small subchemical that repeats. The information carried by the DNA is held in a sequence of bases attached to this backbone. Two backbone-base structures bind together, and their physical shapes and charges make them form a three-dimensional double-helix shape. The backbone is a regular pattern that gives the DNA chemical its structure, and the bases are an irregular pattern that carries the information. Note that it is not the *position* of the bases that is irregular: the bases are always positioned in a regular pattern. It is the choice of *which base* (A, C, G, T) is in

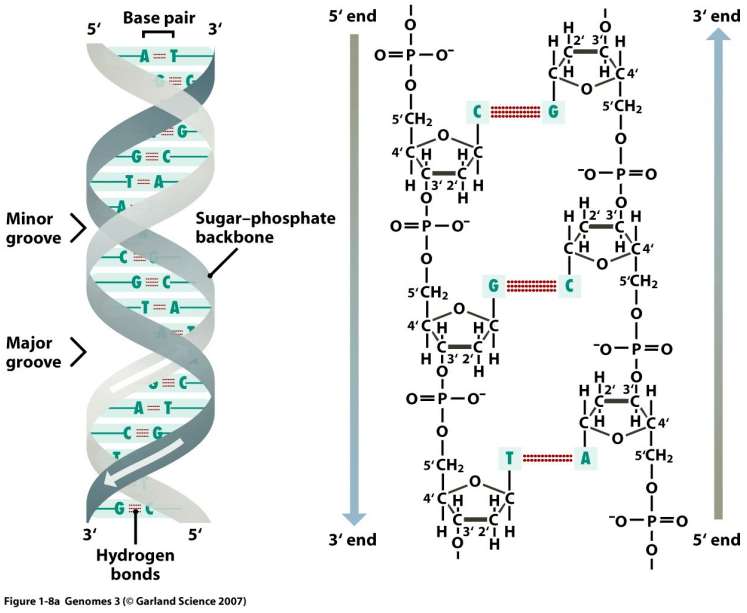


Figure 6.18: Biological DNA is a three-dimensional double-helix structure comprised of a common backbone with bases (information) attached. Figure 1.8a from [15]

which position that is irregular, and hence carries the information. The backbone also gives the DNA chemical a *direction*, allowing the bases of information to be read in a specific direction. (The two ends are called 3′ and 5′ in biology, because of the chemical structure of the backbone at the different ends.)

GraphMol DNA is an abstraction of biological DNA, implementing the ideas of a regular backbone and irregular bases. It uses the benefits and constraints of the GraphMol world, rather than the physical world. Figure 6.19 shows the GraphMol DNA chemical. GraphMol does not have three-dimensional space, as the physical world does, and GraphMol has explicit binding sites. So the GraphMol DNA chemical is a linear sequence of binding sites representing the backbone and bases. The patterns of the backbone binding sites are regular, and indicate which is the *left end* (`lllll`) of the DNA and which is the *right end* (`rrrrr`). The patterns of the base binding sites are irregular, and are either `aaaaa`, `bbbbb`, `ccccc`, or `ddddd`, giving GraphMol DNA four bases: A, B, C and D, analogous to the four biological DNA bases. GraphMol DNA is single-stranded, as opposed to biological DNA that is double-stranded. From an information-carrying viewpoint, there is no difference between single and double stranded DNA. The double-stranded nature of biological DNA is important in the mechanism of recombination (see section 4.6), so if GraphMol were to be used in the future to implement recombination, then

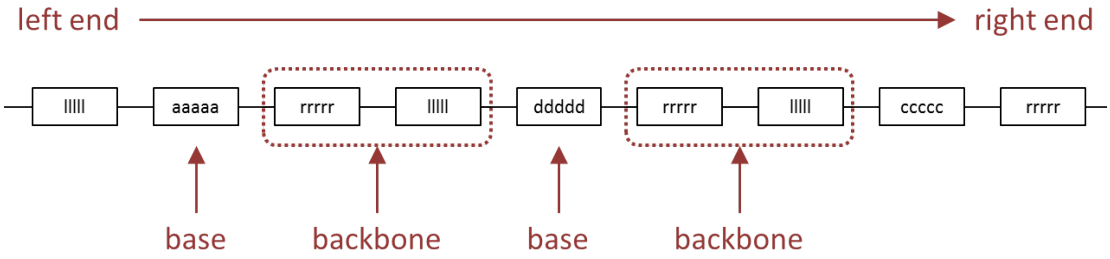


Figure 6.19: GraphMol DNA is a linear structure comprised of *base* binding sites and *backbone* binding sites, separated by regions of junk. The bases each have a pattern of either `aaaaa`, `bbbbb`, `ccccc`, or `ddddd`

```
[start] junk
[lllll] ! junk [XXXXX] ! junk [rrrrr] ! junk
[lllll] ! junk [XXXXX] ! junk [rrrrr] ! junk
...
[lllll] ! junk [XXXXX] ! junk [rrrrr] ! junk
[lllll] ! junk [XXXXX] ! junk [rrrrr] ! junk
[stop]
```

Figure 6.20: The GraphMol DNA chemical as a sequence of atoms (whitespace added for readability only). The `XXXXX` binding sites are the bases carrying the information, with a pattern of either `aaaaa`, `bbbbb`, `ccccc`, or `ddddd`. The junk regions are all of a fixed length (in this case, 40 atoms). The DNA chemical can contain an arbitrary number of repeated base–backbone sections.

designing a double-stranded DNA chemical would probably be useful.

Figure 6.20 shows the complete GraphMol DNA chemical as a sequence of atoms. To allow the copying chemical to begin at the start of the DNA, and finish when it reaches the end, the DNA chemical has a `start` binding site at the beginning of the chemical and `stop` binding site at the end. This allows us to program the copy operation as a *macro-scale reaction*, as described in section 6.3.3.2.

The stop atoms (`!`) are for efficiency, to remove the instruction pointers created on the DNA when binds occur. They are not strictly necessary, but without them the GraphMol chemistry process would spend a lot of time iterating instruction pointers along the DNA that are never going to execute a function atom. We have said previously that we are looking for this kind of redundancy in our mechanisms, to make them more evolvable, but the DNA chemical is not the correct place to put this redundancy. The DNA chemical does not evolve; what evolves is the pattern of bases on the DNA. Thus we need to put redundancy and evolvability into the

copier chemical, rather than the DNA chemical.

The junk regions add distance between the binding sites, which controls the probability of the copier chemical binding to different sites. The probability of two sites binding depends on their graph distance, so the probability of the copier chemical making a *correct* step when it moves (rather than stepping forwards two bases, or backwards) depends on these distances. The junk in the DNA chemical must be long enough that the walker has a low chance of making an erroneous step. But if it is too long, then the chance of the copier making *any* step will be very low. This would still allow the copying mechanism to function, but it would function very slowly. The probability of the copier making an erroneous step also depends on the copier's junk level. But because evolution only changes the copier chemical and not the structure of the DNA chemical (in this system), the copier's junk level can be optimised by evolution whereas the DNA's cannot. So we set the DNA's junk level to be *long enough*, and concern ourselves with the copier's junk level in the experiments described in the following chapters. We set each of the DNA's junk regions to be 40 atoms long.

6.4.2 GraphMol copier chemical

The GraphMol copier chemical is a complicated graph, visualised in figure 6.21 and shown as a sequence of atoms in figure 6.22. It is composed of three regions:

1. Six *feet*, labelled by the folding tags `t1t1t` to `t6t6t` in figure 6.22. These bind to the base and backbone binding sites on the DNA, and are `shown` and `hidden` in sequence to make the copier *walk* along the DNA. This implements the `next` function of the embodied copy operation described in section 5.2.

2. A *start* binding site, labelled by the folding tag `tstst`. This binds to the DNA chemical's `start` binding site and initialises the mechanism ready to begin walking. This implements the `start` function of the embodied copy operation.

3. A *stop* binding site, labelled by the folding tag `tetet`. This binds to the DNA chemical's `stop` binding site and unbinds the copier chemical from the DNA chemical, ending the copying process. This implements the `at-end` function of the embodied copy operation.

The `next` function of the copy operation is *embodied* within this mechanism, because it is implemented as the process of moving the copier's feet from one DNA binding site to the next. The level of junk in the copier affects how this walking process operates, and can cause it to go wrong (explained below), resulting in mutations in the copied DNA (and hence, mutations in any chemicals encoded in the DNA, such as the copier chemical).

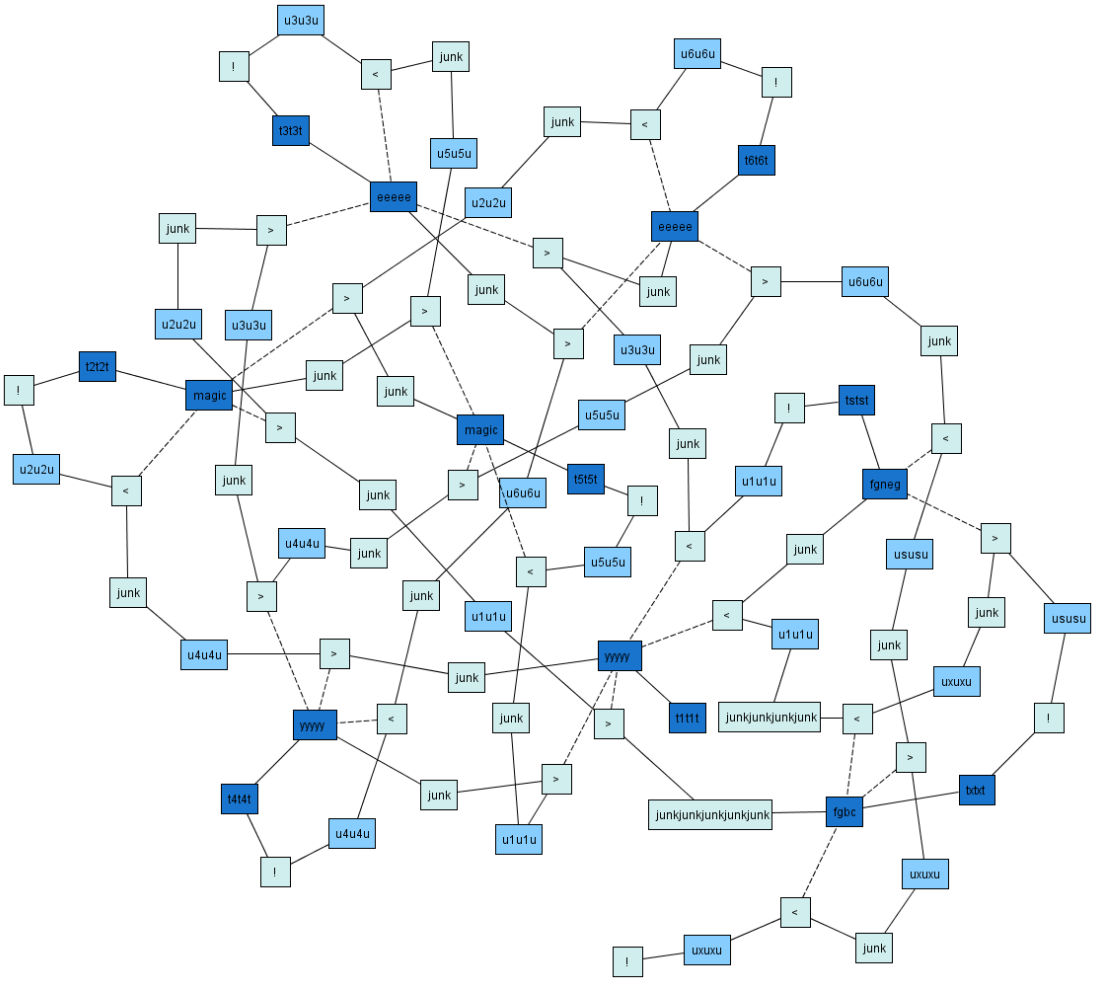


Figure 6.21: The GraphMol copier chemical after folding. Program edges are denoted by solid lines and fold edges by dashed lines. `Shown` binding sites are dark blue, `hidden` binding sites are medium blue, junk and function nodes are light blue. The geometry of how this graph is visualised does not influence its function: it is the topology of the graph that matters. This is a duplicate of figure 6.2, reproduced here for convenience.

```
[t1t1t] [yyyyy] junk >[u4u4u] junk <[u2u2u] !
[t2t2t] [magic] junk >[u5u5u] junk <[u3u3u] !
[t3t3t] [eeeee] junk >[u6u6u] junk <[u4u4u] !
[t4t4t] [yyyyy] junk >[u1u1u] junk <[u5u5u] !
[t5t5t] [magic] junk >[u2u2u] junk <[u6u6u] !
[t6t6t] [eeeee] junk >[u3u3u] junk <[u1u1u] !

[tstst] [fgneg] junk <[u1u1u] junk junk junk
                junk <[ueueu] junk >[ususu] !

[tetet] [fgbc]  junk junk junk junk
                junk >[u1u1u] junk >[u2u2u]
                junk >[u3u3u] junk >[u4u4u]
                junk >[u5u5u] junk >[u6u6u]
                junk <[ususu] junk >[ueueu] junk <[ueueu] !
```

Figure 6.22: The GraphMol copier chemical as a sequence of atoms (whitespace added for readability only). There are six *feet* (`t1t1t`–`t6t6t`), a *start* site (`tstst`) and a *stop* site (`tetet`). The length of the junk sections is varied in the experiment in the next chapter.

6.4.2.1 Start region

When the copier chemical is created, it is initialised with all its binding sites `hidden` except its *start* site. This has a pattern of `fgneg`, which matches the DNA chemical's `start` pattern on its *start* site. As can be seen in figure 6.22, the copier's `fgneg` site is followed by a short program that performs three operations to set up the copying process:

1. `Show` the first foot to start the walking process, allowing the copier to move along the DNA.

2. `Show` the *stop* binding site, allowing the copier to detect when it has walked to the end of the DNA, and trigger the `at-end` function.

3. `Hide` the *start* binding site, removing the bind to the DNA's `start` site and allowing the copier to move beyond the start of the DNA chemical. This step is performed last (and is preceded by a long junk region) because the first foot must bind to the DNA before the program `hides` the *start* site, otherwise there will be no binds between the copier and DNA, and so the reaction will finish.

After executing these three instructions, the `!` function terminates the program.

The `start` function of the copy operation is *crisp* in this mechanism, because the copier and the DNA both have explicit `start` binding sites, and this version of GraphMol uses exact string matching as its binding function. This means that for a given copier and DNA chemical, the copying process either starts perfectly or does not start at all. If the DNA chemical contained extra `start` binding sites spread throughout it, then the `start` function would be stochastic because GraphMol's mixing process would stochastically bind the copier's `start` site to one of the internal `start` sites on the DNA. This would create DNA copies that were missing the start of the DNA. If the world contained mechanisms that could insert `start` binding sites into the DNA, then this would embody the `start` function. Combining this with a rich binding function should make the `start` function evolvable.

6.4.2.2 Stop region

When the copier walks to the end of the DNA chemical, its *stop* binding site (with pattern `fgbc`) will bind to the DNA chemical's `stop` binding site. This causes the copier's *stop* program to execute, which finishes the copying process by `hiding` all the copier's binding sites, causing the two chemicals to unbind and the reaction to finish. There are three technicalities to this process:

1. The program begins with a long junk region. This is because the `stop` sites might bind before the last few bases of DNA have been copied. Adding junk here slows down the *stop* program, giving some extra time to finish the copy.

2. The copier's *start* site is already `hidden` at this point in the mechanism (it was `hidden` by the *start* program). The *end* program `shows` it, to set up the copier for its next copying reaction (so it can bind to the `start` site on the next DNA chemical it encounters via the mixing process).

3. The last thing the *end* program does is `show` the copier's `stop` binding site, immediately after `hiding` it.

   This may seem a strange (and possibly redundant) thing to do, but it addresses a subtle issue with the `at-end` function.

The issue occurs because at the same time as the *end* program is running, other programs can be running in parallel on the copier, started by the copier's feet binding. Most of the time this parallel execution does not cause a problem, but every time a copy ends there is a small chance that one of these programs can cause an error in the ending of the reaction. The error occurs because the programs associated with the copier's feet `show` and `hide` the copiers' feet. `Hiding` feet is not a problem, because the *end* program `hides` all the feet anyway. The problem

occurs if one of the foot programs `shows` a foot after the *end* program has `hidden` it. If this foot binds to the DNA before the *end* program `hides` its `stop` binding site, then the two chemicals will not disconnect, because the foot will keep them bound. The `stop` binding site will now be `hidden`, so the two chemicals would stay bound indefinitely. `Showing` the `stop` binding site at the end of the *end* program, gives the program another chance to `hide` all of the copier's binding sites. The issue is not guaranteed to occur: it only happens with a small probability. So by effectively sending the *end* program into a loop, this `showing` of the `stop` binding site guarantees that all of the copier's binding sites will be unbound from the DNA at some point, and the reaction will end.

The `at-end` function of the copy operation is mostly *crisp* in this mechanism, for the same reasons that the `start` function is crisp: both the copier and the DNA have explicit `stop` binding sites, and GraphMol has a trivial binding function. Unlike the `start` function, adding extra `stop` binding sites into the DNA would not immediately make the `at-end` function stochastic, because the copier mechanism would run its `stop` program as soon as it reached the first `stop` binding site on the DNA. This could be made stochastic by combining it with a non-trivial binding function that would allow the copying mechanism to ignore some `stop` binding sites on the DNA and carry on copying. As with the `start` function, introducing mechanisms into the world that could insert `stop` binding sites into the DNA would embody the `at-end` function within this copying mechanism, and a rich binding function should make it evolvable.

The issue described above, and its resolution, serve to partly embody the `at-end` function of this copying mechanism. When the issue is happening, the `at-end` function is not operating crisply, and the reason for this is the embodiment of the copying mechanism within the GraphMol world. This could potentially have evolutionary consequences, but these are not investigated in this thesis. We just note that the issue and its resolution are interesting properties of this embodied copying mechanism. And sometimes, things that seem like problems (the *end* function not always ending the reaction) can be solved by making the mechanisms more embodied (sending the crisp *end* function into a loop, using mechanisms already available in the GraphMol world).

6.4.2.3 Walking region

The copier chemical's six *feet* embody the `next` function of the copy operation. As seen in figure 6.22, each of the copier's feet is labelled by a folding template from `t1t1t` to `t6t6t`. Each foot contains one (non-folding) binding site, with a pattern of either `yyyyy`, `eeeee` or `magic`. After this binding site comes a short program that `hides` one of the feet and `shows` another. Each of program is terminated by a stop function (`!`), separating the programs associated with each foot.

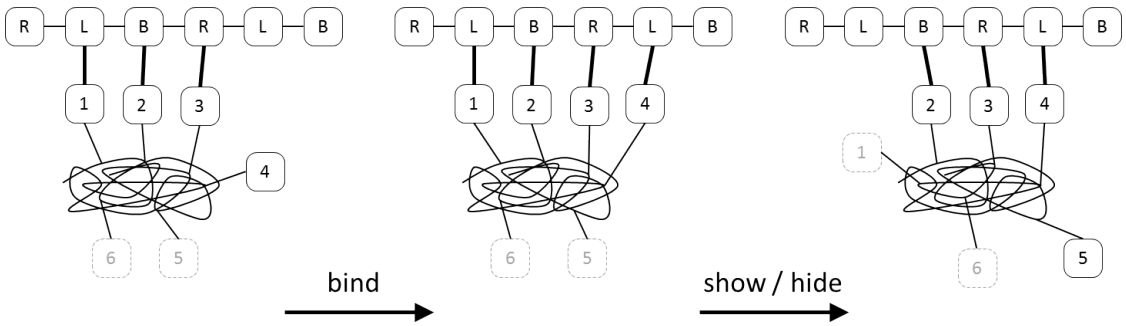


Figure 6.23: One step of the cyclic walking process. Three feet are always bound to the DNA. One new foot binds and runs its program, `hiding` one of the feet (and thus unbinding it from the DNA) and `showing` the next foot that will bind.

Figure 6.23 shows one iteration of the walking mechanism, which proceeds according to the following steps:

- Initially: feet 1–3 are `shown` and bound to the DNA; foot 4 is `shown` and unbound; feet 5 and 6 are `hidden`.
- Foot 4 has a pattern of `yyyyy`, which matches the `lllll` pattern of the DNA backbone's `left` binding site.
- Foot 4 binds to the `left` site on the DNA.
- This causes foot 4's program to run, which `hides` foot 1 and `shows` foot 5.
- The copier chemical has now taken one step, which corresponds to moving one third of a base along the DNA.
- Foot 5 is now ready to bind to the next base on the DNA, and start the next iteration of the cycle (`hiding` foot 2 and `showing` foot 6).

Each foot's binding sites matches one of the binding sites on the DNA. The copier's `yyyyy` sites (1 and 4) match the DNA's `lllll` sites. The copier's `eeeee` sites (3 and 6) match the DNA's `rrrrr` sites. And GraphMol's binding process is modified so that the copier's `magic` sites (2 and 5) match any of the DNA's base sites (`aaaaa`, `bbbbb`, `ccccc` or `ddddd`). This allows the copier chemical to walk along the DNA, binding to each of the DNA's binding sites in turn. The programs associated with each foot `show` and `hide` the other feet in a cycle, ensuring that the correct walker foot is `shown`, to bind with the next binding site on the DNA. Figure 6.24 shows precisely how these programs achieve this, beginning from the initial bind between the `start` sites of the copier and DNA.

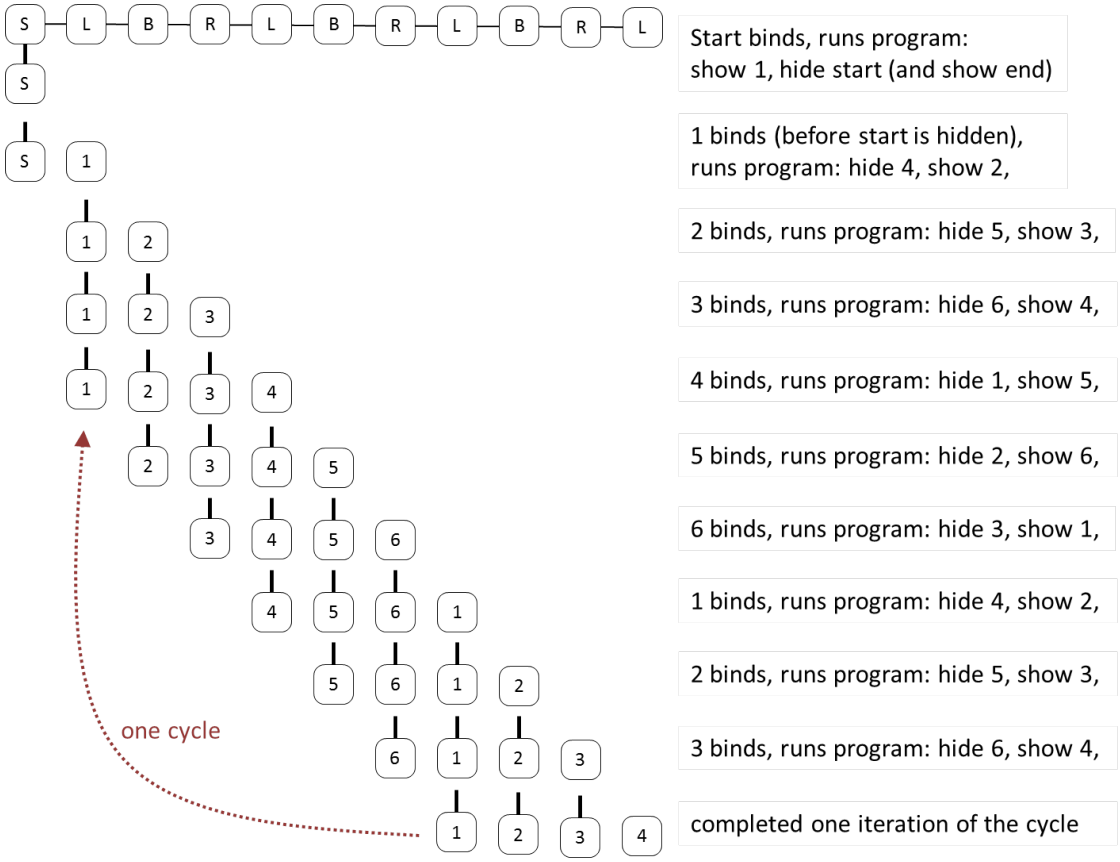


Figure 6.24: A full walking cycle, showing the sequence of binds and `show` and `hide` functions allowing the copier's feet to walk along a DNA chemical. One iteration of the cycle takes six steps and moves the copier over two bases on the DNA. This sequence cycles indefinitely, allowing the copier to walk along a DNA chemical of arbitrary length.

The above, crisp, description of the copier's walking assumes that each step happens perfectly. But this is not actually the case. Each time a foot takes a step, it is *most likely* to bind with the next site on the DNA, but there is a small chance that it could either:

1. step backwards, binding to the previous site; or

2. jump forwards, missing out a site.

These possibilities are shown in figure 6.25. Foot 4 has a pattern of `yyyyy`, so can only bind to left (`lllll`) binding sites on the DNA. The distances to each binding site determine how likely each possibility is. The foot most often binds

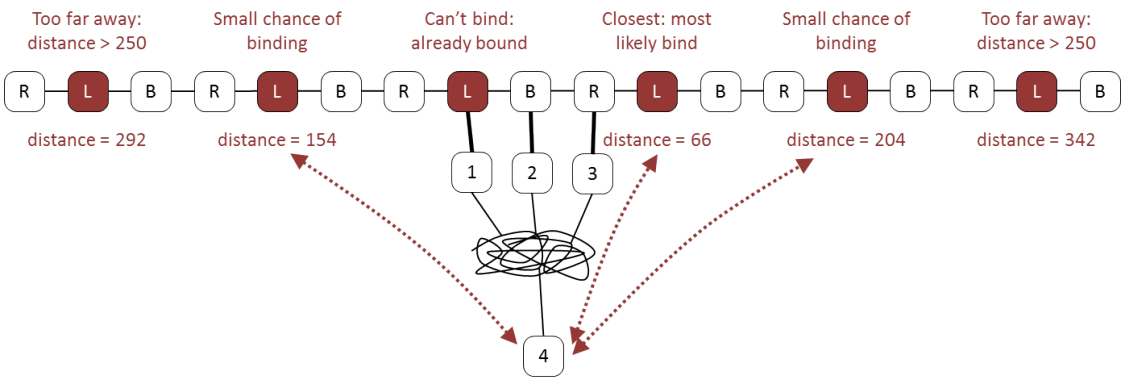


Figure 6.25: The possible binds a foot could make. It is most likely to bind to the next location along the DNA. There is a small chance it could step backwards, or jump forwards. It cannot bind to sites that are too far away, or those that are already bound. (Distances are for a junk level of 10.)

to the next site on the DNA because this is the closest available binding site with a matching pattern. The jumps forwards and backwards involve binding over a longer distance, and so happen with lower probabilities. The foot cannot jump forwards or backwards two (or more) steps, because these binding sites are too far away: beyond the cutoff of the graph distance process. Even if the graph distance process had a larger cutoff, these binds would be very unlikely to happen because their distances are so large.

From an embodiment viewpoint, the copier's feet jumping forwards and backwards correspond to variations in how the `next` function of the copying process operates. But this variation is not performed by a stochastic process existing in a separate world. The variation originates from the stochastic operation of GraphMol's binding process. The effect that this stochasticity has on the copier's feet is modulated by the graph distance process, which is influenced by the level of junk in the copier (and the level of junk in the DNA). Because the copier's junk level can be changed by the copier (via the copying of a DNA chemical that encodes the copier), the copying mechanism can change the influence that the stochastic process has on it's `next` function. This is an example of embodied evolution leading to meta-evolution.

6.4.2.4 Designing the world

We can use the distances above (in figure 6.25) to help us design the GraphMol world. The binding probability function (described in section 6.3.1.2) determines the probability of two sites binding, depending on their distance. This function

takes the form:

$$\text{bind\_probability}(d) = \left(\frac{k}{d}\right)^{\alpha} \tag{6.7}$$

where d is the distance between the two binding sites. This has two parameters: k and α. We can choose these parameters to give the copier chemical the behaviour we want, by assuming the following things:

1. The copier is copying a DNA chemical requiring l steps.
2. We want it to make an average of x errors per copy of the DNA.
3. We want the probability of a 'correct' bind to be p_c.
4. The distance of the bind for a 'correct' step is d_c.
5. The distance of the bind for an 'error' step is d_e.

We can use these assumptions to derive the following values for the parameters in equation 6.7 (full details in appendix A), giving:

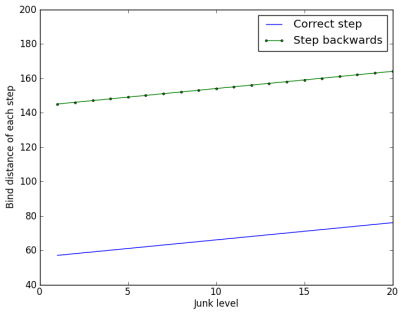
$$\alpha = \frac{\log(x) - \log(l)}{\log(d_c) - \log(d_e)} \tag{6.8}$$

$$k = \exp\left(\log(d_c) + \frac{\log(p_c)}{\alpha}\right) \tag{6.9}$$

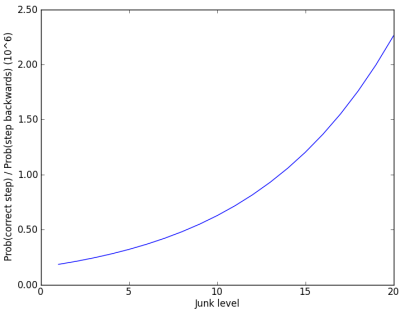
This expresses two parameters of the GraphMol world in terms of requirements that we have for the copier chemical.

Note that this is not a universal method for designing GraphMol mechanisms. If we were to design a different mechanism, with different values of α and k, and put it into the same world as the copier, then we would have a problem. We would have to decide which values of α and k to use, which might break one of the mechanisms. The purpose of this exercise is to show how the parameters of the GraphMol world affect the mechanisms implemented within it. And also, to show that we can analyse the mechanisms before implementing them, to gain an intuition for how they will operate. This is only an intuition, rather than a full understanding, because any analysis must make simplifying assumptions. In this case, for example, we assume that there is only one distance for an 'error' step, when in reality there are two different types of 'error' (stepping backwards and jumping forwards).

To give the copier chemical interesting behaviours over a range of junk lengths, we calculated the values of α and k for a copier chemical of junk length 10. This gives the copier chemical a length of 621 atoms, meaning that its DNA requires



(a) The distances of 'correct' and 'erroneous' steps increase linearly with the copier's junk level.



(b) The likelihood of making a 'correct' step over an 'erroneous' step increases more quickly than linearly.

Figure 6.26: Changing the copier's junk level has a small effect on the geometry of the graph (a), but the binding probability function amplifies this, producing a large effect in the behaviour of the chemical (b).

$l = 621 \times 3 = 1863$ steps to copy. We wanted $x = 0.5$ errors per copy of DNA, i.e. an average of one error every other copy, and the probability of a 'correct' bind to be $p_c = 0.5$. With a copier chemical of junk length 10, $d_c = 66$ and $d_e = 154$ (see figure 6.25). Using equations 6.8 and 6.9, this leads to the values of α and k given in section 6.3.1.2: $\alpha = 9.705$ and $k = 61.451$. After calculating these values, we fixed them in the GraphMol world.

As explained earlier, junk is the means by which GraphMol mechanisms can change the influence the GraphMol world has on them. If we change the sizes of the junk regions in the copier chemical, we change the graph distances between the binding sites in the copier, and so we change the distances of the 'correct' and 'error' steps. Figure 6.26 shows that increasing the junk level in the copier increases the likelihood of making a 'correct' step compared to making an 'error' step. Thus changing the sizes of the junk regions changes the reliability of the embodied walking process. The following chapter describes an experiment investigating exactly how the junk level affects a copier running in the GraphMol world.

The following section describes how the copier's feet jumping correspond to mutations in the copied DNA chemical.

6.4.3 Walking to copying

The walking mechanism described above allows the copier chemical to *move* along the DNA and embody the `next` function of the copy operation, but it does not allow it to actually *copy* the DNA. In other words, we need to add a `char-copy` function to the copier chemical.

We attach a *crisp* `char-copy` function to the copier chemical. In principle, we could build an embodied `char-copy` function in GraphMol, by designing a more complicated copying chemical and adding some more function nodes to the GraphMol world. But this would complicate both GraphMol and the copier chemical, and would combine the effects of an embodied `next` function with an embodied `char-copy`. This is clearly an interesting research direction, but for this thesis we focus on embodying the `next` function by using a crisp `char-copy`.

The copier's `magic` bindings sites are where we attach the crisp `char-copy` function (hence the name *magic*, because these sites operate outside the rules of the GraphMol world). As explained above, these bind to the bases of the DNA, while the copier's other feet bind to the backbone. One *magic* modification of the GraphMol rules is that GraphMol's binding function only allows a binding site's pattern to match one other pattern (its exact complement), but the `magic` pattern is allowed to match the patterns of each of the DNA bases: `aaaaa`, `bbbbb`, `ccccc` and `ddddd`. When a `magic` site binds to one of these bases, we invoke the crisp `char-copy` operation. In a separate (crisp) world, we record the pattern of the base that has been bound to, keeping a record of the order in which the DNA's bases have been walked over. This copying does not consume any additional energy, other than that required to advance the steps of the reaction. When the reaction ends (when both chemicals have completely unbound), the crisp `char-copy` process looks at this record of bases and uses it to construct a new DNA chemical, which it adds to the GraphMol world.

The addition of this *magic* to the copier chemical allows it to implement a full copying process with an embodied `next` function and a crisp `char-copy`. The `start` and `at-end` functions are technically embodied here, but they have not been embodied in a very evolvable way, so for all intents and purposes they are crisp.

When the copier chemical is producing copies of the DNA it walks over, the copier jumping forwards and backwards causes mutations in the copied DNA. Figure 6.27 shows a trace of the copier chemical making a perfect copy of a short piece of DNA. Figure 6.28 shows how a jump backwards causes an insertion of two DNA bases, because the copier goes back to the previous base, repeating the previous base and the 'current' base. Figure 6.29 shows how a jump forwards causes a deletion of one DNA base.

As well as small insertions and deletions, the copier chemical can perform another type of mutation: a large insertion due to its feet *tangling*. Figure 6.30 shows

```
.........................  S
S........................
S........................  1
S1.......................
S1.......................  2
S12......................
S12......................  E
.12......................  E
.12......................  E, 3
.123.....................  E
.123.....................  E, 4
.1234....................  E
..234....................  E
..234....................  E, 5
..2345...................  E
...345...................  E
...345...................  E, 6
...3456..................  E
....456..................  E
....456..................  E, 1
....4561.................  E
.....561.................  E
.....561.................  E, 2
.....5612................  E
......612................  E
......612................  E, 3
......6123...............  E
.......123...............  E
.......123...............  E, 4
.......1234..............  E
........234..............  E
........234..............  E, 5
........2345.............  E
.........345.............  E
.........345.............  E, 6
.........3456............  E
..........456............  E
..........456............  E, 1
..........4561...........  E
...........561...........  E
...........561...........  E, 2
```

```
...........5612..........  E
............612..........  E
............612..........  E, 3
............6123.........  E
.............123.........  E
.............123.........  E, 4
.............1234........  E
..............234........  E
..............234........  E, 5
..............2345.......  E
...............345.......  E
...............345.......  E, 6
...............3456......  E
................456......  E
................456......  E, 1
................4561.....  E
.................561.....  E
.................561.....  E, 2
.................5612....  E
..................612....  E
..................612....  E, 3
..................6123...  E
...................123...  E
...................123...  E, 4
...................1234..  E
....................234..  E
....................234..  E, 5
....................2345.  E
.....................345.  E
.....................345.  E, 6
.....................3456  E
......................456  E
......................456E
......................456E  1
......................456E
.......................56E
........................6E
.........................E
.........................E  S
.........................  S
```

DNA sequence: `ABCDABCD`
Copy produced: `ABCDABCD`

Figure 6.27: The copier chemical making a perfect copy of an 8-base DNA chemical. Dots represent unbound DNA binding sites. Numbers represent the copier's feet, with S representing the *start* binding site and E representing the *end* binding site. The numbers at the right show the copier's binding sites that are `shown` and unbound.

```
.........................  S
S........................
S.........................  1
S1.......................
S1........................  2
S12......................
S12.......................  E
.12.......................  E
.12.......................  E, 3
.123......................  E
.123......................  E, 4
.1234.....................  E
..234.....................  E
..234.....................  E, 5
..2345....................  E
...345....................  E
...345....................  E, 6
...3456...................  E
....456...................  E
....456...................  E, 1
....4561..................  E
.....561..................  E
.....561..................  E, 2
.....5612.................  E
......612.................  E
......612.................  E, 3
......6123................  E
.......123................  E
.......123................  E, 4
.......1234...............  E
........234...............  E
........234...............  E, 5
........2345..............  E
.........345..............  E
.........345..............  E, 6
.........3456.............  E
..........456.............  E
..........456.............  E, 1
..........4561............  E
...........561............  E
...........561............  E, 2
...........5612...........  E
............612...........  E
............612...........  E, 3
............6123..........  E
.............123..........  E
.............123..........  E, 4
.............1234.........  E
..............234.........  E
..............234.........  E, 5
..............2345........  E
...............345........  E
...............345........  E, 6
...............3456.......  E
................456.......  E
................456.......  E, 1
................4561......  E
.................561......  E
.................561......  E, 2
.................5612.....  E
..................612.....  E
..................612.....  E, 3
..................6123....  E
...................123....  E
...................123....  E, 4
...................1234...  E
....................234...  E
....................234...  E, 5
.....................2345..  E
......................345..  E
......................345..  E, 6
......................3456.  E
.......................3456E
........................456E
........................456E  1
.......................456E
.........................56E
..........................6E
...........................E
...........................E  S
.........................  S
```

DNA sequence:	ABCDABCD	ABCDAB--CD
Copy produced:	ABCDABABCD	ABCDABABCD

Figure 6.28: The copier chemical jumping backwards during a copy and making an insertion of two bases. Dashes in the DNA sequence show the position of the insertion.

```
.........................  S
S........................
S.........................  1
S1.......................
S1.......................  2
S12......................
S12......................  E
.12......................  E
.12......................  E, 3
.123.....................  E
.123.....................  E, 4
.1234....................  E
..234....................  E
..234....................  E, 5
..2345...................  E
...345...................  E
...345...................  E, 6
...3456..................  E
....456..................  E
....456..................  E, 1
....4561.................  E
.....561.................  E
.....561.................  E, 2
.....5612................  E
......612................  E
......612................  E, 3
......6123...............  E
.......123...............  E
.......123...............  E, 4
.......123...4...........  E
........23...4...........  E
........23...4...........  E, 5
........23...45..........  E
.........3...45..........  E
.........3...45..........  E, 6
.........3...456.........  E
```

```
.............456.........  E
.............456.........  E, 1
.............4561........  E
..............561........  E
..............561........  E, 2
..............5612.......  E
...............612.......  E
...............612.......  E, 3
...............6123......  E
................123......  E
................123......  E, 4
................1234.....  E
.................234.....  E
.................234.....  E, 5
.................2345....  E
..................345....  E
..................345....  E, 6
..................3456...  E
...................456...  E
...................456...  E, 1
...................4561..  E
....................561..  E
....................561..  E, 2
....................5612.  E
.....................612.  E
.....................612.  E, 3
.....................6123  E
......................123E
......................123E  4
......................123E
.......................23E
........................3E
.........................E
.........................E  S
.........................  S
```

DNA sequence:	ABCDABCD	ABCDABCD
Copy produced:	ABCABCD	ABC-ABCD

Figure 6.29: The copier chemical jumping forwards during a copy and making a deletion of one base.

```
.........................  S
S........................
S........................  1
S1.......................
S1.......................  E
S1.......................  E, 2
S12......................  E
.12......................  E
.12......................  E, 3
.123.....................  E
.123.....................  E, 4
.1234....................  E
..234....................  E
..234....................  E, 5
..2345...................  E
...345...................  E
...345...................  E, 6
...3456..................  E
....456..................  E
....456..................  E, 1
....4561.................  E
.....561.................  E
.....561.................  E, 2
.....5612................  E
......612................  E
......612................  E, 3
......6123...............  E
.......123...............  E
.......123...............  E, 4
.......1234..............  E
........234..............  E
........234..............  E, 5
........2345.............  E
.........345.............  E
.........345.............  E, 6
.........3456............  E
..........456............  E
..........456............  E, 1
.......1..456............  E
.......1...56............  E
.......1...56............  E, 2
.......12..56............  E
.......12...6............  E
.......12...6............  E, 3
......312...6............  E
......312................  E
......312................  E, 4
......312....4...........  E
......3.2....4...........  E
......3.2....4...........  E, 5
.....53.2....4...........  E
.....53......4...........  E
.....53......4...........  E, 6
.....53..6...4...........  E
.....5...6...4...........  E
.....5...6...4...........  E, 1
.....5...61..4...........  E
.....5...61..............  E
.....5...61..............  E, 2
.....5...612.............  E
.........612.............  E
.........612.............  E, 3
.........6123............  E
..........123............  E
..........123............  E, 4
..........1234...........  E
...........234...........  E
...........234...........  E, 5
...........2345..........  E
............345..........  E
............345..........  E, 6
............3456.........  E
.............456.........  E
.............456.........  E, 1
.............4561........  E
..............561........  E
..............561........  E, 2
..............5612.......  E
...............612.......  E
...............612.......  E, 3
...............6123......  E
................123......  E
................123......  E, 4
................1234.....  E
.................234.....  E
.................234.....  E, 5
.................2345....  E
..................345....  E
..................345....  E, 6
..................3456...  E
...................456...  E
...................456...  E, 1
...................4561..  E
....................561..  E
....................561..  E, 2
....................5612.  E
.....................612.  E
.....................612.  E, 3
.....................6123  E
......................123
......................123E  4
.......................23E  4
......................423E
......................4.3E
......................4..E
......................4..E  5
......................45.E
.......................5.E
........................E
........................E  6
........................E
........................E  S
.........................  S
```

DNA sequence:	ABCDABCD	ABCD---ABCD-
Copy produced:	ABCDCBDABCDD	ABCDCBDABCDD

Figure 6.30: The copier chemical tangling and making an insertion of three bases rather than two. (And also making a small insertion at the end of the copy, due to the embodied `at-end` function.)

```
S.....                  ...6...15.....  E, 2    ..261.....  E, 3
S.....  1               ..26...15.....  E       ..261.3...  E
S1....                  ..26...1......  E       ..2.1.3...  E
S1....  2               ..26...1......  E, 3    ..2.1.3...  E, 4
S12...                  ..26..31......  E       .42.1.3...  E
S12...  E               ..2...31......  E       .42...3...  E
.12...  E               ..2...31......  E, 4    .42...3...  E, 5
.12...  E, 3            ..2.4.31......  E       .42..53...  E
.123..  E               ..2.4.3.......  E       .4...53...  E
.123..  E, 4            ..2.4.3.......  E, 5    .4...53...  E, 6
.1234.  E               ..2.453.......  E       .4.6.53...  E
..234.  E               ....453.......  E       .4.6.5....  E
..234.  E, 5            ....453.......  E, 6    .4.6.5....  E, 1
..2345  E               ...6453.......  E       .4.6.5.1..  E
...345  E               ...645........  E       ...6.5.1..  E
...345  E, 6            ...645........  E, 1    ...6.5.1..  E, 2
...3456 E               ...645.1......  E       ...6.5.12.  E
....456 E               ...6.5.1......  E       ...6...12.  E
....456 E, 1            ...6.5.1......  E, 2    ...6...12.  E, 3
....4561 E              ..26.5.1......  E       ...6...123  E
.....561 E              ..26...1......  E       .......123  E
.....561 E, 2           ..26...1......  E, 3    .......123  E, 4
.....5612 E             ..26..31......  E       .......1234 E
......612 E             ..2...31......  E       ........234 E
......612 E, 3          ..2...31......  E, 4    ........234 E, 5
......6123 E            .42...31......  E       ........2345 E
.......123 E            .42...3.......  E       .........345 E
.......123 E, 4         .42...3.......  E, 5    .........345 E, 6
....4..123 E            .42..53.......  E       .........3456 E
....4...23 E            .4...53.......  E       ..........456 E
....4...23 E, 5         .4...53.......  E, 6    ..........456 E, 1
....45..23 E            .4.6.53.......  E       ..........4561 E
....45...3 E            .4.6.5........  E       ...........561 E
....45...3 E, 6         .4.6.5........  E, 1    ...........561 E, 2
...645...3 E            .4.615........  E       ............5612 E
...645..... E           ...615........  E       ............612 E
...645..... E, 1        ...615........  E, 2    ............612 E, 3
...645....1 E           ..2615........  E       .............6123 E
...6.5....1 E           ..261.........  E       .............123 E
...6.5....1 E, 2        ..261.........  E, 3    .............123 E, 4
..26.5....1 E           ..261.3.......  E       .............1234 E
..26......1 E           ..2.1.3.......  E       ..............234 E
..26......1 E, 3        ..2.1.3.......  E, 4    ..............234 E, 5
..26.....31 E           .42.1.3.......  E       ..............2345 E
..2......31 E           .42...3.......  E       ...............345 E
..2......31 E, 4        .42...3.......  E, 5    ...............345 E, 6
.42......31 E           .42..53.......  E       ...............3456 E
.42......3 E            .4...53.......  E       ................456
.42......3 E, 5         .4...53.......  E, 6    ................456
.42.....53 E            .4.6.53.......  E       ................456E 1
.4......53 E            .4.6.5........  E       ................456E
.4......53 E, 6         .4.6.5........  E, 1    .................56E
.4.6....53 E            .4.615........  E       ..................6E
.4.6....5 E             ...615........  E       ...................E
.4.6....5 E, 1          ...615........  E, 2    ...................E S
.4.6...15 E             ..2615........  E       ..................  S
...6...15 E             ..261.........  E
```

DNA sequence:	ABCDABCD	ABC----------DABCD
Copy produced:	ABCBADACABABABABABCDABCD	ABCBADACABABABABABCDABCD

Figure 6.31: The copier chemical tangling and making a large insertion.

an example of a tangle, which happens in the following steps:

1. The copier begins copying perfectly, until one of its feet (here foot 1) jumps backwards, as in an insertion mutation.

2. The following foot (foot 2) proceeds as in a normal insertion, binding after foot 1.

3. Foot 3 does not proceed normally. Instead of binding to the next DNA binding site after foot 2, it binds to the previous site (before foot 1).

4. Normally, foot 6 would be blocking this bind, but because of the insertion, foot 6 is further ahead on the DNA, so foot 3 is able to bind to the site before foot 1, rather than binding to the site after foot 1. The copier is effectively walking backwards at this point.

5. Now, when foot 4 comes to bind, it has a choice. It can either continue the walk backwards, and bind to the site before foot 3, or it can walk forwards, binding to the site after foot 6.

6. It may seem like binding after foot 6 is impossible, because foot 6 has unbound at this point and so the distance of this bind from the walker's current position is very large. But the graph distance process takes time to propagate this information.

7. Foot 4 makes its bind downstream on the DNA, before the graph distance process has propagated the information about foot 6 unbinding. This creates two *regions* on the DNA where the copier's feet are bound. When subsequent feet bind, they choose stochastically which of these regions they bind to.

8. In this case, foot 5 binds upstream and foot 6 binds in the middle of the two regions, ending the tangle and coaxing the copier's feet back into their walking pattern.

In this example, the tangle caused an insertion of three bases rather than the usual two. But the tangle does not have to end this quickly. Figure 6.31 shows an extreme example, where the stochastic choices of which region to bind to have prolonged the tangle, causing an insertion of 16 bases.

It is important to notice that the insertions do not insert *arbitrary* DNA bases into the copy. The insertion happens because the copier's feet walk in a tangled fashion over the DNA. Thus the bases that are inserted depend on which bases are present in the region of the insertion. As an extreme example, if the copier tangled in a region of DNA that contained only `A` bases, then only `A` bases would be inserted into the copy. But in a region containing a mix of all four bases, insertions may

appear arbitrary. Looking at insertions performed by the copier chemical, I have observed two distinct types of insertion. Sometimes the insertions appear arbitrary, and sometimes they repeat two characters in sequence. These two different types of insertion are not mutually exclusive: they can occur in the same insertion event, as seen in figure 6.31. This insertion starts arbitrarily (`BADAC`) before changing to the repeating mode (`ABABABABAB`). These two modes can also be seen in the pattern of the copier's feet in figure 6.31:

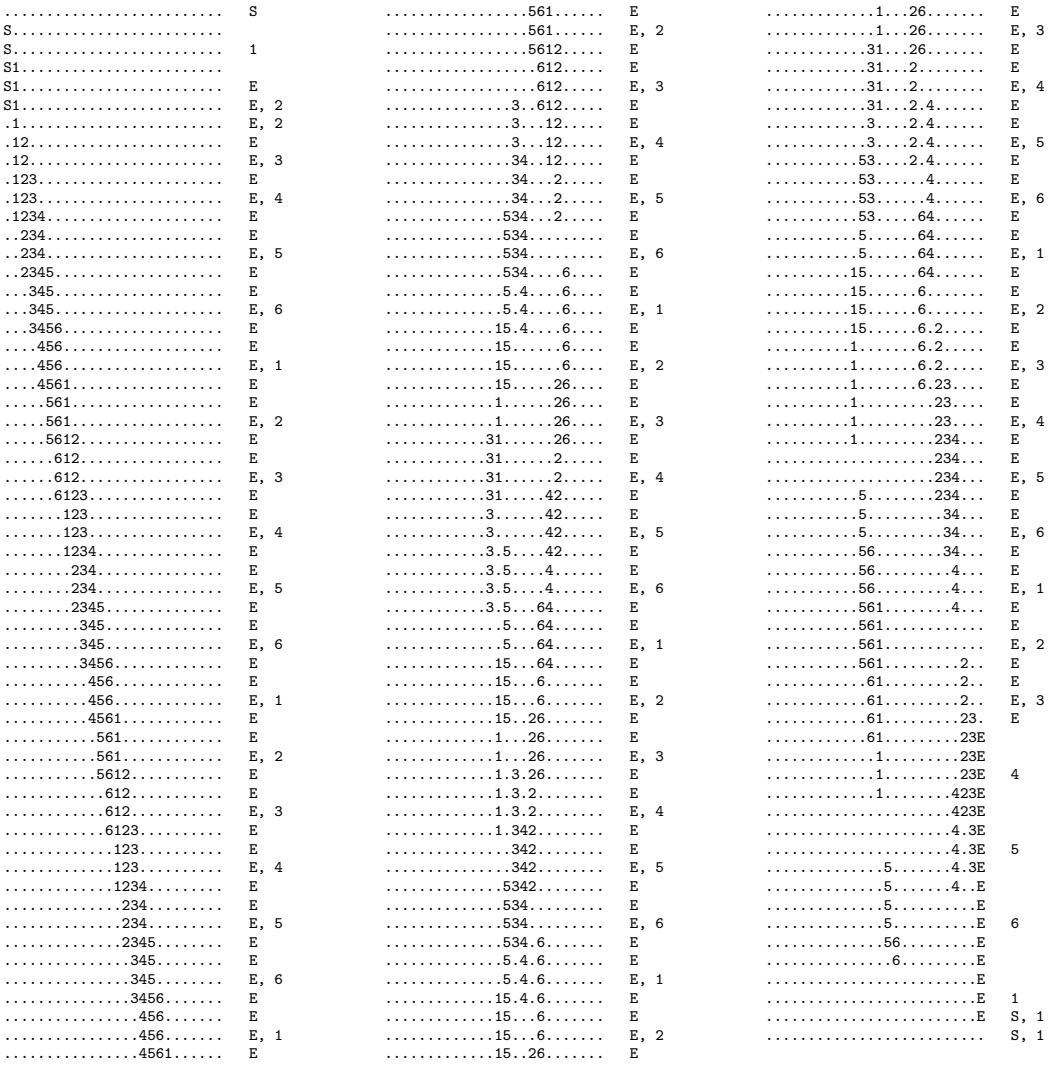
1. In the arbitrary mode, the two regions are far apart. This means that the next foot has a wide choice of different sites to which it can bind (near each region, or in the middle). So, copied bases will be chosen stochastically from any bases appearing on the DNA between the two regions.

2. In the repeating mode, the two regions are very close together. The next `magic` foot always binds to the *other* region, making the copied bases repeat the same two characters.

The reason for the two regions staying close together in the repeating region is purely anthropic. They can drift apart, but this would change the insertion into an arbitrary one. They can come together, but this would stop the insertion and change into the perfect copying mode. But nevertheless, I have observed a higher frequency of repeats (compared to arbitrary insertions) than I would expect by chance. So I conjecture that, in the repeating region, staying in the repeating region is more probable than moving into the arbitrary region.

Tangling is an emergent property of the copier chemical's `next` function. Because this `next` function is embodied within the same world as the other functions of the copier chemical, these functions can interact with each other. Figure 6.32 shows an example of a tangle interacting with the copier's `at-end` function. At one point, the two regions of this tangle are far apart, and so the tangle would continue for a while before the two regions would stochastically come together. But one of these regions reaches the end of the DNA, causing the copier's *end* site to bind to the DNA's `end` site. This triggers the copier's *end* program, that `hides` its binding sites and ends the reaction. This interaction between the copier's `next` function and its `at-end` function is only possible because they are both embodied within the same world.

In summary, the GraphMol language has been designed to allow us to embody the different functions of the copying operation within the same world. We have used GraphMol to build embodied `start`, `next` and `at-end` functions, although the `next` function is the most flexible of these. We have added a crisp `char-copy` function to these embodied mechanisms, to create an embodied copy operation within the GraphMol world. The variation in how the `next` operation works allows this mechanism to perform four different (but related) types of mutation:



DNA sequence:	ABCDABCD ABCDABC---------D-
Copy produced:	ABCDABCACABABDCDDA ABCDABCACABABDCDDA

Figure 6.32: A tangle interacting with the *end* function, to stop the tangle earlier than it would have stopped by itself. An example of the copier's `next` function interacting with its `at-end` function.

1. Single-base deletions, where a foot jumps over a DNA base.
2. Double-base insertions, where a foot jumps backwards and repeats two bases.
3. Repeating insertions, where the feet tangle but stay close together, repeating the same two bases in sequence.
4. Arbitrary insertions, where the feet tangle but move apart into two distinct regions. The bases inserted here are not truly arbitrary, but are arbitrary walks over those bases available nearby on the DNA.

Because these different types of mutation are different behaviours of the GraphMol copier chemical, these different types of mutation are *embodied* within the GraphMol world, and hence can be evolved.

The following two chapters describe experiments investigating how these different types of mutation affect the evolution of the GraphMol copying chemical.

Chapter 7

Junk changes GraphMol copier

The GraphMol copier chemical (described in the previous chapter) can copy a GraphMol DNA chemical, making different types of insertion and deletion errors. When we encode the copier chemical on the DNA being copied, the GraphMol copier chemical can change itself, thus performing meta-evolution.

This meta-evolution can happen because the errors made by GraphMol's copying phenomenon are *embodied* within the same world as the copying mechanism. The errors are not due to a stochastic piece of code implementing the copying phenomenon (which would not allow meta-evolution). Rather, the copying errors occur because the copying phenomenon is embodied within a mechanism implemented in a stochastic world. The stochasticity of the world is fixed and cannot vary, but the copying mechanism can vary, thus changing the influence that the stochastic world has on the copying phenomenon. Letting the copying mechanism *change itself* closes the loop, allowing the copying phenomenon to change its own errors.

The experiment in this chapter is a *feasibility study*, testing the evolvability of the embodied copying process implemented by the GraphMol copier. We want to test whether differences in the low-level *mechanism* of the copier chemical (its junk level) can transfer to the high-level *phenomenon* embodied within it (errors in the copying process), causing variations in the phenomenon *that we can measure*. If we can observe these low-level changes propagating upwards, then this provides some validation of the design of the copier and of GraphMol, showing that both of them are useful components in our investigation of embodiment.

It is important to note that this experiment does *not* attempt to discover exactly how the copier chemical will evolve when allowed to replicate in an open world (which is investigated in the following chapter). This experiment just shows *one way* in which mutations to the GraphMol copier chemical can affect its behaviour. The particular effect demonstrated in this experiment may or may not be useful to an evolving copier chemical, depending upon the world in which it is evolving.

There will hopefully be more different effects that could be seen by making different types of mutation to the copier chemical. This experiment shows just one effect, to demonstrate that this is possible in GraphMol and the copier chemical.

This experiment successfully finds that low-level changes to the copier chemical's mechanisms can transfer to high-level changes in its phenomenon. With more junk, the chemical copies more accurately but also more slowly. With less junk, the chemical copies less accurately but also more quickly. This tradeoff has an optimum junk level, where the rate of *nearly-perfect* copying is maximised.

7.1 Hypothesis

We hypothesise that changing the copier's junk level changes its speed and accuracy of copying. We expect that increasing the junk level will make the copier slower but more accurate, with a *sweet spot* at an intermediate junk level, that trades off speed against accuracy to produce a copier with the highest *rate* of copying. We expect this to be the case, since the GraphMol language and the copier chemical were designed to achieve this.

The *null hypothesis* is that changes to the GraphMol copier's junk level will have no statistically significant effect on the speed or accuracy of the copies produced.

This experiment manages to successfully refute its null hypothesis, showing that the junk level of the copier does affect both the speed of copying and the accuracy of the copies produced, with the appearance of the predicted *sweet spot* at an intermediate junk level.

7.2 Experiment design

To test the above hypothesis, we allow the GraphMol copier chemical to copy multiple DNA chemicals, measuring its speed and accuracy of copying. We repeat this process using copier chemicals with different junk levels, investigating how the copier's junk level affects its speed and accuracy of copying.

One possible approach would be to put copier chemicals with different junk levels into the same world and allow them to compete with each other. Or we could put one copier chemical into a world and allow it to mutate, with its children competing with each other and evolving. But this is not what we do here. Having multiple chemicals competing in the same world is a complicated experiment, with the survival of a particular chemical depending on external factors (such as the energy flux into the world, and which other chemicals it stochastically binds to). We perform that experiment in the following chapter.

In this experiment, we measure two specific properties of the copier chemical (its

speed and its accuracy), in the absence of any other factors (such as evolution). To do this, we put one copier chemical and one DNA chemical into a GraphMol world and allow them to perform one macro-scale reaction (as in the example reactions shown in figures 6.27–6.32 above). When the reaction finishes, we compare the copied DNA chemical with the original, measuring the copier's accuracy. We count the number of GraphMol iterations taken, to measure the copier's speed.

After one reaction has finished, we destroy its world and start again with a fresh GraphMol world, a fresh copier chemical and a fresh DNA chemical. Because the copying is stochastic, we repeat this process 80 times for each junk level, to determine the variation in our measurements. We then repeat the whole process, varying the copier's junk level from one atom up to 20 atoms.

7.2.1 DNA length

For each junk level, the length of the DNA chemical (measured as the number of bases) is the same as the length of the copier chemical (measured as the number of atoms). This means that copiers with different junk levels will have different lengths of DNA to copy (higher junk levels will have longer DNA chemicals). This simulates the fact that if the copier was evolving, then it would have to copy a DNA chemical that encoded itself. So changing its junk level would change the length of its encoding on the DNA.

The copier chemical used in this experiment has the following form (introduced in section 6.4.2):

```
[t1t1t] [yyyyy] J >[u4u4u] J <[u2u2u] !
[t2t2t] [magic] J >[u5u5u] J <[u3u3u] !
[t3t3t] [eeeee] J >[u6u6u] J <[u4u4u] !
[t4t4t] [yyyyy] J >[u1u1u] J <[u5u5u] !
[t5t5t] [magic] J >[u2u2u] J <[u6u6u] !
[t6t6t] [eeeee] J >[u3u3u] J <[u1u1u] !
[tstst] [fgneg] J <[u1u1u] J J J
                J <[ueueu] J >[ususu] !
[tetet] [fgbc]  J J J J
                J >[u1u1u] J >[u2u2u]
                J >[u3u3u] J >[u4u4u]
                J >[u5u5u] J >[u6u6u]
                J <[ususu] J >[ueueu] J <[ueueu] !
```

Each junk region, `J`, comprises a number of characters, `j`, corresponding to the copier's junk length.

This means the length of the copier (and hence the number of bases in its DNA) is:

$$\text{copier\_length} = 311 + \text{junk\_length} \times 31, \tag{7.1}$$

since there are 311 non-junk characters and 31 junk regions of varying length.

The junk regions in the copier chemical are varied in length from one atom to 20 atoms (in steps of one). In this experiment, all of the junk regions within the copier are kept the same length as each other, for simplicity. If the copier was evolving, then it would not need to enforce this. Indeed, unless there was evolutionary pressure for it, evolution would probably not maintain 31 different regions at the same length. So this experiment shows a coarse view of the evolutionary options possessed by the copier. In reality, the copier has a much finer level of control over its junk regions than this experiment shows.

For example, an evolving copier chemical might find it useful to have long junk regions within the programs of its six *feet*, so that it walks accurately over the DNA and accurately implements the *next* function of the copy operation. But being shorter might confer an evolutionary advantage (as it can be copied quicker), so the copier chemical might be able to make gains by shortening some of its less critical junk regions, such as the delay parts of its *start* and *end* programs.

7.2.2 Initialisation

For each macro-level reaction, we create a new DNA chemical of the correct length and set all of its binding sites to be `shown`. This is the default state for new GraphMol chemicals, so no special treatment is needed for the DNA chemical. The pattern of bases on the DNA chemical makes no difference to this experiment, so it is randomly generated each time.

We create a new copier chemical and initialise it by `hiding` all of its binding sites except its *start* site (`fgneg`). This may seem like twisting the rules of the GraphMol world, to create a chemical with some sites `shown` and others `hidden`, but it is not. In a larger system, the copier chemical would exist in a world containing many other chemicals, such as the chaperone folding chemicals described in section 6.2.3.4. We can imagine another chaperone chemical that helps to initialise copier chemicals (and other mechanisms) by binding to their *end* site (`fgbc`). In this case, new chemicals could be created with all their binding sites `shown`, and these chaperones would run the *end* program of each new chemical, initialising it.

Technically, a chaperone chemical binding to the copier's *end* site would not be appropriate for this mechanism, because the copier leaves its *end* site `shown` throughout the reaction, so that it can detect the end of the DNA chemical. The problem here is that a chaperone chemical could bind to the copier's *end* site at any time while the copier was copying a DNA chemical. This would run the copier's

end program, `hiding` all its feet and thus terminating the copy. Instead of the chaperone chemical binding to the *end* site on the copier, the copier could have another, *initialise*, binding site just before its *end* site. Binding to the *initialise* site would still run the *end* program and correctly initialise the copier, but the *initialise* site could be `hidden` by the chaperone chemical after initialisation. This would allow the chaperone chemical to initialise the copier chemical (or any other mechanism) once only, and not interfere with its copying mechanism. It would also allow other mechanisms to `show` the *initialise* site at a later time, if there was any reason to re-initialise the copier chemical.

After initialisation, we create an empty GraphMol world and put into it one DNA chemical and one copier chemical. We bind the copier's *start* site to the DNA's *start* site and run the (macro-scale) reaction until the copier unbinds from the DNA, thus finishing its copy. We then take our measurements of speed and accuracy before destroying this GraphMol world.

7.2.3 Defining accuracy (nearly-perfect copying)

Since the copier can make different types of mutation in its copy of the DNA, there are many different ways to define accuracy. Our definition counts the number of *nearly-perfect copies*. The copier chemical is designed to copy a sequence of DNA perfectly *most of the time*, but occasionally to make small mutations for evolution to exploit. To make this concept rigorous, we define a *nearly-perfect copy* as a copy that differs from the original by at most three bases, i.e. any combination of three insertions or deletions.

The value *three* here is a parameter that could be set to any whole number between zero and the length of the DNA sequence being copied. Biologically, it corresponds to the number of DNA bases that can be changed simultaneously in an organism's DNA before the combination of changes breaks the organism, making it no longer viable. Clearly, the effect of mutations in biological organisms depends on precisely which part of the DNA is mutated, rather than just how many bases are changed. Some regions need to be conserved precisely, while others can undergo huge changes with no phenotypic difference in the organism. But this experiment does not focus on individual mutations to individual organisms: it is looking at the statistical properties of the copying mechanism (how quickly it makes copies with few mutations). The idea is that, for evolution to work properly, we want a copying mechanism that:

1. Makes some perfect copies
2. Makes some copies with small mutations
3. Can (rarely) make copies with large mutations

Since we know that there is always a probability of the copier making a large mutation (section 6.4.3), we concentrate on the small mutations as our measure of *accuracy*. We define a mutation of three bases or less as a *small mutation*. The value of three is not critical: values other than three have been tested and give qualitatively similar results, but larger values are more noisy so more experiments would need to be run to obtain the same error bars on the graphs.

To calculate if a copy is nearly-perfect, we use Smith-Waterman alignment [91]. The Smith-Waterman algorithm measures the length of the longest common subsequence between two strings, taking into account (and penalising) short insertions and deletions. We set the penalty for an insertion or deletion to be 1, to measure the number of errors in the copy (subtracting the length of the original DNA, and taking the absolute value). If the number of errors is three or less, the copy is *nearly-perfect*.

7.2.4 Parameters

The parameters of the GraphMol world are as described in chapter 6, except for the distance spreading cutoff of the graph distance process (section 6.3.1.3), which is increased to 300 atoms. Because this experiment involves copier chemicals with high levels of junk (up to 20 atoms), the cutoff distance is increased to ensure that these extreme copiers are still able to make all of their binds.

7.3 Results

The graphs shown in figures 7.1 and 7.2 refute the null hypothesis and show that the GraphMol copier's junk level does affect its speed and accuracy of copying. Figure 7.3 shows the predicted *sweet spot* in the rate of copying.

7.3.1 Accuracy of copying

As the junk level increases, the copier becomes more accurate at copying its DNA (figure 7.1). This is in spite of there being more DNA to copy at higher junk levels (equation 7.1), so more chances of making an error. Accuracy is measured as the number of nearly-perfect copies produced by the copier chemical, out of its 80 reactions.

The reason for the increase in accuracy is that as the junk level increases, the distances increase between the copier's feet and the binding sites on the DNA. This increase in distance between the binding sites reduces the probability of each bind. The important point for this experiment is that GraphMol's binding probability

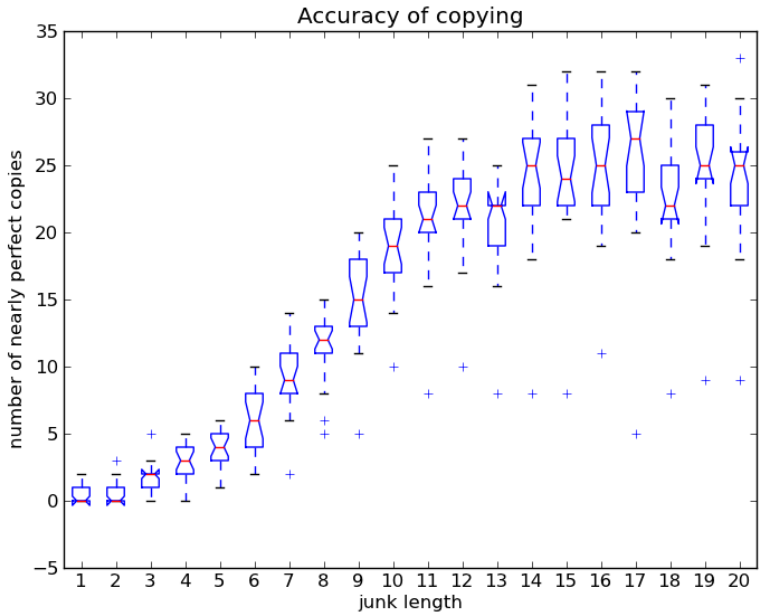


Figure 7.1: Accuracy of copying increases with junk level. The notches show the 95% confidence intervals on the medians.

function is non-linear (see section 6.3.1.2). It reduces the probability of an 'erroneous' bind more than it reduces the probability of a 'correct' bind, as shown in figure 6.26. This means that increasing the junk level decreases the probability of an 'erroneous' step, relative to the probability of a 'correct' step. This makes the copier more accurate with more junk.

7.3.2 Speed of copying

As the junk level increases, the copier takes longer to copy its DNA (figure 7.2). This is for two reasons.

1. More junk makes the distances between binding sites larger (as explained above). This reduces the probability of each bind, making each of the copier's steps take more time.

2. More junk means there is more DNA to copy, so the DNA takes more steps to copy.

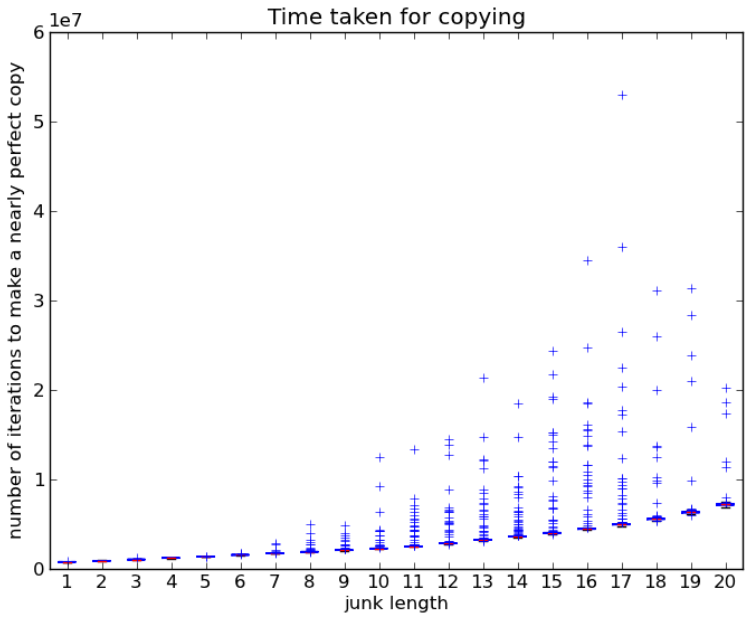


Figure 7.2: Copying time increases with junk level (so speed decreases). These are boxplots; the variation is too small to see (apart from some large outliers at high junk levels, that take a long time to copy).

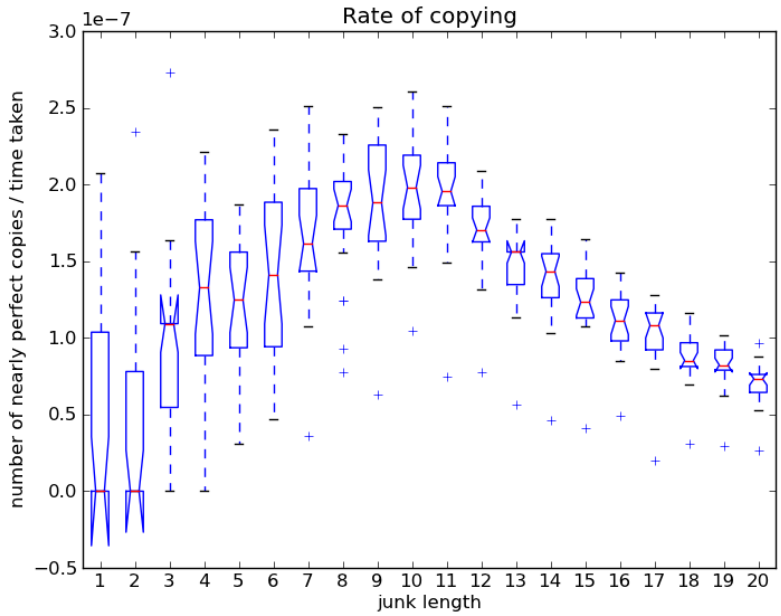


Figure 7.3: Rate of copying has an optimum junk level, trading off speed against accuracy. The notches show the 95% confidence intervals on the medians.

7.3.3 Rate of copying

A copier with a low junk level is fast but error-prone, whereas a copier with a high junk level is slow but reliable. So, the copier can trade off accuracy against speed. We can see this tradeoff by graphing the rate of copying for each junk level (figure 7.3). This is not a quantity that we measure directly, but is calculated as the number of nearly-perfect copies made, divided by the time taken to make them. The graph is noisy at low junk levels because few nearly-perfect copies are made here (as can be seen from the accuracy graph, figure 7.1). The tradeoff can be seen in this graph as a peak at a moderate level of junk. This *sweet spot* is the junk level at which the copier chemical achieves its highest rate of accurate copying.

- Too much junk and the copier copies too slowly, making its *rate* of accurate copying low.

- Too little junk and the copier makes too many errors, making its rate of *accurate* copying low.

The consequences of this *sweet spot* are discussed below.

7.4 Discussion

When the copier chemical is put into a world where it can evolve, it will be able to control its own junk level through mutations that add or remove junk atoms. These results show that changing the copier's junk level changes its position on its speed/accuracy tradeoff for copying. Thus (when it is evolving in different environments) the copier will be able to find, for itself, the tradeoff between speed and accuracy that optimises its survivability in its particular environment.

This tradeoff between speed and accuracy of copying is not specific to the GraphMol copier. It is a common feature of many systems, whether they be virtual (such as artificial chemistries), lab-based (such as self-replicating RNA molecules in a test tube [51]) or biological (such as bacteria and viruses). The precise details of this tradeoff depend on the parameters of the world in which the copying mechanism is embodied. In GraphMol, the important parameter is the binding distance probability equation. The difficult task in designing a virtual world (such as an artificial chemistry) is making the world rich enough that there is structure for the embodied mechanisms to exploit, while keeping it simple enough that the computer code can be run in a reasonable time on modern hardware.

7.4.1 Replicating quickly

As stated at the beginning of this chapter, the purpose of this experiment is to show the general idea of how low-level changes to GraphMol mechanisms can feed through to measurable high-level changes in the embodied phenomena. Thus the *specific changes* demonstrated in this experiment (junk length affecting rate of accurate copying) are less important than the *fact that such changes are possible*. But nevertheless, it is worth discussing the implications of these specific changes.

7.4.1.1 Growth laws

A naïve view of replication is: copiers that can replicate quickly will have a fitness advantage, and will displace slower replicators in a population. This seems intuitively true, but is not always the case. Whether one replicating species will replace another in a population, depends to a large extent on the *growth laws* of each species. These are idealised differential equations modelling how the species would increase in number over time, if it had unlimited resources and no competition. Much work has been done on growth laws relevant to biological systems thought to be related to the origin of life. A concise summary is given in [96], which describes three very different growth laws:

1. Exponential growth, given by the growth law: $\dot{x} = kx$.
   This leads to *survival of the fastest*, where the quickest-replicating species

displaces the others (the species with the highest k).

2. Hyperbolic growth, given by the growth law: $\dot{x} = kx^2$.
   This leads to *survival of the common*, where the species starting with the highest concentration displaces the others (the species with the highest x).

3. Parabolic growth, given by the growth law: $\dot{x} = kx^p$ for $0 < p < 1$.
   This leads to *survival of everybody*, where all species survive despite competing for the same resources.

In these equations: x represents the number of chemicals of the species; $\dot{x}$ represents the derivative of x, which is the rate at which the species is copying (i.e. the quantity I show in figure 7.3); and k is a constant, allowing different copiers with the same growth law to grow at different rates.

Each of these three different growth laws leads to systems with different dynamics. Biologists trying to understand the origin of life are concerned with working out which growth laws could have applied at which times to the precursors of life. In artificial systems, we are able to choose the growth laws we want our systems to have. We do not need to be constrained by how life started; we can design our systems to have the growth laws that give the behaviours we want.

7.4.1.2 Reaction equations

In a well-mixed space such as a *primordial soup* (for the origin of life), a chemostat (for biochemistry experiments), or an aspatial artificial chemistry (such as GraphMol, section 6.3.1.1), these three growth laws can be obtained as follows:

Exponential growth is produced by a copier chemical reproducing asexually, taking in raw materials and making a copy of itself and some waste products:

$$C + M \rightarrow 2C + W, \tag{7.2}$$

where C is the copier chemical, M is the raw materials and W is the waste product.

Hyperbolic growth is produced by a copier chemical reproducing sexually, where two copier chemicals are required to create a third from raw materials:

$$2C + M \rightarrow 3C + W. \tag{7.3}$$

Parabolic growth is more complicated to produce. It requires a copier chemical that attaches raw material to itself, then takes some time before dissociating into two copies:

$$C + M \xrightarrow{c} CC \tag{7.4}$$

$$CC \underset{b}{\overset{a}{\rightleftarrows}} C + C, \tag{7.5}$$

where the rate constants a, b and c are such that $a \gg b > c$. The exponential and hyperbolic growth laws can be clearly seen from these reaction equations. The parabolic case is less obvious, but is explained in [96].

The evolutionary biology literature on reaction equations is concerned with divining those equations that apply to physical systems relevant to the origin of life, and extrapolating their growth laws (see [96] for references). The above reaction equations only yield the growth laws above in a *well mixed* space. The benefit of artificial chemistries is that *we are able to define the space*. GraphMol uses a well-mixed space for simplicity, and because the experiments performed in this thesis do not investigate growth laws. But different artificial chemistries (or modifications of GraphMol) could be made with different definitions of space, so that the above reaction equations lead to different growth laws.

7.4.1.3 Consequences for artificial chemistries

The consequence of this for artificial chemistries is that we have two choices to make, which depend on what we what to do with the artificial chemistry. The choices are:

- Which growth laws we want our system to use.
- Which replicating mechanisms and definition of space we will use to obtain these growth laws.

We might want to see *selective* evolution: sweeps of change where one dominant species is suddenly replaced by a *fitter* species. We can obtain this using exponentially-growing replicators that are able to change their replication rate by low-level mutations. With an aspatial artificial chemistry, this means we have to use copying chemicals that reproduce asexually. Or if we want to use a different copying mechanism, then we have to use a different type of space where that mechanism leads to exponential growth.

If we have hyperbolically-growing replicator species, then changing their replication rate is not enough to see selective sweeps. In this case, we need a different mechanism by which one species can take over from another. One example of this is parasitism, where one species uses the other to replicate itself. It has been shown that some types of parasite can cause selective sweeps under hyperbolic growth [57, p.68].

Alternatively we might want to generate different novel forms of replicator species, without selective sweeps (similarly to how novelty search operates, section 2.5). In this case, parabolic replicators may be more appropriate. Exponential or hyperbolic growth would reduce the diversity of species in the world, whereas parabolic growth (with mutation) would increase diversity.

It is important to note that these growth laws are only a *model* of how an actual system will behave. Necessarily, they make assumptions about the system. The most important of these assumptions are that the space is well-mixed, and that stochastic effects can be ignored. It is easy to implement artificial systems (and physical ones) in which these assumptions break down. For example, if we have chemical species existing in very low numbers (for example, DNA chemicals), then they will be very affected by stochastic events, and by definition they cannot be "well-mixed". Alternatively, if we have an artificial chemistry with a Euclidean definition of space (section 6.3.1.1), if there are high numbers of different species of chemicals then they will experience crowding effects and so will not be well-mixed. Therefore these growth laws should be applied only where they can be shown to be appropriate.

7.4.2 The copier's control over its junk regions

When the copier chemical is free to evolve, it will be able to change the lengths of each of its 31 junk regions independently. So it will have a much finer level of control over its junk regions than this experiment suggests. But although the copier has control over its junk regions *in principle*, this does not necessarily mean that it will be able to accurately control its junk regions *in practice*.

As a thought experiment (and a simplification of the evolving copier), consider the copier's binding sites and functional code to be fixed (the copier cannot mutate them). From the viewpoint of evolution, this situation might occur if they could technically be mutated, but any change to these parts created a copier chemical that was no longer able to make copies, so would not survive. The copier is able to mutate the lengths of each of its 31 separate junk regions. This creates a 31-dimensional *search space* in which the copier can evolve, shown schematically in figure 7.4. The experiment performed in this chapter looks at a 1-dimensional line through this space. It shows that different points in the space produce copiers with different copying rates, which will correspond to different survivabilities in an environment (i.e. different *fitnesses* in an evolutionary sense).

Although the search space is easy to describe, it is not obvious how the evolving copier will be able to move through this space. This is because the copier encodes the ways in which it can move through this space, and these can change as it moves through the space (as it evolves). In a traditional evolutionary algorithm, there would be a *crisp mutation function* describing the possible moves an organism could make through its search space. (For example: moving 1 unit in any dimension; or moving to any point less than 3 units away.) Wherever the organism was in the search space, the mutation function would be the same, providing the same movement options to the organism (as shown in figure 7.5(a)). But with an *embodied mutation function* (as in the GraphMol copier), the mutation func-

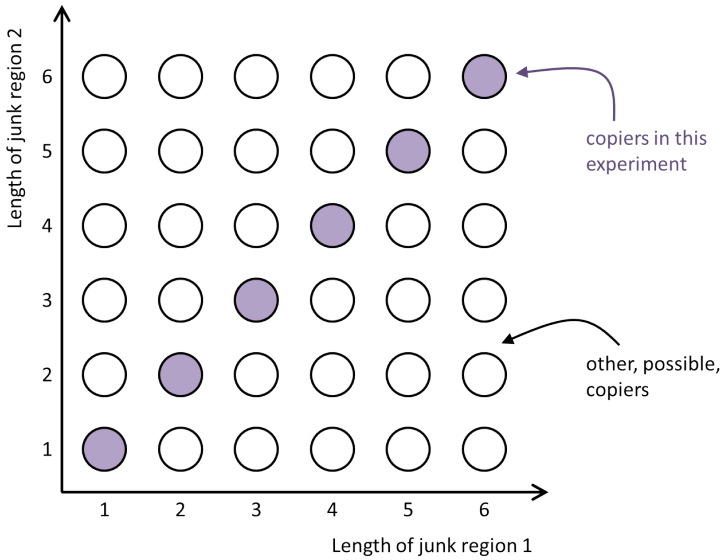
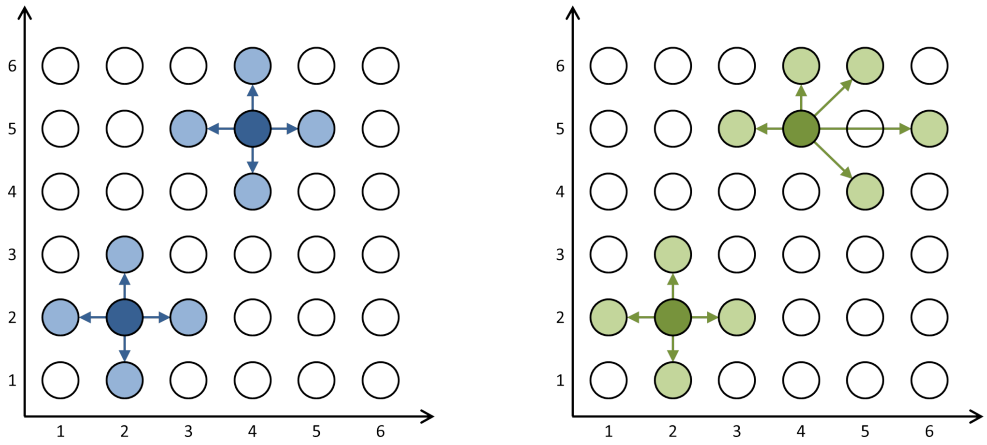


Figure 7.4: A 2-dimensional example of the search space of copier chemicals with different junk lengths. The points investigated in this experiment all have the same junk lengths, but the copier could evolve to different junk lengths by moving through this search space.



(a) Crisp mutation function

(b) Embodied mutation function

Figure 7.5: A crisp mutation function is the same at all points in the search space, but an embodied mutation function can change as the organism moves through its search space. (These diagrams are illustrative only, and are not examples of actual systems.)

tion changes as the organism (copier) moves to different points in its search space (as shown in figure 7.5(b)). So at different points in its search space, the copier chemical (potentially) has different options as to how it can make its next move to another point.

This chapter's experiment illustrates that copiers at different points in their search space can have different *probabilities* of making mutations (i.e. changing their mutation rate). And further to this, we can see that copiers at different points in their search space can make different *types* of mutation. For example, the copier chemical is designed to start its mutations by stepping one of its feet wrongly on the DNA, either jumping forwards or backwards. The copier chemical described in chapter 6 can only jump forwards or backwards one base on the DNA. But with different junk levels, it can jump further. For example, a copier chemical with a junk level of 19 has to bind over a distance of 301 atoms in order to jump backwards two bases. But for a copier chemical with a junk level of 4, this distance decreases to 286. So, if the cutoff of the graph distance process is 300 atoms, then the 4-junk copier can perform a behaviour that the 19-junk copier cannot (jumping backwards two bases). This changes the types of mutation that the copier is capable of performing.

An important point about this version of meta-evolution is that we do not have a separate organism and mutation function co-evolving. The mutation function is *not adapting* to the organism's position in its search space. The mutation function is a *consequence* of the organism's position in its search space. The organism and mutation function cannot be separated and evolved separately, because the mutation function is *part of the organism*. The *organism and mutation function combination* is what adapts to the environment it finds itself in. This is what it means for the mutation function to be embodied within the organisms it is mutating.

7.4.3 Designing embodied mechanisms

When designing embodied mutation functions, it is not possible to know (and so design) all the possible ways in which the mutation function could evolve. But there are things that we can design:

1. We design the initial mechanism, with its initial structure and parameters (e.g. the copier chemical with 6 feet and 31 junk regions each of length 12).

2. We can analyse this initial mechanism to determine the mutations it can perform, or design it to perform specific mutations (e.g. small deletions, small insertions, large insertions).

3. We design the world in which the mechanism exists, which implicitly determines how the mutation function can evolve (even though we cannot necessarily predict this).

Although we do not know *how* the mutation function will evolve, we can design the world and the initial mechanism in such a way that we allow the mutation function *to* evolve. To date, there is no foolproof method for designing evolvable worlds, although this is an ongoing area of research. We feel that worlds can be made more evolvable by avoiding crisp processes in the world, and instead using *rich* ones. For more details about this, see [45] [17] [44].

For systems with a crisp mutation function, the designer has a completely free choice of which mutation function to use. They can choose any possible way of moving from one point in the search space to a set of mutants. But with embodied mutation functions, the mutation function must be embodied within a mechanism implemented in the world (in this case, a copying chemical). This places constraints on the possible mutation functions that can be realised. As well as placing constraints on the initial mutation function that is designed into the mechanism, embodiment also places constraints on the ways in which the mutation function is able to evolve.

It is not clear (to me) whether it is theoretically possible to embody any possible mutation function within a copying mechanism existing in a world. But it feels like this is not possible, at least for all practical purposes. Even if it were possible in theory to embody any mutation function, many of those embodiments might not produce evolvable implementations of the mutation function. Some mutation functions might be so awkward to embody that their embodiments would be effectively crisp implementations of that mutation function within a world that did not allow them to evolve. Doing this would serve no purpose. It would be merely an inefficient way of implementing a crisp mutation function. So if our purpose in embodying mutation functions is to make the mutation function evolvable (i.e. meta-evolution), then there will be some mutation functions that we cannot embody (either by designing them or evolving them).

It is not a bad thing that we cannot embody every possible mutation function. The vast majority of possible mutation functions are arbitrary, not interesting to us and probably not very useful for evolution (for example: "Increase every junk region by 42 atoms, except the seventh junk region, which should be decreased by five atoms, and change the ninth atom into a `show` function."). We are not looking for *every possible* mutation function; we are looking to evolve mutation functions that are interesting to us as humans, or functions that exploit the structure inherent in the problem they are given to solve. The fact that we cannot reach every possible mutation function does not mean that we cannot reach a large range of useful and interesting mutation functions. Indeed, removing many of the arbitrary

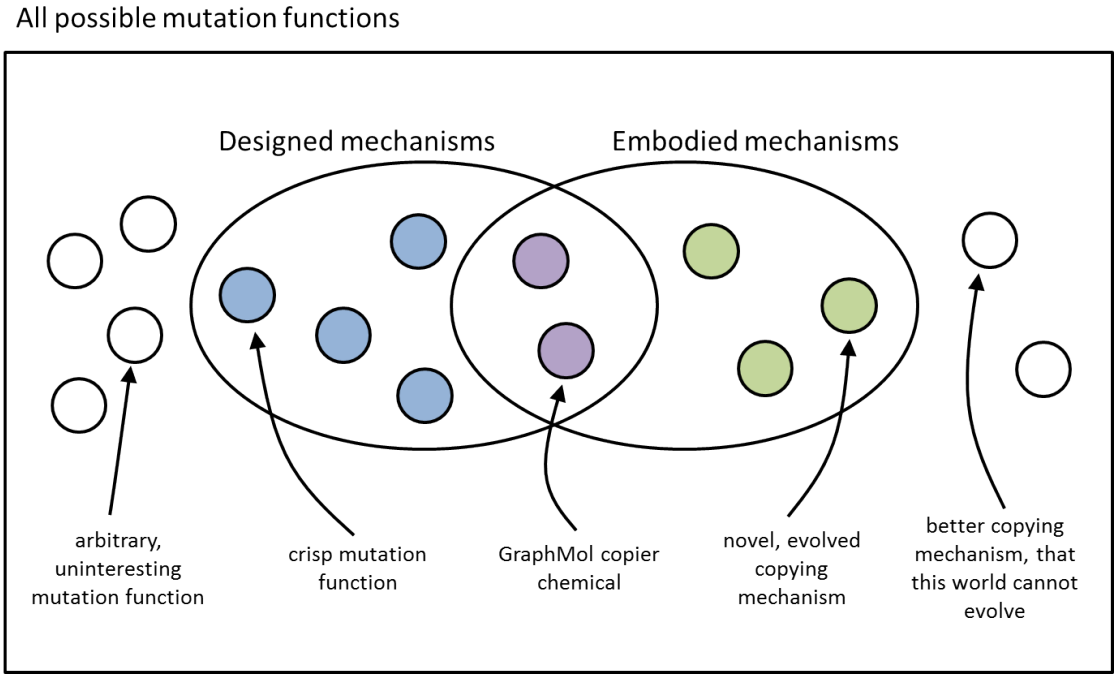


Figure 7.6: There are many possible mutation functions, most of which are arbitrary and uninteresting. Of the mutation functions that humans would consider to design, some are crisp and some are embodied. We hope that embodied designs will be able to evolve into embodied mechanisms that are interesting and useful, but that humans would not have thought to design (or been able to design). For any given world, there may be better copying mechanisms that cannot be embodied within that world.

mutation functions may even help us to reach the interesting and structured ones. This situation is similar to the *no free lunch theorems* [103], which imply that all optimisation algorithms perform equally well, when all possible fitness landscapes are considered. These theorems are not applicable to the types of problems that humans want to optimise, because we do not want to optimise every possible fitness landscape (most of which are very arbitrary and not interesting); we just want to optimise the fitness landscapes that we are interested in. Equally, it does not matter that we cannot embody every possible mutation function, because we are not interested in all of them.

From the viewpoint of novelty-generation algorithms, this situation is a good thing. It provides exactly what we need for novelty-generation. Figure 7.6 summarises this. Embodiment is about moving up a meta-level, for example: embodying copying, rather than explicitly writing a mutation function. To do this, we design an initial mechanism within a world, and hope that this mechanism will

evolve to an interesting place that we would not have thought to design. The evolving mechanism cannot evolve to all of the possible mutation functions we could have implemented; its evolutionary options are limited by the world in which it is embodied. This has both positive and negative implications:

- The positive is: the mechanism cannot evolve to many of the arbitrary, uninteresting mutation functions that we do not want. Any evolved mechanisms will have some structure to them, because they are embodied within a world.
- The negative is: there may be excellent copying mechanisms that we cannot evolve, because of the constraints imposed by the world. But we can explore these by designing different worlds with different constraints, to see what effect they have on the evolvability of mechanisms embodied within them.

This is analogous to the transition from traditional evolutionary algorithms to evo-devo algorithms (section 2.3.3). We can no longer evolve arbitrary results, but all the results that we do evolve are more structured. The form and details of this structure are determined by the *problem representation*. For evo-devo algorithms, this is the development function; for the meta-evolution of embodied mechanisms, it is the way in which we embody these mechanisms within a world.

In order to evolve novel mechanism, we need a world that is suitable for both:

1. Designing embodied mechanisms, so that we can implement an initial mechanism within the world.
2. Evolving the initial mechanism, to a novel place that is interesting to humans.

The previous chapter shows that the GraphMol world fulfills the first point: we can design an embodied copying mechanism in GraphMol. This chapter's experiment shows that *in principle* the GraphMol copier is able to evolve, with low-level changes to its mechanisms being able to propagate upwards to measurable high-level changes in its phenomena. The following chapter performs experiments investigating how well the GraphMol copier evolves *in practice*.

Chapter 8

GraphMol copier evolving

The previous feasibility experiment (chapter 7) shows that GraphMol's embodied copying mechanism can evolve *in principle*. It has a search space of different instantiations of the copier chemical, through which it can potentially evolve. This chapter describes a sequence of experiments designed to investigate how well the copier chemical actually evolves *in practice*.

The previous experiment ran one reaction within a GraphMol world (between a copier chemical and a DNA chemical). It repeated this process multiple times, with multiple GraphMol worlds. This experiment does the opposite. We create a single GraphMol world, in which we allow multiple copier chemicals and DNA chemicals to react. This creates an environment in which the copier chemical can potentially evolve. Copier chemicals copy DNA chemicals that encode copiers. Thus the copier chemical makes mutations in future generations of copier chemicals. These future copiers compete with their parents, giving an intrinsic definition of *fitness*.

This is the standard experiment performed on artificial chemistries: initialise a world with some chemicals and allow them to react, looking out for novelty-generation. This type of experiment is necessary to test the novelty-generation capabilities of an artificial chemistry. To make an analogy with biology:

- In the previous experiment, the world was very sparse and tightly-controlled. This is analogous to a lab-based experiment, where one parameter is varied and the others are controlled for.

- This experiment is analogous to going into the field and observing the system in its natural environment. There are many different parameters affecting the system, but we are not trying to characterise the consequences of each one. We are allowing the system to run on its own and we are observing it, to see what behaviours it displays.

From the viewpoint of novelty-generation, the tightly-controlled *(lab-based)* experiments can be used to help understand the parameters and design decisions of a

novelty-generation algorithm, but they cannot prove that an algorithm is capable of actually generating novelty. They can only prove that an algorithm is *incapable* of generating novelty (for example, if the *sweet spot* had not been found in the previous experiment). Open-ended *in the field* experiments are needed to show that an algorithm actually can generate novelty.

Due to the open-ended nature of this experiment, and the length of time that GraphMol runs take, the results are not as quantitative as in the previous experiment. We present a collection of GraphMol runs with different parameters, and discuss the behaviours displayed by the system. These experiments do not show the results we expected beforehand, but by analysing their results in different ways we are able to narrate the story of each experiment. We identify two distinct types of behaviour displayed by the system: a dynamic equilibrium in which the number of chemicals in the world remains stochastically constant over time, even though the chemicals mutate and new copier species are produced; and a spike-and-death, where the number of chemicals in the world increases massively and then plummets as the world dies.

8.1 Hypothesis

We hypothesise that the behaviour predicted by the previous *lab-based* experiment will be seen *in the field*:

1. When initialising the world with copier chemicals containing low levels of junk, we expect them to evolve, increasing their junk level until they reach an optimum rate of copying.

2. When initialising the world with copier chemicals with high levels of junk, we expect them to reduce their junk level until they reach the same optimum rate of copying.

We do not expect the evolving copier chemical to find precisely the same *sweet spot* as observed in the previous experiment (all junk levels around 9-10 atoms). We expect it might obtain a higher copying rate by having some of its junk regions at different lengths.

One potential caveat here is that we may not see evolution at all. Because GraphMol uses a well-mixed aspatial space, and the GraphMol copier chemical reproduces sexually, we might see hyperbolic growth rather than exponential growth (section 7.4.1). This could cause the novelty-generation of the system to be impaired. But GraphMol is very complicated, and violates many of the assumptions that these types of analysis make. So it is worth trying the experiment to see how well the actual results match with the pure theory. This is discussed towards the end of this chapter.

Because of the open-ended nature of this experiment, we do not presume the ability to predict everything that the evolving copier might do. So we also hypothesise that the copier might evolve a behaviour different from those programmed into it. An example of this is Stringmol's replicase chemical evolving a macro-mutation and hypercycle (as described in section 5.2.2.1). We do not know if a new type of mutation or copying will evolve here, but it is the ideal goal of this type of experiment: showing an example of novelty-generation. So when analysing the results of this experiment, we will keep an open mind as to the possible behaviours that GraphMol might demonstrate.

The *null hypothesis* of this experiment is that the copier chemical will not evolve. It will make copies of itself, and it will make mutated versions of itself, but none of these mutants will have a fitness advantage over the original copiers. Over time, the only dominant chemical species in the GraphMol world will be the original copier chemical.

This experiment manages to refute its null hypothesis: the copier chemical evolves, producing mutant copiers chemicals that displace the original copier chemical from the population. But it does not evolve as neatly as might be expected. We (implicitly) hypothesised that the GraphMol world would have one single dominant copier species, sustaining itself, that would be replaced by a *fitter* mutant. This does not happen in these experiments. Instead we have a collection of copier species that sustain the *phenomenon* of copying, while the particular *species* embodying the phenomenon change over time. Also, collections of *parasites* appear in the world and cause the world to die.

We do see changes in the lengths of the copier chemicals, but not in the clean, idealised manner hypothesised above. The copiers become longer a lot more than they become shorter, because of the bias in GraphMol's embodied mutation process (tangles producing long insertions, but no equivalent process for long deletions). The copier chemicals appear to have a maximum length that they attain in a stable world, but when the world is dying they exceed this.

8.2 General experiment design

Saying that we will *put the copier chemical into a GraphMol world and allow it to evolve* is an imprecise statement. There are different types of copier chemical (with different junk levels), different types of GraphMol world (different parameter values) and there are different ways in which we can allow the copier chemical to evolve. This chapter contains a series of related experiments, exploring different copier chemicals evolving in different GraphMol worlds. In this section, we describe the design decisions that are common to all of these experiments. We then describe each separate experiment, evaluating its results.

The major factors affecting these experiments are:

1. The mechanism of evolution: how a chemical copying DNA can reproduce itself.
2. The copier chemical: its junk level.
3. The GraphMol world: its mixing process and energy model.

Each of these is described in detail below.

8.2.1 Evolution from copying

In the GraphMol world, we have *copier chemicals* and *DNA chemicals*. Copier chemicals can copy DNA chemicals, and DNA chemicals can encode copier chemicals. In order to implement biology's copying phenomenon (as described in section 4.2) we also need a third process, an analogue of the *expression process*: decoding the information contained within a DNA chemical, to build a new copier chemical. We have not implemented an embodied expression process within the GraphMol world. Instead, we implement the expression process crisply. Our method of doing this is described below.

There are three different ways in which we can represent a GraphMol copier mechanism:

1. As a *copier chemical*
2. As a *DNA chemical*
3. As a *genome*

These three representations interact, allowing the copier mechanism to replicate itself.

A *copier chemical* is a chemical in the GraphMol world. It has six feet that walk along a GraphMol DNA chemical and copy it. A *DNA chemical* is a chemical in the GraphMol world that can be copied. It encodes another GraphMol chemical in its pattern of bases. A *genome* is an abstract entity that does not exist within the GraphMol world. It is a sequence of characters (A, B, C, D), that represents an encoding of a GraphMol copier chemical in terms of DNA bases. These three different entities are isomorphic in terms of the copying mechanisms they represent. Any one of these three entities can be changed easily into either of the others. We choose the *genome* to be our canonical representation of a copying mechanism, since it does not exist within the GraphMol world. In order to instantiate a genome within a GraphMol world, we either transform it into a DNA chemical or express it as a copier chemical (illustrated in figure 8.1).

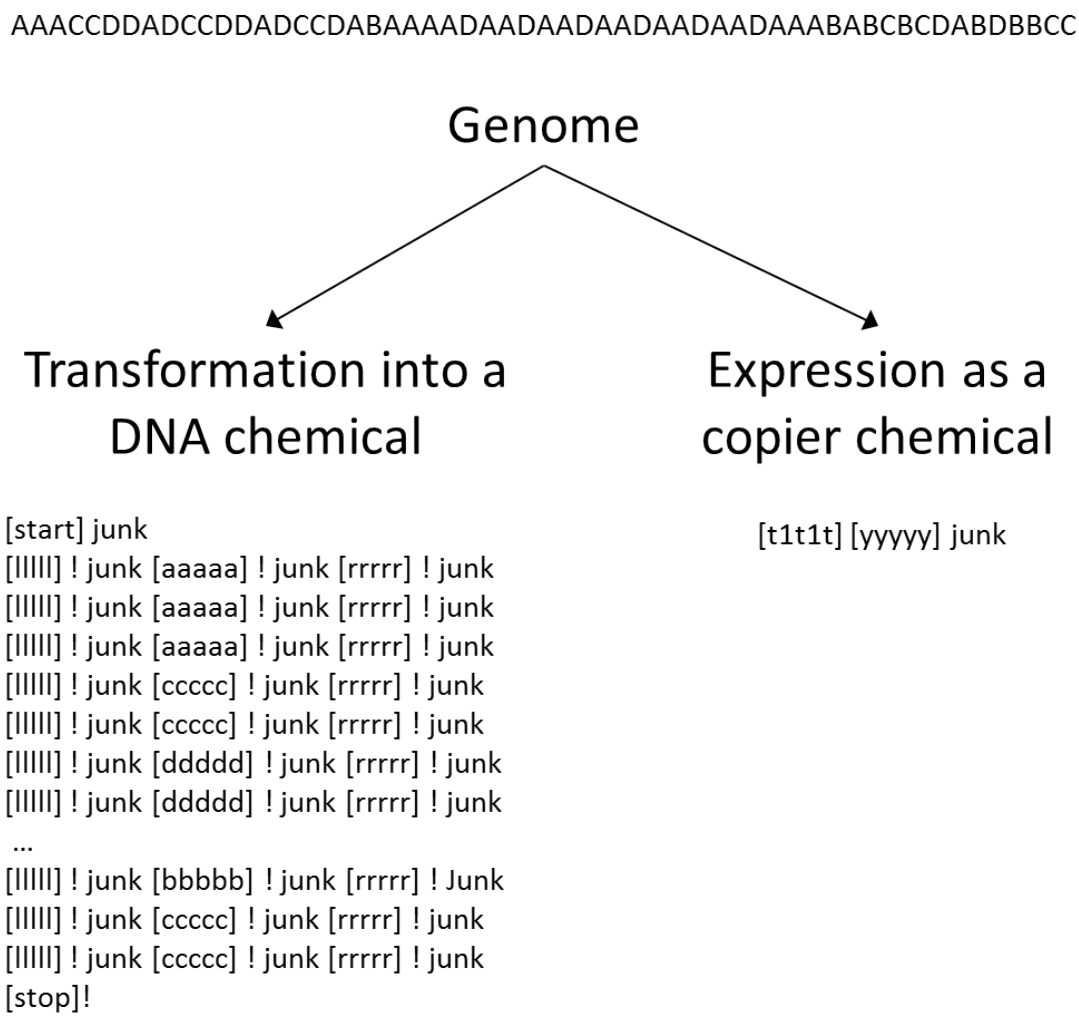


Figure 8.1: Each genome can be processed in two different ways: it can be (1) transformed into a DNA chemical; and it can be (2) expressed as a copier chemical. Note that the DNA chemical here has had some sections removed so it can fit in this figure.

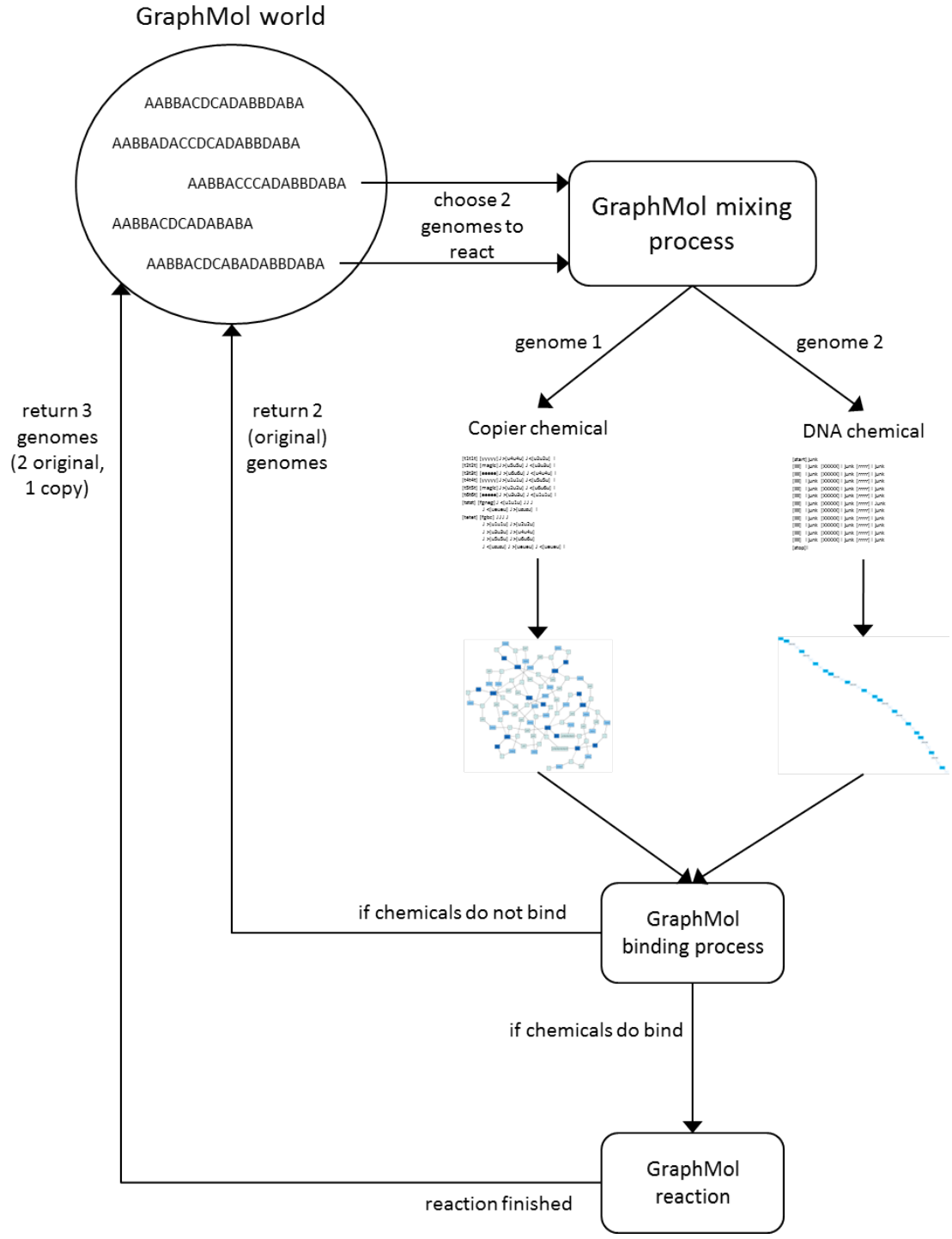


Figure 8.2: The process by which evolution happens in these experiments. Genomes are represented explicitly; a crisp process transforms two genomes into a copier chemical and a DNA chemical, allowing them to react. Multiple reactions happen in parallel, starting and ending asynchronously.

Figure 8.2 shows how these three representations interact, allowing evolution to happen. Into the GraphMol world, we put 20 instances of the genome of the copier mechanism. The GraphMol mixing process chooses pairs of genomes to react. Given a chosen pair of genomes, we transform one into its DNA chemical and the other into its copier chemical. These two chemicals are then allowed to bind and react in a separate GraphMol world. Once the reaction has finished, we put the copied genome into the original GraphMol world, along with the two genomes of the copier and DNA chemicals.

GraphMol's energy model controls the number of chemicals in the world, by decaying chemicals at random and slowing down reactions when too many are happening in parallel.

As the copier chemical mutates, it has the possibility of destroying its *start* binding site, thus preventing it from binding with a DNA chemical. If this happens, then the two genomes are returned to the GraphMol world without a reaction (and copy) happening. The GraphMol mixing process cannot tell in advance if a genome encodes a copier that cannot bind to the DNA; it must express the copier and test if a bind is possible. But it does memoise the results of these tests, so it does not need to pair up non-binding genomes more than once.

Note that whichever genome is transformed into a DNA chemical, the resulting DNA chemical will always have the same structure of backbone and base units, and will always function correctly as a DNA chemical. The only difference will be in the patterns of the base binding sites (`aaaaa`, `bbbbb`, `ccccc` and `ddddd`). Figure 8.1 illustrates this. But when a genome is transformed into a copier chemical, different genomes can encode very different copier chemicals. Some genomes may even encode 'copier' chemicals that have mutated so deleteriously that they no longer function as copiers. The transformation into a DNA chemical simply takes the information contained within the genome, and embeds it into a GraphMol DNA chemical with a fixed format. But the transformation into a copier chemical is different: it decodes the information in the genome, creating an arbitrary GraphMol chemical.

As described in section 7.2.1, the length of the copier chemical is:

$$\text{copier\_length} = 311 + \text{copier\_junk\_length} \times 31, \tag{8.1}$$

for a copier chemical with all of its 31 junk regions the same length. The experiments in this chapter use two different ways of encoding the copier chemical on the genome, but if we assume that three bases are used to encode each atom of the copier chemical (as in biology), then we have that:

$$\text{genome\_length} = 3 \times \text{copier\_length}. \tag{8.2}$$

From this, we can work out the length of the DNA chemical into which this genome

can be embedded:

$$\begin{aligned}&\text{dna\_chemical\_length} = \\ &\quad 14 + \text{dna\_junk\_length} + \text{genome\_length} \times (24 + 3 \times \text{dna\_junk\_length}).\end{aligned} \tag{8.3}$$

For example, copier chemical with junk length 16, has a length of 807 atoms. This equates to a genome of length 2,421 bases and a DNA chemical of length 348,678 atoms (using a DNA junk length of 40, as discussed in section 6.4.1).

8.2.2 Different copier chemicals

As discussed in the previous chapter, we can create different copier chemicals by varying the sizes of each of the copier's junk regions. We expect evolution to adjust the sizes of these junk regions, but we must choose which junk levels to use in the initial *bootstrap* copier chemical that the world is initialised with. The hypothesis section (8.1) explains our strategy of performing different experiments on junk level: initialising the world with a high-junk copier to see if it reduces its junk levels; and initialising with a low-junk copier to see if it increases its junk levels.

For simplicity, we keep two factors constant regarding the copier chemical's junk level:

1. We do not vary the junk levels *within* a copier chemical. This is to reduce the number of permutations of experiments needed, and to test whether evolution will produce copiers with some junk regions at different levels.

2. We initialise the world with copiers of only one junk level. This ensures that any observed behaviours are due to the copier chemical mutating itself, rather than interactions between different types of copier chemical. It also makes these experiments comparable to Stringmol's *diversity from a monoculture* experiments [42].

There are obvious ways of varying these two factors, but they create many different permutations of experiments. There are 26 different junk regions in the copier chemical (section 7.2.1), so varying each of them would require n^{26} different experiments to investigate fully, where n is the number of different lengths tested for each junk region. Initialising the world with copiers of different lengths entails a potentially unbounded number of experiments, although we could use scenarios requiring fewer experiments, such as *short junk versus long junk*. To curtail this potential proliferation of experiments, we keep constant the two factors above.

8.2.3 Mixing process and energy model

GraphMol's aspatial mixing process (described in section 6.3.1.1) and its energy model are the same as Stringmol's. This is to make these experiments comparable

parameter	value(s)	description
v_{world}	100	volume of the world
v_{chem}	10	volume of a chemical
initial energy	1000	energy added to a newly-created world
energy flux	10 – 100	energy added into the world each timestep
decay rate	0.00001	probability of decay each timestep

Figure 8.3: The parameters of the GraphMol world relating directly to this experiment.

with Stringmol's, and because Stringmol has shown that this mixing process and energy model can lead to novelty-generation. The parameters of these processes are shown in figure 8.3, along with their values as used in this experiment.

8.2.3.1 v_{world} and v_{chem}

The parameters v_{world} and v_{chem} are set to values that have been shown useful in Stringmol. The absolute values of these parameters do not affect the algorithm: it is their ratio that matters ($v_{\text{chem}}/v_{\text{world}}$), which represents the *concentration* of chemicals within the world. Having a ratio close to 0 would mean that the solution would be very dilute, so many chemicals would be needed before they were likely to react with each other. Having many chemicals existing in the world would make the code take a long time to run. Having a ratio close to 1 would mean that we could still have a concentrated solution with very few chemicals existing in the world, because the volume of the world would be similar to the volume of a chemical. This would be the same as pairing up each of the chemicals in the world, which would negate the benefit of using this mixing algorithm. A ratio greater than 1 is not sensible (it would lead to reactions having probabilities greater than 1). So intermediate ratios are desired, to balance the benefits of having the concept of volume against the run-time of the code. We use an intermediate ratio of 0.1.

8.2.3.2 Initial energy

The GraphMol world is not very sensitive to the initial energy added to the world when it is created. If the initial energy is set too low, then the copier chemicals cannot start copying instantly. They must wait for the energy flux to provide them with some energy that they can use to react and produce copies. In this case, the copier chemicals have a chance of all decaying before they have managed to make any copies. If the initial energy is set too high, then the copier chemicals will start to produce copies quickly, and the number of chemicals in the world will increase. The population will increase until the initial energy has been used up. At this

point, the energy flux will control the reactions, bringing the population down to a level that is sustainable by the energy flux.

For these reasons, the initial energy is not an important parameter in these experiments, so we will not vary it. It needs to be set *high enough* that the initial population of chemicals is not likely to all decay before they have had a chance to reproduce. And it does not want to be set too high, otherwise the initial population spike will take a long time to die away, slowing down the code.

8.2.3.3 Energy flux and decay rate

The energy flux and the decay rate are important parameters affecting these experiments. The energy flux specifies how much energy is added to the world each timestep, and the decay rate specifies the probability that each chemical decays each timestep. These two parameters work together to embody the carrying-capacity of the world, as a function of the replication rate of the chemicals contained within it.

If the copier chemical operated crisply, producing perfect copies at a constant rate, and if it had unlimited energy, then its population in the world would increase hyperbolically (section 7.4.1). The energy flux provides a limited resource that the copier chemicals must compete for. The decay rate provides a reason for the copiers to compete: they must copy or else they will decay. This transforms the population's hyperbolic growth into a steady-state behaviour: the population will increase or decrease, until it reaches a fixed number of chemicals that balances the copier's growth rate against the decay rate and energy flux. Since our copier chemical operates stochastically, the steady-state will not be a fixed number of chemicals: it will contain some noise. Also, we expect that the value of the steady-state might change as the copier chemical evolves.

The relationship between the energy flux and the decay rate affects the carrying capacity of the world (the number of copier chemicals at steady-state).

- If the energy flux is large compared to the decay rate, then the carrying capacity of the world will be large (the world will be able to support many copier chemicals). While this may help the world to maintain diversity in the mutant copiers that can be supported, it will also slow down the code to run many copier chemicals in parallel.

- If the energy flux is small compared to the decay rate, then the carrying capacity of the world will be small. If the energy flux is too small, then the copier chemicals might not be likely to complete their copies before they decay. This would mean that the copier chemicals would all decay and the world would be left empty. This is not what we want.

AAA [	ABA]	ACA a	ADA e
AAB [	ABB]	ACB b	ADB f
AAC [	ABC]	ACC c	ADC g
AAD [	ABD]	ACD d	ADD h
BAA <	BBA >	BCA i	BDA m
BAB <	BBB >	BCB j	BDB n
BAC <	BBC >	BCC k	BDC o
BAD <	BBD >	BCD l	BDD p
CAA !	CBA x	CCA q	CDA u
CAB !	CBB x	CCB r	CDB v
CAC !	CBC x	CCC s	CDC w
CAD !	CBD x	CCD t	CDD x
DAA y	DBA x	DCA 6	DDA 2
DAB z	DBB x	DCB 7	DDB 3
DAC 0	DBC x	DCC 8	DDC 4
DAD 1	DBD x	DCD 9	DDD 5

Figure 8.4: GraphMol's triplet-based translation table, as used in Experiment 1. Compare this with the biological translation table (figure 4.3).

The ratio of energy flux to decay rate trades off the carrying capacity of the world against the time that the experiments take to run. Since it is their ratio that matters, we do not need to vary both of these parameters. We set the decay rate to be *low enough* that the copier chemicals are not likely to all decay before copying. We then vary the energy flux, to investigate how the copier chemical evolves in GraphMol worlds with different carrying capacities.

8.3 Experiment 1: DNA triplets

The *copier chemical* is a GraphMol chemical graph, composed of a sequence of GraphMol atoms. There are 41 different atoms in the GraphMol alphabet. In biology, *proteins* are chemicals composed of a sequence of amino acids, of which there are 20 (as described in section 4.2.2). Biological *genomes* are a sequence of bases (A, C, G, T), where each triplet of bases codes for one amino acid. The first method we use to define a *genome* in GraphMol, is to copy biology. We choose 4 *base* symbols (A, B, C, D) and encode each GraphMol atom as a sequence of these

bases, as shown in table 8.4.

Unfortunately, this experiment was not successful at generating novelty. The experiment failed to refute the null hypothesis that the copier chemical will not evolve. Using this triplet-based encoding method, the copier chemical is able to copy the DNA chemicals, and make mutations, but these mutations produce copier chemicals that are not functional. The mutants can bind to the start of the DNA, but they are not able to walk along the DNA, so they cannot make copies. This means that the copier chemical produces some perfect copies, and some mutants, but none of the mutants are able to copy. Because the mutants can start a copy but not finish it, they act as parasites. When a mutant reacts with a functional copier, it effectively traps the functional copier until it decays. This causes the functional copiers to decay out of the system, and the mutants soon follow with nothing left to copy them.

The reason that all the mutants are non-functional is the triplet-based encoding method. Each GraphMol atom is specified by three bases, and the copier chemical makes insertion mutations in the genome. This means that this triplet-based encoding causes a more drastic type of mutation to happen to the encoded copier chemical: a frame-shift. If an insertion of, say, 2 bases happens in the middle of the copier chemical's genome, this throws the triplet-based encoding out of line. This changes the decoding of every triplet downstream of this insertion, as shown in figure 8.5. This is extremely likely to destroy the functionality of the decoded chemical. Any insertion that is not a multiple of three bases will cause a frame shift, effectively destroying the functionality of the mutant chemical. There are some functional mutants produced, by an insertion mutation of a multiple of 3 bases, within a junk region. But these are rare compared to the number of parasite mutants, so they are not enough to save the world from death.

Ideally we would like to use a triplet-based encoding, since this provides a clear separation between functional chemicals (the copier) and information-carrying chemicals (the DNA). Biological systems have made good use of this distinction [67, chapters 4 and 5]. But in order to test the evolutionary capabilities of the GraphMol copier, we do not need this triplet-based encoding. So for the next experiment, we move to a direct encoding that is not vulnerable to frame-shift mutations. This allows us to investigate the copier chemical without introducing extra complications.

8.4 Experiment 2: Direct encoding

In the triplet-based encoding above, the genome consists of 4 bases, which are read in triplets and translated into the 41-character alphabet of GraphMol atoms. In contrast, the direct encoding has 41 different genome bases, which are read

```
Original copier: [t1t1t][yyyyy]jjjjjjjjjj>[u4u4u]jjjjjjjjjj<
Frame-shift:     [t1t1t][yyyyy]jjjjjjjjjj>[u4u4u]jjjjjjjjjj<

[u2u2u]![t2t2t][magic]jjjjjjjjjj>[u5u5u]jjjjjjjjjj<[u3u3u]!
[u2u2u]![t2t2t][magic]jjjjjjjjjj>[u5u5u]jjjjjjjjjj<[u3u3u]!

[t3t3t][eeeee]jjjjjjjjjj>[u6u6u]jjjjjjjjjj<[u4u4u]![t4t4t]
[t3t3t][eeeee]jjjjjjjjjj>[u6u6u]jjjjjjjjjj<[u4u4u]![t4t4t]

[yyyyy]jjjjjjjjjj>[u1u1u]jjjjjjjjjj<[u5u5u]![t5t5t][magic]
[yyyyy]jjjjjjjjjj>[u1u1u]jjjjjjjjjj<[u5u5u]![t5t5t][magic]

jjjjjjjjjj>[u2u2u]jjjjjjjjjj<[u6u6u]![t6t6t][eeeee]jjjjjjjjjj>
jjjjjjjjjj>[u2u2u]jjjjjjjjjj<[u6u6u]![t6t6t][eeeee]jjjjjjjjjj>

[u3u3u]jjjjjjjjjj<[u1u1u]![tstst][fgneg]jjjjjjjjjj<[u1u1u]jjjj
[u3u3u]jjjjjjjjjj<[u1u1u]![tstst][fgneg]jjjjjjjjjj<[u1u1u]jjjj

jj--jjjjjjjjjjjjjjjjjjjjjjjjjjjjjj<[uxuxu]jjjjjjjjjj>[usus
jj>jxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx]]0<1>yjxxxxxxxxxx>[0sst

u-]![txtxt][fgbc]jjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjj
ct[ayk3s78z[!<!<!>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

jjjjj>[u1u1u]jjjjjjjjjj>[u2u2u]jjjjjjjjjj>[u3u3u]jjjjjjjjjj>[u
>>>>>><le9e9[>>>>>>>>>>>!thdhd[>>>>>>>>>>>!lhlhl[x>>>>>>>>>><t

4u4u]jjjjjjjjjj>[u5u5u]jjjjjjjjjj>[u6u6u]jjjjjjjjjj<[ususu]jjj
htht[x>>>>>>>>>><th9h9[>>>>>>>>>>>[9gdgd[x>>>>>>>>>>[tctct[]>>

jjjjjjj>[uxuxu]jjjjjjjjjj<[uxuxu]!
>>>>>>>>ytd9f9[x>>>>>>>>>><9bdbl[6
```

Figure 8.5: A frame-shift mutation destroys the functionality of the chemical downstream of the mutation. Dashes have been added into the sequence of the original copier, to show the location of the frame-shift.

individually and 'translated' directly into their counterparts of GraphMol atoms. In the triplet-based encoding, the 4 different genome bases are embodied within the GraphMol DNA chemical as binding sites with patterns of `aaaaa`, `bbbbb`, `ccccc` and `ddddd`. In the direct encoding, the representation is very similar, with each of the 41 genome bases embodied as a binding site with a pattern such as `a0000`, `j0000`, `>0000`, etc. (The atom being represented, followed by four `0` characters.) Since we are not investigating binding in this thesis, the precise details of these patterns do not affect the running system. But with a rich binding function, these patterns would matter. The two binding site atoms `[` and `]` need special treatment, as they cannot be part of GraphMol binding patterns, so their binding site patterns are `bleft` and `bright` respectively. This direct encoding method has no biological basis, and we do not expect it to be as evolvable as a triplet-based encoding in the long run, but for these experiments it allows us to avoid frame-shift mutations and investigate the evolutionary capabilities of the GraphMol copying mechanism.

As explained above, the parameters we vary here are the *energy flux* of the world (section 8.2.3.3), and the *junk length* of the copier chemical (section 8.1).

8.4.1 Technical problems

One huge problem with these experiments is that they take a very long time to run. A single experiment initialising a GraphMol world with 20 copier chemicals and running it until extinction, takes around two months of wall clock time on the fastest computer available to me (a server equipped with four 8-Core AMD Opteron processors running at 2.3 GHz, of which I was able to use 50%-100% of the cores, depending on how many other people were running jobs on the server). Also, the server is not equipped to run jobs for months at a time, and so occasionally it would crash in the middle of a run, or need to be restarted to install an upgrade. When implementing GraphMol, I did not implement the feature of being able to re-start a GraphMol world from a given point. GraphMol has a lot of *state*: the sequences and folding of all the chemicals in the world, which of the binding sites are `shown` and `hidden`, which binding sites are bound to which others, the positions of the instruction pointers, the current state of the distance tables of each node. I did not save all this state regularly, so I could not restore the world after a server crash. But I highly recommend that other people making such systems save all their state and allow their worlds to be restored, particularly if they are to be run for a very long time.

Another problem with these experiments is that, late in the project, I discovered a bug in my code. When a binding site unbound, it did not correctly update its distance table. This biased the probabilities of future binds, increasing the probability of the copier chemical stepping backwards (and hence, increasing the probability of tangles and the lengths of tangles). Thankfully, this did not alter the

fundamental behaviour of the copier chemical. It effectively increased its mutation rate. After fixing the bug, I found the mutation rate of the copier chemical to be very low. I used equations 6.8 and 6.9 to change two parameters of the GraphMol world (α and k, section 6.4.2.3) to compensate for this. This increased the mutation rate of the copier chemical to the rate I had been observing with the bug. I re-ran the experiment in chapter 7 and obtained the same results. All of the experiments and numbers in this thesis are from GraphMol with this bug fixed, except for experiments 2A, 2B and 2C below, which effectively have a higher mutation rate.

For these reasons, it was not possible for me to perform multiple runs of each experiment and calculate statistics. I present a single run for each of these experiments, showing the only complete run that I was able to perform with the time and equipment available. This is not the ideal situation, but even with one run we can still see interesting behaviour and begin to understand how GraphMol's copying mechanism might evolve.

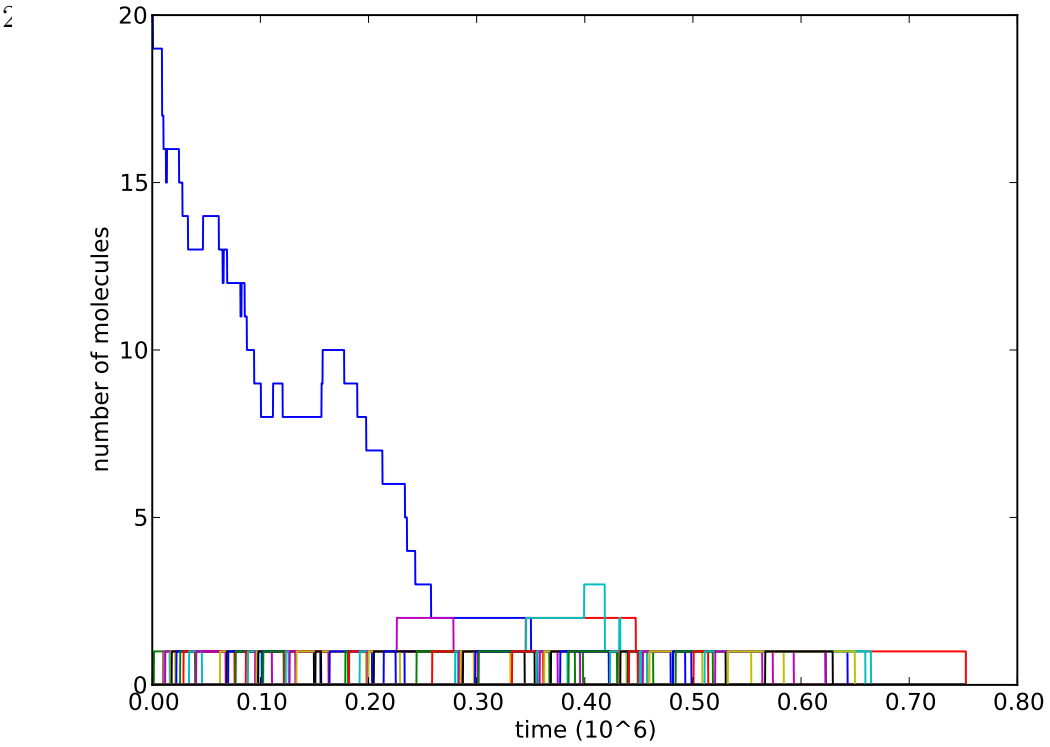
8.4.2 Experiment 2A: Low energy flux

Using a low energy flux (of 10 energy units per iteration), the GraphMol world does not have enough energy to sustain the copying mechanism. Some copies are made, and some mutants are made, but the rate of production of viable copies is too low compared to the decay rate.

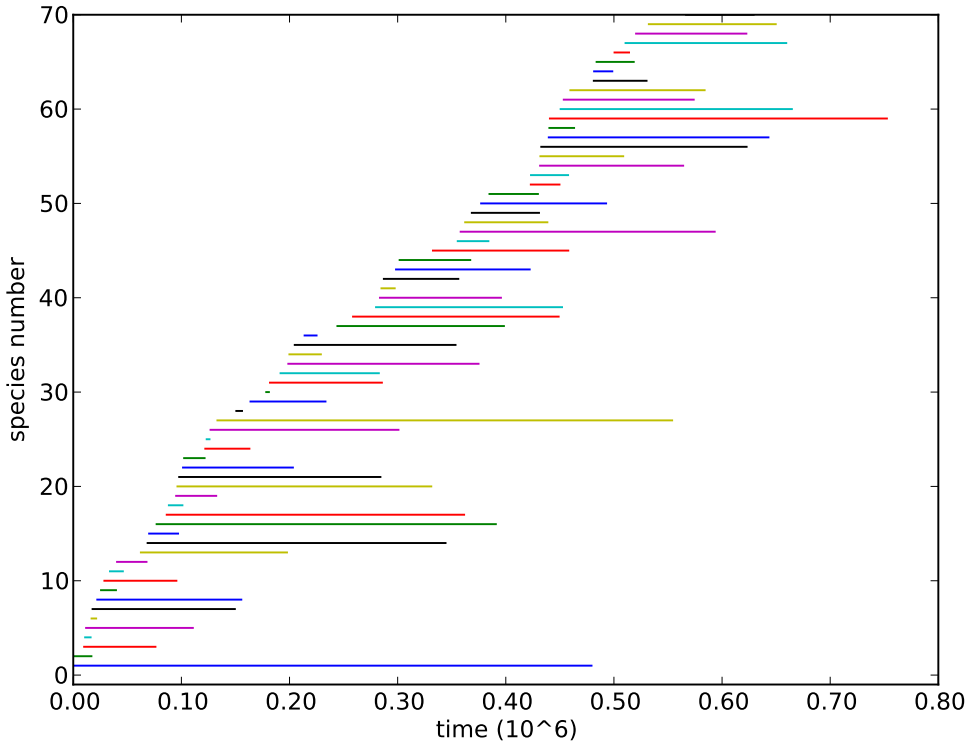
8.4.2.1 Results

Figure 8.6 shows the dynamics of the GraphMol world, initialised with 20 copier chemicals of junk length 16. Figure 8.6(b) shows that many mutants are made, that each have varying lifetimes in the world. But figure 8.6(a) shows that none of these mutants manages to take over the world. Each mutant exists only in very small numbers (between one and three chemicals). The initial copier chemical does not replicate itself quickly enough, and its numbers decrease over time until it eventually dies out. After the death of the initial copier (just before timestep 0.5×10^6), some of the mutants are able to carry on the copying process, but their error rate is even higher than that of the initial copier. This causes every chemical species in the bucket to gradually die out, until they have all gone and the world is dead.

The death of the world can be seen clearly in figure 8.7. Figure 8.7(a) shows the number of individual chemicals in the world over time. This decreases sharply after the death of the initial copier, coinciding with an increase in the energy in the world. This shows that there are not enough chemicals at this time, to use all the energy coming into the world. Before this point, the chemicals had been using all the available energy to do copying reactions, so their rate of copying was

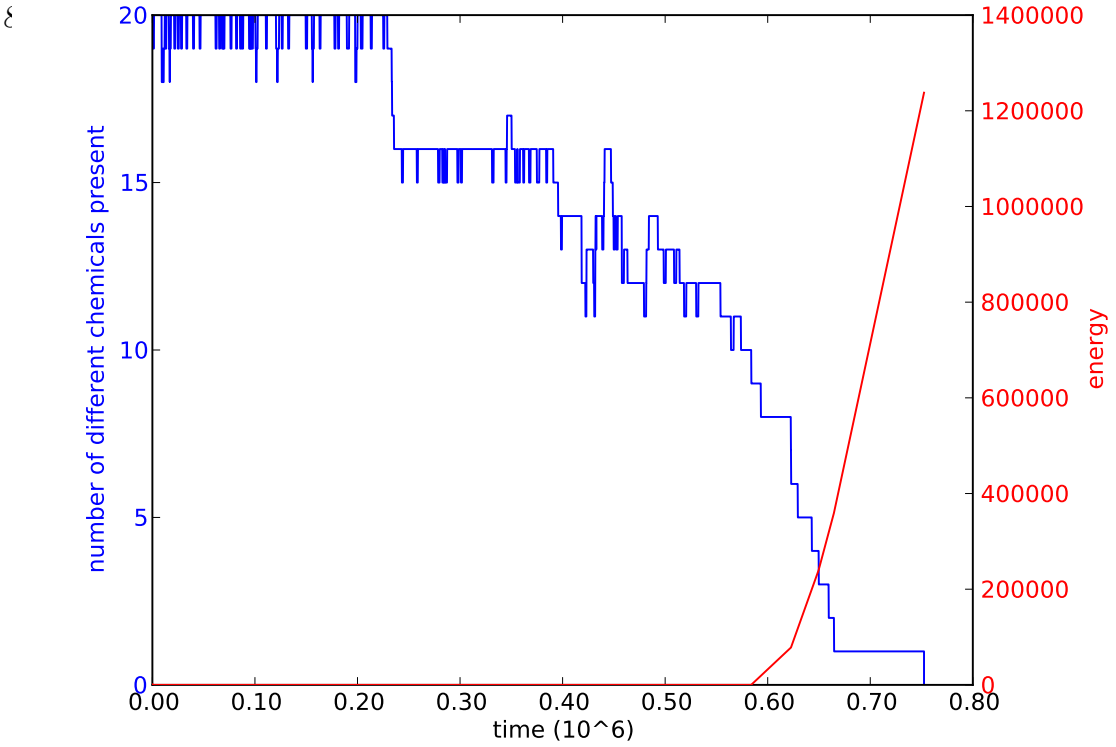


(a) Number of molecules of each species changing over time

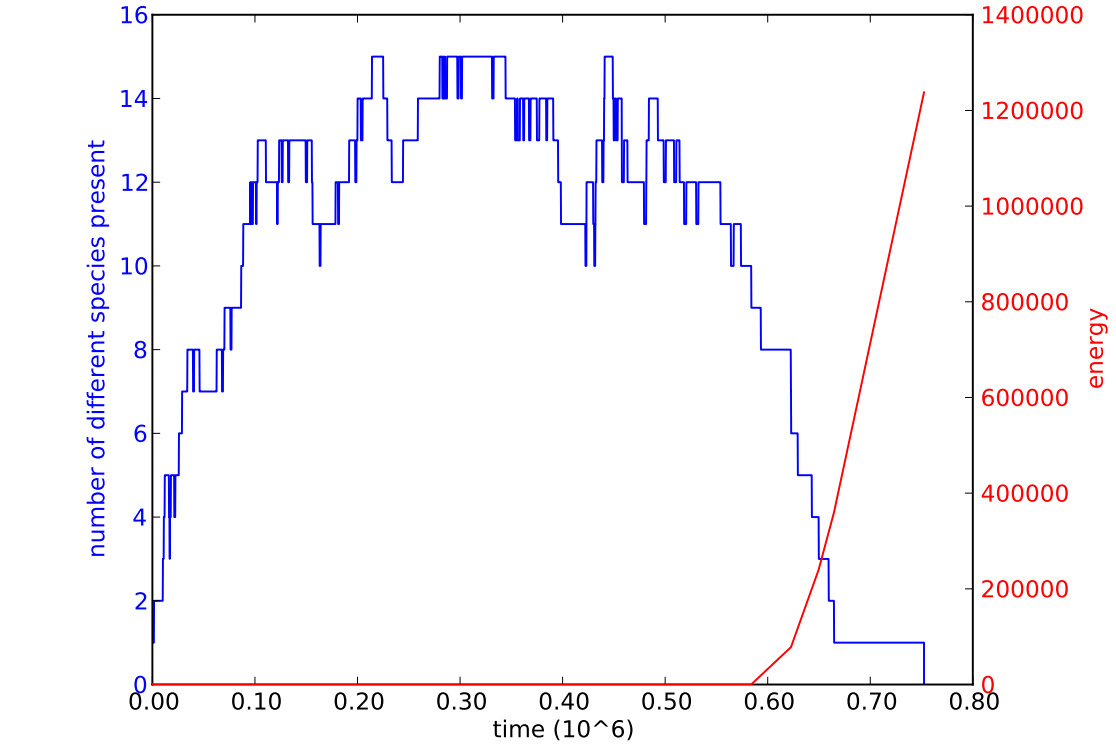


(b) The lifetimes of each different species

Figure 8.6: Dynamics of Experiment 2A (low energy flux). See also figure 8.7



(a) Total number of chemicals in the world



(b) Total number of different species in the world

Figure 8.7: Dynamics of Experiment 2A (low energy flux). See also figure 8.6

limited by the energy flux. When the energy starts to increase, this is because the rate of copying is limited by the number of chemicals present in the world (which is lower than the energy flux). The same story can be seen in figure 8.7(b), which shows the number of different chemical species present in the world (rather than the number of individual chemicals).

8.4.2.2 Discussion

As discussed in chapter 7: in order to be viable, the GraphMol copier chemical needs to have a *high rate of accurate copying*. This can be hampered in two different ways: by the chemical copying *slowly*; or by it copying *inaccurately*. Chapter 7 investigated how the copier's junk level can influence these two factors, but when the copier chemical is reacting in an open world (rather than the constrained test-tube of chapter 7), the world can also influence these two factors. In experiment 2A, the low energy flux reduces the chemical's copying rate, making it copy more slowly because of GraphMol's energy model (section 5.1.1.4).

A low copying rate kills the copier chemical because it cannot produce accurate copies quickly enough to combat the decay rate of the world, as discussed in section 8.2.3.3. An additional factor that makes this worse is the mutants produced by the copier. Some of these mutants are themselves viable copiers, so these will help to keep the world alive. But many of these mutants are not viable copiers. These non-viable copiers act as parasites: chemicals that can be copied, but do not do any copying in return. Parasites drain resources from the viable copiers and speed up the death of the world. A non-viable copier is a copier chemical that has mutations in one (or more) of its functional regions (as opposed to its junk regions). These mutations prevent it walking properly, and so stop it performing a copy when it is chosen as the copier chemical by the mixing process (section 8.2.1). But when it is chosen as the DNA chemical it can still be copied by a viable copier, which prevents that copier reproducing itself for a time and increases the number of parasites in the world.

From the viewpoint of novelty-generation, experiment 2A does not show very much novelty. Figure 8.6(b) shows that many different mutants were created by the copier chemical. This is a type of novelty-generation, but it is not sustainable or useful, as shown by the death of the world. Rather than being an interesting exploration of a search space, these mutations seem to be the death throes of a dying world.

Figure 8.7(b) shows the number of different species in the world over time. This is one measure of novelty-generation. The number of different species represents the diversity of the world, and the number of different copying mechanisms that are being explored in parallel. Changes in this measure correspond to changes in the state of the world, that may help us to understand what is happening within

the world. For example:

- This measure rises at the start of the run. This corresponds to an increase in novelty-generation, because of how the system is initialised (a supply of perfect copiers and energy).
- The decrease in this measure at the end of the run corresponds to a decrease in novelty-generation as the world dies.
- During the run, the measure fluctuates up and down. Some of this is due to the stochasticity of the world, but some of the larger fluctuations could correspond to particular chemical species either being created or going extinct. For example, the trough just before timestep 0.5×10^6 might be caused by the extinction of the original copier.

This measure may be able to tell us interesting things about novelty-generation worlds, or point us to regions of time to investigate in more detail.

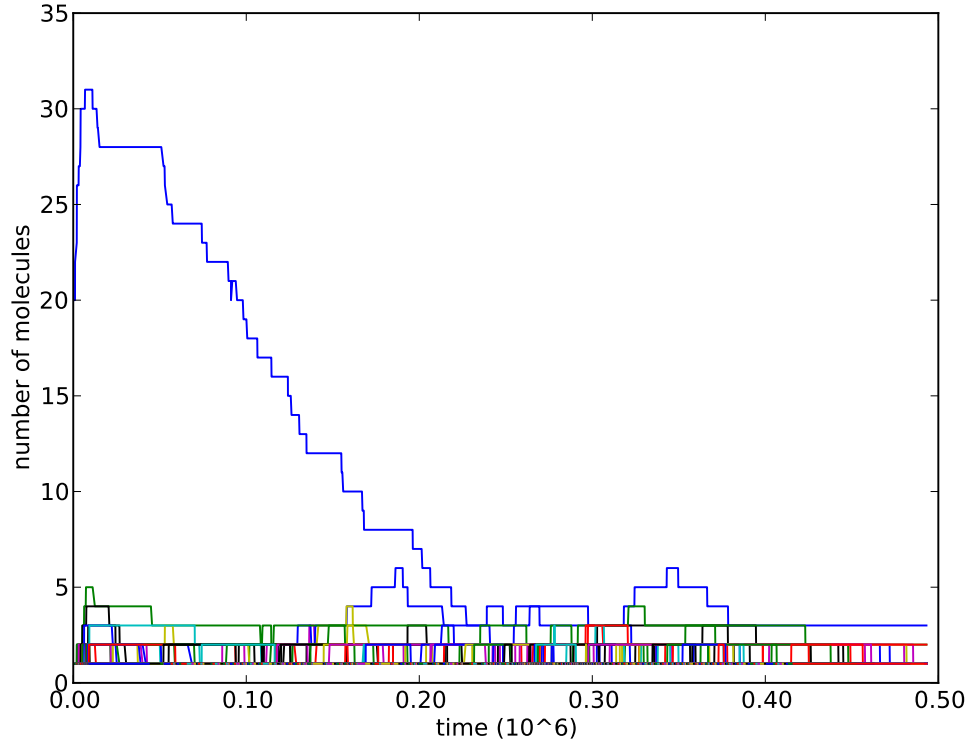
8.4.3 Experiment 2B: High energy flux

Since a low energy flux causes the world to die quickly, with very little novelty-generation, we investigate the opposite extreme: a high energy flux. Using a high energy flux of 100 energy units per timestep, the code runs very slowly: too slowly for us to see much novelty-generation within the time frame of this project (the experiment was terminated after two months).

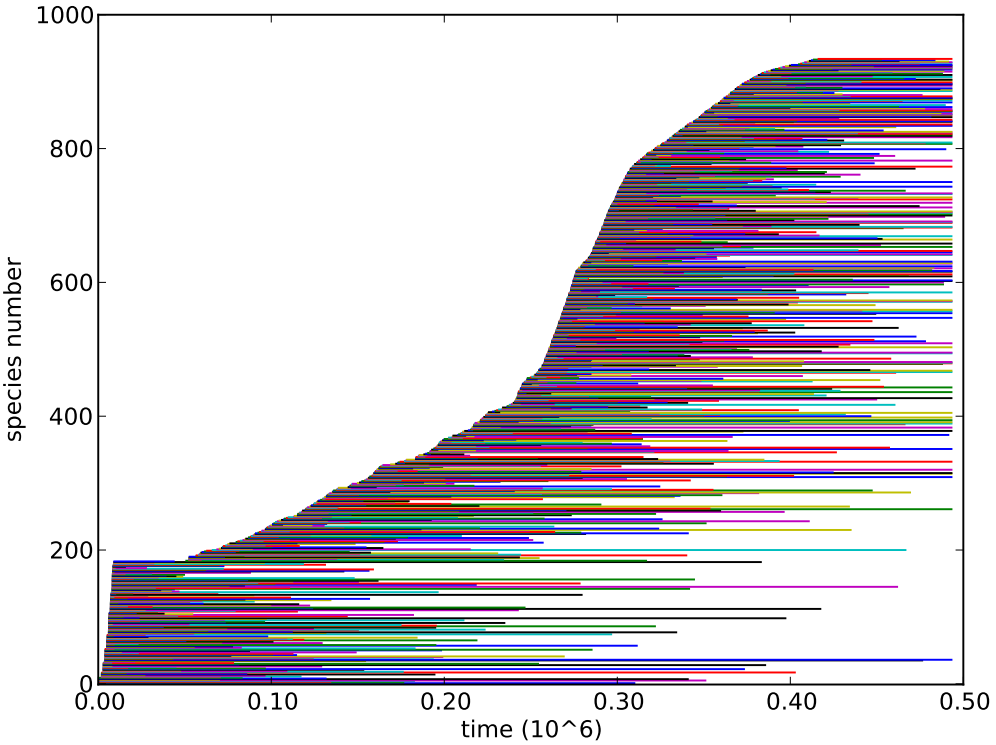
8.4.3.1 Results

Figures 8.8 and 8.9 show the results of a run with high energy flux. Although the number of timesteps in this run is fewer than in the previous experiment, we can see from these figures that many more reactions happen when the world has a high energy flux. This gives us different ways of measuring *time* in GraphMol:

1. From the viewpoint of GraphMol's clock (timesteps), experiment 2B took less time than experiment 2A (0.5×10^6 iterations versus 0.75×10^6 iterations).
2. From the viewpoint of a clock on the wall, experiment 2B took much longer to run than experiment 2A (2 months versus 8 days).
3. From the viewpoint of GraphMol chemicals participating in reactions, experiment 2B performed many more reactions than experiment 2A (non-trivial to quantify because of the changing numbers of chemicals, but the graphs clearly show a large difference).

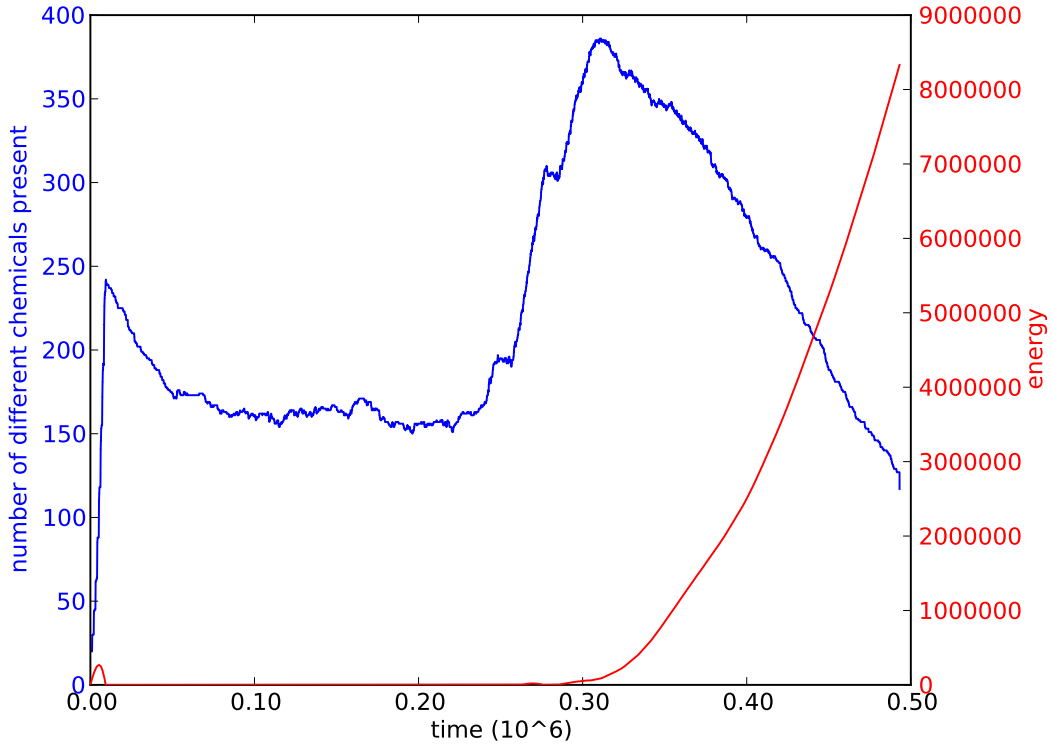


(a) Number of molecules of each species changing over time

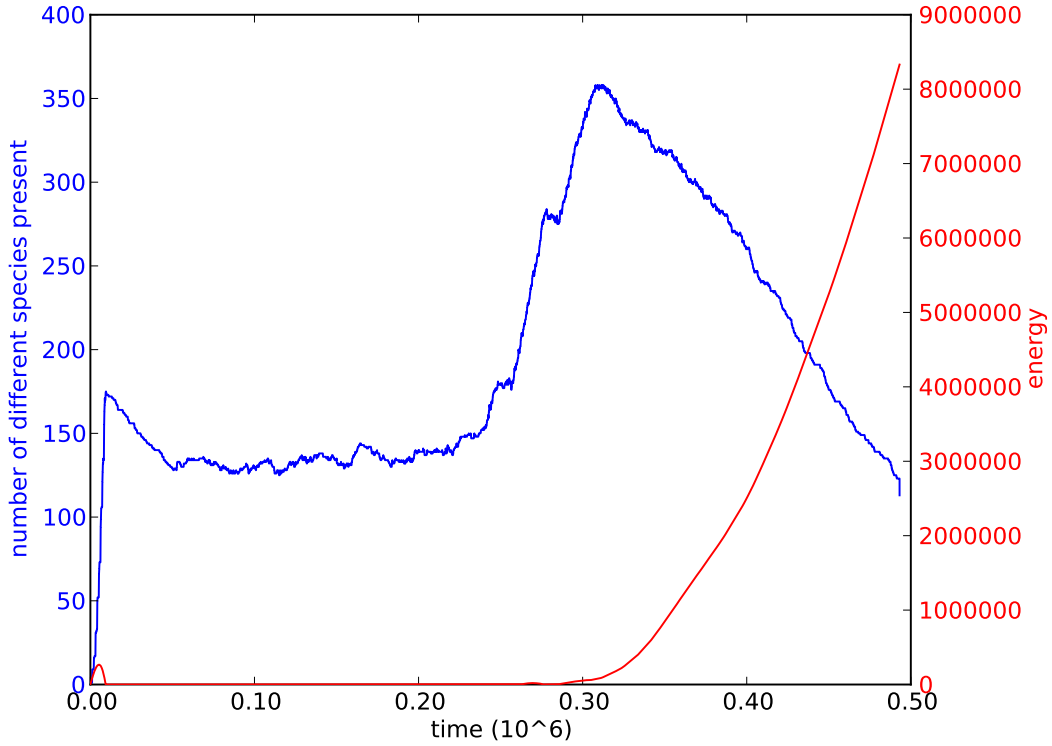


(b) The lifetimes of each different species

Figure 8.8: Dynamics of Experiment 2B (high energy flux). See also figure 8.9



(a) Total number of chemicals in the world



(b) Total number of different species in the world

Figure 8.9: Dynamics of Experiment 2B (high energy flux). See also figure 8.8

Unlike with a low energy flux, a GraphMol world with a high energy flux can sustain a population of copying chemicals. Figure 8.9(a) shows very clearly the number of chemicals increasing, up to the carrying capacity of the world. There is an overshoot, followed by the system settling into a noisy dynamic equilibrium, as predicted in sections 8.2.3.2 and 8.2.3.3. Although the *total* number of chemicals in the world remains constant (with some noise), the makeup of these chemicals is constantly changing. Figure 8.9(b) shows that the number of different chemical *species* is also constant (with noise) during this period, and is similar to the number of chemicals, which means that the number of copies of each chemical species must be low, as can be seen in figure 8.8(a). So during this dynamic equilibrium, the copier chemical is making many copies, each with different errors. Its copying rate is constrained by the energy flux of the world.

Figure 8.8(a) shows that during the dynamic equilibrium phase, the number of original copier chemicals decreases steadily until the original copier chemical dies out just after timestep 0.3×10^6 (as seen in figure 8.8(b)). Around this point there is an explosion of diversity, followed by a decline until the run is terminated. The reason for the run stopping at this point is that the server was restarted by the IT support team. But this does not matter, because we can see from figures 8.9(a) and 8.9(b) that the world is dying. The excess energy in the world increases after timestep 0.3×10^6, which shows that the rate of copying is no longer limited by the world's energy flux, but by the number of viable copiers present in the world. There are some viable copiers at this time (figure 8.8(b) shows new species being produced), but not enough. The energy curve increases in steepness as time progresses, which shows that there are fewer and fewer viable copiers present in the world. And just after timestep 0.4×10^6, figure 8.8(b) shows that new species are no longer being produced, which suggests that all the viable copiers have died out. This does not conclusively prove that the world is dying, but it strongly implies it.

8.4.3.2 Discussion

In the previous experiment (2A), the original copier chemical manages to make some perfect copies of itself as it dies, but in this experiment it does not. Figure 8.8(a) shows that, after a (very quick) initial growth, the number of original copier chemicals declines monotonically (apart from a single increase of one chemical around timestep 0.1×10^6). This may seem counter-intuitive, because there is more energy available in this experiment, so the copier chemical does more copying reactions, so it should produce more perfect copies. This is all true, but the difference is in *what* it is copying perfectly. Because the copier chemical makes many copies, it also makes many mutants. These mutants considerably outnumber the original copier chemical, with a maximum of 31 original copiers out of around 160 chemicals in total. Many of these mutants are still viable copiers, so this zoo of

mutants does not (necessarily) dilute the copying *process*, just the original copier chemical. Although there is more copying happening in this experiment, it is more likely that a mutant will be copied, rather than the original copier chemical. We can see this in figure 8.8(a), by the fact that the number of molecules of some mutants rises above 1. This shows that the mutants are being copied perfectly, as well as being produced by imperfect copying of other chemicals.

The dynamic equilibrium persists even though the original copier chemical is diluted and dies out. This shows an important difference between the *process* of copying and the *particular copying chemical* with which we initialise the world. Many of the mutants must be functional copiers themselves, in order to sustain this dynamic equilibrium for so long. We hypothesised (section 8.1) that we would see one dominant species of copying chemical, that is displaced by a *fitter* species of copying chemical. Instead, we see a collection of closely-related mutants that exist in small numbers and sustain the *process* of copying, rather than sustaining a population of their particular species.

The spike around timestep 0.3×10^6, and subsequent decline, happen at around the same time as the original copier chemical dies out, but I suspect this is a coincidence. As discussed above, the dynamic equilibrium persists while the original copier chemical is being diluted, and it does not seem sensible that less than 20 original copiers out of 160 chemicals would be able to maintain this equilibrium between timesteps 0.1×10^6 and 0.2×10^6. Thus the cause of the spike and decline must be a property of the collection of copying mutants, rather than a consequence of the original copier dying out. In order to test this hypothesis, we would need to repeat this experiment, measuring the time at which the original copier chemical dies out and the time at which the spike and subsequent decline happen (assuming the repeated experiments behave in this way).

From the viewpoint of novelty-generation, experiment 2B is more promising than experiment 2A. We have a system that can sustain itself in a dynamic equilibrium, composed of a collection of ever-changing chemical species. Figure 8.8(b) shows that new species are being continually created during the equilibrium phase, which means that the system is exploring different parts of its search space by generating new mutant copier chemicals. We do not know the exact mechanisms causing the spike and decline, but (as described above) we can infer that it must be a consequence of the collection of copiers deteriorating in copying fidelity. An increased error rate in the collection of copiers causes the production of more mutants with even greater error rates, exacerbating the problem. The world becomes diluted with parasites and then dies.

It seems likely that the spike and decline is an inevitable consequence of the error rate of the copying process growing too high. Other aspatial artificial chemistries die due to parasites (section 2.4.2.1), so this could be the equivalent phenomenon

in GraphMol. Repeated experiments would be able to test this hypothesis.

From a novelty-generation viewpoint, the time required to run this experiment is too long. Of the three different definitions of *time* given above, a large number of reactions is needed to generate novelty (which we have here), but also needed is a short-enough wall-clock time, so that human observers can notice the novelty and use it for something (which we do not have here). The next experiment tries to address this issue.

8.4.4 Experiment 2C: Medium energy flux

With a low energy flux (experiment 2A), we have a short wall-clock time (8 days) but also a low number of reactions. With a high energy flux (experiment 2B), we have a large number of reactions but a very long wall-clock time (2 months). In order to gain the best of both situations, we run an experiment with a medium energy flux (of 20 energy units per timestep). This experiment started to show promising results in a reasonable time (17 days), but the run was terminated due to a server problem. The discovery of a bug in my code meant that it was not worth re-running this experiment.

8.4.4.1 Results and Discussion

This experiment does indeed perform in between experiments 2A and 2B in terms of wall-clock time and number of reactions. Figures 8.10 and 8.11 clearly show a number of reactions in between 2A and 2B. This shows that the energy flux parameter can be used to control the performance of the GraphMol world, as predicted in section 8.2.3.3.

This run starts in a similar way to experiment 2B, with the establishment of a dynamic equilibrium that lasts for around 0.5×10^6 timesteps. Unlike experiment 2B, there is no overshoot this time, because of the lower energy availability. Interestingly, figure 8.11(a) shows that this 'equilibrium' is not a constant number of chemicals (with noise). The number of chemicals decreases very gradually over time. This seems to be a cross between the behaviour seen in experiments 2A and 2B. There is a high enough energy flux to stop the chemicals dying out quickly, as in experiment 2A, but not quite enough energy to maintain the clean equilibrium of experiment 2B. The world is unsustainable, but only just. It is dying, but doing so very slowly. So slowly, in fact, that something interesting happens before it dies completely.

From around timestep 0.55×10^6, there are bursts of activity and the creation of many more chemicals. These bursts are different from the spike and decline in experiment 2B, as can be seen from the energy graphs in figure 8.11. In experiment 2B the viable copiers die out, causing a continual increase in excess energy. But

in this experiment, the excess energy does not increase indefinitely. There are periods of energy buildup, suggesting a decrease in the number of copying reactions happening, but these periods end and the energy is used up. This proves that there is a sizeable population of viable copier chemicals existing at this time. Something happening in the world prevents them from being able to copy (or reduces their numbers so that there are less around to copy), but then this prevention eases (or more viable copies are produced) and they can copy again.

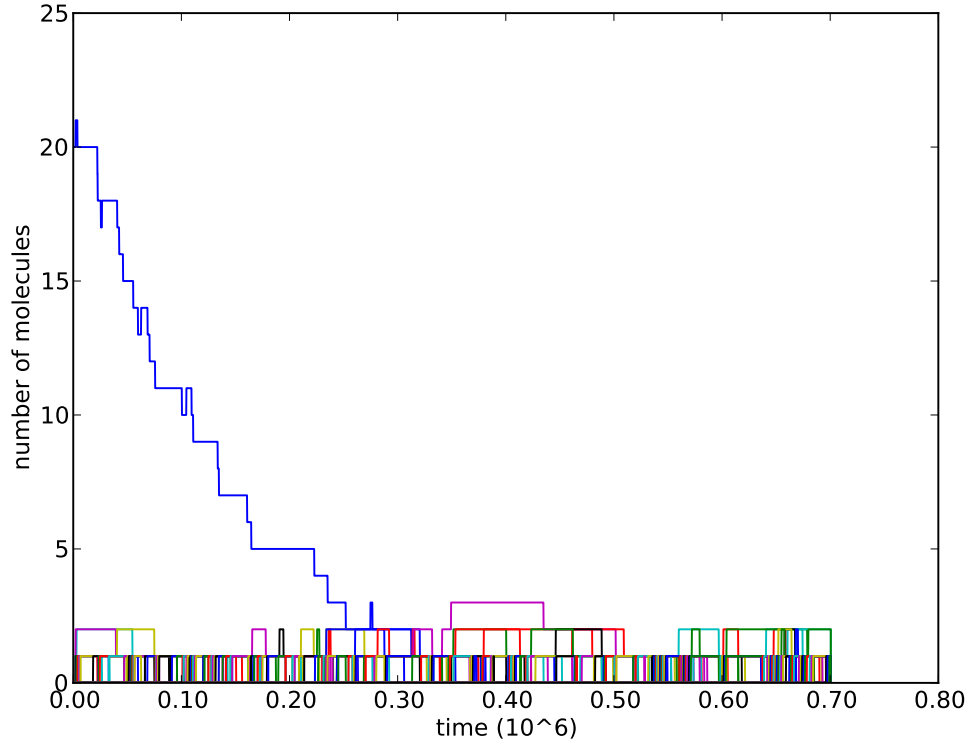
From a novelty-generation viewpoint, the bursts at the end of this run are very interesting. It is not clear whether these bursts are a more drawn-out version of the spike and decline seen in experiment 2B, or whether they are something different. But whichever option is true, both are interesting:

- If this is a drawn-out spike and death, then it shows that the copying process can die in different ways depending on the parameters of the world. Also, the fact that the excess energy keeps being used up in this experiment suggests that there is a force in the world that is pulling the system towards living rather than dying. If this is the case, then understanding this force might allow us to design GraphMol chemicals that could help to control the causes of the world's death, and allow the system to remain alive (even if only for a little longer, before it was killed by a different cause).

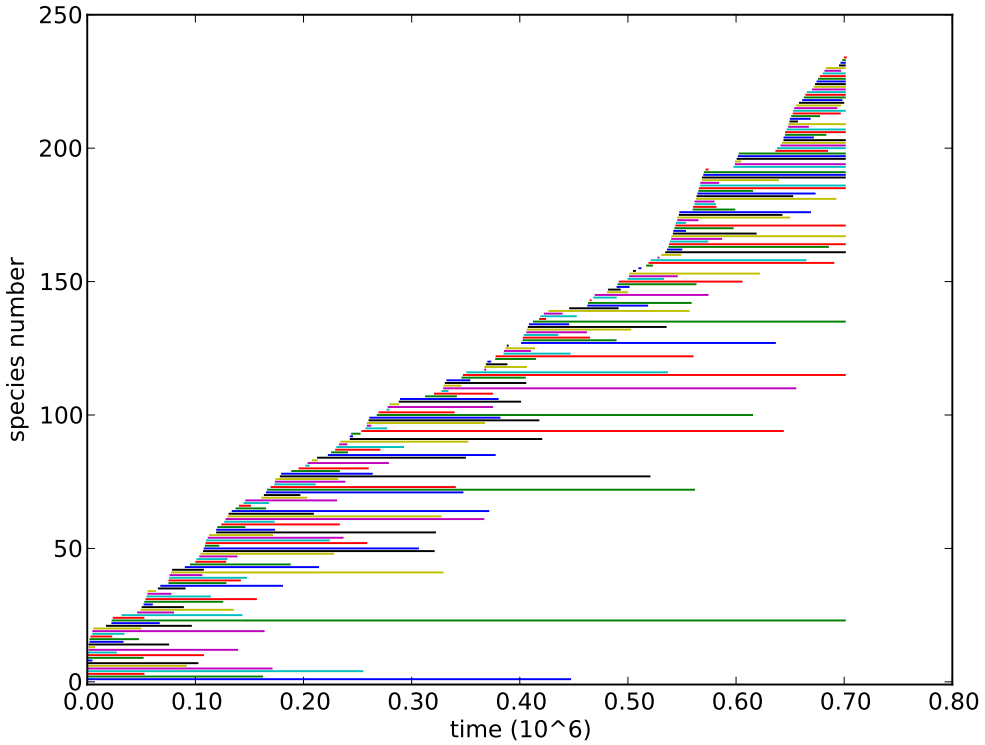
- If this is something different, then we have discovered a new behaviour that the system is capable of displaying. We could analyse the mechanisms that cause these fluctuations in the excess energy, to determine exactly what the chemicals are doing at this time.

This shows a common feature of complex systems: a parameter that can change the behaviour of the system (energy flux), with the most complex behaviour seen at intermediate values of the parameter. With a low energy flux (experiment 2A), the system operates too slowly, so is destroyed by the dissipative forces present in the world (the decay process). With a high energy flux (experiment 2B), the system operates too quickly and destroys itself by working too hard (creating too many mutants). But with an intermediate energy flux (experiment 2C), these two forces oppose each other to produce complex behaviours (spikes of energy and diversity). If this complex behaviour can be harnessed, then we might be able to produce a system that steers itself away from both of these two death scenarios.

It is unfortunate that a server problem caused the run to be terminated. While this experiment was running, I discovered a bug in the GraphMol code, as explained in section 8.4.1. Because of this, I felt it was better to run some experiments with the bug fixed, rather than re-running this experiment or trying to re-start it where it had terminated.

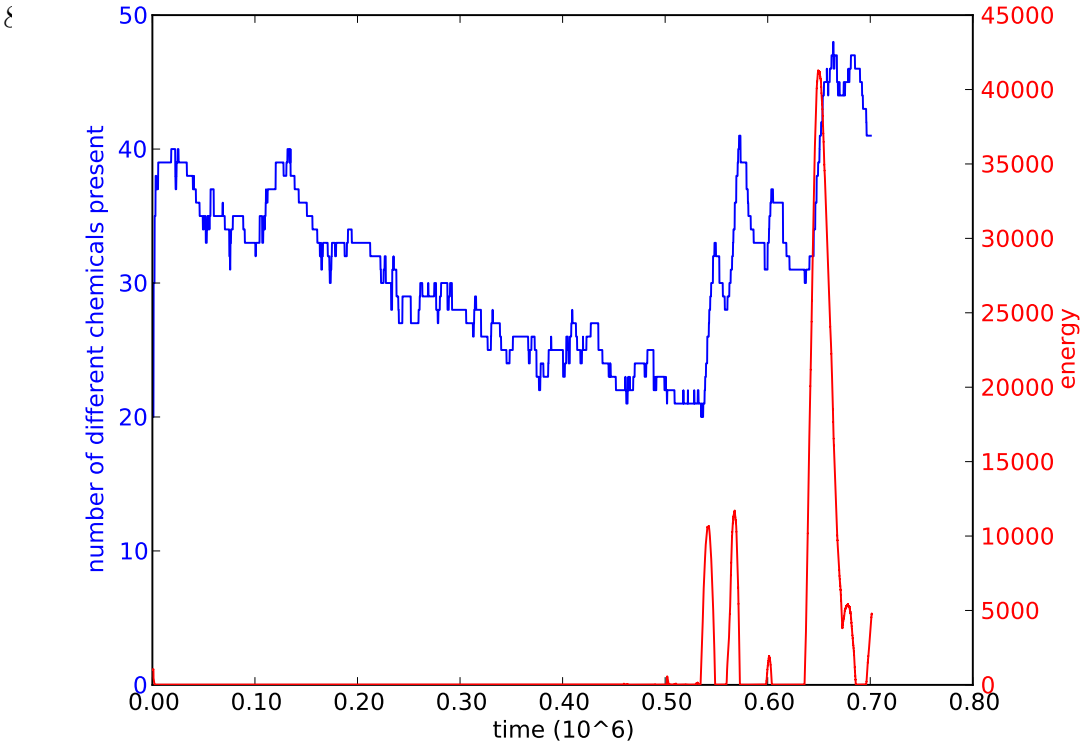


(a) Number of molecules of each species changing over time

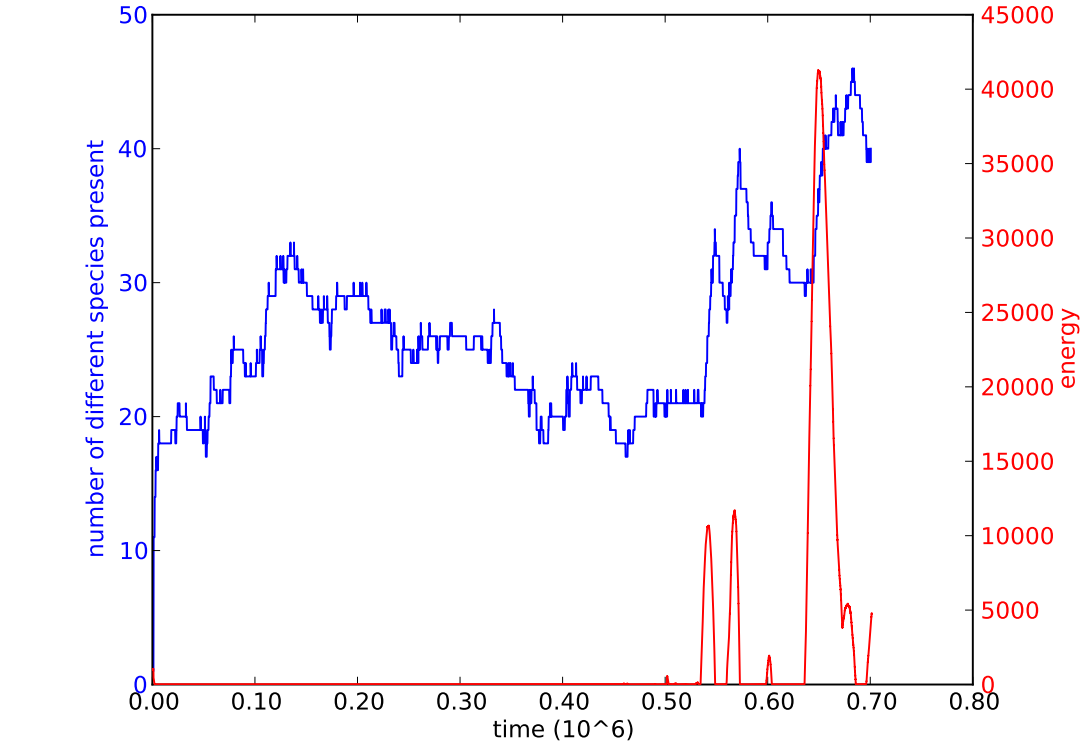


(b) The lifetimes of each different species

Figure 8.10: Dynamics of Experiment 2C (medium energy flux). See also figure 8.11



(a) Total number of chemicals in the world



(b) Total number of different species in the world

Figure 8.11: Dynamics of Experiment 2C (medium energy flux). See also figure 8.10

8.4.5 Experiment 2D: After fixing the bug

Section 8.4.1 describes the bug I found in my code, and how I fixed it. After doing this, I ran an experiment with:

- A medium energy flux, of 20 energy units per timestep, because this showed promising results in experiment 2C.
- A copier chemical with a lower junk length of 4 atoms, compared to the 16-atom copiers of experiments 2A, 2B and 2C. This was to investigate the hypothesis (section 8.1) that a copier chemical with lower junk would increase its junk level.

Again, the server had a problem and crashed the program after 15 days of runtime. But the results from this terminated run, experiment 2D(1), seemed to suggest that one copier chemical might be taking over the population. To test if this was a genuine takeover of a *fitter* copying chemical, I ran another experiment competing this chemical against the original copier, experiment 2D(2). Unfortunately, the chemical is a parasite, and kills off the original copier before destroying the world entirely. But comparing these two experiments is interesting. It shows us in detail how a parasite can destroy a GraphMol world, and illuminates the differences between this and the dynamic equilibrium state where the world is alive and exploring its state space.

8.4.5.1 Experiment 2D(1): Results

Figures 8.12 and 8.12 show that this experiment begins by establishing a stable, dynamic equilibrium that persists for around 0.5×10^6 timesteps. The population size is similar to that of experiment 2C, because the energy flux of these two experiments is the same. In this experiment, the population does not slowly decrease, as in experiment 2C. This is probably because the original copier chemical is shorter in this experiment, so can be copied more quickly. This has the same effect as increasing the energy flux, because it decreases the energy needed for each copy.

Figure 8.13 shows that the dynamic equilibrium is broken by spikes of diversity towards the end of the run, as in experiments 2B and 2C. But the energy signature of this experiment is different from that of both the previous experiments. The energy is not increasing continually, as in experiment 2B, so this does not seem to be the world dying. But there are changes in the numbers of chemicals and species present, without corresponding spikes of energy, as there are in experiment 2C. The largest diversity increase, around timestep 0.65×10^6, corresponds to an energy spike, but there is a smaller diversity spike before this, and a large one afterwards, with no changes to the energy. These diversity spikes could be simply

noise, in which case we would not expect them to come with energy spikes, or they could be a type of behaviour different from that seen in experiment 2C.

Towards the end of the run, we see an increase in the population of one of the mutants (in figure 8.12(a)). It is very unfortunate that the run terminated at this point, because from this data we cannot tell whether this is a *fitter* copier species that is beginning to take over the world, or whether it is just part of the noise, because the overall population size has increased (figure 8.13(a)). To determine which of these is the case, we run an experiment competing 15 copies of this chemical against 15 copies of the original copier chemical. The results of this experiment are described below.

8.4.5.2 Experiment 2D(2): Results

Unfortunately, the *competitor* chemical is a parasite that kills off the original copier chemical, before proceeding to kill off all the mutants and destroy the world.

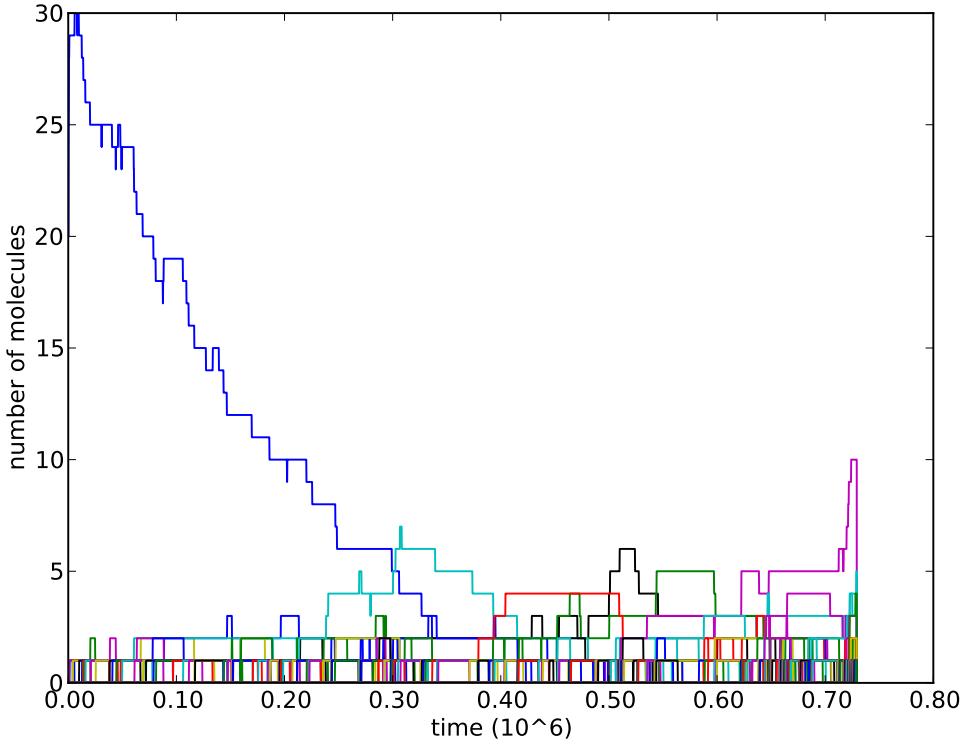
Figure 8.14(a) shows this clearly. The run starts with 15 copies each of the original copier and the competitor. The competitor (highest line) is replicated by the original copier (second highest line at the start). The original copier declines throughout the run, many mutants are made, then the competitor declines, followed by the few mutants remaining in the world. The world's death is also clear from figure 8.15, with the numbers of chemicals and species rising, then falling dramatically until their die out completely, with a corresponding increase in the excess energy. The death of this world shows the same energy signature as experiment 2B (figure 8.9), which lends support to the idea that experiment 2B's world is dying towards the end of its experiment.

Figure 8.16 shows the differences between the sequences of the competitor and the original copier chemicals. We can see that the competitor chemical has been produced by a sequence of insertion mutations. Some of these insertions have increased the lengths of some of its junk regions, but others have increased the lengths of some of its binding sites. GraphMol's binding function is crisp (section 6.3.1.2), so increasing the length of a binding site is the same as destroying it. Thus these mutations render the competitor chemical non-functional as a copier.

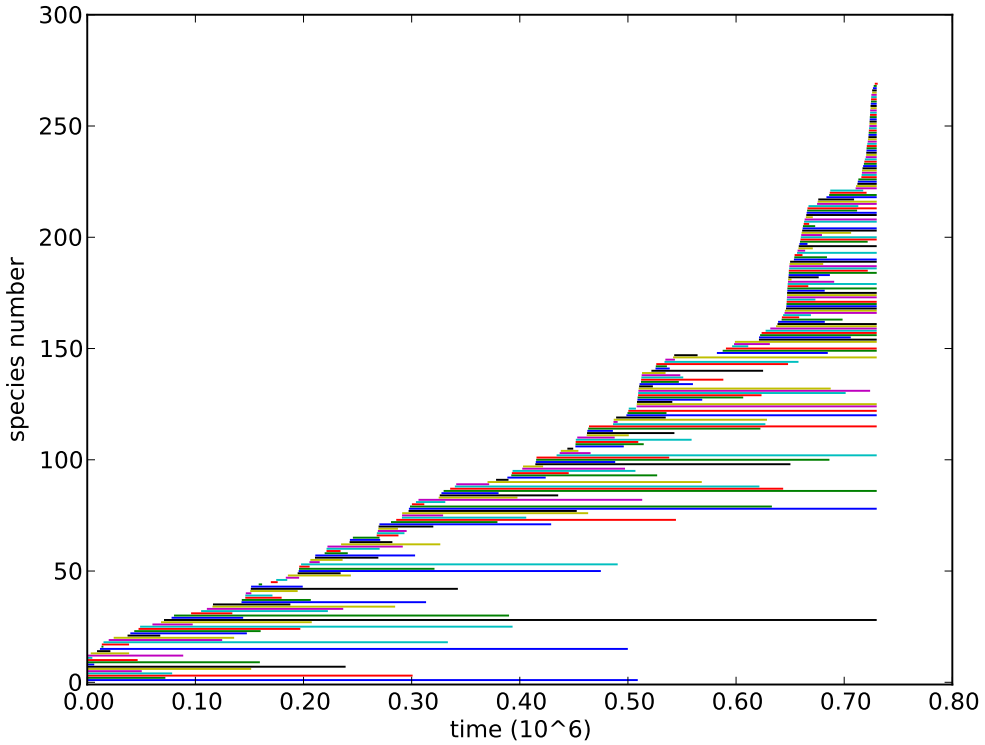
8.4.5.3 Discussion

From a novelty-generation viewpoint, the fact that this experiment establishes a dynamic equilibrium (figure 8.13(a)) is very promising. This was that main (understandable) behaviour identified in the previous experiments, and it still persists after fixing the bug. This strongly suggests that the conclusions drawn from the previous experiments are valid despite the bug.

We can use to our advantage the fact that experiment 2D was terminated

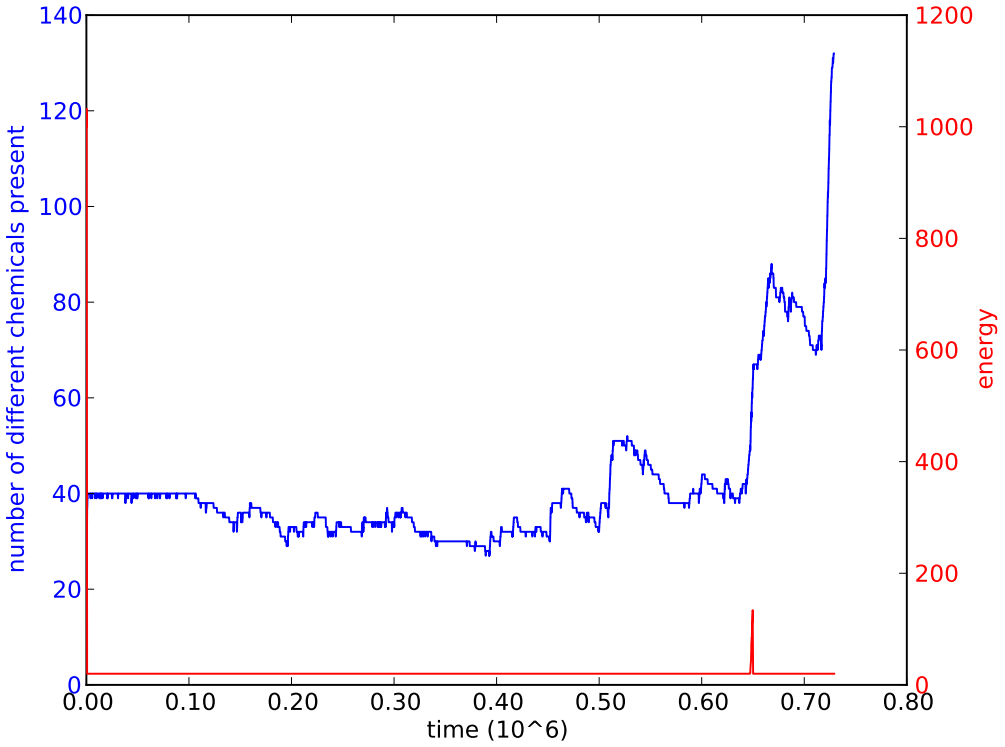


(a) Number of molecules of each species changing over time

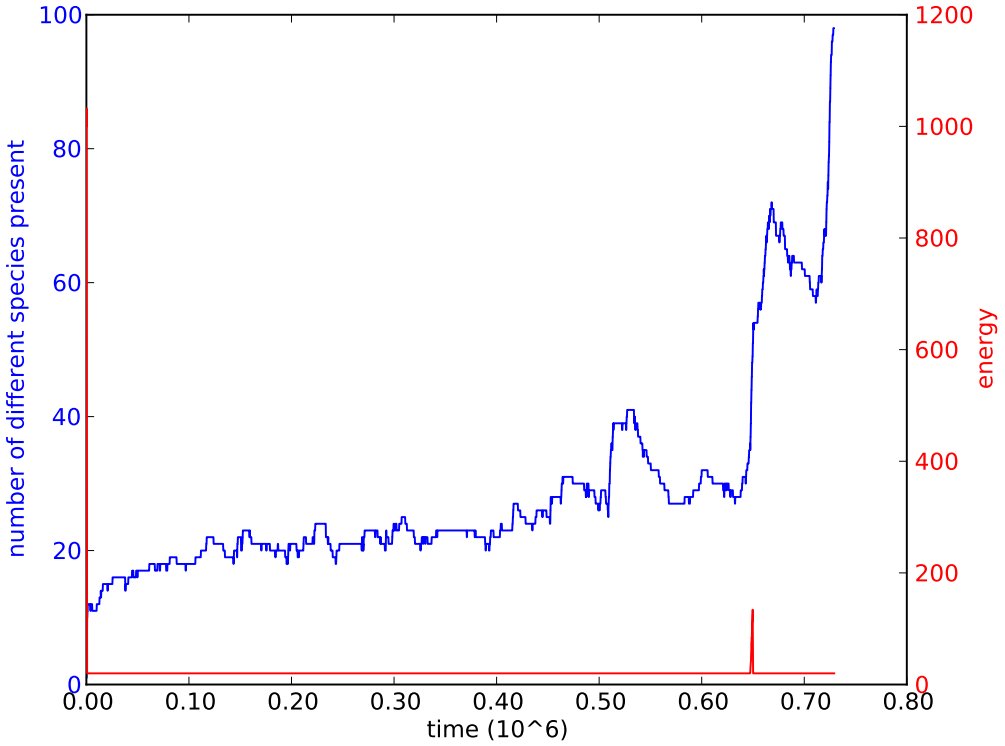


(b) The lifetimes of each different species

Figure 8.12: Dynamics of Experiment 2D(1). See also figure 8.13

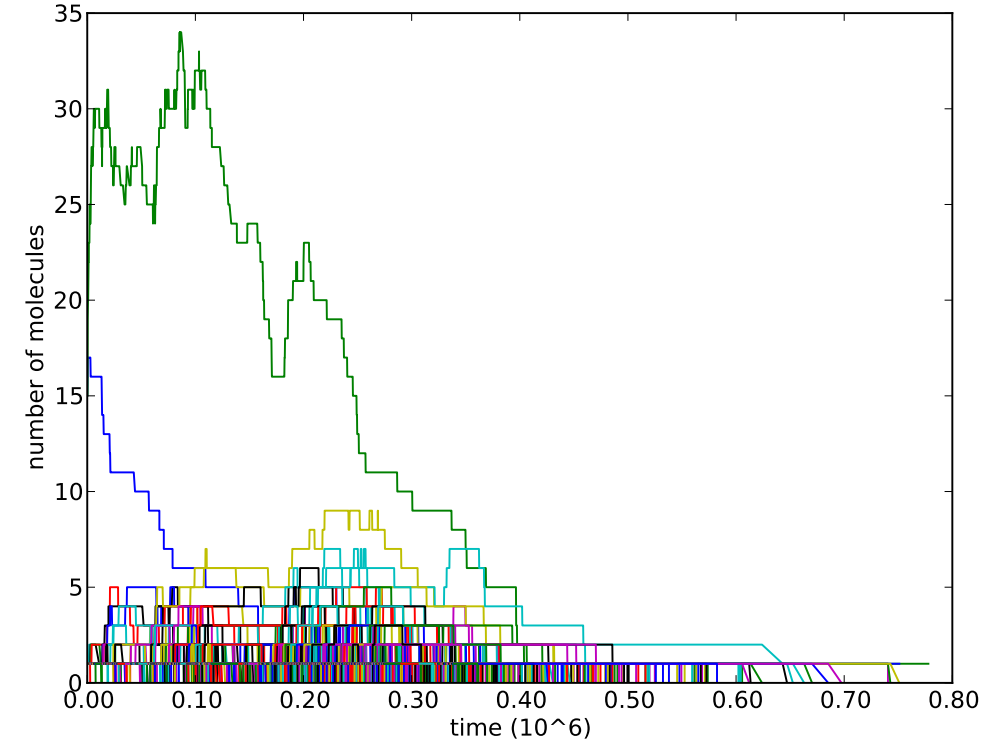


(a) Total number of chemicals in the world

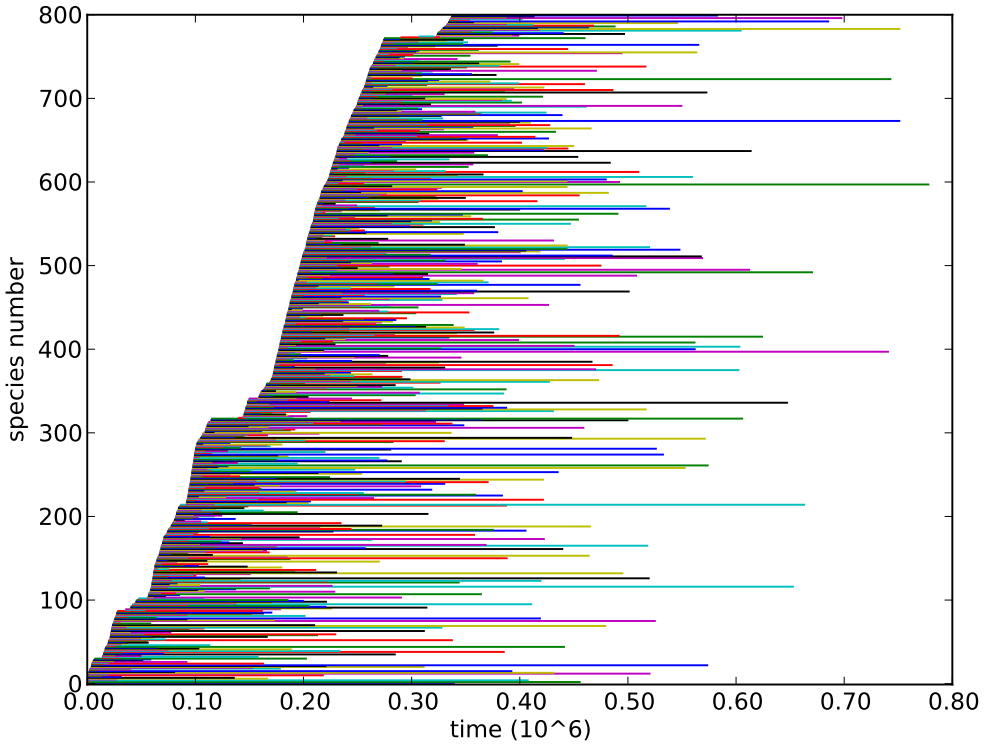


(b) Total number of different species in the world

Figure 8.13: Dynamics of Experiment 2D(1). See also figure 8.12

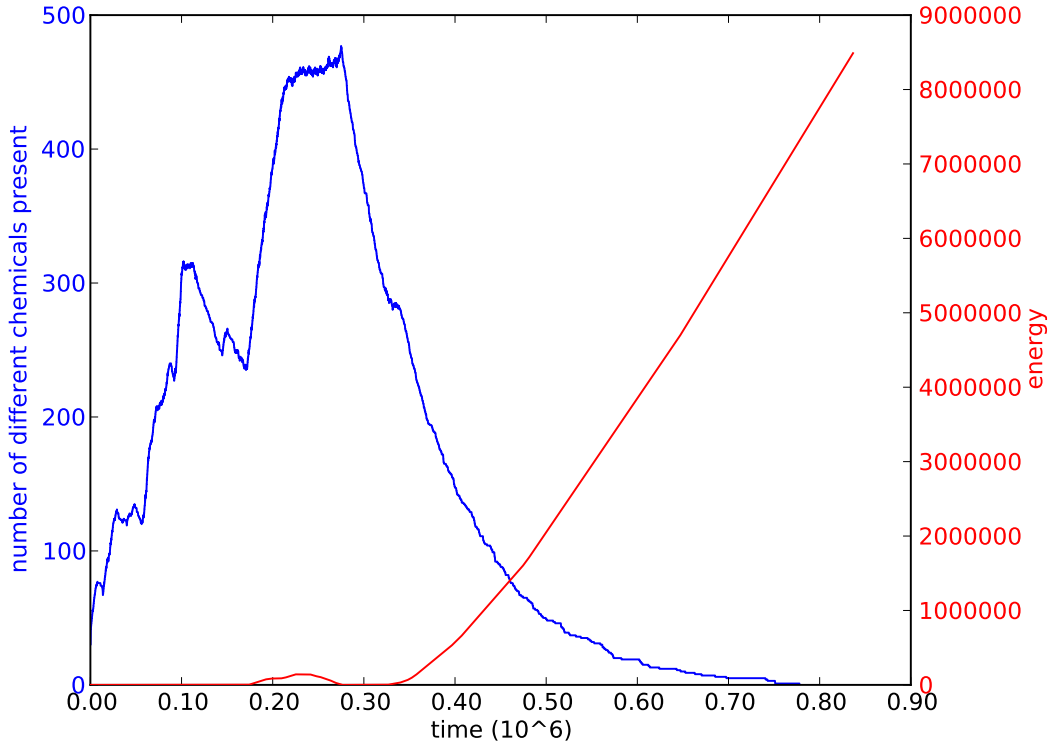


(a) Number of molecules of each species changing over time

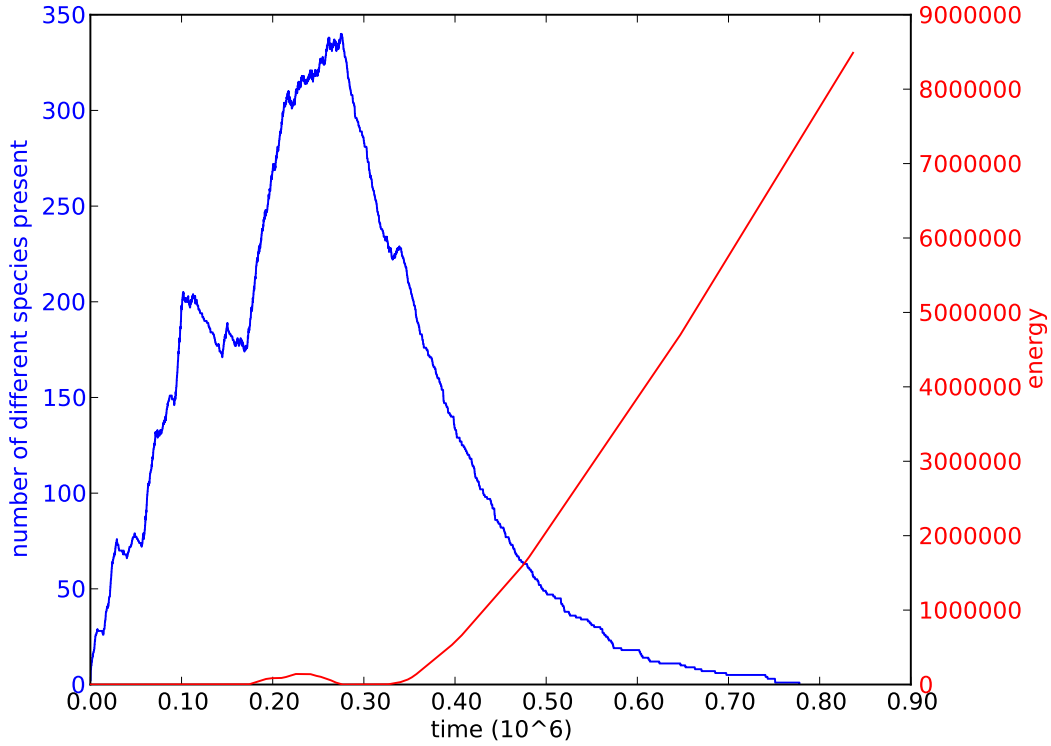


(b) The lifetimes of each different species

Figure 8.14: Dynamics of Experiment 2D(2). See also figure 8.15



(a) Total number of chemicals in the world



(b) Total number of different species in the world

Figure 8.15: Dynamics of Experiment 2D(2). See also figure 8.14

```
Original copier: [t1t1t][yyyyy]jjjj>[u4u4u--]jjjj<[u2u2u]!
Competitor:      [t1t1t][yyyyy]jjjj>[u4u4u4u]jjjj<[u2u2u]!

[t2t2t][magic]jjjj>[u5u5u]jjjj<[u3u3u]![t3t3t][eeeee]jjjj>
[t2t2t][magic]jjjj>[u5u5u]jjjj<[u3u3u]![t3t3t][eeeee]jjjj>

[u6u6u]jjjj<[u4u4u]![t4t4t][yyyyy]j----jjj>[u1u1u--]jjjj<
[u6u6u]jjjj<[u4u4u]![t4t4t][yyyyy]j]y]jjjj>[u1u1u1u]jjjj<

[u5u5u]![t5t5t][magic]jjjj>[u2u2u]jjjj<[u6u6u--]![t6t6t]
[u5u5u]![t5t5t][magic]jjjj>[u2u2u]jjjj<[u6u6u6u]![t6t6t]

[eeeee]jjjj>[u3u3u]jjjj<[u1u1u]![tstst][fg--neg]jjjj<
[eeeee]jjjj>[u3u3u]jjjj<[u1u1u]![tstst][fgfgneg]jjjj<

[u1u1u--]jjjjjjjjjjjjjjj<[uxuxu]jjjj>[ususu]![txtxt][fgbc]
[u1u1u1u]jjjjjjjjjjjjjjj<[uxuxu]jjjj>[ususu]![txtxt][fgbc]

jjjjjjjjjjjjjjjjjjj>[u1u1u]jjjj>[u2u2u]jjjj>[u3u3u]jjjj>
jjjjjjjjjjjjjjjjjjj>[u1u1u]jjjj>[u2u2u]jjjj>[u3u3u]jjjj>

[u4u4u]jjjj>[u5u5u]jjjj>[u6u6u]jjjj<[ususu]jjjj>[u--xuxu]
[u4u4u]jjjj>[u5u5u]jjjj>[u6u6u]jjjj<[ususu]jjjj>[u[uxuxu]

jjjj<[uxuxu]!-
jjjj<[uxuxu]!!
```

Figure 8.16: The mutations of the *competitor* chemical from experiment 2D, as compared to the original copier chemical. Dashes have been added into the sequence of the original copier, to make the alignment easier to read and to show insertion mutations in the competitor.

and restarted. Rather than observing a 'natural' death of the world, as seen in experiment 2A and suspected in experiment 2B, we have a different type of death in experiment 2D(2):

- In experiment 2A, the cause of death is clear: the world has too little energy for any copier chemicals to survive.
- In experiment 2B, the run has been going for a long time before the end, and many reactions have happened, with a long period of dynamic equilibrium before the spike. This makes the precise low-level cause of death difficult to determine, because there are over 350 different chemical species present in the world at the height of the spike. All of these could be contributing in subtle ways to the world's death.
- In experiment 2D(2), we have very few complications. Although many different chemical species are created during this run, which may have subtle interactions, we know that the cause of death is the presence of the competitor species. Without the competitor (experiment 2D(1)), the system enters a dynamic equilibrium. But with the competitor (experiment 2D(2)), the system immediately undergoes a spike and death.

The fact that experiment 2D(2) did *not* manage to establish a dynamic equilibrium shows that the presence of the competitor chemical alone is sufficient to destroy the copying process. This opens up many questions for future work, such as:

1. Experiment 2D(2) has an initial ratio of 50% copier chemical to 50% competitor parasite. Is this always sufficient to destroy the world? How do different ratios fare?
2. When the world dies 'naturally', after a period of equilibrium, is it the presence of a single parasite that initiates the destruction of the world? Or is it the combination of many parasites? (I strongly suspect it is a combination of many parasites, just as the copying process is a combination of many mutated copiers.)
3. In experiment 2B (and other, future, spike-and-decline experiments), can we identify the combination of parasites responsible for the decline? Can we identify the combination of copiers that sustain the equilibrium phase, and show that they are displaced by parasites during the spike phase? (I suspect this will be very difficult but ultimately possible.)

From these results, we are able to identify the two different behaviours of *dynamic equilibrium* and *spike-and-death*. GraphMol is a complex system, generating

a lot of data that can be analysed in many different ways. The following sections analyse the experiments using different techniques. They look at the lengths of the chemicals produced, and the family trees implied by the copying process.

8.4.6 Analysis: Chemical lengths

For this chapter's experiment, we hypothesised (section 8.1) that the length of the dominant copier chemical would change, optimising itself to a *sweet spot* at an intermediate junk level. Figures 8.17, 8.18 and 8.19 show how the lengths of each chemical in the world change over time. Because there are many timesteps in each run, these figures are produced by sampling the world every 10,000 timesteps and plotting a circle for every chemical length present in the world.

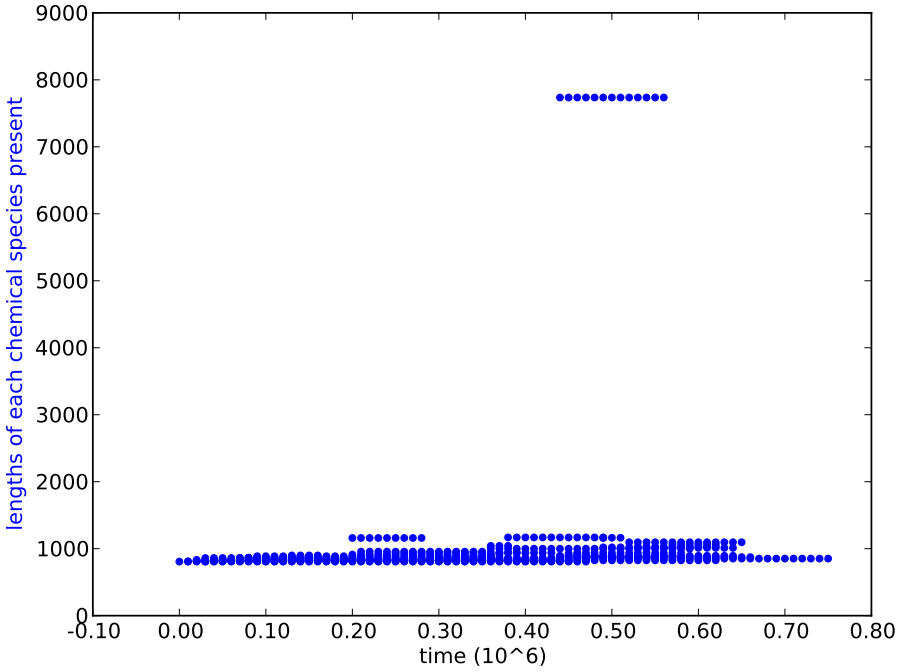
The figures show that the copier chemical does indeed change its length. Because these runs do not have a single dominant species, it is not possible to say how the length of *the copier chemical* is adapting. Instead, it makes more sense to talk about the *collection* of chemical lengths across all the chemicals in the world.

All of these figures have the same y-axis range, allowing them to be compared easily. The only exception is figure 8.17(a), which has an anomalous chemical that is a lot longer than any discovered in the other experiments. I have not looked into this chemical's story in this thesis, because I am focusing on the high-level properties of the system (section 8.4.9.5). But looking into this would be an interesting avenue for future work.

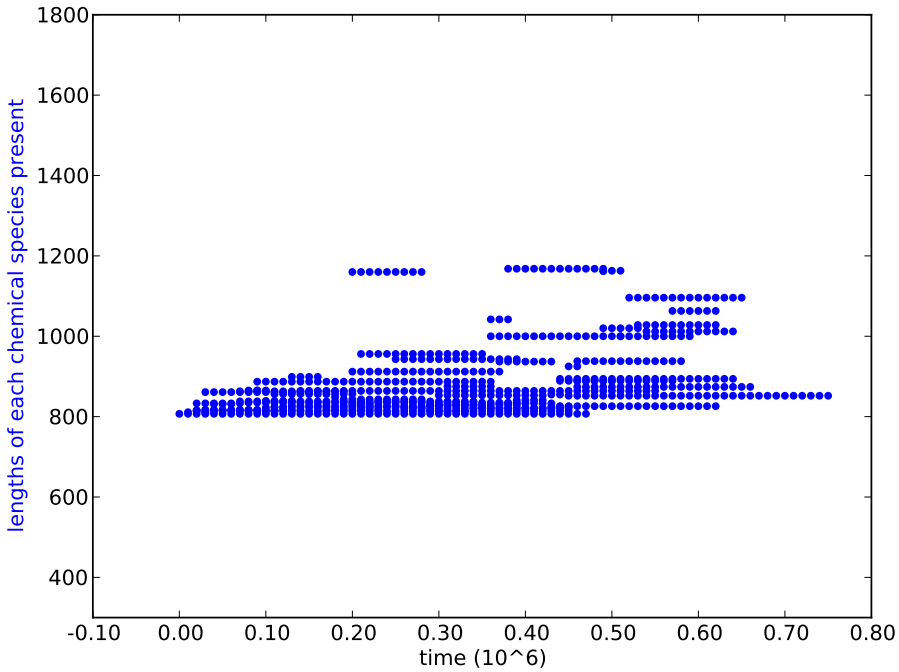
Figures 8.17(b) and 8.18(a) show changes in the chemical lengths consistent with our analysis of these experiments above. With a low energy flux (experiment 2A), the collection of copier chemicals explores a small number of different chemical lengths, seemingly choosing them at random. With a high energy flux (experiment 2B), the system explores a large number of chemical lengths within a certain range. We can see from figure 8.18(a) that when the system is in its equilibrium phase, the range of chemical lengths is roughly constant, with some occasional outliers a small distance away. But when the system enters its spike phase, the range of chemical lengths increases. From this one experiment, we cannot say which way the causality goes. It could be that either:

- The length increase drives the spike. A mutation happens to one of the chemicals that creates some very long mutants. Having a population of long mutants upsets the equilibrium and causes the spike. Or:

- The spike drives the length increase. Something happens in the population of chemicals, to upset the equilibrium (either a single event or a gradual decrease in copying fidelity). This creates a population of copying chemicals with a high error rate, that are more likely to make large insertion mutations and increase the sizes of the chemicals in the world.

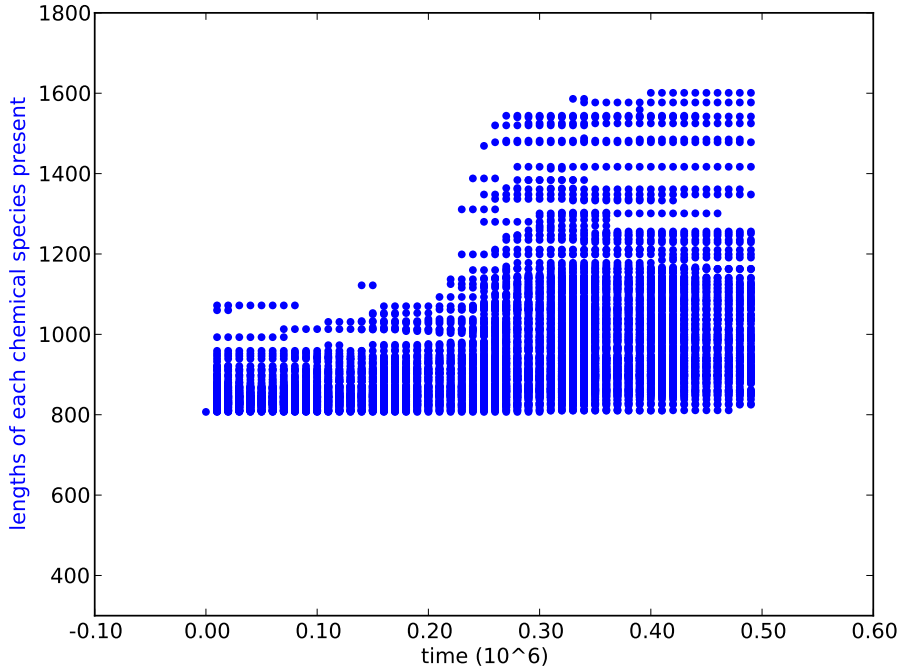


(a) Experiment 2A (low energy flux)

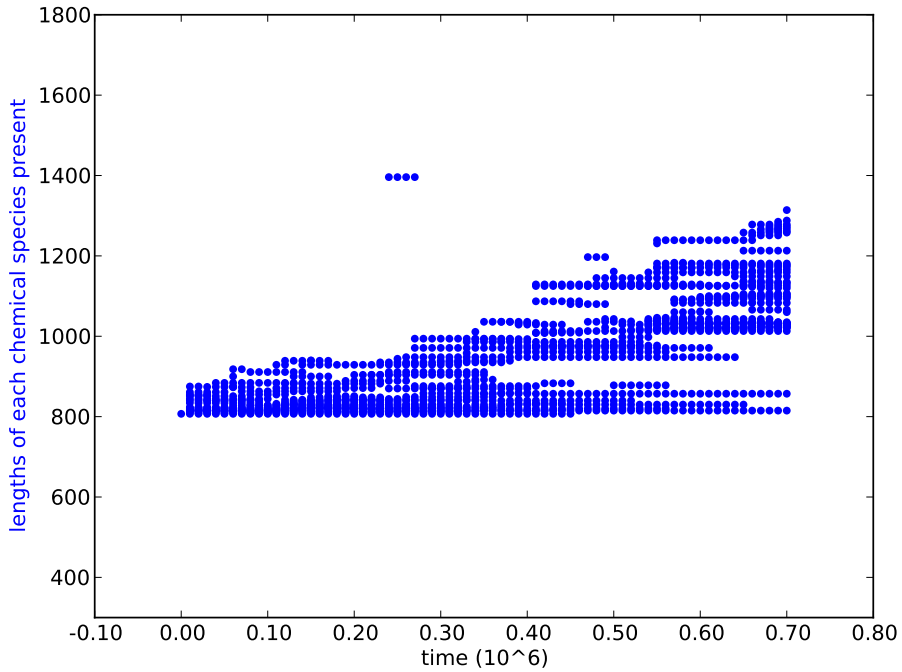


(b) Experiment 2A (low energy flux, without anomaly)

Figure 8.17: Lengths of each chemical in the world. See also figures 8.18 and 8.19.

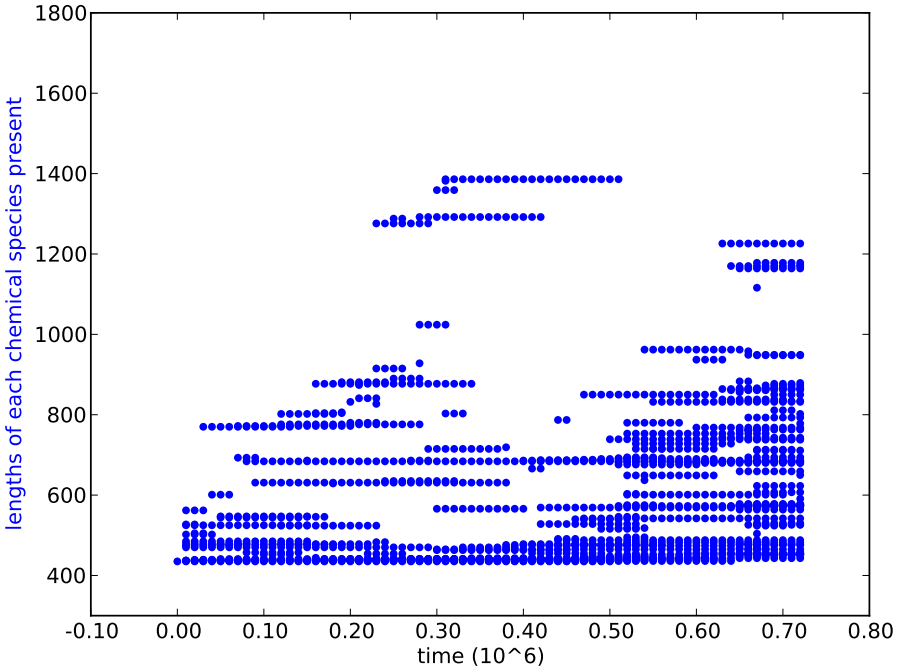


(a) Experiment 2B (high energy flux)

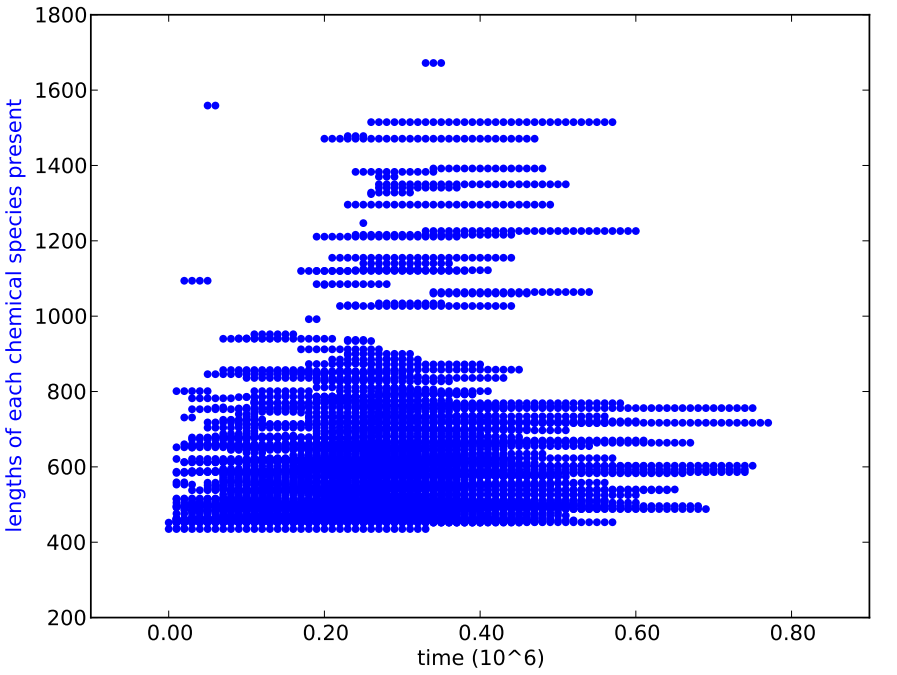


(b) Experiment 2C (medium energy flux)

Figure 8.18: Lengths of each chemical in the world. See also figures 8.17 and 8.19.



(a) Experiment 2D(1) (medium energy flux, low junk copier)



(b) Experiment 2D(2) (original copier vs. parasite)

Figure 8.19: Lengths of each chemical in the world. See also figures 8.17 and 8.18.

With a medium energy flux (experiment 2C, figure 8.18(b)), we see an intermediate state between the two extremes. During the equilibrium phase, there is exploration of different lengths within a range. More different lengths are explored than in the low energy flux case, but not as many as in the high energy flux case. Towards the end of this run, the range of lengths increases. This is similar to the high energy flux case, lending support to the idea that this is a drawn-out spike and death. But the pattern of the length increases is not identical to the high energy flux case. There is an absence of short copier chemicals compared to the numbers of longer copier chemicals. This could be because of a number of different things:

- Random chance. It could be that in this particular run, the short copier chemicals happened to decay more than the longer copiers. Worlds with high energy flux are less susceptible to this, explaining why we do not see a similar depletion of any lengths in the case of high energy flux.

- Long parasites. It could be that the longer chemicals are parasites, and are in the process of killing off the shorter copier chemicals.

- Long copiers. It could be that the collection of copier chemicals is increasing in length, and there is an advantage to being longer. Thus more long copiers are being created and the shorter ones are dying out.

I feel that long parasites is the most likely explanation, because longer chemicals are more likely to be parasites, and copier chemicals with high error rates (producing a lot of parasites) are more likely to produce long copies (because they are more likely to tangle, and slower to recover from tangles). But more experiments would be needed to prove this.

In the final experiment (2D, figure 8.19), we again see the same behaviours. At equilibrium (figure 8.19(a)) there is an exploration of a range of different lengths, with some outliers. During the spike and death (figure 8.19(b)) there is an increase in both the maximum length of the range, and in the number of different lengths explored.

Compared to experiment 2C, experiment 2D shows a greater range of different lengths explored at equilibrium. This is probably because in experiment 2D, the initial copier chemical has a lower junk level. This can be seen by the chemical length at time 0 being just over 400 in experiment 2D (junk length 4), compared to 800 in experiment 2C (junk length 16). This makes the copier chemical explore a greater range of lengths, but the population as a whole does not increase in length.

A constant feature of the equilibrium states is the maximum length reached. Ignoring anomalies, the copier chemical explores lengths up to around 1,000 atoms in experiments 2B, 2C and 2D. This consistency suggests that a length of around

1,000 atoms is an intrinsic feature of the system. It seems likely that having too many copier chemicals above this length upsets that equilibrium and sends the system into a spike and death. The precise mechanisms for this are not known. More experiments would be needed to prove this conjecture and discover its mechanisms.

Throughout the experiments, the copier chemical is able to explore lengths longer than the initial copier chemical, but not shorter. This is because mutants are much more likely to be longer than shorter. The copier chemical was designed to be able to preform small insertion and deletion mutations (section 6.4.2.3). But its embodiment within the GraphMol world causes the appearance of a large insertion mutation due to its feet tangling (section 6.4.3). Because of the embodiment, this large insertion mutation cannot be *removed* from the GraphMol copier chemical, and there is no analogous large deletion mutation that can be *added*. This introduces an asymmetry into the mutation process, allowing the copier chemical to explore longer chemicals with much more ease than smaller chemicals. The copier chemical does make single-base deletion mutations, but these are less easy to see in these diagrams than the large insertion mutations. Of the 269 mutants produced in experiment 2D(1), 10 are single-base deletions, 10 are single-base deletions combined with insertions (multiple, separate mutations in one copy), and the remaining 249 are just insertions.

Looking at these experiments as a whole, we can see general rules governing the behaviour of the GraphMol copying mechanism. From a given starting length, the copier chemical explores longer mutations of itself. It mostly searches lengths under 1,000 atoms, with only occasional anomalies above this. The greater the energy flux, the more chemicals it explores within this range. If it ventures too high above this range, it triggers a spike and death, that destroys the world. An alternative explanation for the world's death is that a build-up of low-fidelity copiers and parasites eventually overwhelms the high-fidelity copiers, an event which can be noticed by the creation of many very long chemicals.

8.4.7 Analysis: Family trees

The above analyses look at how the numbers and lengths of chemicals in the GraphMol world change over time. This section looks at the *relationships* between the different chemicals in the world, focusing on the *ancestry* of mutant chemicals.

Figures 8.22, 8.23, 8.24, 8.25 and 8.26 show family trees of the mutant chemicals produced in each experiment. Each mutant has two parents: one chemical operating as the copier and one chemical acting as the DNA. So in principle, it would be possible to draw family trees for each run, showing how each of the mutant chemicals is related. The problem with this is that GraphMol copier chemicals are more promiscuous than humans. Inbreeding is not a problem in GraphMol, and chemical species survive for a long time compared to their copying speed, so

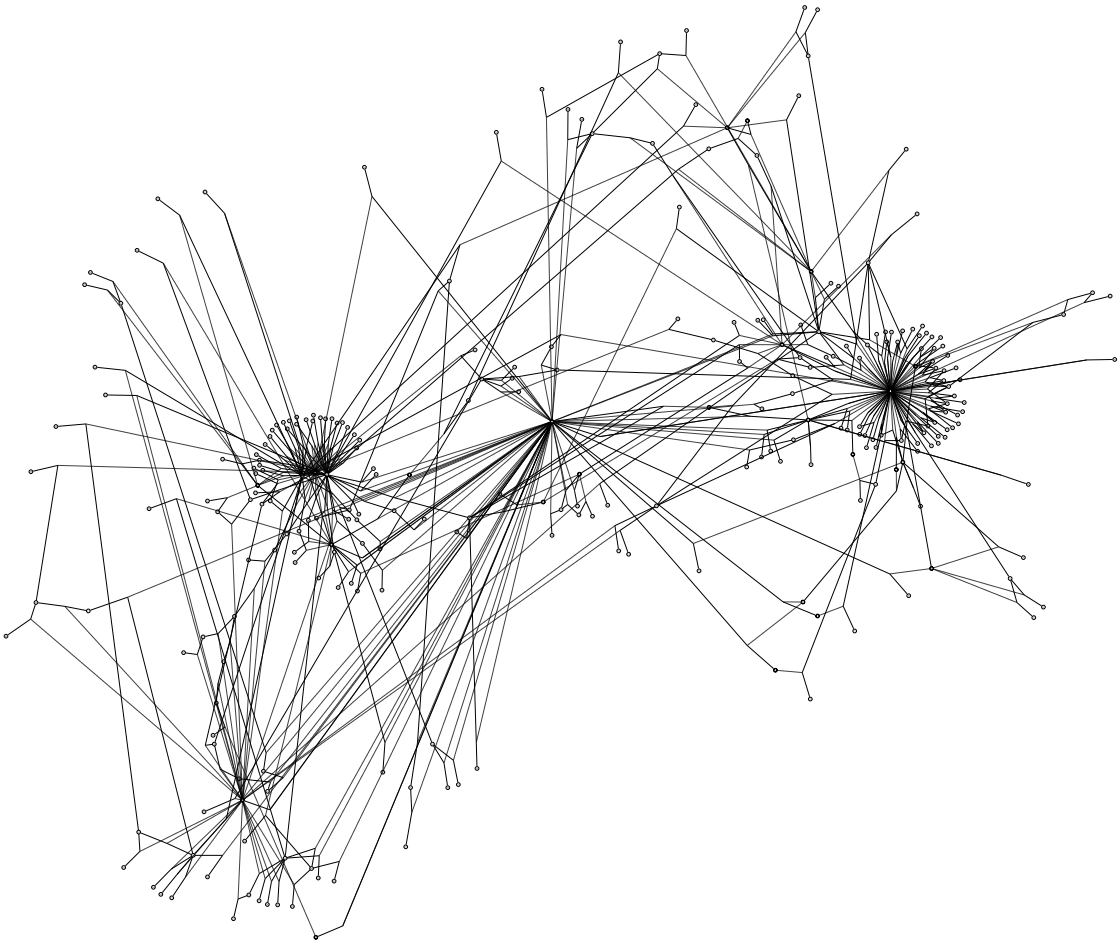


Figure 8.20: Family tree for experiment 2D(1). The promiscuity of GraphMol chemicals makes this diagram very difficult to interpret, so we look at sub-graphs of this overall family tree.

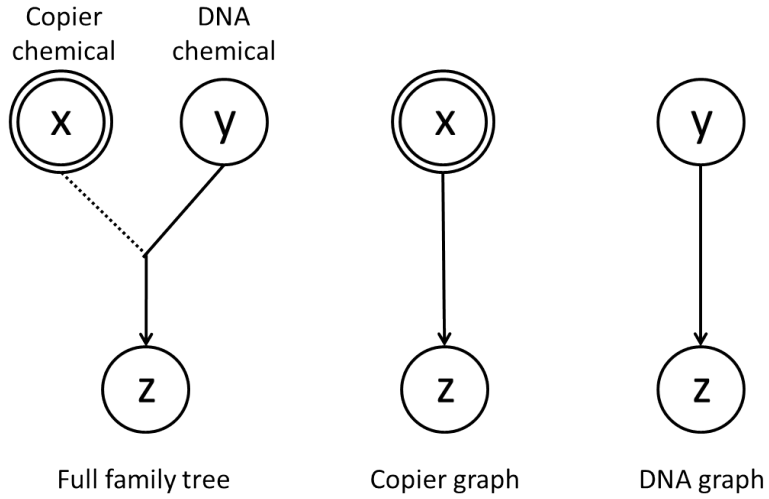


Figure 8.21: An example showing how the full family tree is visualised, and how this is simplified into the copier and DNA graphs.

GraphMol family trees look very tangled and not at all illuminating, as shown in figure 8.20. To extract some information from these tangled graphs, I have split them into four separate images, as illustrated by figure 8.21.

1. Image (a) shows a tree of just the copier chemicals. If, at some point during the GraphMol run, chemical x copies chemical y to produce chemical z, then a line is drawn from chemical x to chemical z.

2. Image (c) shows the situation from the viewpoint of the DNA chemicals. A line is drawn from chemical y to chemical z.

3. Image (b) shows the same tree as image (a), but laid out differently. Image (a) uses an *organic* graph layout algorithm that spaces out the nodes to make the tree structure visible. Image (b) positions the nodes so that their y-coordinate is proportional to the GraphMol iteration number at which they were created. This allows us to see at which time during the run different parts of the tree are created.

4. Image (d) uses the same *hierarchical* layout to display graph (c).

The nodes in these graphs are circles with the species number inside. These are not visible in the printed version of this thesis, but readers with the PDF version can zoom in on these images to see the details. The numbers represent the order in which the species were created during the GraphMol run. Species numbers can be compared across images (a), (b), (c) and (d), but not across different runs (i.e.

species 16 in experiment 2B is not the same as species 16 in experiment 2C). A double circle around a node indicates that this species has made at least one copy of a chemical during this run, so has been verified as a functional copier.

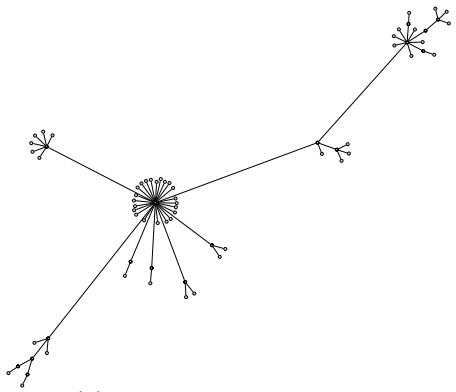
We did not make a hypothesis about the structure of the family trees produced by these experiments (apart from the trivial hypothesis that the copier chemical will make some viable mutants, thus there will be *some* family trees). So in looking at these family trees, we are not trying to find any specific features or trace the ancestry of any particular chemicals; we are looking at the overall structure of the tree. From this, we can see: how much diversity is generated *by each genome*, when it is used as a copier chemical (shown by the degree of each node in the copier trees); how much diversity is generated *from each genome*, when it is used as a DNA chemical (degrees of nodes in the DNA trees); and we can look for any interesting structures in the trees (which would indicate complex behaviour and possible novelty-generation).

The family tree diagrams agree with the previous analyses of each experiment. This supports the conclusions we have been drawing about the behaviour of the GraphMol copying mechanism:

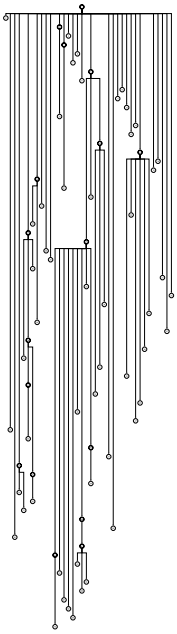
- With a low energy flux (experiment 2A), we see the least amount of diversity and very few interesting structures in the family trees (figure 8.22).
- With a high energy flux (experiment 2B), we see massive diversity but boring (low depth, and uniform) family trees (figure 8.23).
- With a medium energy flux (experiment 2B), we see a medium amount of diversity combined with interesting structures in the family trees (figure 8.24).
- In experiment 2D, we see medium diversity and interesting family trees in the first part of the experiment (figure 8.25), but this degenerates into massive diversity and boring family trees as the world dies (figure 8.26).

Low energy flux (experiment 2A, figure 8.22), while having the smallest family trees (fewest nodes and lowest branching factors), still has a copier tree that branches up to a maximum depth of five. This means that there are five generations of copier chemicals created in this experiment (the original copier chemical is a great great great grandparent). Although the world is dying throughout this experiment, the system is still able to create diversity and explore the space of copying chemicals.

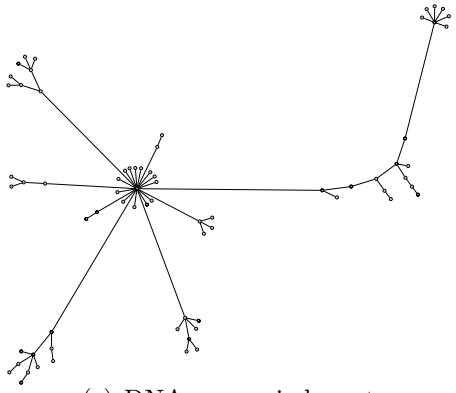
Intermediate nodes in the copier tree (nodes that are not the root and not leaves) represent mutant chemicals that are functional copiers. Some of the leaf nodes may be functional copiers, but they have not performed a copy during this run, so we do not know for certain whether they are or not. Comparing the number



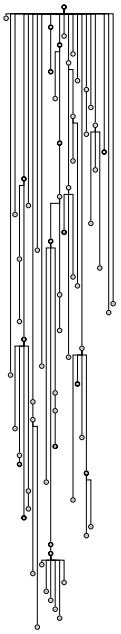
(a) Copiers, organic layout



(b) Copiers, hierarchical layout

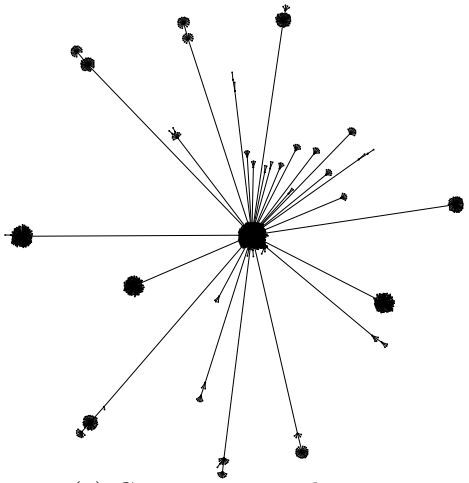


(c) DNAs, organic layout

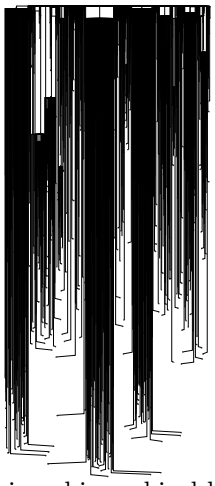


(d) DNAs, hierarchical layout

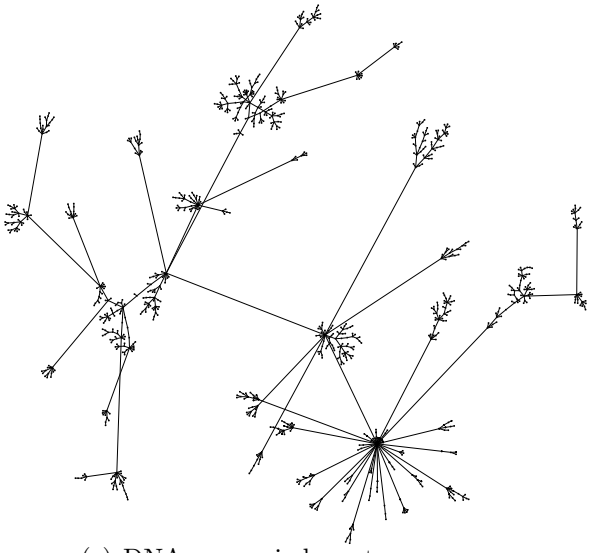
Figure 8.22: Family trees of experiment 2A (low energy flux).



(a) Copiers, organic layout



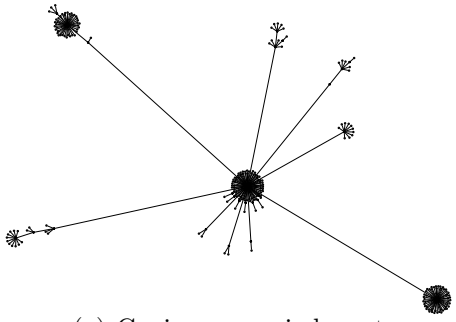
(b) Copiers, hierarchical layout



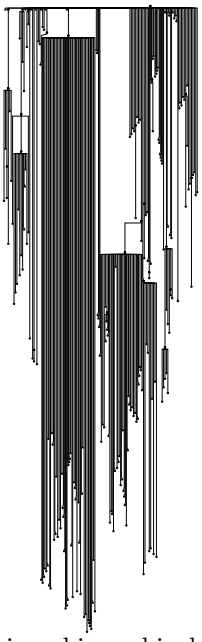
(c) DNAs, organic layout

(d) DNAs, hierarchical layout

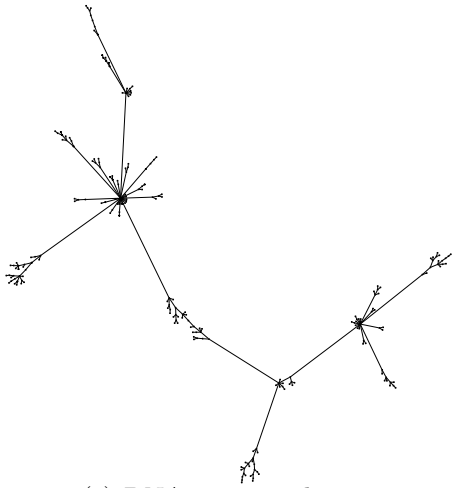
Figure 8.23: Family trees of experiment 2B (high energy flux).



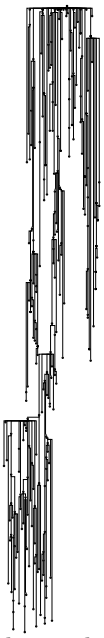
(a) Copiers, organic layout



(b) Copiers, hierarchical layout

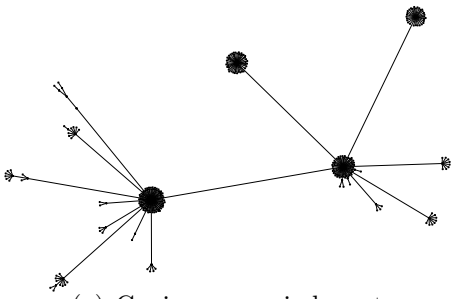


(c) DNAs, organic layout



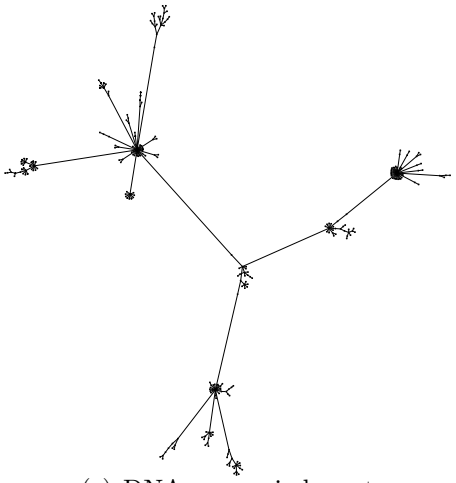
(d) DNAs, hierarchical layout

Figure 8.24: Family trees of experiment 2C (medium energy flux).



(a) Copiers, organic layout

(b) Copiers, hierarchical layout

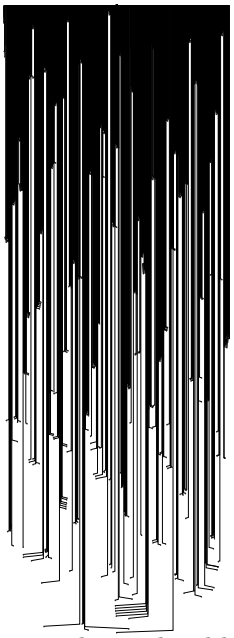


(c) DNAs, organic layout

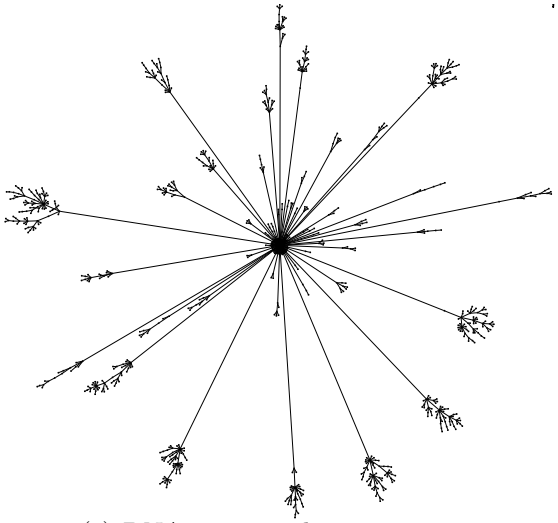
(d) DNAs, hierarchical layout

Figure 8.25: Family trees of experiment 2D(1) (medium energy flux, low junk copier).

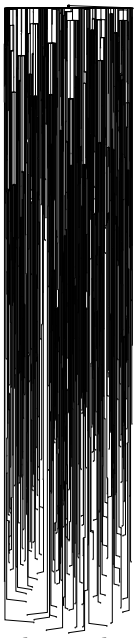
(a) Copiers, organic layout



(b) Copiers, hierarchical layout



(c) DNAs, organic layout



(d) DNAs, hierarchical layout

Figure 8.26: Family trees of experiment 2D(2) (original copier vs. parasite).

of intermediate nodes to the number of leaf nodes shows us how many copier species were created during the run, compared to the number of parasite species (plus copier species that did not manage to make a copy during the run). What the copier tree does not tell us is how many mutations each node has undergone. For example, a level-4 copier has not necessarily undergone 4 mutations. Being a level-4 copier means that this chemical is a mutant that was produced by a level-3 copier. But the level-3 copier could have been copying the original copier chemical,in which case the level-4 copier would have undergone only one mutation. Or it could have been copying a chemical that had undergone 37 previous mutations.

The DNA tree gives us different information about the behaviour of the system to the copier tree. It provides information about how many mutations a chemical has undergone. A level-4 DNA has been produced by a sequence of 4 copying reactions, each of which produced mutant copies (but two or more mutations could have occurred in one copy). But the copier chemicals performing these mutations could have been at any level. To demonstrate that a particular run is using a complex, rich copying mechanism, we look for intermediate nodes in the copier tree that are also deep nodes in the DNA tree. These nodes represent genomes that have undergone a series of mutations, but still remain viable copiers. The overall structure of the DNA tree does not carry much information. This tree is built up randomly, because GraphMol's mixing process effectively pairs up valid copier chemicals with randomly-chosen DNAs from the world. If no species died out in the world, we would expect the DNA graph to be a random small-world network, since it is constructed by a preferential attachment process (the more copies of a particular DNA, the more likely it is to be copied, and hence mutated). Species dying out will alter this, forming DNA graphs with different structures.

8.4.7.1 High energy flux (2B)

An example of *boring* graphs are those produced by the experiment with high energy flux (experiment 2B, figure 8.23). From its copier graph (figure 8.23(a)), we can see that the original copier chemical produces many mutants, but most of them do not produce copies themselves. This could be because most of them are parasites, but it is more likely that these mutants do not get a *chance* to copy, since so many mutants are being produced and competing for a finite energy supply. The same pattern can be seen with those level-1 copiers that do make copies themselves: most of their mutants do not have a chance to copy. This causes the copier graph to be very shallow: even though this experiment produces many more copies than the previous low energy flux experiment, it only generates a copier graph with a maximum depth of three, compared to five for low energy flux. From the viewpoint of novelty generation, this means that although the system is generating many new chemicals, it is not investigating the properties of these chemicals before they decay.

Thus the system is wasting its novelty-generation resources.

Also, the hierarchical graphs (figures 8.23(b) and 8.23(d)) are very homogeneous. This shows that the same types of things are happening to each different chemical species, and this does not change over time (apart from the transition between equilibrium and spike-and-death). Note that this is a very broad, high-level view of the system. There is lots of individual variation in these graphs, but the overall impression is one of homogeneity. Rather than generating novelty and changing its behaviour over time, the system keeps doing the same thing over and over again.

8.4.7.2 Medium energy flux (2C)

In contrast, the experiment with medium energy flux (experiment 2C, figure 8.24) shows interesting structure in its hierarchical graphs. We can see from the hierarchical copier graph (figure 8.24(b)) that different copier species are more active at different times during the run. This shows that the copying process changes as the run progresses. Note that this graph shows only the number of mutant *species* produced by each copier species, rather than the absolute number of *chemicals* copied. But since the absolute numbers of each species are all very low, this should give a good feel for the dynamics of the system. Also, the hierarchical DNA graph (figure 8.24(d)) shows dramatic changes as the run progresses. Since the DNAs in each copying reaction are chosen randomly, this shows that there are substantial changes to the pool of DNAs available during this run.

8.4.7.3 Experiment 2D

As in the previous analyses, experiment 2D behaves in a similar way to experiment 2C during its first part (experiment 2D(1), figure 8.25), and in a similar way to experiment 2B in its second part (experiment 2D(2), figure 8.26). This shows that the boring family trees are a signature of the world dying. They show that all the world's energy is being used to copy a parasite, which generates more parasites and compounds the problem. From a novelty-generation viewpoint, the generation of novelty has stopped. Or, more precisely, all the novelty-generation potential of the system is being used to generate novel parasites that are killing the system.

8.4.7.4 Discussion

Looking across all of the organic copier graphs (sub-figures (a)), we see the original copier chemical in the centre of the graph, surrounded by mutants. Many of these mutants are leaf nodes, indicating that they are either parasites or copier chemicals that did not get a chance to copy (so they acted as parasites). But some are copiers in their own right, and are surrounded by a collection of their own child parasites

and copiers. We talked above about the equilibrium state of the system being sustained by a *collection* of copier species, rather than a single dominant species. I think it is very likely that this collection of copier species is composed mostly of those graph nodes with many children.

Experiment 2B (figure 8.23(a)) shows us that with a high energy flux, the branching factor of this graph is huge and so the space of possible copier chemicals is explored in a very shallow way. Experiment 2A (figure 8.22(a)) shows that with a low energy flux, the branching factor is small so we can travel deeper into the graph but we cannot explore as many different options along the way. Experiment 2C (figure 8.24(a)) shows that with an intermediate energy flux, we can trade off these two extremes. This is clear in experiment 2D(1) (figure 8.25(a)), where we can see that the organic copier graph is not symmetrical about the original copier species. There is at least as much exploration happening from an early copier mutant (species 28) as from the original copier chemical. From the viewpoint of novelty generation, this is evidence of the search process moving to a different area of the search space from where it was initialised.

8.4.7.5 Copier family trees

The images above visualise every *node* in the family tree while dividing its edges into two groups: copiers and DNAs. Each graph discards one set of edges to make the image more understandable. An alternative approach is to visualise every *edge*, but divide the nodes into two groups: copiers and parasites, drawing the copiers and discarding the parasites. Drawing just the parasites would not be worthwhile, because that would produce a completely disconnected graph (as it is the copiers that connect nodes together via copying reactions). Figures 8.27, 8.28 and 8.29 show these graphs. In these images, a dashed line from a parent to a child indicates that this parent was a *copier chemical* in the reaction, and a solid line indicates that this parent was a *DNA chemical*. Where a node appears to have only one parent, this is when a chemical reacted with another copy of itself (there are two lines, but they are drawn over the top of each other).

In these runs, we can say definitively that some chemicals are *copiers*, but we cannot say definitively which are *parasites*. We know that some chemicals have copied others during the run, so these must be copiers. And we label as *parasites* those chemicals have not acted as copiers during the run. This does not mean that they would not function as copiers if given the chance. So we also include in these graphs all of the ancestors of every copier chemical.

Looking at the graphs, we see that some *parasite* chemicals (nodes with only one circle around them) act as DNA chemicals in reactions producing mutants that are valid copiers. From this we can infer that although these chemicals did not perform a copy during this run, they are actually valid copiers and just did

not have a chance to copy due to GraphMol's stochasticity. It would technically be possible for a mutation to turn a genuine parasite into a valid copier, but the probability of this happening is so low that we can discount it. It would require the parasite to be created by a two-atom insertion mutation in one of the copier's binding sites. (This is the most efficient way to create a parasite.) This parasite would then have to be copied by a reaction that happened to make two deletion mutations (since deletions only remove one atom) at the appropriate places in the binding site (or undergo two separate reactions that each performed a deletion mutation in the appropriate place).

These images confirm the story told by the other analysis methods. At low energy flux (experiment 2A, figures 8.27(a) and 8.27(b)), the space of possible copier chemicals is explored deeply but not in very much detail. At high energy flux (experiment 2B, figures 8.27(c) and 8.27(d)), the space is explored broadly but not very deeply. At medium energy flux (experiments 2C and 2D(1), figure 8.28) we trade off these two extremes, with many deep nodes in the graph. And in experiment 2D(2) (figure 8.29), the world is dying rather than exploring its search space.

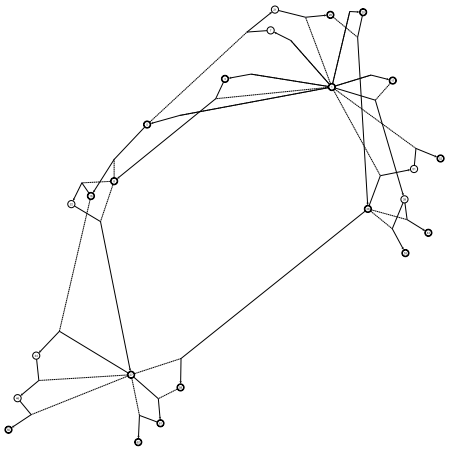
These graphs show more clearly than the previous images the way in which GraphMol is exploring its search space. For example, it is fairly clear from figure 8.29 that experiment 2D(2) is not doing much exploration (if we assume that this figure is representative of the system's exploration). But to discern this fact from the previous images of experiment 2D(2) requires a deeper understanding of GraphMol. Also, it is clear from these images that experiment 2B behaves very differently to experiments 2A, 2C and 2D(1) (which are more similar to each other). This is because experiment 2B it has a much higher energy flux (of 100, compared to 10 for experiment 2A and 20 for experiments 2C and 2D).

8.4.8 Analysis: Hyperbolic parasites

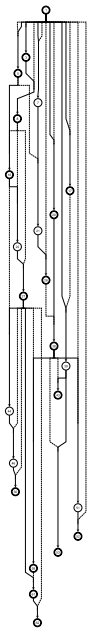
In contrast to the previous analyses, that look at the results of these experiments and try to infer the reasons for observing them, there are different ways to analyse experiments. One such alternative way is to consider in a abstract sense the *types of behaviour* we might expect this type of system to perform, and then look at the results of the experiments to see if this is what we actually observe.

Since the GraphMol copier chemical reproduces sexually in a well-mixed world, we would expect its growth law to be hyperbolic (section 7.4.1). There has been a lot of work done on hyperbolic growth of replicators in physical-world biological systems (see [96] for references), but the analysis most applicable to these experiments is Kelly et. al.'s classification of parasite types in virtual systems undergoing hyperbolic growth. This is introduced in [70], and described fully in [57, pp.60-74].

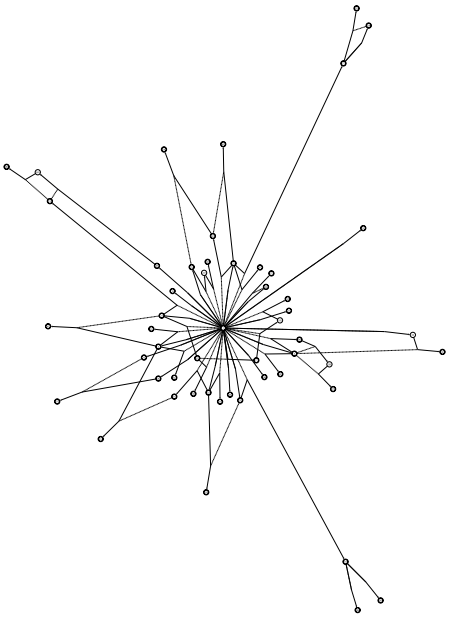
Kelly's work takes the opposite approach to GraphMol. Rather than building



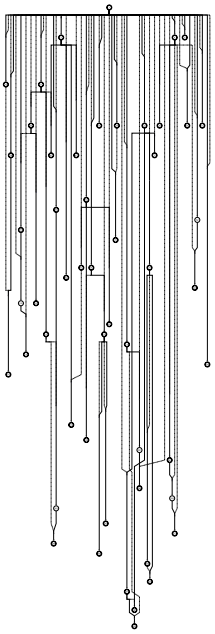
(a) 2A, Reactions, organic layout



(b) 2A, Reactions, hierarchical layout

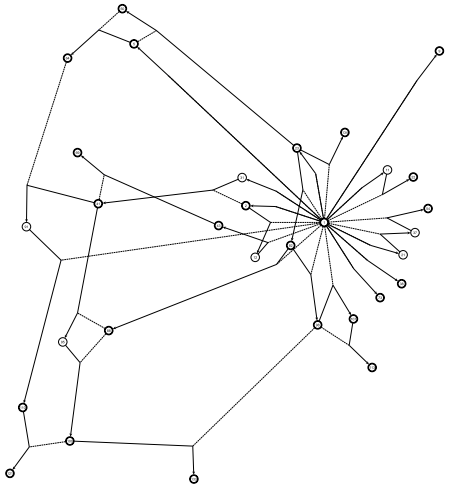


(c) 2B, Reactions, organic layout

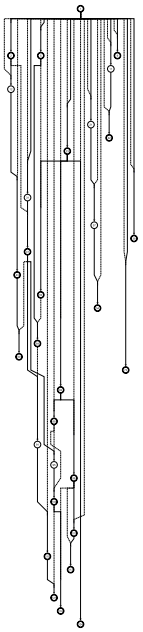


(d) 2B, Reactions, hierarchical layout

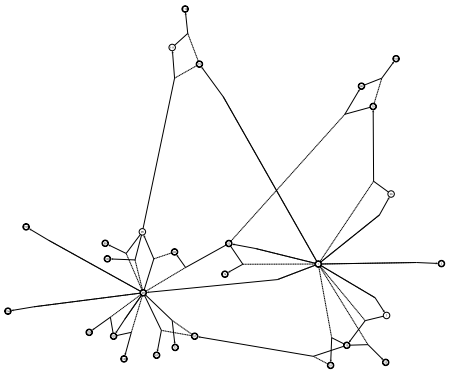
Figure 8.27: Family trees showing only copier chemicals.



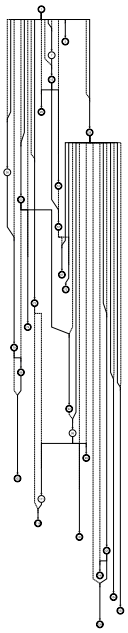
(a) 2C, Reactions, organic layout



(b) 2C, Reactions, hierarchical layout



(c) 2D(1), Reactions, organic layout



(d) 2D(1), Reactions, hierarchical layout

Figure 8.28: Family trees showing only copier chemicals.

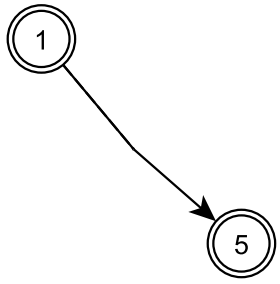


Figure 8.29: Family tree of experiment 2D(2), showing only copier chemicals.

a complicated system inspired by biology and looking for interesting behaviour, Kelly builds a very simple system in which it is possible to enumerate all the different behaviours and analyse them mathematically (under certain simplifying assumptions). Although these two approaches are very different to each other, neither one is *right*, or *better*, than the other. These systems are very complex, and in order to understand them we need a variety of different approaches. I feel that the most important insights will come from combining different approaches to obtain the strengths of both.

8.4.8.1 Assumptions

Kelly's system is intentionally much simpler than GraphMol, so it is not possible to map the two systems onto each other exactly, or make quantitative comparisons between them. But by making clear the differences and similarities between the two systems, it might be possible to see some of the theoretical dynamics described by Kelly reflected in the complicated, practical example of GraphMol.

Space in Kelly's system is the typical artificial chemistry method of pairing up chemicals from a list; it lacks GraphMol's concept of *volume* in aspatial spaces (equation 6.5, section 6.3.1.1), and GraphMol's ability to vary the total number of chemicals present in the world. In Kelly's system it is possible to show mathematically that his definition of space leads to hyperbolic growth of sexual replicators. In GraphMol's more sophisticated definition of space, the mathematics is more complicated (because of equation 6.5). It might be that the GraphMol copier *does* grow hyperbolically, and equation 6.5 merely lowers its growth rate rather than qualitatively changing its growth law, but this is not guaranteed.

In Kelly's system, copying reactions happen instantly. But in GraphMol, longer chemicals take longer to copy, and cannot participate in other reactions until they have finished copying. Also, GraphMol's energy model (Kelly's system has no energy model) effectively lengthens the reaction time of copying reactions when

many reactions are happening in parallel. I do not know what effect this has on the growth law of GraphMol copiers. The fact that GraphMol copying reactions take time to copy makes them similar to the reaction equations of parabolic growth (equations 7.4 and 7.5). But the similarity is not exact, so again, drawing firm conclusions either way is not possible.

Kelly's analysis models the replicating chemicals using Ordinary Differential Equations (ODEs). This comes with the weighty assumption that the effects of stochasticity can be ignored. This can be a dangerous assumption to make, because some systems of chemical reactions display nonlinear effects such as *stochastic focussing* [78]. In this case, noise on the input to a sequence of chemical reactions increases the *average* response of the system. ODE models are not able to predict effects such as this; the stochastic dynamics cannot be regained from the ODEs by adding noise (from any distribution) on top of the ODE solution. Additionally, the system does not need to have low numbers of chemicals for these effects to occur (which is a common misconception). Kelly does point out some of the limitations of ODE models, and validates his models by running stochastic simulations of them. While ODE models are not inappropriate as Kelly has used them, we need to remain cautious about applying them to more complicated systems of chemical reactions in the future.

The final assumption of Kelly's analysis is that there are only two different species of chemical present at any one time, and the chemicals do not mutate. This is necessary to enumerate all the different types of behaviour and analyse them all, as the number of combinations grows very rapidly with the number of species considered. Also, there would be too many combinations to consider all the different ways in which the chemicals could mutate. Kelly considers many different types of mutation, and recognises that he has not considered every permutation.

8.4.8.2 Classes of behaviour

The result of Kelly's analysis is an exhaustive enumeration of the 10 different classes of behaviour that pairs of species can display. The two chemical species are labelled A and B. In Kelly's simplification of the copying process, these species can be: self-replicators (A copies A); parasites (A copies B); or inert (A does not copy A or B). Considering all the permutations of these behaviours, and taking symmetry into account, gives 10 different classes of behaviour:

0. Both species are inert.
   No reactions happen and the numbers of each species do not change.

1. Both species are self-replicators and parasites of each other.
   Both species co-exist at equal levels.

2. A is a self-replicase, B is inert.
   A displaces B in the world.

3. Neither are self-replicators, B is a parasite of A.
   B increases in number, displacing A, at which point the reactions stop.

4. A is a self-replicator, B is a parasite of A.
   In the ODE model, the species are copied but their numbers stay exactly the same. If copies of A mutate into more class-4 parasites, then these eventually destroy the copying process as in class 3.

5. B is a self-replicator, B is a parasite of A.
   B displaces A in the world.

6. A and B are self-replicators, B is a parasite of A.
   B displaces A in the world.

7. A is a self-replicator, both are parasites of each other.
   A displaces B in the world.

8. Neither species are self-replicators, but both are parasites of each other.
   Both species sustain each other, at equal concentrations, in a hypercycle.

9. Both species are self-replicators, neither are parasites.
   The species with the greater numbers displaces the other.

There are more subtleties and further details in [57, pp.60-74], but for our purposes the above description summarises the different classes of behaviour. We need only a high-level description of these behaviours, because we will be able to make only qualitative comparisons with GraphMol.

8.4.8.3 Dynamic equilibrium

The GraphMol behaviour I describe as a *dynamic equilibrium* may be the same as what Kelly calls a *class-1 quasi-species*. This is a collection of chemicals that are all class-1 mutants of each other. *Class-1* means that both species are self-replicators and both are parasites of each other. i.e. A copies both A and B, and likewise B copies both A and B. This can happen in GraphMol if the original copier chemical produces a copy of itself with a small mutation in one of its junk regions. This mutation will change the probabilities of some of the mutant copier's steps, but the mutant will still be able to copy (both itself and the original copier). It is feasible that multiple such mutants could be produced by making small changes to the original copier chemical and to class-1 mutants of the original copier.

According to Kelly's analysis, if a copier species generates class-1 mutants, then these sustain each other in the world at equal levels due to negative frequency-dependent selection. If this is what is happening in these GraphMol experiments, then it explains why the number of original copier chemicals decreases during the dynamic equilibrium phase.

Kelly also says that if the number of possible class-1 mutants is much greater than the carrying-capacity of the world, then it will not be possible to sustain *all* class-1 mutants at their equilibrium level in the ODE model (because their equilibrium levels will be less than one chemical each). This will cause the quasi-species to *drift*, as the particular chemicals representing it at any one time change. If this is what is happening in these GraphMol experiments, then it explains why there is constant production of new species during the dynamic equilibrium, despite the fact that the total number of different species stays roughly constant.

8.4.8.4 Spike and death

The GraphMol behaviour I describe as a *spike and death* may be the same as what Kelly calls *class-4 obligate parasites*. A *class-4 parasite* is a parasite that does not replicate itself. i.e. A copies both A and B, but B performs no copying. This can happen in GraphMol if a copier chemical undergoes a mutation in one of its binding sites. It is feasible that such mutants could arise in GraphMol runs. In GraphMol, these mutants behave slightly differently to Kelly's. In Kelly's system, the class-4 mutants can be copied by their host, but they do nothing in return. In GraphMol, a 'class-4' mutant can be copied by its host, but it does not do *nothing* in return. It initiates a copying reaction with its host, but at some point in the reaction (when the mutated binding site would have bound to the DNA chemical), the reaction fails to proceed. From this point on the reaction cannot complete, but it does not end: it stays in the system, using up energy, until one of the chemicals randomly decays. In Kelly's system, the possible reactions are:

- A copies A;
- A copies B.

But in GraphMol, they are:

- A copies A;
- A copies B;
- B binds to A, preventing both chemicals from participating in further reactions for a long time, wasting energy, before destroying both chemicals.

In GraphMol, mutations to low-level mechanisms can cause changes that happen over a long time, affecting: (1) other mechanisms in the world; (2) the binding model; and (3) the energy model. This complexity was purposely built into GraphMol, but it makes GraphMol difficult to analyse mathematically, and difficult to compare with simplified models such as Kelly's. This does not mean that we *cannot* make comparisons; it just means that we have to know the limits of our comparisons.

According to Kelly's analysis, if a copier species generates a small number of class-4 mutants, then the original copier species and the class-4 mutants co-exist with each other. But the difference between class-4 and class-1 is that in class-4, the parasites are not helping with the replication process, so the overall replication rate is lower. This means that if the copier species *keeps* generating new class-4 mutants, then over time the replication rate will drop until the copier chemical can no longer replicate itself and is diluted out of the system.

Despite the above caveats about the mechanics of GraphMol being very different to Kelly's system, the spike-and-death phenomenon in GraphMol does bear more than a passing resemblance to Kelly's analysis of class-4 obligate parasites in his system. When GraphMol switches between its dynamic equilibrium phase and its spike-and-death phase, it shows very different behaviours in the number of species present and the lengths of chemicals. It would be interesting to see if other systems show similar behaviour when switching between class-1 and class-4 mutants. This would shed some light on whether *class-1 and class-4 mutants* are useful models of GraphMol's behaviour, and would help investigate the transition between class-1 and class-4 behaviour.

8.4.8.5 Unexplained phenomena

Equating GraphMol's dynamic equilibrium with Kelly's class-1 mutants seems to me a fair comparison. Although the two systems are very different, the high-level behaviours seem to be the same: mutually-replicating mutants sustaining each other in a drifting quasi-species.

The comparison between GraphMol's spike-and-death and Kelly's class-4 mutants is more strained. It may be that the class-4 dynamic is precisely what it happening in GraphMol, but GraphMol is too complicated, and these experiments too sparse, to say this for certain at the moment. Since the class-4 dynamic can be slow to act, we need long experiments to show it. Also, since the class-4 dynamic can be slow, complicated systems (such as GraphMol) may contain other mechanisms (not included in Kelly's analysis) that inhibit or prevent the class-4 dynamic before it can destroy the copying phenomenon completely. This would make the class-4 phenomenon difficult to pin down precisely.

Finally, GraphMol has a couple of behaviours that Kelly's analysis does not

explain. In experiment 2D(1) (figure 8.12(a)), the *competitor* species increases in number towards the end of this experiment. This is why I investigated this species in experiment 2D(2). If this was a pure class-4 mutant in the middle of a pure class-4 behaviour, then we would expect all class-4 mutants to exist at the same levels. The increase of this particular mutant could be merely chance, or it could be evidence of a different mechanism at work. Also, Kelly's analysis cannot explain the spikes of energy in experiment 2C (figure 8.11), as Kelly's system has no concept of energy.

8.4.9 Overall discussion

The purpose of this series of experiments was to investigate the behaviour of the GraphMol copying mechanism *in the field*: in an open, evolving world. We wanted to test whether the behaviours predicted by the previous *lab-based* experiment (chapter 7) would carry over into an evolving GraphMol world, and see if any additional behaviours would be present in the evolving case. We find that the previous results about the copier chemical do carry over into these experiments, but in more complex ways than might be assumed. Rather than seeing one dominant copier species, we observe a collection of copiers that sustain each other in the world, and a collection of parasites that destroy the world. We identify two distinct behaviours displayed by the system: a dynamic equilibrium in which the number of chemicals in the world remains constant while their composition continually changes; and a spike-and-death, where parasites are replicated rapidly, overwhelming the system and killing all the viable copiers.

8.4.9.1 Searching for copiers

These experiments show that GraphMol's embodied copying mechanism can survive in an evolving world. It can sustain itself in a constantly-changing environment, while exploring different mutations of the original copier chemical. The original copier chemical can create mutant copiers that are different from the original and are themselves viable copiers. One definition of *successful reproduction* is captured in (one of the many) biological definitions of a *species*: two organisms belong to the same species if they can have children that can themselves have children [15, p.236]. i.e. they are capable of having grandchildren. The family trees presented in this section show that the GraphMol copier is capable of having grandchildren, and is capable of producing mutants that are themselves capable of having grandchildren.

The GraphMol world has a parameter, *energy flux*, that controls the way in which the copier chemical's family tree is searched:

- At low energy flux, the focus is on tree *depth* rather than *breadth*. If the energy flux is too low, then this pressure for depth destroys the world. Some 'breadth' is always needed to replicate the chemicals that are currently present in the world.

- At high energy flux, the focus is on breadth rather than depth. If the energy flux is too high, then this pressure for breadth makes the world run slowly. Looking too closely to home means that it takes a long time to find new and interesting places.

- At intermediate energy flux, we can trade off these opposing forces and observe complex behaviours in the GraphMol world.

8.4.9.2 Chemical lengths

The *sweet spot* in copier length shown in the *lab-based* experiment was not observed *in the field*. This is because the length-changes undergone by the evolving copier chemical are more complicated than might be expected.

Firstly, there is a bias in the copier's embodied mutation function, favouring the creation of long mutants. The copier chemical was designed to make short insertion and deletion mutations, which it does do. But also, it has a behaviour that was not (explicitly) designed into it: its feet can become tangled, resulting in long insertion mutations. This is good from the viewpoint of embodiment and novelty-generation, because it is an example of how embodied mechanisms can have additional behaviours that are relevant to the phenomenon they are embodying (an embodied mutation function has a different type of mutation that was not programmed into it). But this also highlights the inherent complexity of embodied mechanisms, and the need for principled ways of designing them and understanding them.

Secondly, there appears to be a maximum length of around 1,000 atoms, over which copiers are not sustainable. Some chemicals occasionally appear over this length, so it is not a hard limit, but the vast majority of chemicals are under this length while the GraphMol world is alive. A feature of the GraphMol world *dying* is the appearance of many chemicals longer than 1,000 atoms. It is not clear whether the appearance of long chemicals *causes* the world to die, or whether the world dying causes many long chemicals to be produced.

Thirdly, asking what is *the length of the copier chemical* in a GraphMol world is not a well-posed question. In a GraphMol world, there is not just one dominant copier chemical, but a collection of copiers that sustain the world. They can all have different lengths, which makes the simplistic question of whether the length of the copier chemical is *increasing* or *decreasing* more complex.

8.4.9.3 Hyperbolic growth

It might be that the novelty-generation of these experiments was hampered by the hyperbolic growth of the GraphMol copier chemical (sections 7.4.1 and 8.4.8). This is due to the interaction of three different design decisions:

1. GraphMol using an aspatial, well-mixed space.
   In artificial systems we are able to choose our own definition of space, so we could choose whatever gives the growth laws we want.

2. The GraphMol copier chemical reproducing sexually.
   Different copying mechanisms could be designed, with different replication mechanics, to have different growth laws.

3. Mutations in the copier chemical only being able to change its rate and accuracy of replication.
   If we had used a rich binding function in GraphMol, then mutations to its binding site patterns could change the specificity of binding between different copier species.

Changing any of these decisions could change the growth laws of GraphMol copier chemicals, and so potentially change the novelty-generation properties of GraphMol. As described in section 7.4.1, exponential growth would allow selective takeovers of *fitter* species, as we hypothesised might happen in these experiments; and parabolic growth would allow more species to co-exist at the same time, increasing the diversity of species that can persist in the same world.

The purpose of these experiments was not to demonstrate GraphMol's superiority in any particular area, but rather to explore GraphMol's behaviour and assess its potential for novelty-generation. These experiments have amply demonstrated GraphMol's ability to display complex behaviour that is:

- complicated enough that it cannot be analysed completely by traditional approaches such as ODE models; and

- simple enough that we can narrate the story of the experiments and gain some high-level insight into its behaviour.

These experiments have explored some of the consequences of hyperbolic growth in GraphMol, and shown many different ways in which GraphMol could be extended in the future.

8.4.9.4 Novel behaviours

These experiments have conclusively shown two different behaviours that the GraphMol copier system can display:

1. Dynamic equilibrium, where a collection of copier chemicals sustains itself at a constant number over time (with stochastic noise), while the composition of this collection changes continually (the species present, and the number of chemicals of each species).

2. Spike-and-death, where a collection of parasites exploit the copiers to be replicated massively, before the number of parasites destroys the copiers (after which, the parasites decay).

From the viewpoint of novelty generation, the dynamic equilibrium is when the system is exploring the search space of different copier chemicals. There is a (stochastically) constant number of chemicals in the world because at this number, the copying rate of the chemicals (caused by the energy flux) matches the decay rate of the world. GraphMol's *energy model* allows this dynamic equilibrium to happen, which (in the terminology of evolutionary algorithms) embodies a *dynamic population size* within the GraphMol world.

Although the number of chemicals in the world remains constant, their composition is continually changing. There are new mutant chemicals entering the world through copying reactions, and old chemicals leaving the world via decay. This is how the system explores the search space of different copier chemicals. New chemicals can displace old ones in the population, moving the collection of chemicals to different places in its search space.

Unfortunately, parasites upset the dynamic equilibrium after a time, and destroy the GraphMol world. Parasites are a common feature of aspatial artificial chemistries (section 2.4.2.2), so the evolution of parasites in these experiments may well be a consequence of the aspatial world that we have used, rather than anything specific about the GraphMol language. There is a great detail of literature from theoretical biology about the evolution of parasites, including mechanisms for allowing systems to survive despite parasites [67, chapter 4]. These ideas could be used in future versions of GraphMol, to prolong the lifetime of GraphMol runs and increase its novelty generation.

8.4.9.5 Looking forwards

A GraphMol world is a very complicated and complex place. There are multiple copies of different chemical species, all of which are related to each other, interacting in an aspatial world. And different parameter values create different GraphMol worlds with different properties and different behaviours of the copying mechanism.

The analysis provided in this chapter looks at four different worlds from a high-level viewpoint. The figures show different ways of obtaining an overview of the whole GraphMol world, from one perspective (e.g. number of chemicals, lengths of chemicals over time, relatedness of DNAs, etc.) Looking at the world as a whole allows us to follow the behaviour of the copying *phenomenon*, rather than the individual interactions of separate copying *mechanisms*.

An alternative approach would be to focus on the individual chemicals present in the world, and analyse their timestep-by-timestep interactions. Figure 8.16 is an example of this, showing the differences between one particular copier chemical and one particular parasite. In this chapter, I have chosen to take the high-level view because the GraphMol mechanisms are not performing any particular function (such as, for example, flying a helicopter). They are just copying each other. We are not interested here in the *mechanics* of copying; we are interested in how these mechanics affect the high-level *phenomena* (such as dynamic equilibria, and spike-and-deaths). In the future, when novelty-generation systems are being used to discover novel ways of performing a particular function (such as flying a helicopter), then looking in detail at *how* they are performing it will be useful.

Chapter 9

Conclusions and future work

This concluding chapter starts from the GraphMol experiments described in the previous chapters and moves up through the levels of abstraction in this thesis. We critique GraphMol, discussing its positive and negative points, and examine what GraphMol has taught us about embodiment. We then evaluate embodiment, as we have applied it to artificial chemistries, and speculate on what embodiment can tell us about meta-evolution. Avenues for future work are suggested, where this thesis could be built upon in different ways and at different levels.

The novel contribution of this thesis comprises three parts:

1. A review of how meta-evolution has developed historically, through different novelty-generation algorithms, by changes to their biological and computational models. This includes a classification of the different types of meta evolution (chapter 2).

2. Applying the ideas of embodiment to the problem of meta-evolution. This includes:

   (a) A new definition of embodiment (chapter 3).

   (b) A detailed evaluation of how biological systems benefit from embodiment, discussing how they use it to implement meta-evolution and novelty-generation (chapter 4).

   (c) A discussion of how embodiment can be realised in artificial systems, taking inspiration from the biology, using artificial chemistries as a language in which to program embodiment (chapter 5).

3. A new artificial chemistry, GraphMol, designed with embodiment in mind. GraphMol is used to implement an example embodied mechanism (copying), and experiments are performed to evaluate the novelty-generation capabilities of the system (chapters 6, 7 and 8).

The benefit of embodiment is that it gives us a principled and incremental method for improving the novelty-generation capabilities of our algorithms. It is *principled* because it involves changing the biological models of novelty-generation algorithms (section 2.2), to make them more biological. It is *incremental* because we can embody different parts of our algorithms to different degrees (section 5.2). This gives us a method for gradually improving existing algorithms.

9.1 Critique of GraphMol

GraphMol is a novelty-generation algorithm based on artificial chemistries, that allows us to implement embodied mechanisms, as demonstrated by the copying mechanism described above. Previous novelty-generation algorithms (described in chapter 2) implement their mechanisms crisply or stochastically (as described in chapter 3), rather than embodying them within an evolvable world.

Embodying the copying mechanism within the GraphMol world allows us to investigate meta-evolution in a principled and incremental way, but these benefits do not come for free. The GraphMol code runs very slowly and contains many design decisions on different levels, some of which are arbitrary and/or could be improved. This section critiques GraphMol and explores how GraphMol could be changed on three different levels: its *physics*, *chemistry* and *biology*.

9.1.1 GraphMol is slow

One of the main issues with GraphMol is that it takes a long time to run: from three weeks to two months per run, depending on the parameters of the GraphMol world (section 8.4.4). This is partly because I implemented it in the Python programming language, which is an interpreted language and executes slowly compared to compiled languages such as C++. So some speedup could be gained by re-implementing my code in a language such as C++. Also, raw computer power has been increasing exponentially for the last 40 years and shows no signs of slowing, so simply waiting for more powerful computers will increase the speed of GraphMol runs. (Folk wisdom says that one of the reasons for Genetic Algorithms originally becoming popular in the 1990s was that computer hardware became powerful enough to run them.)

But it is not just the implementation of GraphMol that makes it run slowly. The more fundamental reason for GraphMol's slowness is that it is doing a lot of computation. GraphMol has an advanced concept of *space*, which is inherently parallel and computationally-intensive. These issues are explored below.

9.1.1.1 Parallelism

One reason for GraphMol's slow execution is that it contains *inherently parallel* algorithms that are being run on a sequential computer. The experiments described in this thesis were run on a multi-core server, with around 10-20 cores available at any one time (the server was shared with other users). This means that some parallelism was possible: each reaction was run in a separate thread. But this gain is small compared to GraphMol's inherent parallelism: each atom's distance tables, and each instruction pointer, could potentially be handled separately. This gives a conservative estimate[1] of over 100,000 separate GraphMol threads that could be run in parallel, with suitable hardware and a suitable communication mechanism between threads. And this number will only increase as we run larger GraphMol worlds containing more complicated mechanisms. The use of massively parallel hardware would be a potential way of speeding up GraphMol in the future.

9.1.1.2 Space and distance

I have profiled the GraphMol code, measuring the average-case runtime of each process. The part that takes the vast majority of the time (80%–90%) is the *graph distance process* (section 6.3.1.2). This is the algorithm that keeps track of the distances between each pair of binding sites in the GraphMol world, influencing the probabilities of binds between sites.

Section 6.3.4.1 describes how GraphMol's current implementation of this process is not slow due to an inefficient implementation of the algorithms, but is slow because calculating graph distances is genuinely a computationally-intensive task. This highlights the main tradeoff with embodied novelty-generation algorithms: if you want to embody processes in an interesting space, then you must be prepared to pay for that interesting space with runtime.

This process is an abstraction of physical *space*. Simulating a three-dimensional space for the atoms to move in, would be more intensive than GraphMol's *graph distance process*. Graph distances reduce the computation needed, since the only property of space that GraphMol uses is the *distance* between points, so only that is calculated. Novelty-generation algorithms have the benefit of being able to choose their own definition of *space*, rather than being limited to just the physical world (section 6.3.1.1). But once we define non-trivial spaces, running them becomes expensive due to the inherently parallel nature of space. Biological systems avoid this problem by using the massively-parallel computer that is the physical world.

[1] A back-of-the-envelope calculation: a copier chemical with a junk length of 16, encodes as a DNA chemical with 2,421 bases (section 8.2.1). This has three binding sites per base, and there are around 15 DNA chemicals concurrently participating in reactions during the equilibrium phase of experiment 2D(1) (and 15 copier chemicals interacting with them). Ignoring everything else, this gives $2,421 \times 3 \times 15 = 108,945$ binding sites in the world, that could run in parallel.

Different versions of (virtual) space will have different computational requirements, and so it may be useful in the future to look at using different versions of space in GraphMol, to see what they bring to the system in terms of computational efficiency and novelty-generation. Since most artificial chemistries use very simple definitions of space, there is not much to go on from the literature when designing new types of space. So far, we have (in increasing order of runtime):

- The non-existent space in which mathematical functions execute.
- The trivial aspatial pairing up of chemicals from a list.
- Stringmol's (and GraphMol's) more sophisticated aspatial mixer with a concept of *volume*.
- GraphMol's graph-based space, with a concept of *distance*.
- Euclidean spaces in 2 or 3 (or n) dimensions.

Also, there are two different places where space is used in Artificial chemistries: in space in which chemicals mix before binding; and the space in which chemicals undergo reactions. More work is needed to investigate the relationship between these two spaces: when it is appropriate to make one complex and the other simple; whether we gain anything by making them both complex; and when we can perform both of these processes in the same space.

9.1.1.3 Stochasticity improves performance

During the development of GraphMol, I found a useful design pattern for optimising operations on large data structures. Rather than iterating over a large data structure every timestep, GraphMol iterates over a small, randomly-chosen sample of the data structure. This allows an approximate answer to be computed quickly, and over multiple timesteps the accuracy of this answer will improve. This pattern appears twice in the design of GraphMol: in calculating graph distances (section 6.3.1.3) and in choosing which binding sites bind (section 6.3.1.2).

In the case of graph distances, the large data structure is U from algorithm 7: the set of all nodes in the graph that need to update their routing tables. One strategy would be to iterate over all the nodes in U every timestep. This would ensure that we always had a precise measurement of the distances between any pair of nodes in the graph. But GraphMol does not do this. It optimises this process, iterating through a random sample of the nodes in U every timestep. This means that the distance tables used by GraphMol are an approximation to the exact distances through the graph. Figure 6.14 shows that this approximation converges quickly to the exact distances.

In the case of binding, the large data structure is B from algorithm 5: the set of all free binding sites in the world. Again, we do not iterate through every binding site in B every timestep: we iterate through a random sample of them. This means that the value returned by the binding probability function is an approximation of the probability of two sites binding, and this approximation becomes more accurate as time progresses.

Because GraphMol is a virtual world used for its own sake (rather than a simulation of a different world), we can perform these optimisations without worrying about loss of accuracy. It does not matter that we do not always use the precise distances through the graph, because the precise distance through the graph does not have any meaning outside the GraphMol world. We define the GraphMol world, so we can choose to define *graph distance* as the current approximation of the exact distance. The only factor we need to consider is the effect that the approximation has on the (novelty-generation) properties of the GraphMol world, and the mechanisms that can be embodied within it. In this case, the approximation improves the novelty-generation capabilities of the world because it makes the world run faster, so we can perform more experiments.

Making each approximation adds a parameter to the GraphMol world, specifying the size of sample to iterate over each timestep. This allows us to trade off the accuracy of the approximation against the speedup gained.

9.1.1.4 Tradeoffs

The purpose of this thesis is to enrich the biological models of novelty-generation algorithms. Biology achieves its novelty generation by using huge numbers of very complicated systems, interacting over huge time scales. And to help it run this system, biology is able to use the massively parallel computer of the physical universe, and consume (at least) one earth-sized planet's worth of its resources. In light of these observations, we can reasonably conclude that novelty-generation algorithms with realistic biological models will demand a lot of computer power to run.

There is a tradeoff to be made here. It is acceptable to have algorithms that take a long time to run, provided this long run-time is giving us something useful. GraphMol's graph distance process is computationally intensive, but it is useful because it provides a world in which we can embody the `next` function of the copying process. This embodiment allows interesting types of mutation function to be represented and evolved. This is an example of *embodied evolution*, which is one version of meta-evolution.

In the literature, I have not been able to find any other novelty-generation algorithms in which it is possible to embody the `next` function of the copying process. The Stringmol artificial chemistry comes closest, embodying the `start`

and `at-end` functions (section 5.2.2). Stringmol runs more quickly than GraphMol, but is not able to embody the `next` function, as it lacks the ideas of *space* and *distance* — the parts of GraphMol that slow it down.

It may be that embodying the `next` function requires an implementation of space and distance in the world. If this is the case, then GraphMol's high computational cost is necessary to investigate the effect of embodying the `next` function on the evolution of copying mechanisms. On the other hand, it may be that there is a different way of embodying the `next` function, that does not require high computational costs.[2] In this case, GraphMol will be useful as an early example of embodiment, and as a reference system to compare against (to show that any new system could outperform GraphMol).

9.1.2 Changing the physics

The *physics* of GraphMol is the collection of primitive units that make up the GraphMol world, and the processes by which they interact (section 5.1.1). Some parts of GraphMol's physics are fundamental to GraphMol, such as the concepts of graph nodes and edges. It would not make sense to change these aspects of GraphMol (meaning that if we did change them, then we would not call the resulting system 'GraphMol').

But there are some parts of GraphMol's physics that it would be possible to change in the future. Changing different parts of GraphMol's physics can affect the *embodiment* of different phenomena within GraphMol systems, and it can affect the *performance* of GraphMol worlds.

9.1.2.1 Embodiment

We must have *graph edges* in GraphMol, but we could change what types of edge existed, and how these edges can interact with GraphMol mechanisms.

In the current version of GraphMol, the `next` function of the copy operation is embodied through changing *bind edges*. Binds can form and break, and the timings of these can be controlled to a large extent by evolving GraphMol chemicals. So we have a part of the physics (bind edges) that can change (binding sites bind and unbind), and the effects of these changes can be controlled by evolving mechanisms (`show` and `hide` binding sites, and junk regions). This allows a phenomenon (the `next` function) to be embodied within this world. Therefore, we should be able to

[2] I do not believe this will be the case. Even if the `next` function could be embodied without space and distance, I believe there would be a different high cost necessary, replacing space and distance. I do not believe it is possible to abstract away all of biology's huge computational costs. But this is just my opinion — I could be wrong.

embody other phenomena within the GraphMol world by allowing other parts of GraphMol's physics to change.

We have seen one example of this already (section 6.2.3.4), with the chaperone chemical. The chaperone chemical changes the physics of *fold edges*.

- Without chaperones, fold edges are crisp entities that are unable to change. When a GraphMol chemical is created, it is folded by a crisp process that creates fold edges in the appropriate places. These fold edges are static and cannot change after their creation.

- With chaperones, fold edges become changeable by GraphMol mechanisms. Fold edges are still created with GraphMol chemicals, but they are no longer static. They can be re-positioned, created and destroyed by other GraphMol processes, and this changing of fold edges can be programmed into GraphMol chemicals, so can (in principle) be evolved.

Changing the physics of fold edges changes the embodiment of GraphMol's folding process. By changing the low-level physics of a system, we can change whether a phenomenon of the system (folding) is implemented crisply or is embodied. This thesis has not investigated the effects of embodying GraphMol's folding process, but this would be interesting for future work.

A further example for future work is GraphMol's *program edges*. When a chemical is created, its program edges are created crisply between its atoms, and they cannot change during the lifetime of the chemical. But if we allowed program edges to be created and destroyed by new GraphMol processes, then we would be able to implement GraphMol chemicals that could cut other chemicals apart and stick them together. This would open up a wide range of different biological processes that could be implemented in GraphMol, for example, recombination (section 4.6), which could be used to implement many of the biological mechanisms described in chapter 4.

9.1.2.2 Performance

As well as changing the types of embodiment displayed in GraphMol, we could change GraphMol's physics to tune its performance, in terms of novelty-generation and run-time.

There are parameters in GraphMol's mixing process (section 6.3.1.1), its graph distance process (section 6.3.1.2) and its computer programs (section 6.3.2.3). For the experiments performed in this thesis, these parameters have been chosen by a combination of simple experiments and mathematical calculations based on the copier mechanism. This has allowed us to investigate the copier mechanism, but

future work could gain a deeper understanding of GraphMol's possibilities by performing large-scale experiments on the effects of different combinations of parameter settings on GraphMol's novelty-generation capabilities. This understanding could help design GraphMol worlds in which different mechanisms, with different requirements, could be built.

Detailed experiments into parameter values have not been performed in this thesis, because one of the major driving forces in GraphMol's design was to create a system where precise parameter tuning is not necessary. The idea is that embodied mechanisms should be able to adapt themselves to suit the precise details of their world. Junk performs this function in the GraphMol world. The experiment in chapter 7 shows that GraphMol mechanisms with different levels of junk are affected differently by the parameters of the GraphMol physics, while still retaining the ability to perform their function over a range of junk levels. This gives a way in which evolving mechanisms can change the effect that the world has on their operation, allowing them to adapt themselves to different worlds and not require precise parameter tuning.

In spite of the fact that GraphMol's physics parameters do not *need* tuning for its mechanisms to function, different parameter settings can make the GraphMol code *run faster or slower* (section 6.3.2.3). The execution speed of the GraphMol code does not affect the embodiment of phenomena within GraphMol worlds, nor does it affect the novelty-generation capabilities of GraphMol in principle. But it does affect our ability to observe GraphMol's properties and perform experiments. If the code takes too long to run, then we are less able to perform detailed experiments and measure GraphMol's novelty-generation capabilities. Thus future parameter tuning may be useful to improve the run-time of GraphMol code.

In the future, when creating different versions of GraphMol, it will be useful to tune GraphMol's embodiment and performance together. Different physics choices affecting GraphMol's embodiment capabilities will have impacts on its performance. Systems with more embodiment will tend to have better novelty-generation performance but worse run-time performance. And different systems will have different numerical parameters that can be tuned to affect performance in different ways.

9.1.2.3 Growth laws

Most artificial chemistries do not consider their definition of *space*, and use a very simple implementation of an aspatial space (section 6.3.1.1). GraphMol, on the other hand, uses a more sophisticated version of aspatial space (algorithm 4), including a *binding propensity equation* (equation 6.5) that embodies a notion of *volume* in the space, and specifies the probability of two given chemicals being *close enough* in the space that they can bind and possibly react. This version of

space is not specific to GraphMol: it is used by the Stringmol artificial chemistry, and could be very easily used by any other aspatial artificial chemistry.

An important property of replicating chemicals is their *growth law* (section 7.4.1). In artificial chemistries with trivial definitions of space, the only way to control the growth law of a replicating chemical is to change its reaction equations (section 7.4.1.2). This is a drastic thing to do, as it means re-designing the mechanics of how the chemical replicates. It also means that certain replication mechanisms are forced to have certain growth laws, e.g. sexually-reproducing chemicals are forced to have hyperbolic growth. But with a more sophisticated definition of space, we could change this.

GraphMol's binding propensity equation could be changed to give different growth laws. We could either change the current propensity equation for a different one, or we could have a separate propensity equation for each chemical species: somehow deriving the propensity equation from the chemical. We could look for propensity equations that occur in different types of space in biology (e.g. well-mixed primordial soups, 2-d pyrite surfaces, the crowded cytoplasm inside a eukaryotic cell, etc. ...), or we could eschew biological realism and devise propensity equations purely for their computational properties. Using different propensity equations would decouple the reaction equations from their growth law, and allow these two different properties of replicating chemicals to be designed independently. I have not seen this done in artificial chemistries, and it seems to me a very interesting avenue for research.

9.1.3 Changing the chemistry

The *chemistry* of GraphMol is the collection of functions available to its imperative computer programs (section 6.1.2), and the ways in which these functions can affect GraphMol chemicals.

The current version of GraphMol has an intentionally simple chemistry, containing only the functions needed to implement the copier chemical and embody the `next` function of the copying process:

1. Binding sites
2. `Show` function
3. `Hide` function
4. `Stop` function

A simple chemistry is useful for this thesis as it allows us to define GraphMol more easily and focus on the copier chemical, without being distracted by different possible chemistries. But GraphMol's chemistry is designed to be extendable. It is

trivial to add functions that do not require a parameter. GraphMol's folding process (section 6.2.3) makes it possible to add functions that take a single parameter. And it is easy to see how the folding process could be extended to allow functions with multiple parameters (the closest fold site refers to parameter 1, the second closest fold site refers to parameter 2, etc.). In the future, if we want to implement different mechanisms within GraphMol, then we may need to add new functions. If we change GraphMol's physics to allow more processes to be embodied, then we may need to add new functions to allow GraphMol chemicals to interact with these embodied processes.

The chaperone chemical (section 6.2.3.4) provides an example of this. The chaperone chemical changes GraphMol's physics to embody the folding process, by allowing fold edges to be created and destroyed in GraphMol worlds, but this requires two new functions to be added to GraphMol's chemistry. The *unfold* function destroys the fold edges connected to a binding site on another chemical, and the *hide bound target* function hides a binding site on another chemical. These new functions introduce two different types of capability to GraphMol programs: they allow programs to act on fold edges; and they allow one chemical to act on another chemical. These new capabilities allow the chaperone chemical to be implemented and the folding process to be embodied within the GraphMol world.

As well as introducing new functions into GraphMol's chemistry, we could modify existing functions. For example, GraphMol's binding model is more sophisticated than the binding models of many artificial chemistries (section 5.1.1.5), but it could be further improved. It uses a crisp matching function to decide if two binding sites are complementary. This function could be made richer. For a discussion on measuring the richness of binding models, and the consequences of this for novelty-generation, see [17].

9.1.4 Changing the biology

The *biology* of GraphMol is the collection of mechanisms that are implemented within the GraphMol world. The current version of GraphMol has only one mechanism: *copying*, implemented as two chemicals: the GraphMol *copier chemical* and the GraphMol *DNA chemical*. The copier chemical can evolve, but only into different types of copier chemical (including broken copiers). Copying is the only mechanism that has so far been implemented in GraphMol.

The types of mechanism that can be implemented in GraphMol depend on GraphMol's physics and chemistry. By changing the physics and chemistry of GraphMol, we allow different types of biology to be programmed in GraphMol, and to evolve during GraphMol runs. In terms of embodiment, this means that by changing the *world*, we change the types of *phenomena* that can be embodied within that world, and we change the *mechanisms* by which they can be embodied.

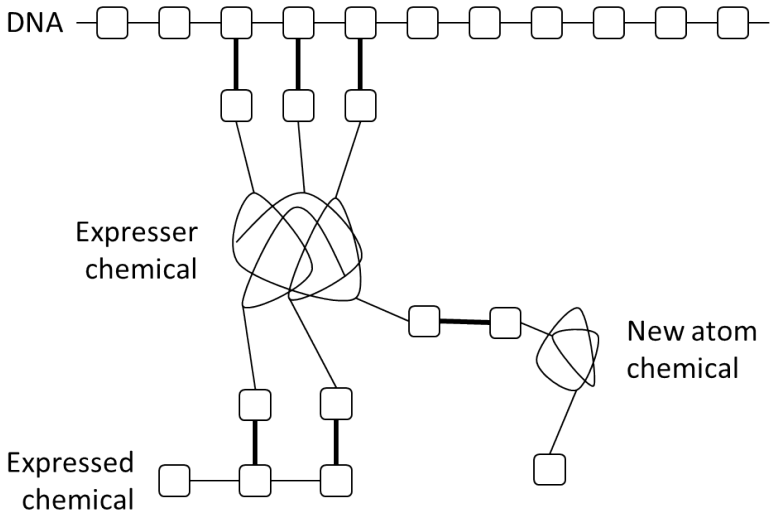


Figure 9.1: One possible version of the *expresser* chemical, where the expresser contains the genotype-phenotype map within it.

9.1.4.1 More processes: Expression

One possible mechanism that would be useful in the future is an *expresser* chemical. Figure 9.1 shows how this chemical would read the DNA three bases at a time and translate each DNA triplet into an atom, expressing the GraphMol chemical coded for on the DNA. This would make the experiments in chapter 8 more biological. Biological systems have DNA chemicals, DNA-copying chemicals and DNA-expressing chemicals (as described in section 4.2). Chapter 8 has DNA chemicals and DNA-copying chemicals, but is missing DNA-expression (this is performed crisply).

There are different ways in which expression could be implemented. Figure 9.1 shows one way, while figure 9.2 shows an alternative. The difference between these two versions is in where the information resides about how to translate triplets of DNA into GraphMol atoms.

1. In figure 9.1, the expresser chemical binds to three DNA bases, and using this information, decides which atom should be added to the expressed chemical. It then `shows` an appropriate binding site, so that a *new atom chemical* will bind to it, and join a new atom on to the expressed chemical. After the new atom has been joined, the expresser moves on to the next triplet of DNA bases.

2. In figure 9.2, we do not use a single chemical to do the translation from 3 DNA bases to 1 GraphMol atom. There are multiple *tRNA* chemicals that

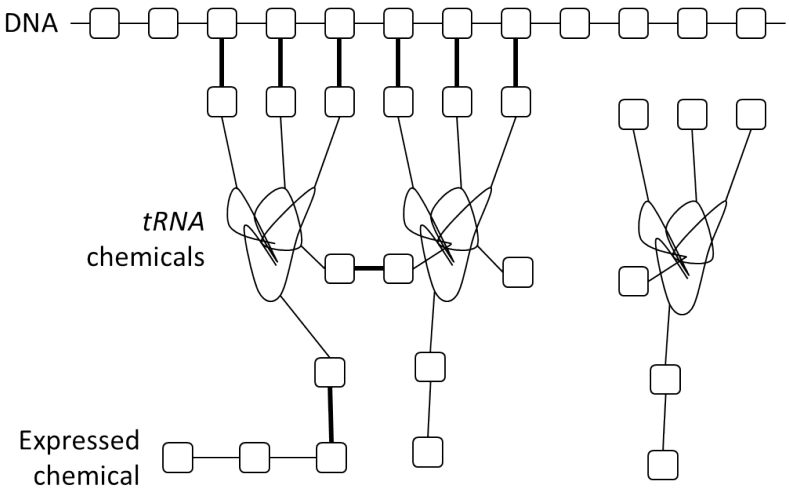


Figure 9.2: A different version of the *expresser* chemical (to figure 9.1), where the genotype-phenotype map is distributed throughout many *tRNA* chemicals.

each specify one mapping of three DNA bases to one GraphMol atom. For each triplet, the appropriate tRNA binds, and when all three of its sites are bound to DNA bases, it communicates with the previous tRNA to attach its atom to the expressed chemical and remove the previous tRNA.

There are many different ways of implementing *expression* in modified versions of GraphMol. The two examples above are not complete designs. They both have details that would need working out, and they both require changes to GraphMol's physics and chemistry to allow the formation (and possibly breaking) of program edges. But they illustrate two very different ways in which one could go about embodying expression within GraphMol. This shows the flexibility of GraphMol's design, and the range of choices one has when thinking in terms of embodiment.

1. Version 1 builds on the GraphMol copier chemical, using its *next* operation to walk along the DNA and along the expressed chemical. But rather than reading single DNA bases, the expresser chemical reads triplets and recruits *new atom chemicals* to build up the expressed chemical.

2. Version 2 is a distributed version. The logic of mapping a DNA triplet into a GraphMol atom might be too complex to implement within a single GraphMol chemical (or might require radical changes to GraphMol's chemistry). Distributing this mapping between different chemicals allows us to use GraphMol's binding function to do some of the computation.

Biological systems use a combination of these two versions (see section 4.2.2, and [15, p.386]). There is a biological *ribosome* chemical that moves incrementally

along the DNA, but it uses a pool of *tRNA* chemicals to implement the translation mapping. In future work, it would be interesting to investigate different mechanisms for embodying expression.

The benefit of embodying expression would be to give evolution control over the expression phenomenon. In the language of Evolutionary Algorithms, *expression* is called the *genotype-phenotype map*. Evo-devo algorithms (section 2.3.3) first introduced the idea of mapping from a genome to a phenome, but the mappings they use are *crisp* and so cannot be evolved. GraphMol allows us the possibility (in the future) of *embodying* this mapping within the same world as the mechanisms produced by the mapping. This could open up some interesting evolutionary possibilities for evolving the problem representations used by novelty-generation algorithms.

9.1.4.2 More embodiment: Chaperones

The expresser chemical described above shows how GraphMol's biology could be extended *horizontally*, to implement different mechanisms. But GraphMol's biology can also be extended *vertically*, allowing GraphMol to implement the same processes in more embodied ways. Embodying more processes should make the overall system more evolvable, but at the cost of increased run-time.

In the experiments presented in this thesis, GraphMol's folding process was implemented crisply. But the running example of GraphMol *chaperone* chemicals (section 6.2.3.4) shows how this process could be embodied within a GraphMol world. Future work could see what effect this would have on the evolvability of different GraphMol mechanisms.

Embodying the folding process allows evolution to change the folding process. This could help evolution in two different ways (as far as I can see):

1. Evolution could change a chaperone chemical, changing the way in which folding works, allowing a new type of fold to happen. This new type of fold could allow other GraphMol mechanisms to mutate more easily, using the new fold to smooth their mutation landscapes.

2. Evolution could create a single, mutant chaperone chemical, that does not survive for long or become fixed in the population. But this mutant chaperone could mis-fold another chemical in such a way as to cause a large-scale mutation that had an evolutionary effect that did become fixed genetically.

It would be interesting to see if either of the two options above actually happen during GraphMol runs. Their likelihood would depend on what GraphMol mechanisms the chaperones were folding, so it might be worth performing these experiments when we have a range of different GraphMol mechanisms evolving.

9.1.4.3 Different embodiments: Copying

Implementing more mechanisms within GraphMol can change its processes in three different ways:

1. Allowing it to implement *more processes* (e.g. expression).

2. Allowing it to implement the same processes in *more embodied ways* (e.g. chaperones).

3. Allowing it to implement the same processes in *different ways*.

Section 5.2 describes how the copying process can be broken down into four functions: `start`, `at-end`, `char-copy` and `next`. GraphMol's embodied copying mechanism is one way of embodying the copying process, by implementing these functions in terms of a combination of crisp, stochastic and embodied processes (as described in section 6.4.2). The current version of GraphMol focuses on embodying the `next` function, but future work could look at the other functions, to see how they could be embodied within GraphMol.

Another part of the copying process is the embodiment of the new string being produced. The Stringmol artificial chemistry embodies the new string, by appending it onto the end of the copier chemical. The current version of GraphMol does not embody the new string. A crisp process operating in a separate world monitors the GraphMol copier's *magic* binding sites, recording which DNA bases they bind to. Future work could look at representing the new chemical explicitly within the GraphMol world. A starting point is the expresser chemical described above, which explicitly represents its expressed chemical. Embodying the new string produced by the copying process might be an incremental way to start building the expresser chemical.

9.2 Moving between levels

Embodiment is inherently hierarchical. Section 3.2.3 describes how embodied processes can be nested within embodied processes, with different parts of a system at different levels implemented crisply, stochastically or embodied. Changing the embodiment of a system is one way of addressing some of the limitations of the system. The previous section (and most of this thesis) discusses different ways of making a system more evolvable by making it *more embodied*, at the expense of run-time. But conversely if we care very much about run-time, and not as much about evolvability, then we could take a slow, evolvable, embodied system and make it less embodied to improve its run-time, at the expense of evolvability.

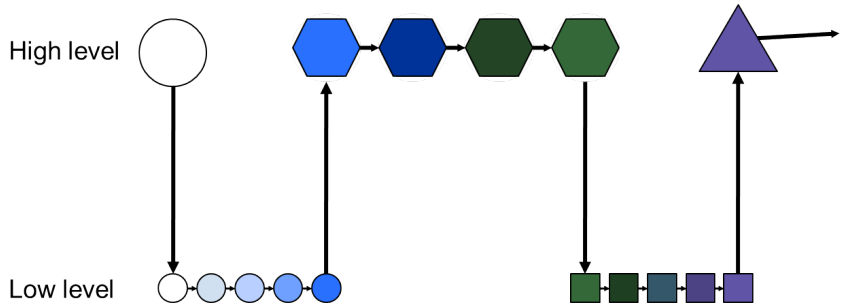


Figure 9.3: A multi-level evolutionary process. Evolving on two different levels allows the system to reach parts of the search space that would be impossible with either level alone.

This could be useful in a (future) multi-level evolutionary system. A common problem with current evolutionary algorithms is that they can only generate a certain amount of novelty before they become stuck and can evolve no further. They reach a local optimum in the search landscape generated by their fitness function and problem representation. But moving between levels provides a way of *changing the problem representation*, allowing the system to evolve further and generate more novelty. Figure 9.3 shows a schematic of this idea:

1. A hand-designed organism (solution to a problem) is implemented in a high level system that is not very embodied but is suited to designing organisms by hand.

2. This high-level organism is transformed into a low-level, embodied system, where it can evolve until it becomes stuck in a local optimum.

3. The evolved organism is transformed back into the high-level language. The evolved organism looks very different from the hand-designed one, and it can be evolved on the high level.

4. After the high-level evolution becomes stuck, the organism is again transformed into a low-level, embodied system. But because of the high-level evolution, the system is now in a different place in its low-level search landscape (or perhaps, in a completely different low-level landscape). Thus it can continue evolving, and the system can continue to generate novelty.

Clearly, there are many potential issues with this multi-level evolutionary system. It is only an idea at the moment, and much future work would be needed to investigate and understand the details. We would need to produce implementations of possible high levels and low levels, along with ways of mapping between

them. We would need to perform experiments on different systems, to see whether two levels of evolution can actually help systems to evolve in new ways, or whether the run-time is better spent simply running the low level evolutionary system for longer, and/or spending more time varying its parameters. Also, it might be the case that the high level evolution and low level evolution go into a cycle, where the high level moves the system to a new place, but the low level brings it back again. i.e. the high level evolution fails to move the system out of the basin of attraction of the low level local optimum. These types of issue would need careful investigation.

The process of moving between levels could be done by hand, but it would be more useful if it could be automated. The general problem of moving a system between arbitrary levels of description is very difficult. But artificial chemistries provide us with a general model of a wide range of systems, as discussed in section 5.1. Artificial chemistries can represent systems at different levels (section 5.1.3), and so they provide an ideal set of candidate high level and low level systems that we can use to start looking at moving between levels. I have designed an algorithm to aid in the process of moving down from a high level artificial chemistry to a low level one. This is described in appendix B, along with some ideas about how to move upwards from low levels to higher levels.

Moving upwards is all about making approximations of the lower level and removing information to describe the low-level system in a more concise way. Making these approximations will necessarily lose properties from the low-level system. The problem is deciding which types of property to retain and which to lose. Moving downwards is all about putting in low-level mechanisms to replace the ones that have been removed. These low-level mechanisms must remain subject to the constraints implied by the high-level system (which might not be obvious to a cursory inspection of the high-level system). The problem is determining the constraints implied by the high-level system, and finding low-level systems that can implement them.

This multi-level evolutionary idea is an example of why it is useful to think about novelty-generation systems in terms of embodiment. Levels of abstraction are a common theme in computer science, and computer programmers regularly move ideas between different levels *at the design stage* when deciding how to structure their programs. But embodiment allows us to think about how to move *implementations* of phenomena between levels *at run-time*. The idea of embodiment gives us a language in which we can talk about moving between levels. We can think about properties of different worlds, and analyse systems to determine which processes they embody to what levels. We can also generate new systems at different levels by changing their worlds to embody different processes to greater or lesser degrees.

9.3 Designing embodiment

This section discusses what GraphMol has taught us about embodiment.

For any phenomenon that we might want to embody, there are many different mechanisms that could potentially embody it, within many different worlds. Sections 3.2.3 and 3.3 describe a process that we can go through to embody a given phenomenon within the virtual world of a computer program:

1. Firstly, the phenomenon must be decomposed into intermediate stages. This can be done in as much or as little detail as required, and is a hierarchical process (i.e. intermediate stages can also be decomposed).

2. The lowest-level stages of the decomposition constitute the *world* in which the phenomenon is embodied.

3. The intermediate stages are the *mechanisms* by which elements of the world combine to embody the overall phenomenon.

Different decompositions of the same phenomenon can embody the same phenomenon within different worlds, and can embody the same phenomenon using different mechanisms.

Section 5.2 describes one way of decomposing the phenomenon of *copying a string*. GraphMol decomposes this further, in its embodiment of the copying phenomenon. This is shown in figure 9.4, and is described in detail in section 6.4.2. Stringmol is an artificial chemistry that embodies copying in a different way to GraphMol. Stringmol's decomposition can be seen in figure 9.5, and is described in detail in section 5.2.2. These two figures highlight the fact that the novelty-generation capabilities of a system stem from the system's embodiment, which is a hierarchical nesting of different worlds within each other. Different decompositions of the same phenomenon lead to different hierarchies of worlds.

- Stringmol's novelty-generation comes from its embodied `start` and `at-end` functions (section 5.2.2.1). These are embodied within the Stringmol world of chemicals as strings, interacting with each other via the Smith-Waterman algorithm, which is a computational version of the biological process of binding.

- GraphMol's novelty-generation comes from its embodied `next` function (section 6.4.3). This is embodied within the GraphMol world of chemicals as graphs that bind and run programs based on graph distances. This is a computational version of how biological machines operate on each other (for the biology, see section 4.2).

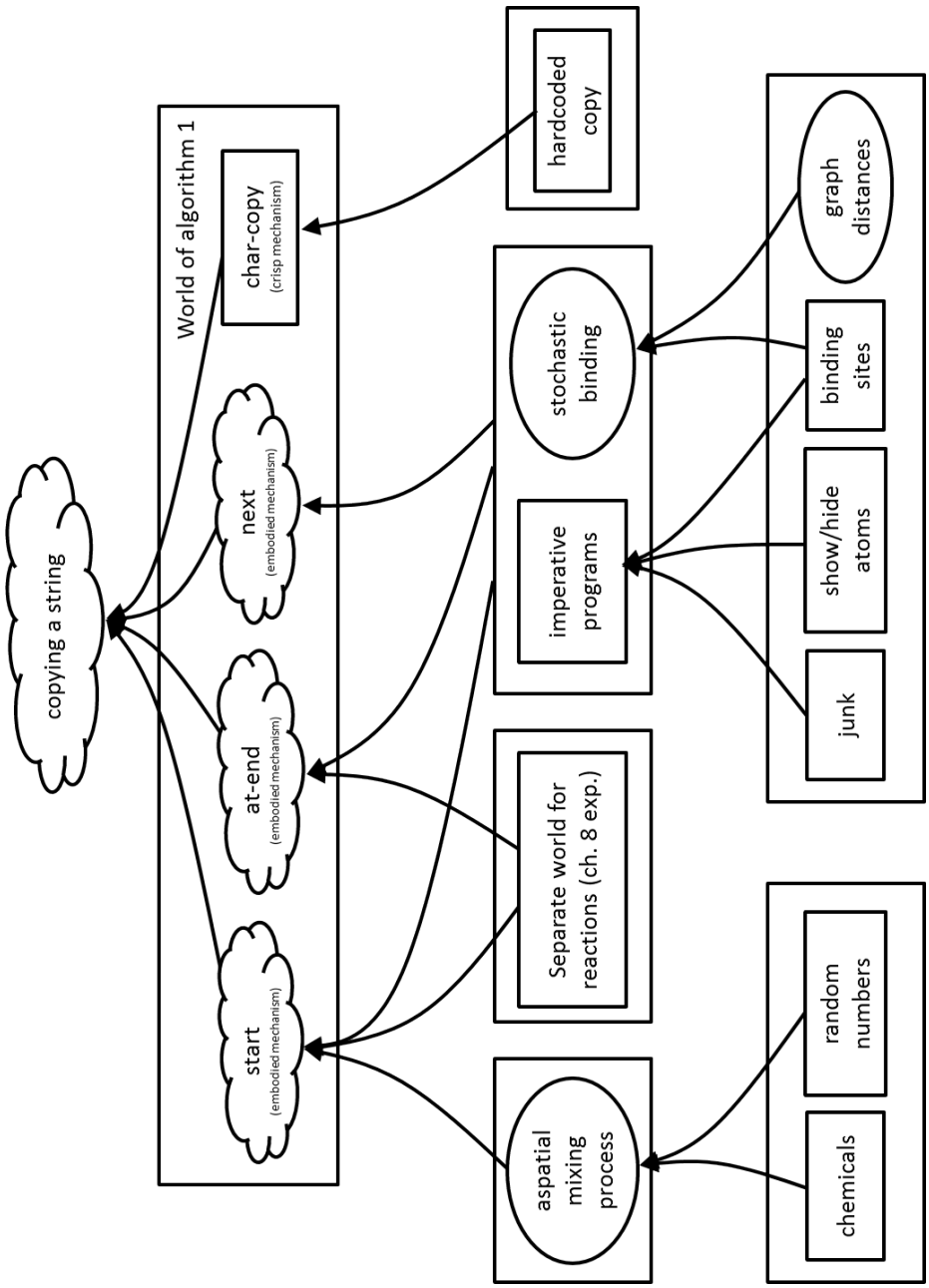


Figure 9.4: The mechanisms and worlds used by GraphMol to embody the phenomenon of copying a string. Compare with figure 9.5.

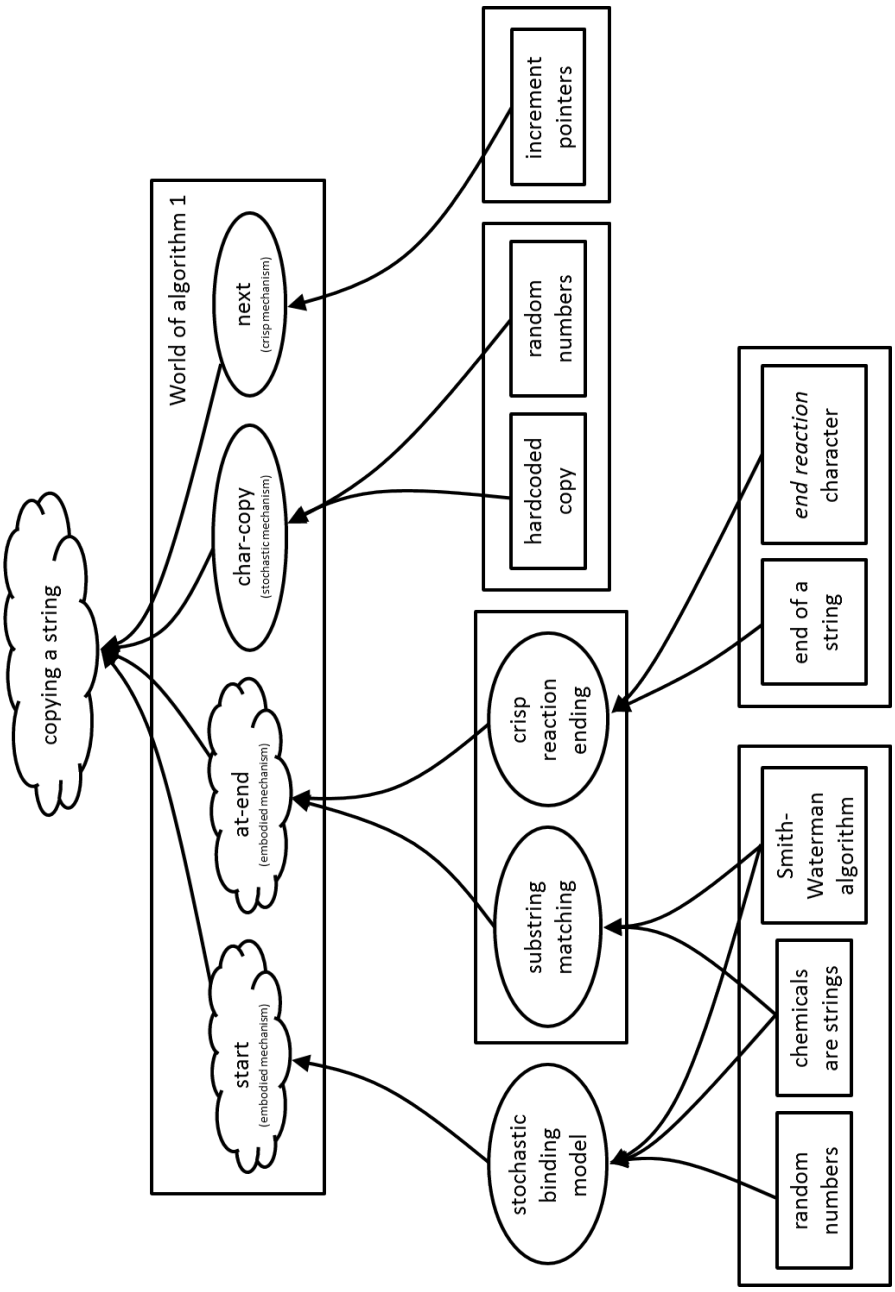


Figure 9.5: The mechanisms and worlds used by the Stringmol artificial chemistry to embody the phenomenon of copying a string. Compare with figure 9.4. (This is a duplicate of figure 5.2, reproduced here for convenience.)

In both cases, we have changes to the biological model of artificial chemistries. The biological model informs the computational model (section 2.2), which defines the different *worlds* that need to be implemented to produce a running computer program. (The worlds are the enclosing rectangles in figures 9.4 and 9.5.) The influence of the biological model, reflected in these worlds, filters up through the layers of embodiment. The choices made when designing the biological model manifest as the novelty-generation that we observe in the phenomenon of interest (the cloud at the top of the embodiment hierarchy).

Different properties of novelty-generation systems can be seen in the trees representing their embodiment hierarchies:

1. In parts of the embodiment hierarchy that are more embodied (have deeper trees), we see greater novelty-generation displayed in the running system (Stringmol's `start` and `at-end` function; GraphMol's `next` function).

2. Conversely, in parts of the hierarchy that are less embodied (have shallower trees), we do not see these aspects of the system playing key roles in the system's examples novelty-generation (Stringmol's `char-copy` and `next` functions; GraphMol's `char-copy` function).

3. Finally, in parts of the hierarchy that are of mixed depth (some deep subtrees, some shallow subtrees), we see some influence on novelty-generation, but not as much as that shown by the more embodied parts (GraphMol's `start` and `at-end` functions are embodied, but are not as influential as its `next` function).

In summary, the *physics* and *chemistry* of an embodied system define the lowest-level components and interactions. These are the parts that we implement explicitly to create a virtual world. The low-level components define the level of abstraction that the system will start from. If this level is too low, then the system will take a long time to run, but if it is too high then the system will not be able to embody the desired phenomena. The *biology* of an embodied system is the collection of mechanisms that can be implemented within the world. These mechanisms embody the phenomena of interest, and evolve as the system runs. The world (physics and chemistry) stays the same; the mechanisms (biology) are created, mutated and destroyed; and the phenomena (that we are interested in) evolve over time. In order for the phenomena to evolve in novel and interesting ways, the world must be rich enough that the mechanisms are able to mutate and change, while still embodying the same phenomena. We need mechanisms that can embody the same phenomena in different ways, some of which are novel and not predicted beforehand by the algorithm designers. And we need the mechanisms to be able to mutate between these different embodiments, as the novelty-generation algorithm runs.

These ideas could be used to design future novelty-generation systems. Thinking explicitly about the embodiment hierarchy when designing a system could improve its novelty-generation capabilities. I am well aware that the above example only uses two data points (GraphMol and Stringmol), and Stringmol's hierarchy was written down only after its novelty-generation properties were known. So it may be that in drawing these diagrams, I have been biassed to highlight those parts of the systems that I know influence their novelty-generation capabilities, but I do not believe this to be the case. Nevertheless, future work should investigate the embodiment of other systems, ideally before they are run, to see how effective this is at predicting the novelty-generation capabilities of a system, and predicting which parts of a system will most influence its novelty-generation. At the moment, this enterprise is hampered by the fact that existing novelty-generation systems are not very embodied (and not very successful at generating novelty). GraphMol and Stringmol are the only examples I know of that are embodied in a non-trivial way. In the future, when more embodied systems have been designed, this problem will disappear.

9.4 Thoughts on meta-evolution

This section discusses what embodiment has taught us about meta-evolution.

The reason for investigating embodiment in this thesis is that embodiment allows evolution to be an *emergent property* of a collection of interacting organisms (section 2.6.4.2). This is one type of *meta-evolution* (section 2.6.5) that is a natural extension to the novelty-generation capabilities of previous algorithms (section 2.4.3). It allows us to improve these algorithms in a principled and incremental way.

Emergent evolution is a large and difficult research area, and the purpose of this thesis is *not* to demonstrate a computer program displaying emergent evolution. Rather, my contribution is the identification of emergent evolution as a specific type of meta-evolution (this had not been done before), including a classification of the different types of meta-evolution (section 2.6.5), and an investigation of how biological systems benefit from their emergent evolution in the physical world (chapter 4). To investigate emergent evolution from a theoretical viewpoint, I used the ideas of *embodiment*. This included presenting a new definition of embodiment (section 3.2.3), and investigating ways in which the biological benefits of embodiment in the physical world can be transferred across to virtual worlds (chapter 5). In order to relate these ideas to as broad a range of computational systems as possible, I used *artificial chemistries* as a general model for virtual worlds.

In my view, the important research areas for emergent evolution are:

1. Different ways of representing organisms (representation of *mechanisms*, and representation of *DNA* encoding the mechanisms).
2. Different ways to encode an organism for copying (transforming *DNA* into *mechanisms*).
3. Different ways to copy (replicating DNA using different mechanisms).
4. Different ways to interact with an environment and obtain energy (and use energy to do things).
   (a) Different types of environment
   (b) Different ways to perceive / interact with an environment
   (c) Whether *environment* can include other organisms or not.

Chapter 2 describes how other authors have addressed point 1: *representing organisms*. This review describes how representations in novelty-generation algorithms have improved, becoming more embodied. Stringmol (section 5.2.2) and GraphMol are two different examples of representing organisms in an embodied way. Different representations have different levels of evolvability, and so future work could look at different embodied representations, to expand the number of embodied representations beyond the two that currently exist.

Point 2 is about *encoding organisms*, for example on DNA. Current biological organisms all use DNA (section 4.2), but this has not always been the case (see *RNA world* and other discussions about the origin of life [67, chapter 3]). And in our virtual worlds, we do not need to be constrained by how biology solves its problems: we can consider different ways of encoding organisms. Most artificial chemistries use a direct (RNA-world) encoding method, where chemicals are simply copied verbatim. Stringmol does this, but GraphMol introduces DNA as an intermediate encoding method (as current biology does). Future work could investigate different encoding methods and evaluate their effects on novelty generation.

GraphMol and its embodied copying mechanism directly target point 3: *Different ways to copy*. GraphMol shows one copying mechanism, that can (potentially) change itself into other mechanisms. But its evolution is very limited. All of GraphMol's evolved copying mechanisms operate in the same way, just with different parameters. In novelty-generation terms, this is an improvement over previous algorithms that evolve their parameters explicitly (sections 2.6.1.1 and 2.6.1.2), because in GraphMol the parameters are encoded (and evolved) *implicitly*, by being embodied within a world. Future work is needed to make this world more *evolvable*, so that as well as varying parameters, embodied mechanisms can also

evolve the mechanisms themselves. Stringmol is a promising step in this direction, as it shows one copying mechanism evolving into a very different type of copying mechanism (section 5.2.2.1). This small success suggests that these research goals could be achieved with future work.

Obtaining *energy* from an *environment* (point 4) is very important. This thesis (and GraphMol) is heavily focussed on the environment, presenting it as the *world* in which mechanisms are embodied. This also has the (desirable) effect of dissolving the separation between *system* and *environment*, seeing both as different parts of the same *world* (section 3.2.3). In this thesis, I have investigated the effect that the world has on the *types of mechanism* that can be embodied. But the world also affects the way(s) in which organisms can obtain *energy* (and hence do work), and the ways in which organisms can interact.

Stringmol has a more sophisticated energy model than any previous artificial chemistries (section 5.1.1.4), so I have used this in GraphMol. Much future work could be done to investigate different ways in which organisms can obtain energy from an external environment, and distribute it among their mechanisms. For example, we could have one energy bucket per cell full of mechanisms (as in Stringmol), or each mechanism could have its own energy reserve.

One way to create an interesting environment is to consider other organisms as part of an organism's environment, by allowing organisms to interact with each other. All artificial chemistries (including GraphMol) have a microcosm of this, in terms of chemical interactions. Treating an individual chemical as an *organism*, each chemical has all the other chemicals in its environment, because it can react with them. But if an *organism* in an artificial chemistry is a cell full of reacting chemicals, then we need a way in which cells can interact with each other. In terms of artificial chemistries, this introduces many complications such as: how to represent the membranes surrounding cells — whether these are chemicals or a different class of object; permeability of membranes — which chemicals travel across which membranes and when; defining the space in which cells exist — whether or not it is the same as the space inside the cells. These are all interesting avenues for future work.

In terms of embodiment, we could see cells as another level in the embodiment hierarchy. Chemical mechanisms form the *world* in which we could embody cells: from the interaction of different chemicals, we could obtain partially-permeable membranes enclosing volumes of space. Something that seems to introduce many complications can be understood more easily by using embodiment. By describing a system as a hierarchy of worlds, we obtain a flexible method by which we can increase the complexity of our systems, while still being able to describe and understand them. This is absolutely necessary if we are to build virtual worlds capable of generating increasing novelty.

9.5 Final words: The far future

The contribution of this thesis is applying the ideas of embodiment to the problem of computational novelty-generation inspired by biology, via the idea of meta-evolution and the technology of artificial chemistries. I have not solved the problem, but I have defined the problem and described it in more detail than previous authors. I have highlighted some of the main branches comprising this tree of knowledge, and have begun to tentatively peer along them.

This is only the start of a long journey. Biological novelty-generation is incredibly complex, but embodiment gives us a language that might help with handling this complexity and implementing it within virtual worlds. Embodiment is no more than (and no less than) the well-known ideas of abstraction, applied to the problem of designing biologically-inspired novelty-generation algorithms.

It is at present unknown (and is perhaps unknowable) whether or not it is possible to replicate the complexity of the physical world within virtual worlds, including the amazing capacity of the physical world to generate novelty. And related to this, it is unknown whether or not the physical world's capacity for generating novelty is truly unbounded. But whatever the answers to these questions, I believe it is definitely beneficial to try for computational novelty-generation. If it is possible to obtain unbounded novelty-generation within a virtual world, then we definitely need to try, otherwise we will never realise this possibility. And if it is not possible, then trying is still useful. By attempting this grand challenge, we create new and exciting virtual worlds in which previously unseen organisms exist. By peering into these new worlds, we learn more and more about computation and life as it could be, which can only be a good thing for the future of technology and knowledge.

Appendix A

The binding probability function

As described in section 6.4.2.3, the GraphMol binding probability function has the form:

$$\text{bind\_probability}(d) = \left(\frac{k}{d}\right)^{\alpha} \tag{A.1}$$

where d is the distance between the two binding sites. This has two parameters: k and α.

We can choose these parameters to give the GraphMol copier chemical the behaviour we want, by assuming:

1. The copier is copying a DNA chemical requiring l steps.

2. We want it to make an average of x errors per copy of the DNA.

3. We want the probability of a 'correct' bind to be p_c.

4. The distance of the bind for a 'correct' step is d_c.

5. The distance of the bind for an 'error' step is d_e.

We can use equation A.1 to derive the following values for α and k:

$$\alpha = \frac{\log(x) - \log(l)}{\log(d_c) - \log(d_e)} \tag{A.2}$$

$$k = \exp\left(\log(d_c) + \frac{\log(p_c)}{\alpha}\right) \tag{A.3}$$

A.1 Derivation of α

Given a DNA chemical requiring l steps to copy.

We want to make an average of x errors per copy.

So the ratio of error : correct step is $x : l$.

$$\frac{\text{prob(error)}}{\text{prob(correct)}} = \frac{x}{l} \tag{A.4}$$

$$\frac{\left(\frac{k}{d_e}\right)^{\alpha}}{\left(\frac{k}{d_c}\right)^{\alpha}} = \frac{x}{l} \tag{A.5}$$

$$\left(\frac{d_c}{d_e}\right)^{\alpha} = \frac{x}{l} \tag{A.6}$$

$$\alpha \log\left(\frac{d_c}{d_e}\right) = \log\left(\frac{x}{l}\right) \tag{A.7}$$

$$\alpha(\log(d_c) - \log(d_e)) = \log(x) - \log(l) \tag{A.8}$$

$$\alpha = \frac{\log(x) - \log(l)}{\log(d_c) - \log(d_e)} \tag{A.9}$$

A.2 Derivation of k

$$p_c = \text{prob(correct)} \tag{A.10}$$

$$p_c = \left(\frac{k}{d_c}\right)^{\alpha} \tag{A.11}$$

$$\log(p_c) = \alpha(\log(k) - \log(d_c)) \tag{A.12}$$

$$\log(k) = \log(d_c) + \frac{\log(p_c)}{\alpha} \tag{A.13}$$

$$k = \exp\left(\log(d_c) + \frac{\log(p_c)}{\alpha}\right) \tag{A.14}$$

Appendix B

Moving between levels

This appendix describes an algorithm for aiding in the process of moving down from a high level system to a low level one, as introduced in section 9.2. It does not perform the whole process of moving downwards. Rather, it analyses a high level system and generates a set of *constraints* that any low level system must obey, if it is to implement the high level system. These constraints could be used by humans to help hand-design a low level system, or they could be fed into an automated process that searches for suitable low-level systems or enumerates different possible low-level systems satisfying the constraints. This algorithm is not perfect, but it is a start on the difficult problem of automatically moving between levels. Much future work could be done to improve this algorithm.

This algorithm is described in terms of artificial chemistries, but artificial chemistries are a model for many different types of system. So although this presentation of the algorithm uses terms such as *chemical*, *reaction* and *conservation of mass*, these words do not constrain the algorithm to only operate on systems that are models of physical-world chemistry. The idea of moving between levels is inherently applicable to systems on different levels, using different terminology.

This appendix is based on [73].

B.1 Moving downwards

For systems that are models of the physical world, there is always a lower level of description that the system could be described on (until we reach the level of our understanding of particle physics). Also, for physical-world systems, some information about this lower level is always known (we know that organisms are composed of cells, which are composed of molecules, and so on.)

For artificial systems, however, the implementation of any level is arbitrary (and is often chosen to make the program execute efficiently). So when describing an

artificial system in terms of a lower level, there are arbitrary implementation choices to be made, some of which are constrained by the higher level. Looking for these constraints can give insight and information about the higher level, and resolve some of the seemingly arbitrary design choices for the lower level. These kinds of insight can also be gained about physical-world systems as well as computational ones.

The high-level entities are *symbols*. On the lower level, each of these high level symbols is expressed as a collection of lower-level —it components. For example, the decomposition of hydrogen peroxide into water and oxygen can be written as:

$$2 \text{ hydrogen-peroxide} \rightarrow 2 \text{ water} + \text{oxygen} \tag{B.1}$$

But the same equation can be written in terms of the lower level of atoms, instead of in terms of the higher level of molecules:

$$2 \ H_2O_2 \rightarrow 2 \ H_2O + O_2 \tag{B.2}$$

Here, 'hydrogen-peroxide' is a symbol on the higher level, that is expressed as two 'H' components and two 'O' components on the lower level. Likewise, 'oxygen' is a symbol on the higher level, that is expressed as two 'O' components on the lower level. Note also the constraint: 'oxygen' and 'hydrogen-peroxide' are different symbols on the higher level, but they share common components on the lower level: 'hydrogen-peroxide' contains all the components of 'oxygen', along with some others.

On the high level, information about the system is contained in the reaction equations. On the low level, it is contained in the structure of the chemicals (how their components are arranged). So the task of describing a high-level system on a lower level is about *moving information from reaction equations to chemical structures*.

This movement can be performed by humans looking at reaction equations and diagrams. But as the lists of equations become longer and the number of different symbols increases, the problem becomes harder and more tedious to solve. Also, if evolutionary algorithms are to evolve symbolic systems, then this problem needs to be solved hundreds of times for each generation of the evolutionary algorithm. This is why it is useful to have an algorithm for automatically performing this process.

B.1.1 Conservation of mass

The above reasoning relied on the assumption that *mass* is conserved in the high-level reaction equations: if $\alpha + \beta \rightarrow \gamma$, then all the low-level components making

up α and β are present in γ, and γ contains no new components that have not come from α or β.

This condition is not difficult to fulfill on the high level, as new symbols can be introduced to account for any mass gained or lost in a reaction. For example, if $\alpha + \beta \to \gamma$, but mass is lost ($\gamma$ does not contain all of the components of α and β), then the reaction $\alpha + \beta \to \gamma$ can be replaced by $\alpha + \beta \to \xi + \gamma$, where the symbol ξ does not appear anywhere else in the system. ξ represents the mass that is lost in the reaction. Likewise, if mass is gained in the reaction (γ contains a component that does not come from α or β), then $\alpha + \beta + \zeta \to \gamma$ can be used, where ζ represents the mass gained in the reaction. These two patterns can be applied to any reaction. If they are applied at the same time, they can represent reactions in which some components are lost and some are gained.

Given a high-level system of reaction equations that conserve mass, we can deduce constraints on how the high-level symbols are composed of low-level components. We can also put constraints on the possible masses that the symbols can have. Technically, we deduce a partial order on the masses of the symbols, with constraints of the form: 'χ has more mass than ψ'. We can also use this to work out if a system conserves mass or not, so we do not need to know beforehand. If we encounter a contradiction when building the partial order, then we have proved that the system does not conserve mass. If we can build the partial order with no contradictions, then we have proved that the system does conserve mass.

B.1.2 Multiple meanings

Some high-level reaction equations can have more than one interpretation on the lower level. These can be disambiguated by modifying the reaction equations to include intermediate steps. Different disambiguations lead to different low-level constraints for the same high-level system.

B.1.2.1 3 chemicals or fewer — unambiguous reactions

There are five kinds of reaction equation that have only one interpretation on the lower level: they involve three molecules or fewer.

1. nothing $\to \alpha$ (influx)
2. $\alpha \to$ nothing (outflux, or decay)
3. $\alpha \to \beta$ (isomerisation)
4. $\alpha + \beta \to \gamma$ (composition or association)
5. $\gamma \to \alpha + \beta$ (decomposition or dissociation)

Reaction types (1) and (2) give no information about the lower level (other than saying "α is a symbol that exists"), so are ignored in later analysis.

If we have four or more chemicals participating in a reaction, then this can have more than one interpretation on the lower level. The different permutations are described below.

B.1.2.2 3 → 1 reactions.

Reactions of the form $\alpha + \beta + \gamma \rightarrow \delta$ imply that chemical δ is a composite of chemicals α, β and γ. The ambiguity lies in the order in which α, β and γ combine to form δ. Because the probability of three molecules reacting with each other at the same instant is negligibly small, two of α, β and γ must react first, the other one reacting with the intermediate complex. This implies the existence of an intermediate complex, ξ, (in the lower-level description) that is the combination of the first two components. There are three possibilities for the order in which α, β and γ combine:

$$\alpha + \beta \rightarrow \xi \; ; \quad \xi + \gamma \rightarrow \delta \tag{B.3}$$
$$\alpha + \gamma \rightarrow \xi \; ; \quad \xi + \beta \rightarrow \delta \tag{B.4}$$
$$\beta + \gamma \rightarrow \xi \; ; \quad \xi + \alpha \rightarrow \delta \tag{B.5}$$

If $\alpha + \beta + \gamma \rightarrow \delta$ were the only reaction in the system, then these three disambiguations would be equivalent. But if α, β and γ participate in other reactions, then the order in which they combine to form δ could have implications on the lower level.

B.1.2.3 1 → 3 reactions.

Similarly to the 3 → 1 reactions, reactions of the form $\alpha \rightarrow \beta + \gamma + \delta$ can also have multiple interpretations. The chemical α must be composed of chemicals β, γ and δ, and so it must be composed of their low-level components, held together in a certain structure. It must release one of β, γ or δ first, because the probability of a reaction releasing all three at the same instant is negligibly small. This again implies the existence of an intermediate chemical, ξ, that is the combination of two of β, γ and δ. There are three possibilities:

$$\alpha \rightarrow \beta + \xi \; ; \quad \xi \rightarrow \gamma + \delta \tag{B.6}$$
$$\alpha \rightarrow \gamma + \xi \; ; \quad \xi \rightarrow \beta + \delta \tag{B.7}$$
$$\alpha \rightarrow \delta + \xi \; ; \quad \xi \rightarrow \beta + \gamma \tag{B.8}$$

B.1.2.4 $2 \to 2$ reactions.

Reactions of the form $\alpha + \beta \to \gamma + \delta$ can have multiple interpretations, but these interpretations are of a different kind from those above. The earlier interpretations are about the order in which three chemicals come together to form a complex (or come apart from a complex). The interpretations for $\alpha + \beta \to \gamma + \delta$ reactions concern symbols being transformed into other symbols, which corresponds, on the lower level, to chemicals undergoing isomerisations. There are three possibilities for how this isomerisation can occur:

1. α is an isomer of γ; and β is an isomer of δ.
2. α is an isomer of δ; and β is an isomer of γ.
3. Both α and β contain some components of γ and δ.

Depending on precisely how the lower level will be implemented, point (3) may or may not be possible.

For the purpose of reducing every ambiguous reaction to unambiguous reactions, the reaction $\alpha + \beta \to \gamma + \delta$ can be replaced with the two reactions:

$$\alpha + \beta \to \xi \; ; \quad \xi \to \gamma + \delta \tag{B.9}$$

This again introduces an intermediate complex, ξ. Replacing the equation in this way does not remove the underlying ambiguity. We must make another disambiguation by choosing one of the three cases above.

B.1.2.5 More than 4 chemicals

In the same way that reactions involving four chemicals can be reduced to unambiguous reactions involving three chemicals or fewer, reactions with more than 4 chemicals can be reduced to unambiguous reactions by the repeated application of the above reductions.

B.1.2.6 Disambiguation

The first step in the analysis of a high-level system is to pre-process the reactions, reducing them to unambiguous reactions. This involves making choices about how to decompose ambiguous reactions, as described above. If only one ambiguous reaction needs to be decomposed, then the choice made is somewhat arbitrary. But if multiple choices need to be made, then there is the possibility that choices can affect each other.

Any set of choices will always lead to a valid disambiguation, and every disambiguation can always be reversed (by removing the intermediates) to return to the

same set of ambiguous equations. However, different disambiguations of the same equations can differ in the number of intermediates introduced. If two reactions need to be disambiguated, then this will introduce two new intermediate symbols (one for each reaction). These intermediates are different symbols on the high level, but if there is extra information in the system about the reactants and products of the ambiguous reactions, then it may be possible to relate the intermediates on the lower level, seeing them as isomers of each other (i.e. realising they are composed of the same components). If, however, the equations were disambiguated using different choices, then it might not be possible to relate the intermediates on the lower level. This can also carry over to some of the non-intermediate symbols as well. One disambiguation may make it possible to infer that two non-intermediate symbols are isomers of each other, but a different disambiguation may not make it possible to infer this.

Note that this is not a mistake in the disambiguation process: it is a choice that must be made about how to interpret the high-level equations. If an equation is ambiguous about how one reaction happens, then this ambiguity can carry over to other parts of the system. If application-specific information is available about how ambiguous equations should be disambiguated, then they can be disambiguated by hand before running the analysis. Or if the equations are being generated by a computer program, then this program can be instructed to produce unambiguous equations of the correct form. If it is not known which way the equations would be best disambiguated, then any disambiguation will give a valid representation of the equations. If there is reason to believe that one representation will be better than others, but it is not known which, then all disambiguations can be enumerated. The analysis can be run on all disambiguations and the results compared to see if multiple representations are possible. If the most compact representation is desired (i.e. the representation that sees the greatest number of symbols as isomers of each other), then this can be found by comparing the different representations. The fact that multiple representations are possible via different disambiguations, highlights the fact that the lower level contains more information than the higher level. Thus we cannot map directly to a low-level description from a high-level description; we can only obtain constraints on the lower level.

B.1.3 Algorithm

Once the high-level set of reaction equations has been disambiguated, they can be reasoned about to obtain constraints on the low-level implementation of the system. This reasoning will give us:

- L : a list of low-level components.

- H : a list of the high-level symbols and how they are composed of low-level components.
- I : a list of which high-level symbols are isomers of each other.
- P : a partial order on the masses of the components and symbols.

The low-level components here represent constraints on how the lower level must be implemented. These components are simply those high-level symbols that do not need to be broken down into other symbols. If a high-level symbol does not *need* to be broken down on the lower-level, this does not mean that it *must* not be broken down; it just means there is no information in the high-level reaction equations *requiring* it to be broken down.

A set of high-level reaction equations can be thought of as an implicit description of how some symbols in the system are composed of other symbols. The purpose of this algorithm is to make this implicit description explicit. This uses a form of unification [5]. The word *unification* has a specific meaning in Computer Science (that applies here), but it can be thought of more generally as a way of taking information that is implicit and spread out; making it explicit and bringing it into one place. In this situation, the information is implicitly spread throughout the high-level reaction equations. We are bringing it into an explicit description of how the high-level symbols are composed of low-level components. Off-the-shelf unification algorithms are not suited to this particular situation, as here there is only one function (composition), and it is commutative. So we have designed a special-purpose unification algorithm (algorithms 9 and 10) to exploit the structure of this problem.

B.1.3.1 Algorithm 9 — set-up

Before we can perform the unification, we need some equations to unify. These will be of the form $\alpha = \beta + \gamma$, representing the fact that the high-level symbol α is composed of the same low-level components as a β symbol combined with a γ symbol. These equations are stored in the data structure D. After the pre-processing steps of disambiguation and removal of influx and outflux reactions, we have isomerisation, composition and decomposition reactions. Algorithm 9 processes these reactions, putting their information into the data structures D, I and P. The decomposition reactions are added as-is into D; the composition reactions are reversed, and added to D. The isomerisation reactions do not need to be put into D, instead their information can be put directly into I. As each equation is added, its information about the partial order on the masses is added to P. When every equation has been processed, the unification can begin. We check the partial order to see if the system conserves mass, and stop now if it does not

Algorithm 9 The first half of the 'downwards' algorithm: Setting up the decompositions of symbols.

```
P := ∅ {partial order on the masses}
I := ∅ {high-level symbols that are isomers}
D := ∅ {decompositions being unified}
for all reaction in high-level-reactions do
  if reaction is α → β {isomerisation} then
    add isomer 'α = β' to I
    add order relation 'α = β' to P
  else if reaction is α + β → γ {composition}
       or γ → α + β {decomposition} then
    add decomposition 'γ = α + β' to D
    add order relations 'α < γ' and 'β < γ' to P
  end if
  if there is a contradiction in P then
    return failure: the system does not conserve mass
  end if
end for
```

(because the unification would fail). If there is a contradiction in the partial order, then the high-level system does not conserve mass. If there is not a contradiction then this does not necessarily mean that the system does conserve mass; there is another conservation of mass check during the unification.

B.1.3.2 Algorithm 10 — unification

After the initial set-up stage, the data structure D is filled with the equations to unify. Algorithm 10 performs this unification and completes the *downwards* algorithm. D contains a list of equations of the form $\omega = \chi + \psi$, where ω, χ and ψ are symbols from the high-level system (or intermediates generated by disambiguation). The equation $\omega = \chi + \psi$ means that the symbol ω is composed of the same low-level components as the symbols χ and ψ. But D could contain another equation: $\omega = \tau + \upsilon$. These two equations both describe how ω is composed, and need to be considered together during this step of the algorithm. During this step we iterate through the equations in D, grouping together all equations describing the same symbol (e.g. ω). So in a typical iteration we might consider the decompositions $d = d_1 = d_2$, where d is ω, d_1 is $\chi + \psi$ and d_2 is $\tau + \upsilon$. So the notation $d = d_1 = d_2$ means that we are considering the two equations, $\omega = \chi + \psi$ and $\omega = \tau + \upsilon$.

For each of these sets of decompositions, we apply one of five operations (in

Algorithm 10 The second half of the 'downwards' algorithm: Unifying the decompositions

L := ∅ {low-level components}
H := ∅ {high-level symbols to be broken down}
PA := ∅ {partial decompositions}
while D is not empty **do**
  for all $d = d_1 = d_2 = \cdots = d_n$ **in** D **do**
    if $n = 1$ **then**
      add decomposition '$d = \mathrm{PA}(d) \cup d_1$' to H
      remove decomposition '$d = d_1$' from D
    else if common symbols in $d_1 = d_2 = \cdots = d_n$ **then**
      cancel the common symbols
      add the common symbols to PA(d)
    else if more than one of $d_1 = d_2 = \cdots = d_n$ are length 1 **then**
      $s_1 = s_2 = \cdots = s_m$ are these decompositions
      for all unique pairs s_i, s_j **do**
        add isomer '$s_i = s_j$' to I
        add order relation '$s_i = s_j$' to P
        if there is a contradiction in P **then**
          return failure: system not conserve mass
        end if
      end for
      remove all but one of $s_1 = \cdots = s_m$ from D
    else if at least one of $d_1 = d_2 = \cdots = d_n$ contains a chemical in I **then**
      for all matching chemicals c **do**
        replace c with its common identifier from I
      end for
    else
      find the first d_i in $d_1 = d_2 = \cdots = d_n$ with a match in H
      replace '$d = d_i$' in D with '$d = H(d_i)$'
    end if
  end for
end while
L := {all high-level symbols} \ (H ∪ I)
return success: L, H, I, P

order) to simplify the equations. This process is iterated until no equations remain. Then the equations have been unified and the process is complete. When simplifying the equations, we may find a way to partially decompose a symbol but not know its full decomposition yet. This information is stored in the temporary variable PA, which is like H but stores partial information about decompositions. The operations that we perform are:

1. If a symbol has only one decomposition ($d = d_1$) then this symbol has been fully decomposed. We add this decomposition to H, including any partial decomposition already done to d.

2. If there are common symbols in the decompositions of a symbol ($d = \chi + \psi = \tau + \psi$) then we cancel these and add them to the partial decomposition of d.

3. If any decompositions of a symbol contain only one symbol themselves, then we can cancel them. We remove all but one of these decompositions from D, and add into I the fact that these symbols are all isomers of each other. We also update the partial order, P, with the fact that these symbols all have the same mass (and we check the partial order for contradictions).

4. Because different symbols can be isomers of each other, we replace all instances of these isomers with a common identifier so they can be cancelled from the equations by operation 2.

5. If none of the above operations can be performed, then we search the decompositions for the first symbol that we know how to decompose (it has an entry in H). We replace this symbol in its equation by its decomposition. So if $\omega = \psi + \chi = \tau + \upsilon$ and $\tau = \rho + \sigma$ then we end up with $\omega = \psi + \chi = \rho + \sigma + \upsilon$.

After all the equations have been unified, the set of low-level components, L, can be read off as those high-level symbols that cannot be decomposed (are not in H) and are not isomers of a different high-level symbol (are not in I).

B.2 Moving upwards

The above algorithm describes a method for moving down from a high-level system of chemical reaction equations to a set of constraints on the components of any low-level implementation of the system. The reverse of this is moving up from a low-level system to a high-level system. Moving upwards is more difficult than moving downwards. Because the high-level system contains more information than the low-level system, moving downwards is about making arbitrary design choices, constrained by the information in the high-level system. But moving upwards is

Algorithm 11 Going upwards from a low-level description to a high-level description of a system.

```
S := ∅ {set of high-level symbols}
R := ∅ {set of high-level reactions}
while long timescale has not expired do
  while short timescale has not expired do
    run the low-level system
  end while
  observe low-level system
  for all new structures not seen before do
    create a new symbol for this structure
    add this new symbol to S
  end for
  for all structures, S, at the start of this timescale do
    if S was in a reaction during this timescale then
      A := { structures that S reacted with }
      B := { products remaining after these reactions }
      create a new symbolic reaction: A → B
      add this reaction to R
    end if
  end for
end while
```

about approximation: removing information from the low-level system to leave a more concise description of the system's behaviour. Working out *which information to remove* is the difficult part, that I have not solved here. The following are some thoughts and ideas on how to go about moving upwards.

The precise implementation details of the lower level system do not matter for the process of moving up to the higher level. However the low-level system is implemented, it will consist of components that interact with each other and join together to form structures. (For example, two hydrogen atoms and one oxygen atom may join to form a water molecule structure.) These structures are symbols on the higher level. The reactions on the higher level summarise the low-level mechanisms by which these structures interact. To produce a high-level description of a low-level system, two things are needed: (1) a list of high-level symbols; and (2) a list of reactions involving these symbols. The symbols represent the structures formed by the low-level components, and the reactions represent the dynamics happening on the lower level. Algorithm 11 gives the pseudocode of an algorithm to do this.

To produce a list of symbols, it is necessary to run the low-level system and

observe the structures that form. The length of time the system is observed for has an impact on the structures observed. If the system has the potential to form very involved structures, but takes a long time to form them, then we will not observe these if we do not run the system for long enough. Likewise if some structures form quickly but rarely, they may also not be observed if the system is not run for long enough. This highlights the fact that the high-level system is an approximation of the low-level system, capturing those structures that form within a certain timescale.

There is a second timescale associated with the observation of the low-level system. When observing structures within the system, a short timescale must also be chosen. Because the low-level components are constantly interacting with each other, a complicated structure goes through intermediate stages in its formation. These intermediate stages may not be appropriate to represent in the high-level system: the only thing required may be the resulting structure. The separate, simple operations happening on the low level are combined into one complicated operation on the high level. This again highlights the fact that the high-level system is an approximation of the low-level system. Separate events that happen in sequence on the low level are collected together into just one event on the high level. A complicated structure that forms through intermediate stages on the lower level springs into being in one step on the higher level.

Observation of the low-level system gives a list of reaction equations as well as a list of symbols. The short timescale is used to approximate a series of intermediate structures by one symbol: the end product of the series. This approximation gives a reaction equation. Whatever structures were present in the area of interest at the start of the short timescale are the reactants in the reaction equation, and whatever structures were left over after the short timescale are the products of the reaction equation. Thus the observation of symbols also gives a list of reaction equations. For a new symbol to be observed, there must have been a process taking place by which the symbol was formed. This process is observed and approximated by the short timescale. This gives a new symbol (or symbols), and a reaction creating the symbol(s). Repeating this observation of the low-level system for the whole duration of the long timescale gives a list of high-level symbols and a list of reaction equations. This is a high-level description of the system.

Clearly there are issues with this approach to moving upwards, but it is a starting point. One major problem might be if different structures form at different rates in the low-level system. Observing the system on a fixed (short) timescale could perfectly capture some structures but struggle with others forming at different rates. If they form more slowly, then it could capturing some of their intermediate stages. If they form more quickly, then they could have started another interaction before being captured.

B.3 Applying this to GraphMol

Section 9.2 describes how automatically moving between levels could be used to create a multi-level evolutionary system. We do not yet have all the different parts of this jigsaw, but we have some.

GraphMol is ideal as a low-level system, since it is an embodied artificial chemistry. A symbolic artificial chemistry could be used as the high-level system, although in the future we might prefer a system at a slightly lower level. The *moving downwards* algorithm described above would generate constraints describing how many GraphMol chemicals were needed in the world, and which chemicals need to react with each other. This would tell us how many binding sites each GraphMol chemical needed, and which sites should have affinities for each other. There would be design choices to make about precisely *how* to implement the reactions within the GraphMol world, but the constraints would specify *which* reactions were needed.

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, first edition, 1985.

[2] C. Adami. *Introduction to Artificial Life*. Springer, 1999.

[3] P. S. Andrews, F. A. C. Polack, A. T. Sampson, S. Stepney, and J. Timmis. The CoSMoS process, version 0.1: A process for the modelling and simulation of complex systems. Technical Report YCS-2010-453, Department of Computer Science, University of York, Mar. 2010.

[4] F. J. Ayala. Darwin's greatest discovery: Design without designer. *Proceedings of the National Academy of Sciences*, 104(Suppl 1):8567–8573, May 2007.

[5] F. Baader and W. Snyder. Unification Theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 447–533. Elsevier Science, 2001.

[6] T. Back, F. Hoffmeister, and H.-p. Schwefel. A survey of evolution strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9, 1991.

[7] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pages 14–21, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[8] W. Banzhaf, G. Beslon, S. Christensen, J. A. Foster, F. Képès, V. Lefort, J. F. Miller, M. Radman, and J. J. Ramsden. Guidelines: From artificial evolution to computational evolution: a research agenda. *Nature reviews. Genetics*, 7(9):729–735, Sept. 2006.

[9] W. Banzhaf, P. Dittrich, and H. Rauhe. Emergent computation by catalytic reactions. *Nanotechnology*, 7:307–314, 1996.

[10] M. F. Barnsley. *Fractals Everywhere: The First Course in Deterministic Fractal Geometry*. Academic Press, 1988.

[11] M. A. Bedau and N. H. Packard. Evolution of evolvability via adaptation of mutation rates. *Biosystems*, 69(2-3), 2003.

[12] H.-G. Beyer and H.-P. Schwefel. Evolution strategies – A comprehensive introduction. *Natural Computing*, 1(1):3–52, May 2002.

[13] C. Biemont and C. Vieira. Genetics: Junk DNA as an evolutionary force. *Nature*, 443(7111):521–524, Oct. 2006.

[14] R. A. Brooks. Elephants don't play chess. *Robotics and Autonomous Systems*, 6(1-2):3–15, June 1990.

[15] T. Brown. *Genomes 3*. Garland Science, 2006.

[16] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *In Proceedings of 39th Annual ACM Symposium on Theory of Computing*, pages 590–598, 2007.

[17] E. Clark, A. Nellis, S. Hickinbotham, S. Stepney, T. Clarke, M. Pay, and Young. Degeneracy enriches artificial chemistry binding systems. In *ECAL 2011, Paris, France, August 2011*, pages 133–140. MIT Press, Aug. 2011.

[18] J. Clune, B. E. Beckmann, C. Ofria, and R. T. Pennock. Evolving coordinated quadruped gaits with the HyperNEAT generative encoding. In *CEC 2009, Trondheim, Norway, May 2009*. IEEE Press, May 2009.

[19] N. L. Cramer. A Representation for the Adaptive Generation of Simple Sequential Programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 183–187. L. Erlbaum Associates Inc., 1985.

[20] F. Crick. Central Dogma of Molecular Biology. *Nature*, 227(5258):561–563, Aug. 1970.

[21] F. H. C. Crick. Origin of the Genetic Code. *Nature*, 213, Jan. 1967.

[22] C. Darwin. *The origin of species by means of natural selection, or the preservation of favoured races in the struggle for life*. John Murray, London, 6th edition, 1876.

[23] E. D. De Jong and J. B. Pollack. Ideal Evaluation from Coevolution. *Evol. Comput.*, 12(2):159–192, June 2004.

[24] K. Deb. Evolution's Niche in Applied Optimization and Informatics. In *CEC 2009, Trondheim, Norway, May 2009*. IEEE Press, May 2009.

[25] J. Decraene, G. G. Mitchell, and B. McMullin. Unexpected evolutionary dynamics in a string based artificial chemistry. In *ALife XI*, pages 158–165. MIT Press, 2008.

[26] A. K. Dewdney. Recreational Mathematics – Core Wars. *Scientific American*, May 1984.

[27] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries–a review. *Artificial Life*, 7(3):225–275, 2001.

[28] A. Droop and S. Hickinbotham. Properties of Biological Mutation Networks and Their Implications for ALife. *Artificial Life Journal*, 17:353–364, 2011.

[29] G. A. Edgar. Darwin: a survival game for programmers. *Comput. Language*, 4(4):79–86, 1987.

[30] A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, Mar. 2011.

[31] A. Faulconbridge, S. Stepney, J. Miller, and L. Caves. RBNWorld: Sub-symbolic artificial chemistry. In *ECAL 2009, Budapest, Hungary, September 2009*. LNCS. Springer, Sept. 2009.

[32] A. Faulconbridge, S. Stepney, J. F. Miller, and L. Caves. RBN-world: The hunt for a rich AChem. In *ALife XII*, pages 261–268. MIT Press, 2010.

[33] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345+, June 1962.

[34] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, 1966.

[35] W. Fontana. Algorithmic Chemistry. *Artificial Life II*, pages 159–210, 1992.

[36] W. Fontana and L. Buss. The arrival of the fittest: Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 56(1):1–64, Jan. 1994.

[37] I. A. Glover and P. M. Grant. *Digital Communications*. Prentice Hall (UK), second edition, Sept. 2003.

[38] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[39] P. J. B. Hancock. An Empirical Comparison of Selection Methods in Evolutionary Algorithms. In *Selected Papers from AISB Workshop on Evolutionary Computing*, pages 80–94. Springer, 1994.

[40] D. Harel. Can we computerize an elephant? In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, MEMOCODE'09, page 77. IEEE Press, 2009.

[41] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Molecular microprograms. In *ECAL 2009, Budapest, Hungary, September 2009*. LNCS. Springer, Sept. 2009.

[42] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Diversity From a Monoculture: Effects of Mutation-On-Copy in a String-Based Artificial Chemistry. In *ALife XII*, pages 24–31. MIT Press, 2010.

[43] S. Hickinbotham, E. Clark, S. Stepney, T. Clarke, A. Nellis, M. Pay, and P. Young. Specification of the Stringmol chemical programming language version 0.2. Technical Report YCS-2010-458, Univ. of York, 2012.

[44] S. Hickinbotham, A. Faulconbridge, and A. Nellis. The Blind Watchmaker's Workshop: Three Artificial Chemistries in the Context of Eigen's Paradox. In *ALife XII, Odense, Denmark, August 2010*. MIT Press, 2010.

[45] S. Hickinbotham, S. Stepney, A. Nellis, T. Clarke, E. Clark, M. Pay, and P. Young. Embodied genomes and metaprogramming. In *ECAL 2011, Paris, France, August 2011*, pages 334–341. MIT Press, 2011.

[46] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. In *CNLS '89*, pages 228–234. North-Holland, 1990.

[47] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[48] J. Horn, N. Nafpliotis, and D. E. Goldberg. A niched Pareto genetic algorithm for multiobjective optimization. In *IEEE World Congress on Computational Intelligence*, pages 82–87 vol.1. IEEE Press, 1994.

[49] G. S. Hornby. Functional Scalability through Generative Representations: the Evolution of Table Designs. *Environment and Planning B: Planning and Design*, 31(4):569–587, July 2004.

[50] G. S. Hornby and J. B. Pollack. Evolving L-Systems To Generate Virtual Creatures. *Computers and Graphics*, 25(6):1041–1048, 2001.

[51] D. L. Kacian, D. R. Mills, F. R. Kramer, and S. Spiegelman. A Replicating RNA Molecule Suitable for a Detailed Analysis of Extracellular Evolution and Replication. *Proceedings of the National Academy of Sciences*, 69(10):3038–3042, Oct. 1972.

[52] K. Kaneko and T. Ikegami. Homeochaos: dynamics stability of a symbiotic network with population dynamics and evolving mutation rates. *Physica D: Nonlinear Phenomena*, 56:406–429, June 1992.

[53] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Empirical analysis of different levels of meta-evolution. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, pages 2086–2093. IEEE, 1999.

[54] W. Kantschik, P. Dittrich, M. Brameier, and W. Banzhaf. Meta-evolution in graph GP. In *Second European Workshop on Genetic Programming, Goteborg*. Springer, 1999.

[55] S. A. Kauffman. Autocatalytic sets of proteins. *Journal of Theoretical Biology*, 119(1):1–24, Mar. 1986.

[56] S. A. Kauffman. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford University Press, first edition, 1993.

[57] C. Kelly. *Design and Characterisation of a Novel Artificial Life System Incorporating Hierarchical Selection*. PhD thesis, Dublin City University, 2010.

[58] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. MIT Press, 1 edition, 1992.

[59] P. Kreyssig and P. Dittrich. Reaction Flow Artificial Chemistries. In *ECAL 2011, Paris, France*, pages 431–437. MIT Press, 2011.

[60] V. Kvasnicka, J. Pospichal, and T. Kaláb. A Study of Replicators and Hypercycles by Typogenetics. In *ECAL '01: Proceedings of the 6th European Conference on Advances in Artificial Life*, pages 37–54, London, UK, 2001. Springer-Verlag.

[61] C. G. Langton. *Artificial Life: An Overview*. MIT Press, 1995.

[62] J. Lehman and K. Stanley. Exploiting open-endedness to solve problems through the search for novelty. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 329–336. MIT Press, Cambridge, MA, 2008.

[63] J. Lehman and K. O. Stanley. Abandoning objectives: evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, June 2011.

[64] D. Lenat. Eurisko: A program that learns new heuristics and domain conceptsThe nature of Heuristics III: Program design and results. *Artificial Intelligence*, 21(1-2):61–98, Mar. 1983.

[65] D. B. Lenat and J. S. Brown. Why am and eurisko appear to work. *Artificial Intelligence*, 23(3):269–294, Aug. 1984.

[66] A. Lovelace. Note G. In R. Taylor, editor, *Scientific Memoirs: Selected from the Transactions of Foreign Academies of Science and Learned Societies, and from Foreign Journals*, volume 3, page 722. 1843.

[67] J. Maynard Smith and E. Szathmáry. *The major transitions in evolution*. Perseus Books, 1999.

[68] B. McClintock. The Origin and Behavior of Mutable Loci in Maize. *Proceedings of the National Academy of Sciences of the United States of America*, 36(6):344–355, June 1950.

[69] B. McMullin. John von Neumann and the evolutionary growth of complexity: looking backward, looking forward... *Artificial Life*, 6(4):347–361, Sept. 2000.

[70] B. McMullin, C. Kelly, and D. O'Brien. Multi-level selectional stalemate in a simple artificial chemistry. In *Proceedings of the 9th European conference on Advances in artificial life*, ECAL'07, pages 22–31, Berlin, Heidelberg, 2007. Springer-Verlag.

[71] J. F. Miller and P. Thomson. Cartesian Genetic Programming. In *Proceedings of the 3rd European Conference on Genetic Programming*, number 1802 in LNCS, pages 121–132. Springer, 2000.

[72] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, third printing edition, 1998.

[73] A. Nellis and S. Stepney. Automatically moving between levels in Artificial Chemistries. In *ALife XII, Odense, Denmark, August 2010*, pages 269–276. MIT Press, Aug. 2010.

[74] A. Nellis and S. Stepney. Embodied copying for richer evolution. In *ECAL 2011, Paris, France, August 2011*, pages 597–604. MIT Press, 2011.

[75] G. Nitschke. Co-evolution of cooperation in a pursuit evasion game. In *Intelligent Robots and Systems, 2003*, volume 2, 2003.

[76] S. Nolfi and D. Floreano. Coevolving Predator and Prey Robots: Do "Arms Races" Arise in Artificial Evolution? *Artif. Life*, 4(4):311–335, 1998.

[77] A. Pargellis. The evolution of self-replicating computer organisms. *Physica D: Nonlinear Phenomena*, 98(1):111–127, 1996.

[78] J. Paulsson, O. G. Berg, and M. Ehrenberg. Stochastic focusing: Fluctuation-enhanced sensitivity of intracellular regulation. *Proceedings of the National Academy of Sciences*, 97(13):7148–7153, June 2000.

[79] H. Pearson. 'Virophage' suggests viruses are alive. *Nature*, 454(7205), Aug. 2008.

[80] M. D. Preston, J. W. Pitchford, and A. J. Wood. Evolutionary optimality in stochastic search problems. *Journal of The Royal Society Interface*, 7(50):1301–1310, Sept. 2010.

[81] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1991.

[82] T. Quick, K. Dautenhahn, C. L. Nehaniv, and G. Roberts. The essence of embodiment: A framework for understanding and exploiting structural coupling between system and environment. In *American Institute of Physics Conference Series*, volume 517 of *American Institute of Physics Conference Series*, pages 649–660, May 2000.

[83] T. Rajala, A. Häkkinen, O. Smolander, O. Yli-Harja, and A. S. Ribeiro. Effect of Pauses on Transcriptional Noise. In *BioSysBio 2009, Cambridge, UK, 2009*, Mar. 2009.

[84] S. Rasmussen, C. Knudsen, P. Feldberg, and M. Hindsholm. The coreworld: emergence and evolution of cooperative structures in a computational chemistry. *Physica D: Nonlinear Phenomena*, 42(1-3):111–134, 1990.

[85] T. S. Ray. An approach to the Synthesis of Life. In *Artificial Life II*, pages 371–408. Addison Wesley, 1992.

[86] I. Rechenberg. Cybernetic solution path of an experimental problem. In *Royal Aircraft Establishment Translation No. 1122, B. F. Toms, Trans.* Ministry of Aviation, Royal Aircraft Establishment, Farnborough Hants, Aug. 1965.

[87] A. V. Samsonovich and K. A. De Jong. Pricing the 'free lunch' of meta-evolution. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*. ACM, 2005.

[88] H. Sayama. A New Structurally Dissolvable Self-Reproducing Loop Evolving in a Simple Cellular Automata Space. *Artificial Life*, 5(4):343–365, Oct. 1999.

[89] K. Sims. Evolving 3D morphology and behavior by competition. *Artif. Life*, 1(4):353–372, 1994.

[90] K. Sims. Evolving virtual creatures. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 15–22. ACM, 1994.

[91] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[92] L. Spector and A. Robinson. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. 3(1):7–40, 2002.

[93] S. Stepney. Embodiment. In D. Flower and J. Timmis, editors, *In Silico Immunology*, chapter 12, pages 265–288. Springer, 2007.

[94] S. Stepney, R. E. Smith, J. Timmis, A. M. Tyrrell, M. J. Neal, and A. N. W. Hone. Conceptual Frameworks for Artificial Immune Systems. *International Journal of Unconventional Computing*, 1(3):315–338, July 2005.

[95] T. Stibor, J. Timmis, and C. Eckert. On the Use of Hyperspheres in Artificial Immune Systems as Antibody Recognition Regions. In H. Bersini and J. Carneiro, editors, *Proceedings of the 5th international conference on Artificial Immune Systems*, volume 4163 of *Lecture Notes in Computer Science*, chapter 17, pages 215–228. Springer Berlin / Heidelberg, 2006.

[96] E. Szathmáry and J. M. Smith. From replicators to reproducers: the first major transitions leading to life. *Journal of theoretical biology*, 187(4):555–571, Aug. 1997.

[97] J. N. Thompson. *The Coevolutionary Process*. University Of Chicago Press, 1 edition, 1994.

[98] J. Timmis and M. Neal. A resource limited artificial immune system for data analysis. *Knowledge-Based Systems*, pages 121–130, June 2001.

[99] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[100] L. Van Valen. A new evolutionary law. *Evolutionary Theory*, 1:1–30, 1973.

[101] J. von Neumann. *Theory of Self-Reproducing Automata*. In Burks (1966). Based on transcripts of lectures delivered at the University of Illinois in December, 1949.

[102] R. A. Watson, R. Mills, and C. L. Buckley. Global Adaptation in Networks of Selfish Components: Emergent Associative Memory at the System Scale. *Artificial Life*, 17(3):147–166, May 2011.

[103] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, Apr. 1997.

[104] J. P. Young, L. Crossman, A. Johnston, N. Thomson, Z. Ghazoui, K. Hull, M. Wexler, A. Curson, J. Todd, P. Poole, T. Mauchline, A. East, M. Quail, C. Churcher, C. Arrowsmith, I. Cherevach, T. Chillingworth, K. Clarke, A. Cronin, P. Davis, A. Fraser, Z. Hance, H. Hauser, K. Jagels, S. Moule, K. Mungall, H. Norbertczak, E. Rabbinowitsch, M. Sanders, M. Simmonds, S. Whitehead, and J. Parkhill. The genome of Rhizobium leguminosarum has recognizable core and accessory components. *Genome biology*, 7(4):R34+, 2006.

[105] J. Zupan, D. Ward, and P. Zambryski. Inter-kingdom DNA transfer decoded. *Nat Biotech*, 20(2):129–131, Feb. 2002.