Speeding up Graph Programs

Ziad Ismaili Alaoui

MASTER OF SCIENCE BY RESEARCH

UNIVERSITY OF YORK

Department of Computer Science

December 2024

UNIVERSITY OF YORK

Abstract

Department of Computer Science

Master's by Research

Speeding up Graph Programs

by Ziad ISMAILI ALAOUI

This thesis presents an improvement in rule-based graph programming which enables the design of graph programs that achieve the time complexity of imperative linear-time algorithms. Achieving such complexity in conventional languages using graph transformation rules is challenging due to the high cost of graph matching. Previous work demonstrated that with rooted rules, certain algorithms can be executed in linear time using the graph programming language GP 2. However, for non-destructive algorithms that preserve the structure of input graphs, achieving linear runtime required input graphs to be both connected and of bounded node degree.

In this thesis, we overcome these limitations by enhancing the graph data structure generated by the GP 2 compiler and exploiting this new structure within the programs. We present four case studies: a program that checks graph's connectedness, a program that two-colours a graph, a cycle detection program, and a program that computes the shortest weighted distances from a single source to all other nodes in a weighted graph. The first three programs run in linear time on both connected and disconnected input graphs with arbitrary node degrees. The fourth program achieves a time complexity comparable to that of conventional implementations in imperative programming languages. For each program, we provide a formal analysis of its correctness and complexity, supplemented by empirical benchmarking.

Parts of this thesis have been published previously within the following publications:

> Ziad Ismaili Alaoui and Detlef Plump. "Linear-Time Graph Programs for Unbounded-Degree Graphs". In: *Graph Transformation.* Springer Nature Switzerland, 2024, 3–20. ISBN: 9783031642852. DOI: 10.1007/978-3-031-64285-2_1. LNCS 14774.
>
> Ziad Ismaili Alaoui and Detlef Plump. "Linear-Time Graph Programs without Preconditions". Proc. 14th and 15th International Workshop on Graph Computation Models (GCM 2024 and GCM 2025). Electronic Proceedings in Theoretical Computer Science 417, pages 39-54, 2025. DOI: 10.4204/EPTCS.417.3.

Declaration of Authorship

I, Ziad Ismaili Alaoui, declare that this thesis is a presentation of original work and I am the sole author, except where explicit attribution has been given. This work has not previously been presented for an award at this, or any other, university. All sources are acknowledged as references.

Ziad Ismaili Alaoui, June 2025

Acknowledgements

I would like to express my sincere appreciation to all my supportive friends and family members, both locally and internationally, who have been part of my academic journey at the University of York. Special thanks go to my supervisor, Detlef Plump, with whom I greatly enjoyed working. His meticulous guidance and unwavering support throughout my Master's have been invaluable. I would also like to thank the staff at the Department of Computer Science at the University of York for their assistance in helping me access the services I needed, as well as my lab mate for the engaging discussions we had, often over cold coffee.

A special acknowledgement goes to my youngest sister, Amira, who, at the time of writing this thesis, has kept me in her thoughts and loved me unconditionally, despite my long absences from home and her young age.

Contents

List of Figures

Chapter 1

Introduction

Graphs provide a natural and intuitive way to represent relationships between objects in a complex system. Since systems are often dynamic, their changes can be effectively modelled using frameworks in the domain of graph rewriting, particularly rule-based graph transformation.

Rule-based graph transformation modifies graphs by applying specified rules. Each rule consists of two components: a left-hand side and a right-hand side, which define how the transformation affects the resulting graph. Since the 1970s, when the algebraic approach was introduced in Germany by Ehrig, Pfender and Schneider [EPS73], rule-based graph transformation has been extensively studied. Its simplicity and ability to facilitate reasoning and formal analysis have contributed significantly to its recognition and popularity. Beyond theory, applications of rule-based graph transformation are plentiful; they include software engineering [Kol+22; FH00], UML diagrams [FM07], natural language processing [BGP18], logical [CRPP91] and functional programming [PE93], and even chemistry [BFS03b; BFS03a]. Several graph programming languages, based on different theoretical frameworks, have been designed; they include GReAT [Agr+06], GROOVE [Gha+12], GrGen.Net [JBK10], Henshin [Str+17], PORGY [FKP19] and GP 2. We begin this thesis by outlining the motivation behind it and our aims in Section 1.1, and finally we present the structure of the thesis in Section 1.2.

1.1 Motivation and Aim

Designing and implementing languages for rule-based graph transformation poses significant performance challenges. Typically, programs written in these languages do not achieve the same runtime efficiency as those written in conventional imperative languages, such as C or Java. The main obstacle is the cost of graph matching, where matching the left-hand graph L of a rule within a host graph G generally requires time $|G|^{|L|}$, with $|X|$ denoting the size of graph X. (Since L is fixed, this is a polynomial.) As a consequence, standard imperative graph algorithms running in linear time (see [Cor+09; Ski20] for examples) may exhibit non-linear, polynomial runtimes when recast as rule-based graph programs.

As a way to address this issue, the graph programming language GP 2 [Plu12] supports *rooted* graph transformation rules, initially proposed by Dörr [Dör95]. This approach involves designating certain nodes as *roots* and matching them with roots in the host graphs. Consequently, only the neighbourhoods of host-graph roots need to be searched for matches, which can often be done in constant time under some light conditions. The GP 2 compiler [Bak15] maintains a list of pointers to roots in the host graph, facilitating constant-time access to roots if their number remains bounded throughout the program's execution.

The first linear-time graph problem implemented by a GP 2 program was one that two-colours a graph. In [PB12; Bak15], it is shown that this program colours connected graphs of bounded node degree in linear time. Since then, the GP 2 compiler has received some major improvements, particularly related to the runtime graph data structure used by the compiled programs [CRP20]. These improvements made a linear-time worst-case performance possible for a wider class of programs, in some cases even on input graph classes of unbounded degree.

Despite this progress, programs that retain the structure of input graphs (i.e. non-destructive programs), such as the aforementioned two-colouring program, have until now required non-linear runtimes on disconnected graphs. The issue arises because, after a connected component is visited, the number of failed attempts to match a non-visited node in a different connected component increases. As a result, in disconnected graph classes, this number may grow quadratically in the graph size, which leads to a quadratic runtime.

We aim in this thesis to address these challenges. We present two updates to the GP 2 compiler that eliminate the need for the host graph to be connected and have a bounded node degree in order to achieve linear runtimes for certain programs. The solution involves improving the internal graph data structure. Nodes are now stored in separate linked lists based on their marks (red, green, blue, grey, or unmarked), and each node is associated with a two-dimensional array of linked lists that store all incident edges based on their marks (red, green, blue, dashed, or unmarked) and orientation (incoming, outgoing, or looping). This allows the matching algorithm to find, in constant time, a node with a specific mark or an edge with a specific mark and orientation. For example, if a red node is needed, a single access to the list of red nodes will either locate the node or confirm its absence. Similarly, if an outgoing green edge is required, a single access to the corresponding linked list will either find such an edge or determine that none exists.

1.2 Thesis Structure

This thesis is organised into chapters. Chapter 2 provides an overview of the theoretical framework underlying the language GP 2, including its syntax and a sample program to illustrate its use. Chapter 3 discusses how the compiler performs graph matching and identifies two key issues that contribute to the non-linear runtimes observed in unbounded-degree graph classes for graph-traversal programs in GP 2. Chapter 4 presents several case studies demonstrating how the changes introduced in the previous chapter help achieve the thesis objectives. Each case study includes a formal analysis of correctness and time complexity. To the best of our knowledge, no other works in the literature provide such graph programs along with formal proofs of correctness and complexity.

Empirical measurements are also provided for each case study, covering various graph classes with both bounded and unbounded degree. Chapter 5 concludes the thesis.

Chapter 2

Preliminaries and Background

GP 2 is a graph programming language that takes a graph as input, transforms it following a user-specified program, and outputs a resulting graph. It was designed and introduced by Plump [Plu12], and Bak implemented it as a GP 2-to-C compiler [Bak15] (Figure 2.1). The aforementioned compiler was later updated in 2020 by Campbell, Romö and Plump [CRP20], and subsequently in 2024 by this thesis' author and Plump [IAP24].

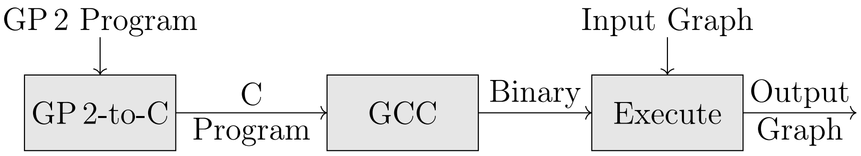


FIGURE 2.1: GP 2-to-C compiler.

As depicted in Figure 2.1, the GP 2-to-C compiler takes a user-written graph program specified in the semantics of GP 2 and translates it into an equivalent C program. The latter code is then fed to the GCC compiler, which translates it into machine-readable binary. The C runtime system then executes the binary in conjunction with a user-provided input graph and outputs a resulting graph.

GP 2 is known to be computationally complete in that any function on graphs can be programmed in the language [Plu17]. It is, in principle, conceptually non-deterministic in that a rule application can result in different graphs and that it allows for sets of rules to be called non-deterministically; however, the implemented compiler, being the subject of this thesis and the previous studies on GP 2, is deterministic in that any execution of a program on an input graph will always yield the same output graph.

Graphs in GP 2 are attributed; that is, nodes and edges carry information in the form of labels. A label consists of a list of (possibly zero) variables (integers, strings and characters) and a mark. Lists can be heterogeneous, i.e. two distinct elements need not be of the same type. Marks include blue, red, green, grey (exclusive to nodes) and dashed (exclusive to edges). If a node or edge does not have a mark, it is *unmarked*. GP 2 graphs are directed, with parallel and looping edges allowed. The language also supports so-called *roots*, a subset of distinguished nodes with access separate from other nodes.

In Section 2.1, we introduce the theory under double-pushout transformation, the backbone of GP 2. Section 2.2 is devoted to attributed graph transformation, where transformation with labelling is presented. And finally, Section 2.5 presents the semantics and syntax of the language.

The concepts, definitions, framework, examples and structure presented in this chapter are largely derived from the work of Courtehoute [Cou23] (Chapter 2), who explored similar topics in his thesis on the time complexity of GP 2 programs.

2.1 Double-Pushout Graph Transformation

2.1.1 Graphs

We consider finite, directed graphs that allow for looping edges (edges connecting a node to itself) and parallel edges (multiple edges between two nodes).

Definition 2.1.1 (Graph)**.** Let $\mathcal{L}$ be a set of labels. A graph G over $\mathcal{L}$ is defined as the tuple $\langle V_G, E_G, s_G, t_G, l_G, m_G, p_G \rangle$, where:

- V_G and E_G are finite sets of nodes and edges, respectively;
- $s_G : E_G \to V_G$ and $t_G : E_G \to V_G$ are the source and target functions for edges;
- $l_G : V_G \xrightarrow{\text{par}} \mathcal{L}$ is a partial node labelling function;
- $m_G : E_G \to \mathcal{L}$ is the edge labelling function; and
- $p_G : V_G \xrightarrow{\text{par}} \{0, 1\}$ is the partial rootedness function, where $p_G(v) = 1$ indicates v is a root, and $p_G(v) = 0$ indicates v is unrooted.

A graph G is called *totally labelled* if its node labelling function l_G is total. Nodes $v \in V_G$ for which l_G is undefined are referred to as *unlabelled* and are denoted by $l_G(v) = \perp$, where $\perp \notin \mathcal{L}$. This allows us to extend l_G to a total function $l_G : V_G \to \mathcal{L} \cup \{\perp\}$. Similarly, the graph G is said to have defined *rootedness* if its rootedness function p_G is total. Nodes $v \in V_G$ with undefined rootedness are denoted by $p_G(v) = \perp$, enabling the extension of p_G to a total function $p_G : V_G \to \{0, 1, \perp\}$.

Rootedness is a special status that, when assigned to a small set of nodes, enables fast access to the local area around those nodes. Intuitively, rooted nodes can be seen as the "current node" in graph traversal algorithms. Root nodes in this thesis are represented with double borders. They are modelled using a partial function that maps a node to either 0 (unrooted) or 1 (rooted). (The use of a partial function is analogous to the modelling of node labels and allows for flexibility in cases where a node's rootedness is undefined.) Note that a graph with defined rootedness does not necessarily have any roots; it simply means that each node is either rooted or unrooted. We require both node labelling and rootedness functions to be partial because this abstraction allows us to map nodes independently of their labels and rootedness.

Figure 2.2 gives a representation example of a graph. In this thesis, nodes are drawn as circles or pill shapes (when their labels are longer than a character), with root nodes drawn using double borders. Nodes with undefined rootedness are represented with dashed double borders. Edges are shown as arrows pointing from their source node to their target node. Nodes that have an undefined label are marked with the symbol $\perp$.

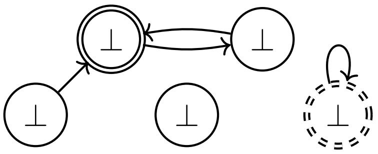


Figure 2.2: Example of a graph.

2.1.2 Graph Morphisms

In this subsection, we explore how we map graphs into each other with structure-preserving functions (called graph morphisms). In this thesis, the word *graph* is often omitted since we do not use other kinds of morphisms.

For some function $f : X \to Y$, we define $\mathrm{dom}(f)$ as the *domain* of f (i.e. $\mathrm{dom}(f) = X$).

Definition 2.1.2 (Graph Morphism)**.** Let G and H be graphs. A (graph) morphism $g : G \to H$ is a pair $g = \langle g_V, g_E \rangle$ of functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ such that the following holds.

- *Preservation of sources and targets:* for each edge $e \in E_G$, we have $g_V(s_G(e)) = s_H(g_E(e))$ and $g_V(t_G(e)) = t_H(g_E(e))$.
- *Preservation of edge labels:* for each edge $e \in E_G$, we have $m_G(e) = m_H(g_E(e))$.
- *Preservation of defined node labels:* for each node of defined label $v \in \mathrm{dom}(l_G) \subseteq V_G$, we have $g_V(v) \in \mathrm{dom}(l_H)$ and $l_G(v) = l_H(g_V(v))$.
- *Preservation of defined rootedness:* for each node of defined rootedness $v \in \mathrm{dom}(p_G) \subseteq V_G$, we have $g_V(v) \in \mathrm{dom}(p_H)$ and $p_G(v) = p_H(g_V(v))$.

We hereby define some properties of morphisms.

Definition 2.1.3 (Morphism Properties)**.** Let $g : G \to H$ be a morphism.

- We say g is *injective/surjective* if both g_V and g_E are *injective/surjective.*
- We say g is *bijective* if it is both injective and subjective.
- We say morphisms $g : G \to H$ and $f : I \to J$ are *equal*, denoted by $g = f$, if $G = I$, $H = J$, $g_V = f_V$, and $g_E = f_E$.
- We call $\mathrm{id}_G : G \to G$ where $\mathrm{id}_G = \langle \mathrm{id}_{V_G}, \mathrm{id}_{E_G} \rangle$ the *identity (graph) morphism.*

Definition 2.1.4 (Graph Isomorphism)**.** A (graph) morphism $g : G \to H$ is a *(graph) isomorphism* if it is a bijective morphism that preserves undefined labels and undefined rootedness, i.e. for each $v \in V_G$, if $l_G(v) = \perp$ then $l_H(g_V(v)) = \perp$, and if $p_G(v) = \perp$ then $p_H(g_V(v)) = \perp$. If such an isomorphism exists, we call G and H isomorphic, denoted by $G \cong H$.

Definition 2.1.5 (Graph Inclusion)**.** A (graph) morphism $g : H \to G$ is a *(graph) inclusion* if for each $v \in V_H$ and each edge $e \in E_H$, we have $g_V(v) = v$ and $g_E(e) = e$.

Definition 2.1.6 (Subgraph)**.** If a (graph) morphism $g : H \to G$ is an inclusion, we may use the notation $g : H \hookrightarrow G$, and we say H is a *subgraph* of G, denoted as $H \subseteq G$.

Definition 2.1.7 (Operations on Morphisms)**.** Let $g : G \to H$ be a morphism.

- Given morphisms $g : G \to H$ and $f : H \to I$, we define their *composition* $f \circ g : G \to I$ as $f \circ g = \langle f_V \circ g_V, f_E \circ g_E \rangle$. We sometimes omit the composition's name by writing $G \to H \to I$.

- Given a morphism $f : G \to H$, and a subgraph I of G with a **non-surjective** inclusion $g : I \hookrightarrow G$, we define the *restriction* of f to I as $f|_I : I \to H$ where $f|_I = \langle f_V|_{V_I}, f_E|_{E_I} \rangle$.

Note that the composition of two morphisms, isomorphisms, or inclusions is itself a morphism, isomorphism, or inclusion, respectively. Additionally, restriction preserves the properties of being a morphism or an inclusion. While it does not preserve isomorphisms, it does preserve injectivity.

2.1.3 Rules

The fundamental construct in graph computation is a rule, consisting of a left-hand side graph L and a right-hand side graph R. Informally, to apply a rule to a graph G, we first identify a morphism from L to a subgraph of G. Then, the image of L in G is transformed to resemble R.

Let us lay down some formal definitions surrounding the construct.

Definition 2.1.8 (Rule)**.** A *rule* $\langle L \leftarrow K \to R \rangle$ consists of graph K with unlabelled nodes of undefined rootedness and no edges, totally labelled graphs with defined rootedness L and R, and inclusions $K \hookrightarrow L$ and $K \hookrightarrow R$. We call L the *left-hand side*, R the *right-hand side*, and K the *interface*.

The graph K consists of unlabelled nodes with no defined rootedness. This design enables the inclusions to map these nodes to either rooted or unrooted nodes in L and R. If such flexibility were absent, the labels and rootedness of corresponding nodes in L and R would need to match exactly. This restriction would make it impossible for a rule to relabel nodes or modify their rootedness, which is an essential feature of the GP 2 programming language. More details on this are provided in Subsection 2.4.4.

Furthermore, edge labels do not require explicit modifications within rules, as edges can be deleted and reintroduced with new labels connecting the same pair of nodes.

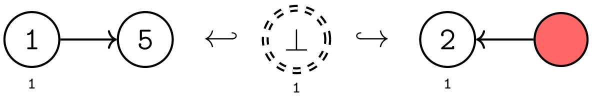


FIGURE 2.3: Example of a rule.

An example rule is illustrated in Figure 2.3. The rule uses GP 2 labels, including numeric values 1, 5 and 2, as well as the colours red and blue, which are defined in Subsection 2.4.1. Unlabelled nodes in the figure are assumed to carry the empty GP 2 label for clarity. Nodes annotated with small numbers underneath denote interface nodes, which specify how inclusions map these nodes. As such, when representing a rule $\langle L \leftarrow K \rightarrow R\rangle$, it is unnecessary to explicitly depict K; so the rule is simply expressed as $L \Rightarrow R$.

2.2 Double-Pushout Graph Rewriting

2.2.1 Double-Pushouts

The foundational framework for transformation in GP 2 is double-pushout (DPO) graph rewriting, which allows us to apply a rule to a host graph and obtain a unique outcome. We hereby define some of its notions and properties.

Definition 2.2.1 (Pushout, Pullback, Natural Pushout)**.** Let A, B, C, and D be graphs with morphisms $A \to B$, $B \to D$, $A \to C$, and $C \to D$, as illustrated in Figure 2.4. Assume square (1) commutes ($A \to B \to D = A \to C \to D$).

We say square (1) is a *pushout* if for all graphs D', and all morphisms $B \to D'$ and $C \to D'$ such that $A \to B \to D' = A \to C \to D'$ (see the left-hand side of Figure 2.4), there is a unique morphism $D \to D'$ such that triangles (2) and (3) commute ($B \to D' = B \to D \to D'$ and $C \to D' = C \to D \to D'$).

We say square (1) is a *pullback* if for all graphs A', and all morphisms $A' \to B$ and $A' \to C$ such that $A' \to B \to D = A' \to C \to D$ (see the right-hand side of Figure 2.4), there is a unique morphism $A' \to A$ such that triangles (2) and (3) commute ($A' \to B = A' \to A \to B$ and $A' \to C = A' \to A \to C$).

We call a square (1) a *natural pushout* if it is both a pushout and a pullback.

We call B and C *pushout/pullback complements*.

$$\begin{tikzcd}
& B \arrow[dr] \arrow[drr, bend left] & & & & & B \arrow[dr] & \\
A \arrow[ur] \arrow[dr] & (1) & D \arrow[r, dashed, "(2)", "(3)"'] & D' & A' \arrow[r, dashed, "(2)", "(3)"'] \arrow[urr, bend left] \arrow[drr, bend right] & A \arrow[ur] \arrow[dr] & (1) & D \\
& C \arrow[ur] \arrow[urr, bend right] & & & & & C \arrow[ur] &
\end{tikzcd}$$

FIGURE 2.4: A pushout (left) and a pullback (right).

Pushouts can be thought of as a way to glue graphs: the graph D is constructed by merging B and C along their shared nodes and edges with A. And in contrast, pullbacks instead represent intersections: the graph A is formed by identifying the portions of B and C that overlap in D and combining them in the same fashion.

Definition 2.2.2 (Natural Double-Pushout)**.** Let the squares in Figure 2.5 be natural pushouts. Then we call the diagram a *natural double-pushout*.

$$\begin{tikzcd}
L \arrow[d] \arrow[dr, phantom, "\text{(NPO)}"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{(NPO)}"] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

Figure 2.5: A natural double-pushout.

An important advantage of natural double-pushouts is their ability to facilitate node relabelling, as discussed in Subsection 2.4.4. This method of relabelling nodes using pushouts is also presented in [HP02]. Since node labels and rootedness are modelled in a similar way, the same technique can also be applied to modify rootedness (see [CRP20] for more details).

2.2.2 Derivations

We leverage natural double-pushouts, as shown in Figure 2.5, to apply a rule $\langle L \hookleftarrow K \hookrightarrow R\rangle$ to a graph G, which results in graph H. It is important to note that when applying a rule, we want an injective morphism $g : L \to G$ that matches L with part of G. To achieve this, we use a construct called *direct derivation* (an example using the rule from Figure 2.3 is given in Figure 2.6).

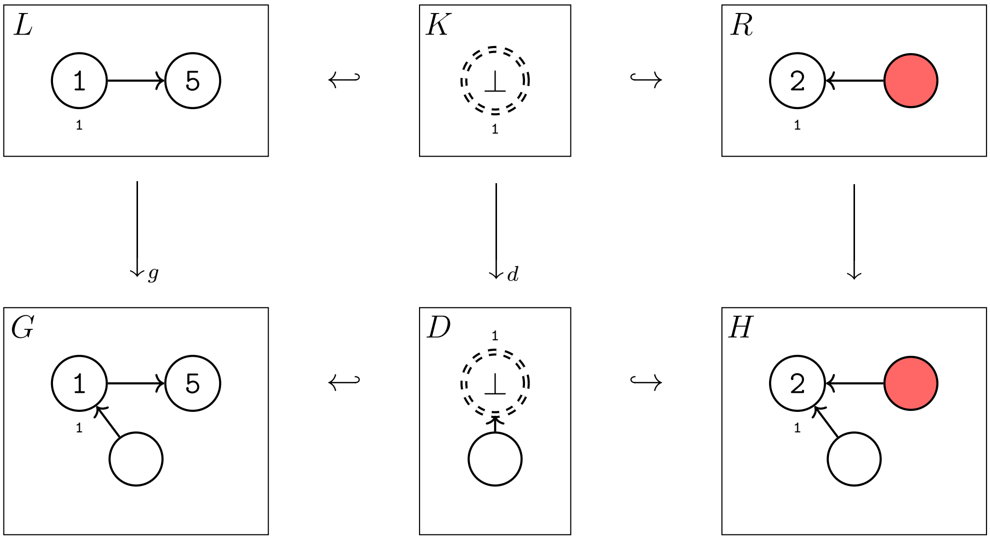


Figure 2.6: An example of a direct derivation.

Definition 2.2.3 (Direct Derivation, Derivation)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R\rangle$ be a rule, G a totally labelled graph, and $L \to G$ an injective morphism. Suppose D and H are graphs such that the squares in Figure 2.5 form a natural double-pushout. Then $G \Rightarrow_r H$ is called a *direct derivation*. In this thesis, we write $G \Rightarrow H$ when the specific rule is either irrelevant or can be unambiguously inferred from the context.

The *derivation* relation $\Rightarrow^*$ is defined as the reflexive (up to isomorphism) and transitive closure of $\Rightarrow$. In other words, $G \Rightarrow^* H$ holds if either:

- $G \cong H$ (isomorphic graphs), or
- there exists a finite sequence of direct derivations $G \Rightarrow \cdots \Rightarrow H$.

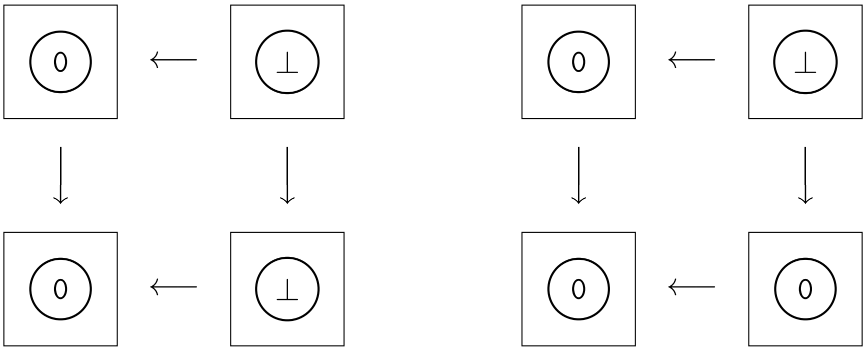


FIGURE 2.7: Two different pushout complements.

Direct derivation must exist and be unique up to isomorphism. This uniqueness is partly ensured by the fact that the double-pushout is natural, which guarantees the uniqueness of the pushout complement. Figure 2.7 illustrates two different pushout complements, where only the pushout on the left is natural.

In a direct derivation, L is totally labelled, while K is completely unlabelled and has no edges. The natural aspect of the double-pushout forces nodes in $d(K) \subseteq D$ to be unlabelled.

It is important to note that while nodes without labels in K are actually unlabelled, nodes without labels in L, G, R, and H are treated as having the *empty label*. For more details, a precise definition of labels is provided in Subsection 2.4.1. In D, unlabelled nodes in $d(K) \subseteq D$ are truly unlabelled, and any other unlabelled nodes are considered to have the empty label. This distinction does not apply to edges; that is because K contains no edges by the definition of a rule, so D can adopt the edge labels of G without violating the double-pushout property.

2.3 Constructing a Direct Derivation

The existence of a direct derivation can be demonstrated simply by constructing it, as outlined in [HP02]. However, a key prerequisite for this construction is the *dangling condition*.

When a rule deletes a node from G, any edges incident to that node may be left without a source or target (dangling), which ultimately produces an invalid graph. To circumvent this, the dangling condition requires that nodes deleted by the rule (those not included in the interface K) must not have any incident edges in G.

Definition 2.3.1 (Dangling Condition)**.** Given a rule $\langle L \hookleftarrow K \hookrightarrow R \rangle$ and a graph G, an injective morphism $g : L \to G$ satisfies the *dangling condition* if no edge in $E_G - g_E(E_L)$ is incident to a node in $g_V(V_L - V_K)$.

A direct derivation is constructed through the following steps in order:

1. The graph D is constructed by removing nodes and edges from G that are part of the left-hand side L but not included in the interface K. Nodes that also belong to K are stripped of their labels, and their rootedness is assigned as undefined.
2. The graph H is formed by merging D and R along their common parts derived from K. During this merging, nodes take their labels and rootedness from R given that their counterparts from K and D are unlabelled and lack rootedness.

More on the properties of direct derivations and gluings are given in [Cou23].

2.4 Attributed Graph Transformation

In graph programming with GP 2, it is desirable for rules to be more expressive than the conventional graph transformation rules discussed previously. To achieve this, rules are extended to support variables in labels and the inclusion of application conditions while we retain double-pushouts.

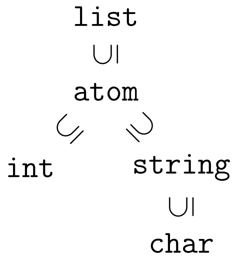


FIGURE 2.8: GP 2 subtype hierarchy.

2.4.1 Labels

The labels we use follow the subtype hierarchy shown in Figure 2.8. They include the following types:

- single characters (`char`);
- sequences of characters (`string`);
- integers (`int`);
- either strings or integers (`atom`); and
- lists of atoms (`list`).

Lists can be empty, and an atom is considered a list of length 1. Similarly, every character is treated as a string of length 1.

In GP 2, labels are specified using context-free string grammars. The abstract syntaxes of these grammars are shown in Figures 2.10 and 2.11. We refer to them as abstract because the grammars they define are inherently ambiguous; indeed, the expression `6/9+23` can be generated through several distinct derivations. This ambiguity is resolved for parsing by introducing parentheses to clarify the order of operations.

For parsing, we make use of concrete syntaxes that are unambiguous, as detailed in [Bak15]. The use of abstract syntaxes is advantageous because it simplifies the association of types to labels. We employ two distinct grammars: one for the host graphs, which represent the structures being modified, and another for rule graphs, which describe the modifications being made. Moreover, we use intermediate graphs for relabelling, as defined below.

Definition 2.4.1 (Intermediate, Host, and Rule Graph)**.** An *intermediate graph* is a graph over the set of labels generated by the grammar from Figure 2.10. A totally labelled intermediate graph is a *host graph.*

A *rule graph* is a totally labelled graph over the set of labels generated by the grammar in Figure 2.11.

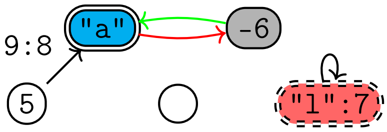


FIGURE 2.9: Example of a host graph.

The labels produced by the grammar in Figure 2.10 form a subset of those generated by the grammar in Figure 2.11. Therefore, host graphs can be viewed as a subset of rule graphs. Furthermore, by definition, host graphs are also a subset of intermediate graphs.

An example of how we represent a host graph in this thesis is shown in Figure 2.9. Again, recall that the grey mark is reserved for nodes and the dashed mark for edges. Label expressions are drawn inside their node or next to their edge. Items with the empty label are drawn without label (i.e. blank). Unlabelled nodes (undefined labelling function) are drawn using ⊥ in order to avoid confusion with empty labels. However, in future chapters, nodes will always have defined labelling and rootedness.

HostLabel ::= HostList [HostMark]
HostList ::= `empty` | HostAtom {‘:’ HostAtom}
HostAtom ::= HostInteger | HostString
HostInteger ::= [‘-’] Digit {Digit}
HostString ::= “‘ ’{Character}‘ ”’
HostChar ::= “‘ ’Character‘ ”’
HostMark ::= `red` | `green` | `blue` | `grey` | `dashed`
Character ::= Printable characters except for ‘"’
Digit ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

FIGURE 2.10: Abstract syntax of host graph labels.

RuleLabel ::= RuleList [RuleMark]
RuleList ::= LVar | `empty` | RuleAtom | RuleList ‘:’ RuleList
RuleAtom ::= AVar | RuleInteger | RuleString
RuleInteger ::= IVar | [‘-’] Digit {Digit} | ‘(’RuleInteger‘)’ |
RuleInteger (‘+’ | ‘-’ | ‘∗’ | ‘/’) RuleInteger |
(`indeg` | `outdeg`) ‘(’NodeID‘)’ |
`length` ‘(’(LVar | AVar | SVar)‘)’
RuleString ::= SVar | RuleChar | ‘“’{Character}‘”’ |
RuleString ‘`.`’ RuleString
RuleChar ::= CVar | ‘“’Character‘”’
RuleMark ::= `red` | `green` | `blue` | `grey` | `dashed` | `any`

FIGURE 2.11: Abstract syntax of rule graph labels.

In the grammars shown in Figures 2.10 and 2.11, terminals are denoted using a typewriter font and enclosed in single quotation marks. Nonterminals, on the other hand, begin with capital letters. Nonterminals that end with ‘Label’ serve as the starting nonterminals. The rule graph label syntax utilises the definitions of HostMark, Character, and Digit from the host graph label syntax. NodeID is a set of unique identifiers assigned to each node in a graph. Additional details about these can be found in the concrete syntax of GP 2 in [Bak15]. Sets ending in ‘Var’ correspond to typed variables (which we will discuss further shortly). For rule graph labels, we define types for strings of terminals by naming them accordingly:

- *(List) expression*: string of terminals generated by RuleList.
- *Atom expression*: expression generated by RuleAtom.
- *Integer expression*: atom expression generated by RuleInteger.
- *String expression*: atom expression generated by RuleString.
- *Character expression*: string expression generated by RuleChar.

In contrast to the labels of host graphs, the labels of rule graphs support more general expressions and variables, which are designed to correspond to host graph labels. The function `indeg` returns the indegree of a node (i.e. the number of incoming edges incident to it), while `outdeg` provides its outdegree (i.e. the number of outgoing edges incident to it). The `length` function returns the length of a variable (see Appendix A of [Bak15] for how variable lengths are defined in the language). The `any` mark matches any of the marks in a host graph, but not the absence of a mark. More precisely, `any` matches `red`, `green`, `blue`, `grey` (for nodes), or `dashed` (for edges).

2.4.2 Rule Schemata

Due to the presence of variables in rule schema labels, some ambiguity may arise. For example, when two string variables, `x` and `y`, are concatenated as `x.y`, multiple variable assignments may lead to the same resulting string. A similar issue occurs with lists.

Definition 2.4.2 (Simple Label/Expression)**.** We say a rule graph label is *simple* if its expression is simple. An expression is simple if it has the following properties:

- it contains no length, degree, or arithmetic operators, except for the unary minus (e.g. `-2`);
- it contains at most one list variable; and
- every string expression it contains has at most one string variable.

Definition 2.4.3 (Rule Schema)**.** A *rule (schema)* $\langle L \hookleftarrow K \hookrightarrow R\rangle$ is a rule where L and R are rule graphs, and K an intermediate graph such that:

- all labels in L are simple;
- all variables occurring in R must also occur in L; and
- all nodes in R with the `any` mark must be the image of an interface node whose image in L also has the `any` mark.

A *(conditional) rule (schema)* $\langle L \hookleftarrow K \hookrightarrow R, c\rangle$ is composed of a rule schema and an *(application) condition* c, which is generated by the grammar in Figure 2.13. Naturally, it is required that all variables appearing in c must also appear in L, as the latter is matched to the host graph.

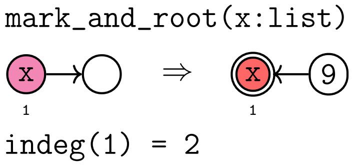


Figure 2.12: Example of a conditional rule.

An example of how a conditional rule schema is represented in this thesis can be found in Figure 2.12. The rule is named `mark_and_root`, with the list variable `x` declared immediately following the rule name. The condition is written directly underneath and specifies that node `1` must have exactly 2 incoming edges. Note that the nodes fed to the functions `indeg` and `outdeg` need not be in the interface.

The rule application condition is a Boolean statement depending on properties of the host graph. The left-hand side of the rule will only be matched if the condition evaluates to true. The functions `int`, `char`, `string`, and `atom` are *subtype predicates* used to check if a given variable belongs to a specific type.

Condition ::= (`int` | `char` | `string` | `atom`) '('(LVar | AVar | SVar)')'
| RuleList ('`=`' | '`!=`') RuleList
| RuleInteger ('>' | '>=' | '<' | '<=') RuleInteger
| `edge` '(' NodeID ',' NodeID [',' RuleLabel] ')'
| `not` Condition
| Condition (`and` | `or`) Condition
| '(' Condition ')'

FIGURE 2.13: Abstract syntax of conditions.

The `edge` function verifies whether an edge (possibly with a specified label) exists between two given nodes. Additionally, we have equality comparisons for list expressions (e.g. `x = 4:"a"`), inequality comparisons for integer expressions (e.g. `w != 7-z`), and logical operators (e.g. `w = 1 or w = 2`).

To apply a rule schema, we need a method for mapping rule graphs with variable labels to host graphs with constant labels. We accomplish this by using two mechanisms: *variable assignments*, which assign values to variables, and *premorphisms*, which map one graph to another while disregarding labels.

Definition 2.4.4 (Variable Assignment)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ be a conditional rule schema. Let LVar_r, AVar_r, IVar_r, SVar_r, and CVar_r be the set of variables used in r of the corresponding type. A variable assignment is a family of functions $\alpha = (\alpha_X)_{X\in\{\mathrm{L,A,I,S,C}\}}$, where $\alpha_L : \mathrm{LVar}_r \to \mathrm{HostList}$, $\alpha_A : \mathrm{AVar}_r \to \mathrm{HostAtom}$, $\alpha_I : \mathrm{IVar}_r \to \mathrm{HostInteger}$, $\alpha_S : \mathrm{SVar}_r \to \mathrm{HostString}$, and $\alpha_C : \mathrm{CVar}_r \to \mathrm{HostChar}$.

We often omit the subscript from α since only one function is applicable to a given variable.

Definition 2.4.5 (Graph Premorphism)**.** Let G and H be graphs. A *(graph) premorphism* $g : G \rightharpoonup H$ is a pair $g = \langle g_V, g_E\rangle$ of functions $g_V : V_G \to V_H$ and $g_E : E_G \to E_H$ such that the following holds:

- *Preservation of sources and targets*: for each edge $e \in E_G$, we have $g_V(s_G(e)) = s_H(g_E(e))$ and $g_V(t_G(e)) = t_H(g_E(e))$.
- *Preservation of defined rootedness*: for each node of defined rootedness $v \in \mathrm{dom}(p_G) \subseteq V_G$, we have $g_V(v) \in \mathrm{dom}(p_H)$ and $p_G(v) = p_H(g_V(v))$.

We can now instantiate a rule schema. This means that variables and `any` marks are substituted according to a variable assignment and a premorphism.

Definition 2.4.6 (Rule Schema Instance)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ be a conditional rule schema, $g : L \rightharpoonup G$ an injective premorphism, where G is a host graph, and α a variable assignment function. A *rule schema instance* is a tuple $r^{g,\alpha} = \langle L^{g,\alpha} \hookleftarrow K \hookrightarrow R^{g,\alpha}, c^{g,\alpha}\rangle$, where $L^{g,\alpha}$, $R^{g,\alpha}$, and $c^{g,\alpha}$ are obtained from L, R, and c by substituting variables with their assignments, replacing the `any` mark with the corresponding mark in G, and evaluating any arithmetic, list, and logic operations, as well as statements about the host graph G (i.e. `indeg`, `outdeg`, and `edge`), all with respect to the mapping g.

Since all variables have been substituted and all operations evaluated, $L^{g,\alpha}$ and $R^{g,\alpha}$ can be viewed as either host, rule, or intermediate graphs.

2.4.3 Rule Schema Application

To apply a conditional rule schema $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ to a host graph G, we must find an injective premorphism $g : L \rightharpoonup G$ and a variable assignment α such that the Boolean expression $c^{g,\alpha}$ is true, and the labels of L correspond to those of G. In other words, this means there exists a morphism $g : L^{g,\alpha} \to G$, where $L^{g,\alpha}$ is obtained from the premorphism g by replacing the nodes and edges of L with those of the instance $L^{g,\alpha}$. We use the same name g for both the morphism and the premorphism, as they map nodes and edges in the same manner.

Additionally, for the rule to be applicable, g must satisfy the dangling condition to maintain the structural integrity of the host graph; otherwise, edges may be left dangling (i.e. without adjacent nodes).

We now formally define the notion of *matches*.

Definition 2.4.7 (Match)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ be a conditional rule schema, G a host graph, $g : L \rightharpoonup G$ an injective premorphism, and α a variable assignment. We say $\langle g, \alpha\rangle$ is a *match* for r in G if the following holds:

- $g : L^{g,\alpha} \to G$ is an injective morphism that satisfies the dangling condition; and
- $c^{g,\alpha}$ evaluates to true.

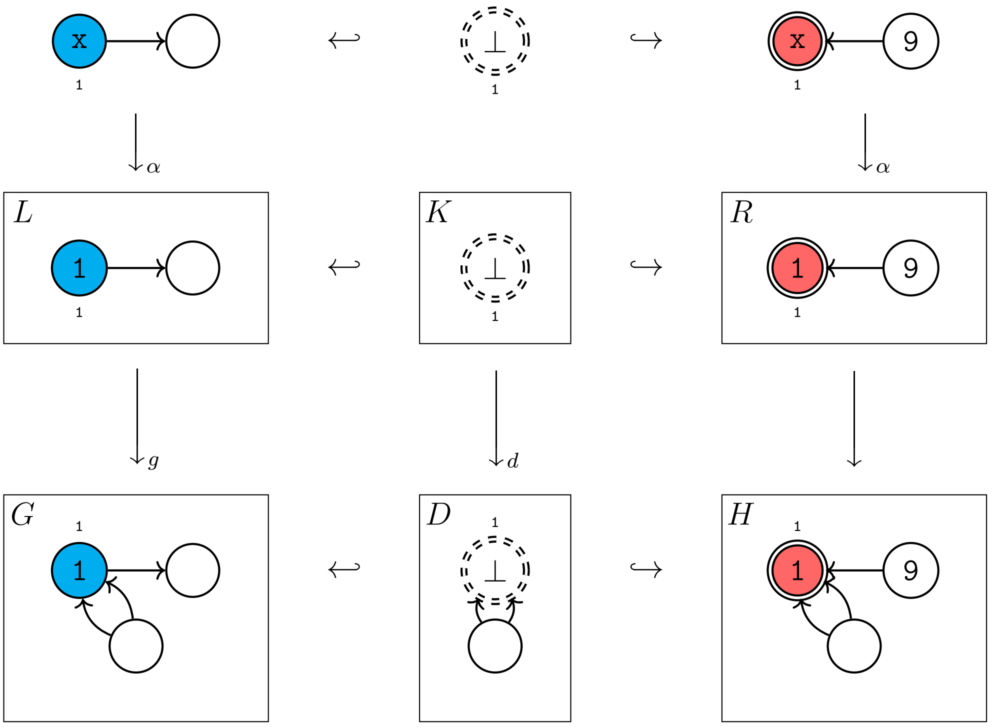


FIGURE 2.14: An example of a rule application.

We then apply the rule to G using natural double-pushouts from the previous section, along with the instantiated rule, the injective morphism g, and graph G. An example of this rule application, based the rule `mark_and_root` from Figure 2.12 is shown in Figure 2.14.

Definition 2.4.8 (Derivation Using Attributed Graphs)**.** Let $r = \langle L \hookleftarrow K \hookrightarrow R, c\rangle$ be a conditional rule schema, G a host graph, and $\langle g, \alpha\rangle$ a match for r in G. We say $G \Rightarrow_r H$ if $G \Rightarrow_{r^{g,\alpha}} H$ (in the way described in the previous section with a natural double-pushout).

We define *derivation* $\Rightarrow^*$ using attributed graphs analogously to derivation in Definition 2.2.3 as the reflexive (with respect to isomorphism) and transitive closure of $\Rightarrow$.

2.4.4 Changing Labels and Rootedness

In traditional double-pushout graph transformation, only fully labelled graphs are used. Since morphisms preserve labels, this means that node labels cannot be changed. As stated in Subsection 2.2.2, to allow for relabelling, we introduce partially labelled graphs while still requiring the double-pushout to be natural; this would allow for a unique direct derivation [HP02].

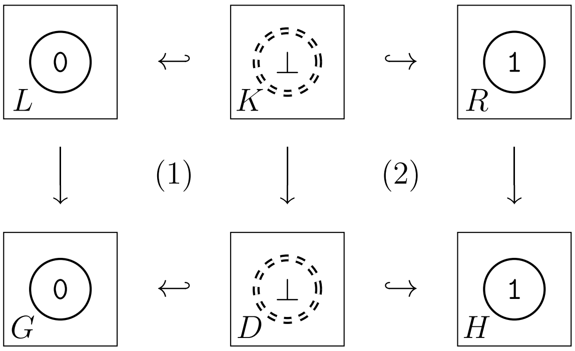


FIGURE 2.15: A derivation labelling a node.

Figure 2.15 depicts a direct derivation that relabels a node. We assume L and G are totally labelled, while K is totally unlabelled. If we only require (1) to be a pushout, the pushout complement D could be either labelled or unlabelled. However, we require (1) to be a pullback as well, which forces D to be unlabelled. Under this condition, H can be uniquely constructed, assuming that (2) is a pushout.

Figure 2.16 illustrates a derivation that roots an unrooted node. Since morphisms like g preserve defined rootedness, we cannot match an unrooted L with a rooted G. This restriction prevents unexpected behaviour, such as a rooted node remaining rooted instead of rooting an unrooted node.

Just as with labels, the undefined rootedness of K and the unrooted nature of both L and G imply that if (1) is a pullback, D must also have undefined rootedness. Under this condition, H can be uniquely derived, provided that (2) is a pushout.

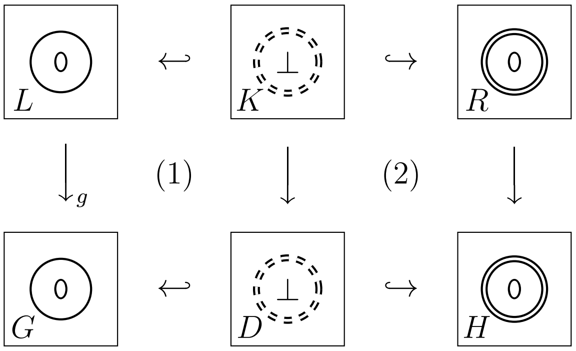


FIGURE 2.16: A derivation rooting a node.

The correct handling of roots depends on several factors: morphisms that preserve rootedness, the ability for nodes to have undefined rootedness, and the double-pushout being natural [CRP20].

2.5 GP 2 Programs

Having established rules as the core operations for transforming graphs in GP 2, we now turn to the definition of programs. The primary components of GP 2 programs are loops and conditional branching statements, which enable the implementation of entire graph algorithms.

2.5.1 Syntax

Figure 2.17 presents the abstract syntax of a GP 2 program in Extended Backus-Naur Form (EBNF). This abstract syntax is designed to write programs in human-readable way. However, programs provided to the compiler use the concrete syntax, which is designed for compiler interpretation. The concrete syntax can be found in Appendix A of [Bak15].

Prog ::= Decl {Decl}
Decl ::= RuleDecl | ProcDecl | MainDecl
ProcDecl ::= ProcId '=' ['[' LocalDecl ']'] ComSeq
LocalDecl ::= (RuleDecl | ProcDecl) {LocalDecl}
MainDecl ::= `Main` '=' ComSeq
ComSeq ::= Com {';' Com}
Com ::= RuleSetCall | ProcCall
| `if` ComSeq `then` ComSeq [`else` ComSeq]
| `try` ComSeq [`then` ComSeq] [`else` ComSeq]
| ComSeq '!' | ComSeq `or` ComSeq
| '(' ComSeq ')' | `break` | `skip` | `fail`
RuleSetCall ::= RuleId | '{' [RuleId {',' RuleId}] '}'
ProcCall ::= ProcId

FIGURE 2.17: Abstract syntax of GP 2 programs.

A GP 2 program is composed of various types of declarations. The MainDecl consists of a sequence of commands executed when the program runs. This sequence may include procedures, which are predefined command sequences defined in ProcedureDecl. When a procedure is invoked by its name, the corresponding command sequence is executed. Procedures may also include local declarations, which are rules and procedures within the scope of that procedure alone. Rules are defined in RuleDecl, although their textual representation is omitted here, as this thesis focuses on the graphical representation of rules, as shown in the previous section (like Figure 2.3). Each rule is associated with a unique name, referred to as RuleID.

Command sequences (often simply dubbed "sequences" in this thesis) consist of commands separated by semicolons. Commands can be either an `if` statement (with an optional `else` branch), a `try` statement (with optional `then` and `else` branches), a command sequence in brackets, a loop denoted by '!', an `or` statement, or a simple command.

2.5.2 Command Sequences

As shown in the syntax, both GP 2 programs and procedures are composed of command sequences. When a command sequence is applied to a host graph, there are three possible outcomes: it can produce a new host graph (which may have been modified by rules), it can result in failure (if a rule is inapplicable or the `fail` command is executed), or it may simply not terminate.

Calling a rule set $\{r_1, \ldots, r_n\}$ results in the non-deterministic application of one of the rules. A rule is applied if its left-hand graph matches a subgraph of the host graph, and both the dangling condition and the application condition of the rule are satisfied. If none of the rules can be applied to the host graph, the call fails. In this thesis, we also use the term *invocation* to refer to calls.

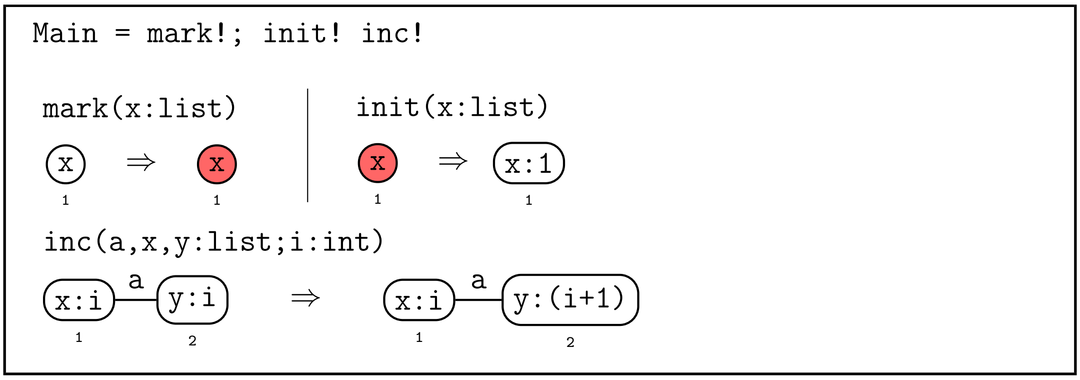
The command `if` C `then` P `else` Q is executed on a host graph G by first applying C to G. If C produces a graph (i.e. C is applicable), then P is executed on the original graph G; if C fails, Q is executed instead. The `try` command works in a similar way, but in this case, the difference is that P is actually executed on the result of C's execution if C succeeds. That is, the command `try` C `then` P `else` Q executed on a host graph G results in the host graph being modified by C should it apply.

The loop command $P!$ executes the body P repeatedly until it fails. When this is the case, $P!$ terminates with the graph on which the body is entered for the last time. The `break` command inside a loop terminates that loop with the current graph and transfers control to the command following the loop.

A program P or Q non-deterministically chooses to execute either P or Q, which can be simulated by a rule set call and the other commands [Plu12]. The commands skip and fail can also be expressed by the other commands: `skip` is equivalent to an application of the rule $\emptyset \Rightarrow \emptyset$ (where $\emptyset$ is the empty graph) and `fail` is equivalent to an application of $\{\}$ (the empty rule set).

2.5.3 Example Program

In this subsection, we consider an example program. The program in Figure 2.18 shows a GP 2 program that produces a node colouring, i.e. every node gets assigned an integer (each integer represents a different colour) such that no two adjacent nodes share the same integer.



```
Main = mark!; init! inc!
```

Figure 2.18: A GP 2 program producing a node colouring.

In this thesis, we separate the rules sharing a common row using a vertical black line for clarity. The loop `mark!` repeatedly applies the rule `mark` until no further applications are possible, marking each node as red. A red node indicates that it has not yet been assigned an integer. The loop `init!` appends the integer `1` to the label of every node. Finally, the loop `inc!` ensures that as long as there are adjacent nodes with the same integer, one of the integers gets incremented, thereby producing a colouring. Note that the edges in the rule `inc` are bidirectional, meaning that their direction is irrelevant while matching then with edges in the host graph, provided nodes `1` and `2` are still endpoints.

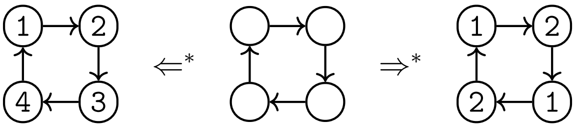


Figure 2.19: Two possible results from applying the colouring program.

Recall that GP 2 programs are inherently non-deterministic, particularly when it comes to rule matching. The resulting colouring can vary depending on how the rules called within `inc!` match and the order in which they are applied. This non-determinism is illustrated in Figure 2.18, where two different colourings are produced due to the different locations where the rule `inc` matched.

Chapter 3

The GP 2 Compiler

In this chapter, we introduce the GP 2-to-C compiler (Figure 2.1), a tool that takes a GP 2 program and an input graph specified in the language's syntax, and produces equivalent C code. The runtime system (separate from the compiler) then parses the input file issued by the user using the C files generated by the compiler [CRP20; Bak15].

While GP 2 is, in principle, non-deterministic, as demonstrated in Subsection 2.5.3 with the colouring program, the implemented compiler operates deterministically. Given a specific input graph and program, the compiler will consistently produce the same output graph. This deterministic behaviour is a deliberate design choice aimed at improving computational efficiency [Cou23; Bak15].

The chapter is organised as follows. First, we examine the mechanisms used by the compiler for graph matching, along with their assumed time complexities. Next, we address the core issue of the thesis: identifying two implementation problems that caused certain GP 2 graph programs to underperform relative to their imperative counterparts. Finally, we propose solutions for each issue, which will be exploited in Chapter 4 to achieve faster programs.

The majority of this chapter is derived from earlier work presented in [IAP24] and [IAP25], for which the author of this thesis was the main author.

3.1 The Compiler

3.1.1 The Graph Data Structure

As explained in [CRP20], the GP 2 compiler organises the storage of the host graph as follows:

- A global array stores all nodes in the host graph, including those that have been deleted.

- A global array stores all edges in the host graph, including those that have been deleted.

- A linked list, called `NodeList`, where each entry consists of two pointers: one pointing to an existing node (i.e. a node present in the host graph) in the array of nodes, and the other pointing to the next element in `NodeList`.

The use of linked lists, implemented by Plump, Campbell, Romö and Plump [CRP20], improves the efficiency of deletions during the execution of GP 2 programs. Deleted nodes leave so-called holes in the node array, and without proper handling, iterating through the array to find an existing node could become a linear-time operation. The introduction of linked lists allows for the compiler to manage deletions efficiently. Indeed, when a node is deleted, its corresponding link is also removed from the linked list. That ensures that searching for nodes in the array remains unaffected by holes.

Each node stores its outdegree, indegree, label (recall that labels include the mark) as well as two linked lists: one of pointers to edges in the array of edges that are incoming from the node, and one for outgoing edges.

3.1.2 Finding a Match for a Rule

The *search plan* [Dör95] is the underlying algorithm in the generated GP 2 compiled code for finding matches in the host graph [Bak15], inherited from GP 1 [MP08].

In essence, at execution time, the algorithm breaks down the process of matching a rule's left-hand side into a sequence of primitive matching operations. It then executes them sequentially, one by one, to find a match in the host graph (or determine that none exists). As described in [Bak15], search plans are constructed using an undirected depth-first search on the left-hand sides of rules. When a node or edge is visited, the corresponding matching operation is added to the search plan. Each search begins at a root node, if one exists; otherwise, an unvisited non-root node is selected. These operations are:

- match an unrooted node;
- match a root node;
- given a matched edge, match its target;
- given a matched edge, match its source;
- given a bidirectional edge, match one of its incident nodes;
- match an edge;
- given a matched node, match one of its outgoing edges;
- given a matched node, match one of its incoming edges; and
- given a matched node, match one of its looping edges.

The algorithm identifies graph items (i.e. nodes and edges) that meet the search plan's objectives by flagging them. Flagging does not happen until the items' labels are verified (after assignment).

3.2 First Problem: Unbounded-Degree Graphs

Previous versions of non-destructive[1] GP 2 algorithms which were based on depth-first search showed a linear-time complexity on graph classes of bounded degree but a non-linear runtime on graph classes of unbounded degree [CCP22].

For example, consider the program `is-connected` in Figure 3.1 which checks whether a given graph consists of exactly one connected component. Input graphs have arbitrary node and edge labels of type `list`, unmarked edges and grey nodes. The program fails on a graph if and only if the graph is disconnected.

The rule `init` selects an arbitrary grey node as a root (if the input graph is non-empty) and the loop `DFS!` performs a depth-first search of the connected component of the node chosen by `init`. The rule `forward` marks each newly visited node blue, and `back` unmarks it once it is processed. The procedure `DFS` terminates when `back` fails to match, indicating that the search is complete. The rule `match` checks whether a grey node still exists in the graph following the execution of `DFS!`. This is the case if and only if the input graph contains more than one connected component. In this situation, the program invokes the command `fail`; otherwise, it terminates by returning the graph resulting from the depth-first search.

```
Main  = try init then (DFS!; Check)
DFS   = forward!; try back else break
Check = if match then fail
```

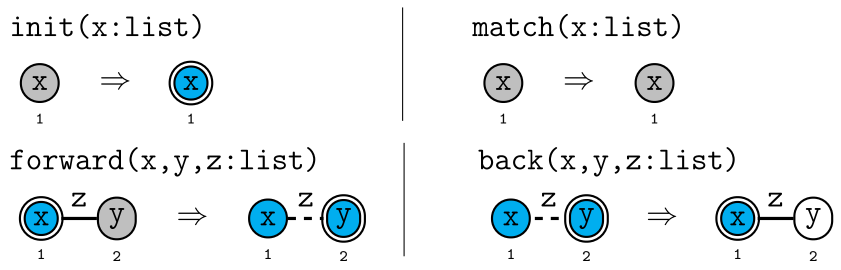


FIGURE 3.1: The old program `is-connected`.

It can be shown that the program `is-connected` runs in linear time on classes of graphs with bounded node degree [CCP22]. However, as the following example shows, the program may require non-linear time on unbounded-degree graph classes. Figure 3.2 shows an execution of `is-connected` on a star graph with 8 edges. The numbers below the graphs show the ranges of attempts that the matching algorithm may perform. For instance, in the second graph of the top row, either a match is found immediately among the edges that connect the central node with the grey nodes, or the dashed edge is unsuccessfully tried first.

In order to find a match for the rule `forward`, the matching algorithm considers, in the worst case, every edge incident with the root. When the node

---

[1] A program is non-destructive if it is still possible to recover the input graph from the output graph.

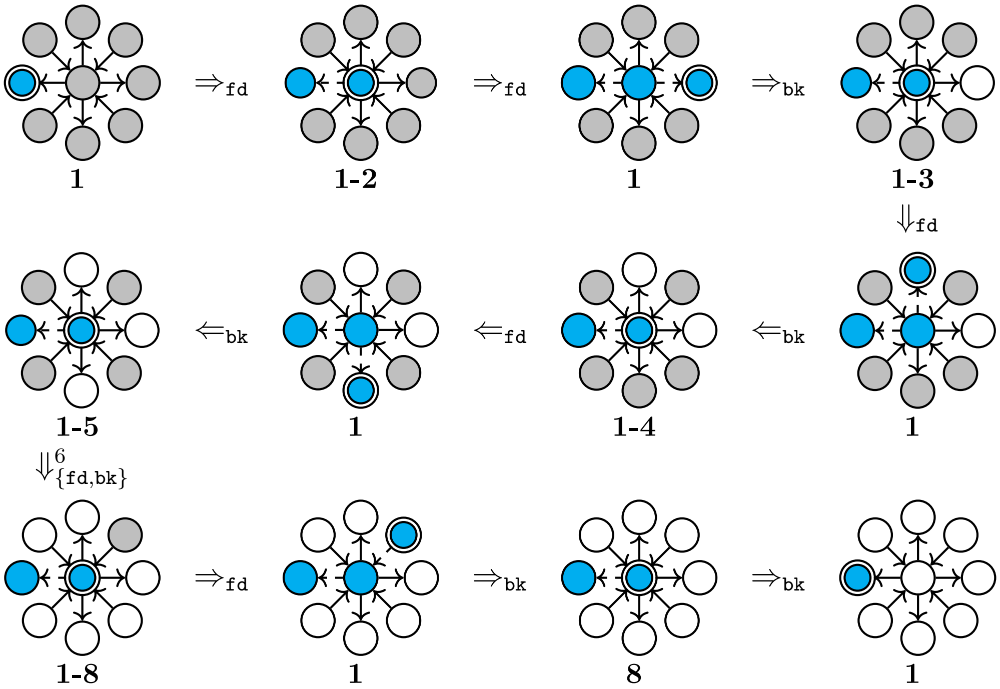


Figure 3.2: Matching attempts with the `forward` rule. `fd` and `bk` denote `forward` and `back`, respectively.

central to the graph is rooted and the rule `forward` is called, the matching algorithm, as described in Subsection 3.1.2, may first attempt a match with the dashed edge, and then all edges incident with an unmarked node. Therefore, the maximum number of matching attempts for `forward` grows as the root moves back to the central node. As can be seen from this example, the worst-case complexity of matching `forward` throughout the program's execution in the general case for this class of graph is $2|E| + \sum_{i=1}^{|E|} i = \mathcal{O}(|E|^2)$, where E is the set of edges.

3.3 First Enhancement

To address the problem described in Section 3.2, we amended the implementation of the data structure in the GP 2 compiler described in [CRP20]. We refer to the version prior to the changes as the *2020 compiler*. We call the version introduced in this thesis the *new compiler*[2].

Recall that the 2020 compiler stores the host graph's structure as one linked list pointing at every node in the host graph in the global array of nodes, with each node storing two additional linked lists: one for incoming edges and one for outgoing edges. When iterating through edge lists to find a particular match for a rule edge, the 2020 compiler had to traverse through edges with marks incompatible with that of the rule edge. This resulted in performance issues, especially if nodes could be incident to an unbounded number of edges with marks incompatible with the edge to be matched. For example, consider the

---

[2] Available at: `https://github.com/UoYCS-plasma/GP2`.

	in	out	loop
unmarked	. . .	. . .	. . .
dashed	. . .	. . .	. . .
red	. . .	. . .	. . .
green	. . .	. . .	. . .
blue	. . .	. . .	. . .

FIGURE 3.3: Two-dimensional array of linked lists of edges.

rule `move` from Figure 4.16. Initially, the matching algorithm matches node `1` from the interface with a root node in the host graph. Subsequently, it iterates through the node's edge lists to locate a match for the red edge. In the 2020 compiler, all edges incident to this node were stored within two lists, one for each orientation, irrespective of their marks. However, if the node were incident to a growing number of unmatchable edges (because of mark changes), the matching algorithm would face, in the worst case, a growing number of iterations through the edge lists to find a single red edge.

When considering a match for a rule edge, host edges with incorrect orientation and incompatible marks do not match; thus, the matching algorithm need not iterate through them. By re-organising edges into homogeneous linked lists as array entries based on their marks and orientations, the matching algorithm can selectively consider linked lists of edges of correct orientation and mark. More precisely, in the new compiler, we update the graph structure generated by the 2020 compiler by replacing the two linked lists with a two-dimensional array. Each element of the array stores a linked list containing edges of a particular mark and orientation. We also consider looping edges to be a distinct type of orientation, separate from non-looping outgoing and incoming edges. The 2D array, as depicted in Figure 3.3, therefore consists of 5 rows (unmarked, dashed, red, blue, green) and 3 columns (incoming, outgoing, loop), totalling 15 cells, each one storing a single linked list.

3.4 Second Problem: Finding Nodes in Constant Time

Consider the program is-discrete from Figure 3.4. The program fails if and only if the input graph is not discrete; that is, if it contains at least one edge (including looping-edges). Here, we assume that the input graph is unmarked (i.e. nodes and edges are not coloured, and edges are not dashed). The program is non-destructive in that the output graph is isomorphic to the input graph up to node marks, should it not fail. `is-discrete` is composed of a looping procedure followed by a test. The rule `mark` in the looping procedure marks and roots an arbitrary unmarked node in the host graph, and `isolated` checks whether the degree of the node rooted by `mark` is 0. Notice that the right-hand side node of the rule `isolated` is not in the interface; hence, for it to match, it must satisfy the dangling condition (i.e. have no edge outside the rule incident to it). If `isolated` does not apply after an application of `mark`, a node whose degree is non-zero was found and the looping procedure breaks. The rule `root` checks if a root exists in the host graph, which can happen if and only if `isolated` failed to apply.

```
Main = (mark; try isolated else break)!; if root then fail
```

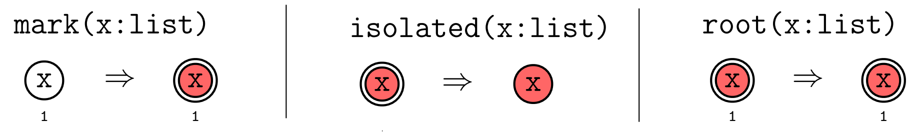


FIGURE 3.4: The non-destructive program `is-discrete`.

The 2020 compiler matches the rule `mark` with a complexity of $\mathcal{O}(n)$, with n being the number of nodes in the host graph at the application of `mark`. Indeed, in order to find an unmarked node, this version of the compiler has to iterate through a unique linked list consisting of all nodes in the host graph. As a result, since the looping procedure can be invoked at most n times, the overall complexity of `is-concrete` is $\mathcal{O}(n^2)$ under that implementation, as empirically evidenced in Figure 3.5.

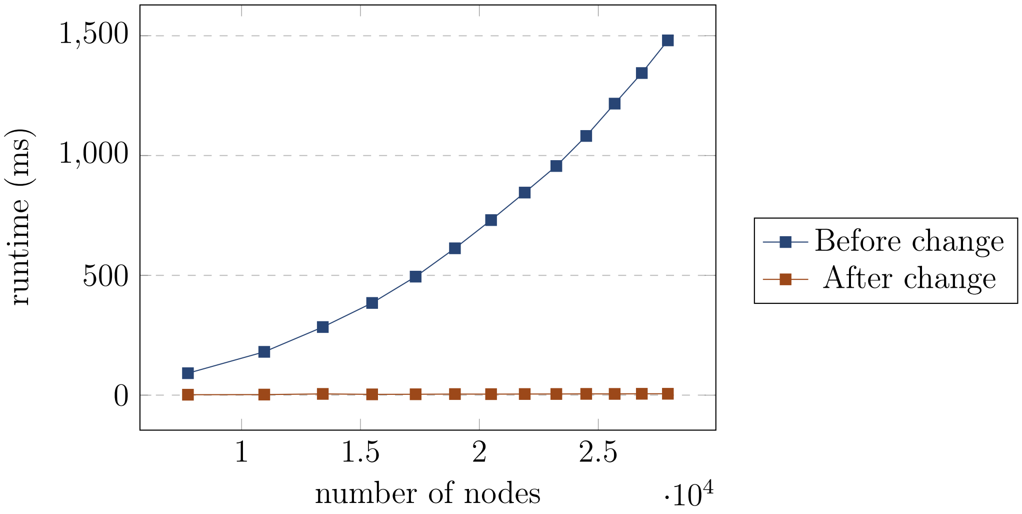


FIGURE 3.5: Measured performance of the program `is-discrete` on discrete graphs under the previous and the new compilers.

3.5 Second Enhancement

To address the problem described in Section 3.4, we again modified the internal graph data structure the GP 2 compiler generates.

Recall that the 2020 compiler stores the host's graph data structure as one linked list containing every node in the graph. One major problem with the

unmarked	. . .
grey	. . .
red	. . .
green	. . .
blue	. . .

FIGURE 3.6: Array of linked lists of nodes.

structure used to store nodes is the unnecessary lookups the matching algorithm carries out to find a match for a node in the host graph, especially when there exist other nodes whose marks could not match. For instance, in order to match node `1` of the rule `mark` in `is-discrete` (Figure 3.4), the matching algorithm would have to iterate through a single linked list consisting of every node in the host graph, including those of incompatible marks (such as red-marked nodes). Hence, in the case where every node in the host graph were red beside an unmarked one, the matching algorithm could, in the worst case, look up (i.e. attempt matching with) every red node in the host graph until it finally reaches the unmarked node. It is clear that under such a model, matching `mark` is linear to the number of nodes, provided the number of non-red nodes in the class under which the host graph falls is not bound to a constant.

A solution presented in this thesis, supported by the new compiler, is the storage of nodes in several linked lists, each consisting of nodes of a particular, distinct mark (unmarked nodes are stored in a distinct linked list). Each linked list is stored in a unique cell of a global one-dimensional array, allowing for constant access to the first element of each linked list. This model permits the rule `mark` to match in constant time, since the matching algorithm need not iterate through nodes of incompatible marks. Indeed, the matching algorithm directly inspects the linked list consisting of all red nodes in the host graph. Since the rule `mark` allows for any red node of arbitrary label to be matched, the first element found in the linked list by the matching algorithm matches successfully. As a result of these changes, the time complexity of the program `is-discrete` under the new compiler is reduced down to $\mathcal{O}(n)$. Figure 3.5 highlights the difference in runtimes of the program `is-discrete` run under both compilers.

Chapter 4

Case Studies

While the compiler changes introduced in Chapter 3 allow for faster access and computation of GP 2 programs, a new approach to programming has to be considered to achieve better runtimes on classical graph programs. Despite the changes, the program `is-connected` from Figure 3.1 would still run in quadratic time on graph classes on unbounded degree, given that the rule `forward` would require linear time to match as the root node may be adjacent to an unbounded number of blue nodes.

In this chapter, we explore several graph algorithms and provide implementations whose time complexity matches that of conventional imperative programming languages. Section 4.1 introduces a linear-time algorithm for checking whether a graph is connected. Then, building on this, Section 4.2 presents a program for two-colouring graphs (or determining if a graph is not bipartite) using techniques from the previous section. Section 4.3 describes a GP 2 program that preserves the structure of the input graph while checking for the presence of directed cycles. Finally, Section 4.4 focuses on an $\mathcal{O}(nm)$ implementation of the Bellman-Ford algorithm, a single-source shortest-path algorithm. To the best of our knowledge, this is the first graph programming language implementation of the Bellman-Ford algorithm to achieve conventional time complexity alongside a rigorous complexity and correctness analysis.

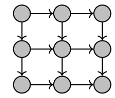


FIGURE 4.1: Grid graph.

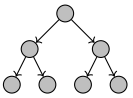


FIGURE 4.2: Binary tree.

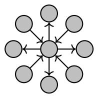


FIGURE 4.3: Star graph.

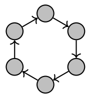


FIGURE 4.4: Cycle graph.

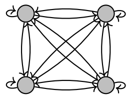


FIGURE 4.5: Complete graph.

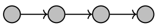


FIGURE 4.6: Linked list.

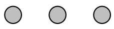


FIGURE 4.7: Discrete graph.

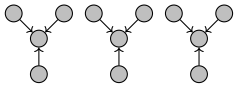


FIGURE 4.8: k k-star graphs.

Each section is structured as follows. First, a brief introduction to the algorithm and its objectives is given. Second, a proof of correctness is provided given certain specifications. Finally, a proof of (time) complexity is offered alongside empirical benchmarks to support the analysis.

To reason about programs, it is crucial to lay down assumptions on the complexity of certain elementary operations. Figure 4.9 showcases the complexity assumptions of the basic procedures of the search plan, adapted from [CCP22]. The grey rows indicate existing procedures updated by the changes introduced

Procedure	Description	Complexity
`alreadyMatched`	Test if the given item has been matched in the host graph.	$O(1)$
`clearMatched`	Clear the `is matched` flag for a given item.	$O(1)$
`setMatched`	Set the `is matched` flag for a given item.	$O(1)$
`firstHostNode(m)`	**Fetch the first node of mark `m` in the host graph.**	$O(1)$
`nextHostNode(m)`	**Given a node of mark `m`, fetch the next node of mark `m` in the host graph.**	$O(1)$
`firstHostRootNode`	Fetch the first root node in the host graph.	$O(1)$
`nextHostRootNode`	Given a root node, fetch the next root node in the host graph.	$O(1)$
`firstInEdge(m)`	**Given a node, fetch the first incoming edge of mark `m`.**	$O(1)$
`nextInEdge(m)`	**Given a node and an edge of mark `m`, fetch the next incoming edge of mark `m`.**	$O(1)$
`firstOutEdge(m)`	**Given a node, fetch the first outgoing edge of mark `m`.**	$O(1)$
`nextOutEdge(m)`	**Given a node and an edge of mark `m`, fetch the next outgoing edge of mark `m`.**	$O(1)$
`firstLoop(m)`	**Given a node, fetch the first loop edge of mark `m`.**	$O(1)$
`nextLoop(m)`	**Given a node and an edge of mark `m`, fetch the next loop edge of mark `m`.**	$O(1)$
`getInDegree`	Given a node, fetch its incoming degree.	$O(1)$
`getOutDegree`	Given a node, fetch its outgoing degree.	$O(1)$
`getMark`	Given a node or edge, fetch its mark.	$O(1)$
`isRooted`	Given a node, determine if it is rooted.	$O(1)$
`getSource`	Given an edge, fetch the source node.	$O(1)$
`getTarget`	Given an edge, fetch the target node.	$O(1)$
`parseInputGraph`	Parse and load the input graph into memory: the host graph.	$O(n)$
`printHostGraph`	Write the current host graph state as output.	$O(n)$

FIGURE 4.9: Updated runtime complexity assumptions. Modified procedures are highlighted in grey. n is the size of the input.

in this thesis. The empirical evidence collated from various case studies showcased later in the thesis corroborates the complexity assumptions implied by the changes brought to the new compiler.

The majority of this chapter (especially Section 4.2) is derived from earlier work presented in [IAP24] and an unpublished manuscript, for which the author of this thesis was the main author.

4.1 Checking a Graph's Connectedness

Checking for the connectedness of a graph is a fundamental application of depth-first search (DFS). In this section, we introduce the program `is-connected`[1] (Figure 4.10) determines whether an input graph is connected, in that each pair of nodes are connected by a sequence of edges of arbitrary orientation. The program fails if and only if the input graph is not connected. This can be achieved by conducting a DFS from an arbitrarily chosen node while marking newly visited nodes. Since the DFS cannot propagate beyond the connected component it started in, the presence of an unmarked node following the DFS traversal indicates that the host graph is not connected. This program improves implementations of the same problem in [Bak15; CCP22] (shown in Figure 3.1) by achieving linear-time complexity on arbitrary input graphs (including star graphs).

4.1.1 Proof of Correctness

Let us examine the correctness of `is-connected`[2].

---

[1] The concrete syntax of the program available at: `https://gist.github.com/ismaili-ziad/7b419a12a8d156abf5ad8ac8c352713a`.

[2] Magenta represents the *wildcard* mark.

```
Main    = try init then (DFS!; Check)
DFS     = FORWARD!; try back else break
FORWARD = next_edge; {move, ignore}
Check   = if match then fail
```

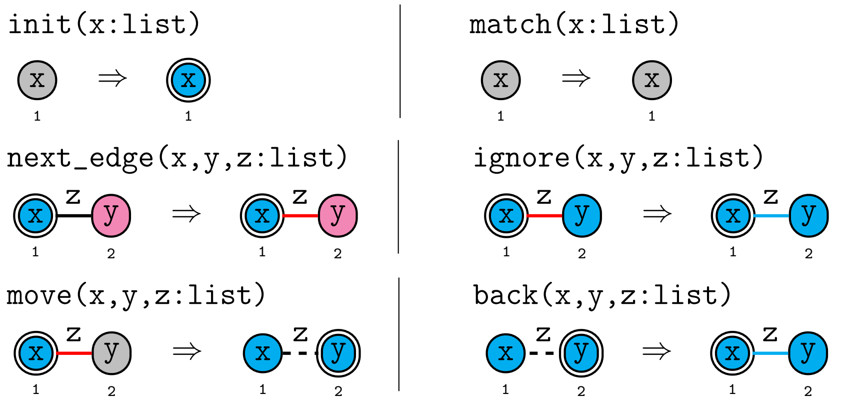


FIGURE 4.10: The (new) program `is-connected`.

First of all, we establish what we mean by an input graph in the context of the `is-connected` program.

Definition 4.1.1 (Connectedness)**.** A graph G is *connected* if and only if, for any pair of distinct vertices u and v in G, there exists a path of edges (regardless of direction) from u to v.

Definition 4.1.2 (Input Graph)**.** An *input graph*, in the context of the connectedness problem, is an arbitrarily-labelled GP 2 host graph such that:

- every node is marked grey,
- every node is non-rooted, and
- every edge is unmarked.

The choice of having input nodes grey is not an aesthetic one. Indeed, it is necessary for unvisited nodes adjacent to the root to match with the mark `any`. The new compiler does not separate edges in terms of the marks of their targets or sources.

Proposition 4.1.1. Throughout the execution of `is-connected` on an input graph, all non-grey nodes in the host graph share a connected component.

Proof. All nodes in the input graph are initially grey. Upon inspection of the rules, `init` and `move` are the only rules altering the mark of a grey node, where `init` is only called once at the start of the program. Let us show inductively that the invariant holds.

If `init` is applied at the start of the program, the host graph will contain exactly one non-grey node, which does not violate the invariant. If `init` is not applied, `move` is never called, and the invariant remains true.

Now, assume that the invariant is true at some stage of the execution. If `move` is applied, the non-grey node v that turns blue has to be adjacent to a non-grey node. Thus, if all non-grey nodes shared a connected component prior to the application of `move`, v has to be part of that component as well. Thus, the invariant remains satisfied. ☐

Lemma 4.1.1 (Termination of `is-connected`)**.** The program `is-connected` terminates on any input graph.

Proof. The program `is-connected` contains two looping bodies: `DFS!` and `FORWARD!`. To show termination, consider a measure $\#(X)$ consisting of the number of unmarked edges in the host graph X. The rule `next_edge` is invoked at the beginning of `FORWARD`, and if it fails to match, the loop breaks. Clearly, an application of `next_edge` reduces the measure $\#$. Since the number of edges is finite and no rule in `is-connected` creates or unmarks an edge, `FORWARD!` terminates. We now show that the upper body `DFS!` terminates. This time, define $\#(X)$ to be the number of non-blue edges in the host graph X. The loop `DFS!` breaks if and only if `back` is called and fails to apply. Let H be the resulting graph of an application of `back` on G, that is, $G \Rightarrow_{\texttt{back}} H$. Clearly, $\#(H) < \#(G)$ since the rule marks a dashed edge blue and preserves the size (i.e. $|H| = |G|$). Similarly, given that blue edges retain their marks throughout the execution of `is-connected`, the number of edges is finitely fixed and `FORWARD!` is known to terminate, dashed edges are eventually exhausted, the `break` command is called, and `DFS!` terminates. ☐

Proposition 4.1.2. Throughout the execution of `DFS!`, there exists exactly one path of dashed edges such that every node on the path is blue, and the endpoint of the path is the root node.

Proof. Upon inspection of the rules, `move` is the only rule dashing edges in the program. We proceed by induction on the execution steps of the program.

Initially, before `move` is applied, no dashed edges exist in the host graph. The rule `init` creates a non-grey root node, and `back` cannot be applied. Thus, the invariant holds vacuously. Now, assume the invariant holds at some stage of execution. Clearly, the rule `move` extends the dashed path by adding one edge, and `back` reduces the dashed path by removing one edge. Both operations preserve the invariant and ensure the dashed path remains unique, its nodes remain blue, and the endpoint remains the root. Therefore, the invariant is maintained throughout the execution. ☐

Proposition 4.1.3. In the output graph of `try init then DFS!` executed on a non-empty input graph, there is a connected component in which every node is non-grey.

Proof. The proposition is trivially true for an input graph consisting of a single node: `init` marks the node blue, and no rule marks a non-grey node grey. Assume now that the input graph contains at least two nodes. By Proposition 4.1.1, all non-grey nodes belong to the same connected component. Therefore, it suffices to show that a connected component containing at least one non-grey node cannot also contain a grey node.

For the sake of contradiction, suppose the connected component containing all non-grey nodes in the output graph also includes at least one grey node. Let v and u be such that v is grey, u is non-grey, and v is adjacent to u, which must occur if they share a connected component.

Upon inspection of the rules, `init` and `move` are the only rules altering the mark of a grey node; both mark a grey node blue and root it. Thus, v must have been rooted at some point during the execution. Let us consider two cases based on whether v remains rooted in the output graph.

Case 1. The procedure `DFS!` terminates only if `back` fails to apply, which occurs when `next_edge` fails to find an unmarked edge. However, u, being non-grey, shares an unmarked edge with v. Since u is blue and v is rooted, the rules `next_edge` and `move` must have been applied, which contradicts the assumption that v is grey in the output graph.

Case 2. v is not rooted in the output graph. Throughout the execution of `DFS!`, there is exactly one root node at any time, given that `init` initialises a single root, and `move` and `back` relocate the root while preserving the invariant. After the first application of `move`, the root is always incident with exactly one dashed edge, until `back` becomes inapplicable.

Since v is not rooted, it must have been unrooted by either `move`, which dashed an edge incident with v, or `back`. Two scenarios arise:

1. If `back` was applied to v, it would have unrooted v only after no rules in `FORWARD` were applicable. However, since v is adjacent to u, and u shares an unmarked edge, `next_edge` and `move` would have applied, turning u non-grey. This contradicts the assumption.

2. If `back` was not applied to v, v must be part of a path of dashed edges with a root node as the endpoint (by Proposition 4.1.2). The termination of `DFS!` implies that the rule `back` was no longer applicable, which is, again, a contradiction to the fact that there exists some root node adjacent to a blue node sharing a dashed edge (which follows from the fact that v must be incident with at least one dashed edge).

Given that both cases contradict the assumption, at the termination of `try init then DFS!` on a non-empty graph, there must be a connected component in which every node is non-grey. □

Theorem 4.1.1 (Correctness of `is-connected`)**.** The program `is-connected` is totally correct with respect to the following specifications:

Input:	An input graph.
Output:	The program fails if and only if the input is not connected. The empty graph is considered connected.

Proof. Termination follows from Lemma 4.1.1. Let G be the input graph. First, assume that the input graph G is empty. Since the rule `init` is not applicable, the program immediately terminates and no `fail` command is invoked.

Now, assume G contains at least one node. Consider the graph H as the output graph of `try init then DFS!` on G, i.e. $G \Rightarrow_{\texttt{init}} I \Rightarrow_{\texttt{DFS!}} H$ for some intermediate graph I. By Proposition 4.1.3, there exists some connected component in H where every node is non-grey, let us call it C, and by Proposition 4.1.1, it follows that $H - C$ contains no non-grey node. Therefore, G is not connected if and only if $H - C$ is non-empty. The procedure `Check` is executed after `DFS!`, and verifies that no grey node, which must belong to $H - C$ should one exist, exists. If a grey node exists, $H - C$ is non-empty and the program fails, indicating that G is not connected. If none exists, the program terminates without invoking the `fail` command, indicating that G is connected. ☐

4.1.2 Proof of Complexity

Proposition 4.1.4. Throughout the execution of `is-connected`, there is at most one red edge in the host graph.

Proof. Upon inspection of the rules, `next_edge` is the only rule that marks an edge red in the host graph (note that the program is structure-preserving). It is easy to see that when `next_edge` is applied, either `move` or `ignore` has to apply, as node `2` of `next_edge` can either be blue or grey. Therefore, it follows that the number of red edges in the host graph is bounded to 1. ☐

Theorem 4.1.2 (Complexity of `is-connected`)**.** On any class of input graphs, the program `is-connected` terminates in time linear to the size of the input graph.

Proof. We first show that, for every rule, there is at most one matching attempt with respect to the complexity assumptions of the updated compiler (Figure 4.9).

The rule `init` matches a single node. If the graph is non-empty, `init` immediately applies on the first match as every node is grey. Otherwise, no node is considered and the rule fails. Therefore, there is at most one matching attempt for `init`.

Since there is at most one root in the host graph (`init` is the only rule creating or deleting a root), node `1` of `next_edge`, `move` and `ignore`, and node `2` of `back` match in constant time. The rule `next_edge` matches any unmarked edge incident to the root and incident to an any-marked node. Since any node adjacent to the root is marked, and non-loop edges are stored in distinct lists with respect to their marks, there is at most one matching attempt for `next_edge`. An analogous argument can be made for all rules beside `init` and `match`, as it is known from Proposition 4.1.4 that there is at most one red edge in the host graph, and Proposition 4.1.2 establishes that a root node is incident to at most one dashed edge, hence limiting the number of possible matches to a single edge. The rule `match` requires the compiler to find a grey node, which the procedure `firstHostNode(m)` of the updated compiler (Figure 4.9) executes in constant time. Now, let us look at the number of calls during the execution of the program for each rule. We define a call to be the invocation of a rule. For the purpose of this proof, let n and m be the number of nodes and edges, respectively.

The rule `init` is only called once at the beginning. It succeeds if the input graph is non-empty; otherwise, it fails. Let us show that `back` is called at most $m + 1$ times. The number of calls is the sum of successful applications and unsuccessful ones. Since the loop `DFS!` terminates at the inapplicability of `back`, there can be, at most, one unsuccessful application. Observe that the rule `back` marks an edge blue. Upon inspection of the rules, it can be seen that blue edges retain their mark. Hence, since there are m edges, there can be at most m successful applications of `back`.

Similarly, we demonstrate that the rules `move` and `ignore` are called a linear number of times. Let us look at the number of successful applications first. The rule `ignore` marks an edge blue. As previously stated, blue edges retain their mark. Therefore, there can be at most m successful applications of `ignore`. The rule `move` marks a grey node blue. Non-grey nodes retain their mark, thus implying that there can be at most $n - 1$ successful applications of `move` (there are n nodes and `init` has already turned one node non-grey, hence $n - 1$). Again, since `move` and `ignore` are only invoked after a successful application of `next_edge`, it is easy to see that these rules cannot fail.

The rule `next_edge` marks an unmarked edge, implying that there can be at most m successful applications. An unsuccessful application of `next_edge` terminates the loop `FORWARD!` and invokes the rule `back`, which either succeeds or terminates the `DFS!` loop. Thus, there can be at most as many unsuccessful applications of `next_edge` as there are successful applications of `back`, that is, m. The number of calls of `next_edge` is bounded to $m + m = 2m$.

Finally, the rule `match` is only invoked once. The overall complexity of the program `is-connected` is therefore linear. ☐

Figure 4.11 showcases the empirical benchmarks of `is-connected` on the graph classes of Figures 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6. The measured runtimes do not account for graph parsing, building and printing, as these operations have a linear-time complexity with respect to the input size (Figure 4.9). Compilation time is also not included.

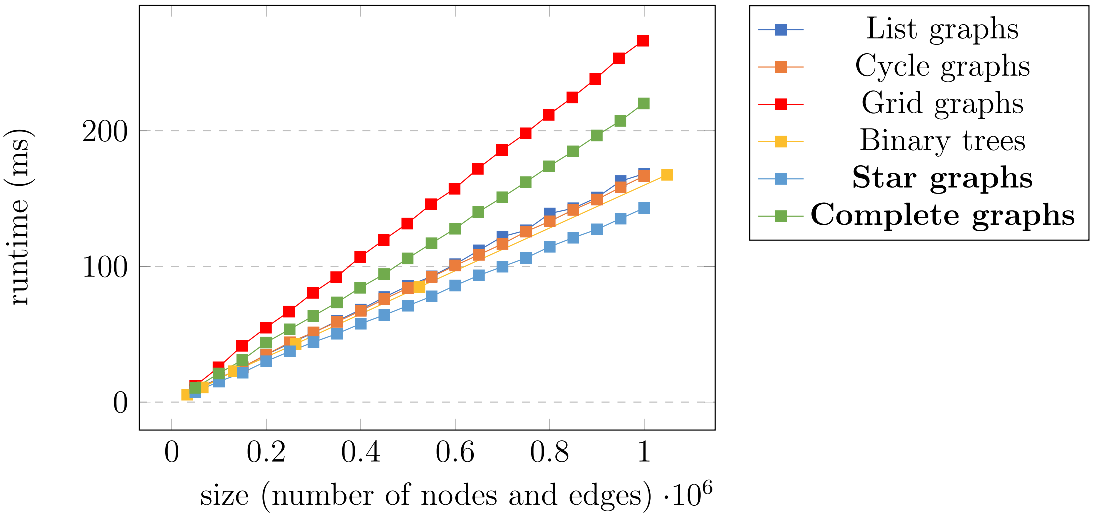


FIGURE 4.11: Measured performance of the program `is-connected`. Unbounded-degree graph classes are boldened in the legend.

4.2 Two-Colouring Graphs

Vertex colouring is a well-known graph problem which has applications in domains such as scheduling, compiler optimisation and register allocation [Ski20]. In 2012, Bak and Plump came up with a GP 2 program for the 2-colouring problem [BP12]. The program achieved linear time complexity on bounded-degree graph classes but exhibited quadratic time complexity on graph classes of unbounded degree [BP16].

In this section, we discuss the `2-colouring` program of Figure 4.12, which achieves linear runtime on both bounded- and unbounded-degree graph classes using the same input and output conditions as Bak and Plump's program. This became possible thanks to the improvements to the compiler described in Section 3.3, namely, the separation of edge lists with respect to marks and orientation.

```
Main    = try init then (DFS!; Check)
DFS     = FORWARD!; try back else break
FORWARD = next_edge;
          try  {colour_red, colour_blue, blue_red, red_blue}
          else (unroot; break)
Check   = try unroot else fail
```

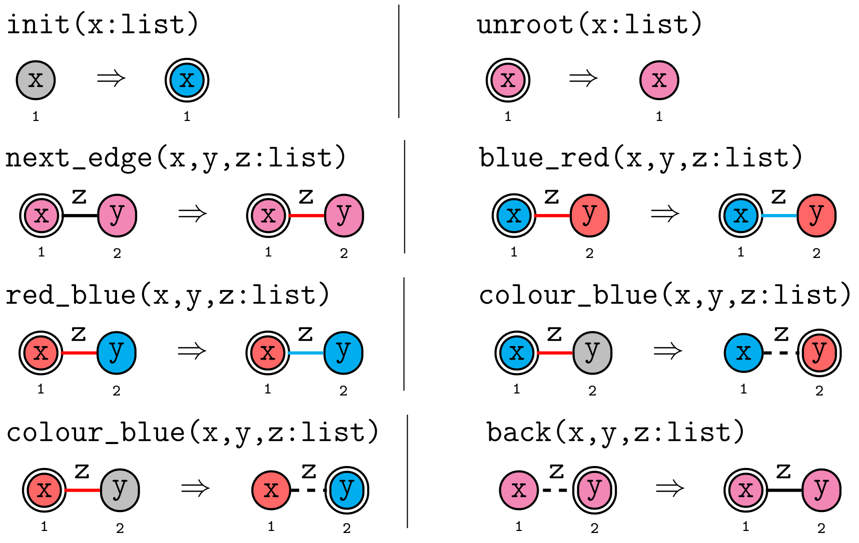


Figure 4.12: The program `2-colouring`.

The program `2-colouring`[3] expects a host graph satisfying the input conditions of Definition 4.2.1. It attempts to 2-colour the graph by performing a depth-first search (DFS) from an initial node, colouring each newly visited node in the colour contrasting that of the node it is visited from (either red or blue). The program fails if an edge is found to be incident with two non-grey nodes of the same colour. Figure 4.13 provides a sample execution of `2-colouring`.

In contrast to previous implementations of the 2-colouring, the linearity of this program's runtime is primarily attributed to an invariant ensuring that a single edge incident to the root is marked red, allowing rules implementing a depth-first search to process both forward and cross edges, instead of forward edges only. Proposition 4.2.4 shows that there can be, at most, one red edge at a time in the host graph throughout the execution of the program, allowing for instantaneous access with the compiler's recent optimisations. Furthermore, a blue mark on an edge indicates that its processing has ended, eliminating the need for it to be matched again as a forward or cross edge in the DFS traversal.

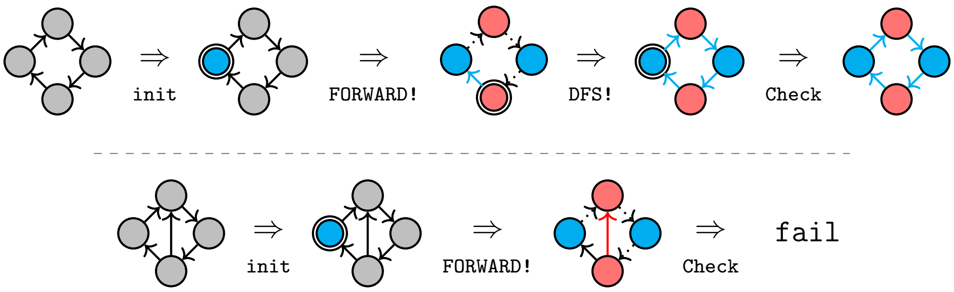


FIGURE 4.13: Sample executions of `2-colouring` on a 2-colourable (top) input graph and a non-2-colourable (bottom) input graph.

In this section, we define `COLOUR` to be the set {`colour_red, colour_blue`} and `IGNORE`, the set {`blue_red, red_blue`}. When we refer to the application of `COLOUR`, we mean that either `colour_red` or `colour_blue` is applied. Similarly, applying `IGNORE` means applying either `blue_red` or `red_blue`.

We lay down the definition of an input graph in the context of the `2-colouring` program.

Definition 4.2.1 (Input Graph)**.** An *input graph*, in the context of the two-colourability problem, is an arbitrarily-labelled *connected* GP 2 host graph such that:

1. every node is marked grey,
2. every node is non-rooted, and
3. every edge is unmarked.

---

[3] The concrete syntax of the program available at: https://gist.github.com/ismaili-ziad/51cc29fa3ea49a49d1922acd560ce3ee.

4.2.1 Proof of Correctness

Let us first examine the correctness of `2-colouring`.

Proposition 4.2.1. Throughout the execution of `2-colouring` on a given input graph, the following invariant holds: edges marked as blue and nodes marked as either blue or red retain their respective marks unchanged.

Proof. Upon inspection of the rules, none modifies the mark of a blue node, a red node, a blue edge or an *any*-marked element. Thus the invariant holds. □

Proposition 4.2.2. Throughout the execution of `2-colouring`, the following invariant holds: there is at most one root in the host graph and is incident to at most one dashed edge.

Proof. `init` and `unroot` are the only rules that do not preserve the cardinality of roots in the host graph. The `init` rule increases the number of roots by 1, while the `unroot` rule decreases it by 1. Since `init` is only called once, there is at most one root node throughout the execution of `2-colouring`.

For the following argument, observe that `colour_red` and `colour_blue` are the only rules generating dashed edges in the host graph. Let u, v and e denote node `1`, node `2` and the edge of either rule, respectively.

Consider the step $G \Rightarrow_{\texttt{COLOUR}} H$ and let the nodes u' and v' be the images of u and v by that production, respectively. Define $d(x)$ as the number of dashed edges incident to node x. Then, $d(v') = d(v) + 1$ and $d(u') = d(u) + 1$. To maintain the invariant, $d(v)$ must be 0. To show this, consider the fact that v is grey. Upon inspection of the program, there is no rule that generates a dashed edge incident to a grey node. Indeed, any generated edge must be incident to either a red or a blue node. Furthermore, as established in Proposition 4.2.1, red and blue nodes retain their mark, implying that a non-grey node incident to a dashed edge cannot subsequently turn grey.

Hence, since v is grey, it is guaranteed not to be incident to a dashed edge prior to the application of `COLOUR`; thus, $d(v') = d(v) + 1 = 1$. Given that the rule `back` does not increase the incidence of a node to dashed edges, the property remains satisfied. □

Proposition 4.2.3. Upon the execution of `2-colouring` on an input graph G, the rule `unroot` in `DFS!` is applied if and only if G is not 2-colourable.

Proof. The lemma is trivially true if G is empty. Assume G contains at least one node. Let us split the proof into two distinct cases.

Case 1. *G is 2-colourable.* Let X and Y be disjoint subsets of V_G with respect to the bipartition of G such that $X \cup Y = V_G$ and every edge in G shares endpoints in both X and Y. Thus, no pair of distinct nodes within the same subset are adjacent in G. Clearly, given that the program is structure-preserving, X and Y remain the same throughout the execution of `2-colouring`. Let G' be the initialised graph such that $G \Rightarrow_{\texttt{init}} G'$. Suppose that `init` roots and marks a node blue in X. Consider the graph H such that $G' \Rightarrow^{n}_{\texttt{FORWARD}} H$ for some $n \geq 0$. By induction on n, we show that two non-grey nodes sharing the same mark are never adjacent.

When $n = 0$ (base case), the body `DFS!` has not been invoked once. Hence, H consists of (possibly zero) grey nodes and a single blue root node in X created by `init`, which does not violate the property.

Now, assume the property holds in H for some n; we show it still holds for $n + 1$. At the invocation of `FORWARD`, the rule `next_edge` is called. If there is no unmarked edge incident to the root node, the rule fails to match, and the loop exits, satisfying the condition. Otherwise, `next_edge` matches node `1` to the root node and node `2` to some arbitrarily marked node adjacent to the root via an unmarked edge. Since the host graph is 2-colourable, `next_edge` preserves marks and it is assumed, by the induction hypothesis, that no pair of adjacent nodes are both marked blue or red, node `2` must be part of the subset opposite to the root's and marked differently. That is, if node `1` is in X, node `2` is in Y, and vice versa. Otherwise, it would imply two nodes within the same subset share an edge, violating the assumption on G. At the invocation of `try {colour_red, colour_blue, blue_red, red_blue}`, following `next_edge`, node `2` must either have a mark opposite to the root's (i.e. blue if the root is red and vice versa) or be grey. Hence, one rule within that body must apply and move the root to the opposite subset while alternating its mark, preserving the property.

Therefore, upon the execution of `FORWARD`, either `next_edge` fails to apply and exits the loop, or it does apply and a rule in the `try` condition consequently matches and applies. Given that the rule `back` does not affect marks, `unroot` in `DFS!` is never invoked.

Case 2. *G is not 2-colourable.* By definition, there exists no assignment of marks from the set $\{blue, red\}$ to every node in G such that every edge has endpoints of different marks. Thus, a program that colours nodes in G either blue or red has to violate the two-colourability condition in that at least one pair of adjacent nodes share the same colour. Indeed, the node-marking rules of `DFS!` are `colour_red` and `colour_blue`, and each rule colours a node in the contrasting colour to that of the adjacent root. As such, an eventual application of these rules will result in two non-grey nodes sharing the same mark. Let w be node `2` following such an application of either `colour_red` or `colour_blue` on the host graph. w is either blue or red, rooted and adjacent to some node x that shares its mark. Any edge that connects w and x at this instance must be unmarked since a mark would imply that the edge was previously matched by `next_edge`, and subsequently by a rule in `try {colour_red, colour_blue, blue_red, red_blue}`. One of two situations occurs at the next invocation of `FORWARD`: either `next_edge` matches w and x (1), or it matches w and a different neighbour distinct from x (2).

1. Each rule in `COLOUR` and `IGNORE` is invoked, but none matches. Consequently, the `unroot` rule, applicable to w, is invoked and applied, leading to the call of `break` and the termination of the `DFS!` loop.

2. Let y be the neighbour of w distinct from x matched by the rule `next_edge`. Three subcases emerge: y admits of the same mark as w (a); y is marked in the colour opposite to w (b); y is grey (c).

(a) The same reasoning as Situation (1) applies.

(b) As both w and y are of contrasting colours and connected by a red edge, `IGNORE` applies. This ends the current instance of `FORWARD`, although not necessarily terminating the loop, and brings us back to the beginnings of Situations (1) and (2).

(c) After `COLOUR` is applied, w is unrooted, and y becomes the new root, marked with the colour opposite to w. Additionally, the edge matched by the rule is dashed. Proposition 4.2.2 establishes that the root node is incident to at most one dashed edge. Consequently, any subsequent application of `back` successfully backtracks the root to its ancestral node in the depth-first search tree.

The loop `DFS!` terminates upon the failure of the `back` rule. There are two possible scenarios: either the root eventually backtracks to node w before `DFS!` terminates, returning us to Situations (1) and (2), or `DFS!` breaks prior to w being rooted again. The latter can only happen if `back` is no longer applicable.

However, since `COLOUR` creates a path of dashed edges with a single endpoint rooted, the inapplicability of `back` can only occur due to the removal of the root, i.e. the application of `unroot`. In either case, `unroot` is invoked, causing `FORWARD!` to break, `back` to fail, and `DFS!` to terminate.

Therefore, executing `2-colouring` on a non-2-colourable input graph results in the eventual application of `unroot` in `DFS!`.

□

The next lemma demonstrates that termination of `2-colouring` is ensured by showing that the body of each loop reduces a measure that assigns a non-negative integer to each host graph, thereby showing that the loop body eventually fails.

Lemma 4.2.1 (Termination of `2-colouring`)**.** The program `2-colouring` terminates on any input graph.

Proof. An argument analogous to Lemma 4.1.1 can be made with respect to the program `2-colouring`. □

Lemma 4.2.2 shows that the program 2-colours the graph, should it be 2-colourable. It demonstrates that the existence of an unvisited (grey) node following the termination of `DFS!` on a 2-colourable graph leads to a contradiction.

Lemma 4.2.2. Consider a *2-colourable* input graph G. Upon termination of `DFS!` on G, the host graph contains no grey nodes.

Proof. The lemma is trivially true if G is empty since the rule `init` fails to apply and the body `(DFS!; Check)` is not invoked. Suppose G consists of at least one node, thereby making `init` applicable, and consider G' and H such that $G \Rightarrow_{\texttt{init}} G' \Rightarrow_{\texttt{DFS!}} H$. For the sake of contradiction, assume that a grey node exists in H. The rule `init` creates a blue node in G'. Note, upon inspection of the rules, that the program is structure-preserving. Therefore, both G' and H are 2-colourable and connected. As per Proposition 4.2.1, once a node is turned blue, its mark is no longer modified. This implies at least one grey node in H (assumption) is adjacent to some non-grey node by the connectedness of H. Let u and v be the non-grey (blue or red) and grey nodes, respectively. We show that u and v are matched by a rule in `COLOUR` prior to the termination of `DFS!`, thereby contradicting the assumption.

Given that u is non-grey in G, and non-grey nodes preserve their mark, it must have been matched by either `init` or a rule in `COLOUR`. Either way, u must have been a non-grey root node. Recall that `DFS!` terminates if and only if `back` fails to apply, which can only occur following the termination of the loop `FORWARD!`. The latter breaks if either condition holds: `next_edge` fails to apply; no rule in `COLOUR` $\cup$ `IGNORE` applies. The second condition implies that the body `(unroot; break)` is then called. However, as shown in Proposition 4.2.3, `unroot` is never invoked if G is 2-colourable. Therefore, the second condition can never hold; that is, a rule in `COLOUR` $\cup$ `IGNORE` is always applicable upon its invocation. Regarding the first condition, let us examine the implicit data structure `DFS!` generates. The dashed edges form a path of non-grey nodes, wherein an endpoint is rooted. This models a stack of nodes where the root represents the top element. Specifically, `init` initialises the stack, `COLOUR` executes the *push* operation, and `back` performs the *pop* operation. Given that a root node can be incident to, at most, one dashed edge, it is guaranteed to backtrack to its ancestral node in the depth-first search tree as `back` is applied. The inapplicability of `back` can be seen as an exhaustion of the stack, which can only occur if there are no dashed edges in the host graph (as previously stated, the application of `unroot` is not a possibility). It is easy to observe that the remaining root at the end of `DFS!` is the initial node of the stack (i.e. the first pushed node). If u is rooted, then it is the initial node (i.e. the node `init` was applied to). The invocation of `back` follows the termination of `FORWARD!`, which only occurs if `next_edge` is not applicable, provided the graph is 2-colourable. However, `next_edge` is applicable on u and v, hence a contradiction. If u is non-rooted, it must have been a root at some point during the execution of `DFS!` prior to it being popped from the so-called stack. Again, an analogous argument shows that this leads to a contradiction.

Since u and v ought to have been matched by `next_edge`, one of two outcomes must have occurred: either `COLOUR` successfully applied, or the statement `(unroot; break)` was invoked. However, since G is 2-colourable, the `unroot` rule cannot be applied. Therefore, `COLOUR` must have been applicable, resulting in v being marked non-grey. As established in Proposition 4.2.1, non-grey nodes retain their colour assignment. Hence, v cannot be grey in H.

Given the following properties:

- nodes in the host graph can only be grey, blue or red;
- a node adjacent to a non-grey node cannot be grey;
- G is 2-colourable and connected;

- blue and red nodes retain their mark; and
- the rule `init`, prior to `DFS!`, creates one non-grey node in the host graph;

it follows that every node in H (i.e. the resulting graph following the execution of `DFS!`) is non-grey. □

Building upon the previous lemmata and propositions, we now show the correctness of `2-colouring`.

Theorem 4.2.1 (Correctness of `2-colouring`)**.** The program `2-colouring` is totally correct with respect to the following specifications:

Input: An input graph.
Output: The program fails if and only if the input is not 2-colourable. Otherwise, it outputs a 2-colouring of the input such that every node is either blue or red, and no pair of adjacent nodes share the same mark.

Proof. Termination follows from Lemma 4.2.1. Loop edges do not affect the 2-colouring and are omitted in the program. Let G be the input graph. If G is empty, `init` fails to match, and the program terminates, outputting the empty graph and satisfying the specifications. Suppose G consists of at least one node. We then split the remainder of this proof into two cases.

Case 1. *G is 2-colourable.* Since it is nonempty, `init` applies and `DFS!` is invoked. Consider G' and H such that $G \Rightarrow_{\texttt{init}} G' \Rightarrow_{\texttt{DFS!}} H$. It follows from Lemma 4.2.2 that H only contains red and blue nodes. Furthermore, as per Proposition 4.2.3, the rule `unroot` in `DFS` is never invoked, indicating that no violation of the 2-colouring has been encountered and the existence of a single root in H. Upon termination of `DFS!`, the procedure `Check` is called, and the rule `unroot` unroots the unique root node. The program then terminates and returns the 2-coloured host graph.

Case 2. *G is not 2-colourable.* Since it is nonempty, `init` applies and `DFS!` is invoked. Analogously to the previous argument, consider G' and H such that $G \Rightarrow_{\texttt{init}} G' \Rightarrow_{\texttt{DFS!}} H$. It follows from Proposition 4.2.3 that `unroot` is applied at the last execution of `FORWARD!`, breaking the `DFS!` loop. Therefore, there is no root node in H. The procedure `Check` is called, and since H contains no root, the rule `unroot` fails to match, and the `fail` command is invoked, failing the entire program.

□

4.2.2 Proof of Complexity

We now examine the complexity of `2-colouring`. Prior to doing so, we establish another invariant of the program in Proposition 4.2.4 so as to argue for the constant-time matching of rules in $\texttt{COLOUR} \cup \texttt{IGNORE}$.

Proposition 4.2.4. Throughout the execution of `2-colouring`, there is at most one red edge in the host graph.

Proof. Upon inspection of the rules, `next_edge` is the only rule that creates a red edge in the host graph. Following its application, either a rule in the body `{colour_red, colour_blue, blue_red, red_blue}` applies and removes the unique red edge, or `(unroot; break)` is called, breaking the `FORWARD!` loop, making `back` inapplicable and, subsequently, terminating the `DFS!` loop. No rule involves red edges afterwards. □

Theorem 4.2.2 (Complexity of `2-colouring`)**.** On any class of input graphs, the program `2-colouring` terminates in time linear to the size of the input graph.

Proof. We first show that, for every rule, there is at most one matching attempt with respect to the complexity assumptions of the updated compiler (Figure 4.9).

The rule `init` matches a single node. If the graph is non-empty, `init` immediately applies on the first match as every node is grey. Otherwise, no node is considered, and the matching fails immediately. Therefore, there is at most one matching attempt for `init`. The rule `unroot` matches a single node. Given that there is a bounded number of roots in the graph (Proposition 4.2.2), the number of matching attempts is also bounded.

Since there is at most one root in the host graph, node `1` of `next_edge`, `blue_red`, `red_blue`, `colour_red`, `colour_blue` and node `2` of `back` match in constant time. The rule `next_edge` matches any unmarked edge incident to the root incident to an *any*-marked node. Since any node adjacent to the root is marked, and non-loop edges are stored in distinct lists with respect to their marks, there is at most one matching attempt for `next_edge`. An analogous argument can be made for all rules beside `init` and `unroot`, as it is known from Proposition 4.2.4 that there is at most one red edge in the host graph, and Proposition 4.2.2 establishes that a root node is incident to at most one dashed edge, hence limiting the number of possible matches to a single one. Therefore, there is at most one matching attempt for every rule in `2-colouring`. Now, let us look at the number of calls during the execution of the program for each rule. We define a call to be the invocation of a rule. For the purpose of this proof, let n and m be the number of nodes and edges, respectively.

The rule `init` is only called once at the beginning. It succeeds if the input graph is non-empty; otherwise, it fails. Let us show that `back` is called at most $m + 1$ times. The number of calls is the sum of successful applications and unsuccessful ones. Since the loop `DFS!` terminates at the inapplicability of `back`, there can be, at most, one unsuccessful application. Observe that the rule `back` marks an edge blue. It has been established in Proposition 4.2.1 that blue edges retain their mark. Hence, since there are m edges, there can be at most m successful applications.

Similarly, we demonstrate that the rules `blue_red`, `red_blue`, `colour_red` and `colour_blue` are called at most $n + m$ times each. Since the reasoning applies analogously to each of these rules, let `r` represent one of them. It can be observed that an unsuccessful application of `r` does not necessarily trigger the invocation of `(unroot; break)`, as the latter is called if and only if every single rule in `COLOUR`$\cup$`IGNORE` fails. Let us then look at the number of successful applications first. The rules `red_blue` and `blue_red` mark an edge blue. As previously stated, blue edges retain their mark. Therefore, there can be at most m successful applications of `COLOUR`. The rules `colour_red` and `colour_blue` mark a grey node blue or red. Non-grey nodes retain their mark, thus implying that there can be at most $n-1$ successful applications of `IGNORE` (there are n nodes and `init` has already turned one node non-grey, hence $n-1$). Given that the failure of every rule in $\texttt{COLOUR} \cup \texttt{IGNORE}$ triggers the invocation of `(unroot; break)` and terminates both `FORWARD!` and `DFS!`, each rule can fail to apply at most as many times as some rule in $\texttt{COLOUR} \cup \texttt{IGNORE}$ succeeds and one more time, marking the terminations of the loops `FORWARD!` and `DFS!`. Hence, each rule application can be unsuccessful at most $(n-1)+(m)+1 = n+m$ times.

The rule `next_edge` marks an unmarked edge, implying that there can be at most m successful applications. An unsuccessful application of `next_edge` terminates the loop `FORWARD!` and invokes the rule `back`, which either succeeds or terminates the `DFS!` loop. Thus, there can be at most as many unsuccessful applications of `next_edge` as there are successful applications of `back`, that is, m. The number of calls of `next_edge` is bounded to $m+m=2m$. Finally, it is easy to see that `unroot` is called at most twice. Once in `DFS!` (its call terminates the loop) and once in `Check`.

Therefore, taking into account that all rules are constant, the overall time complexity of the program `2-colouring` is linear. ☐

Figure 4.14 showcases the empirical benchmarks of `2-colouring` on the graph classes of Figures 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6.

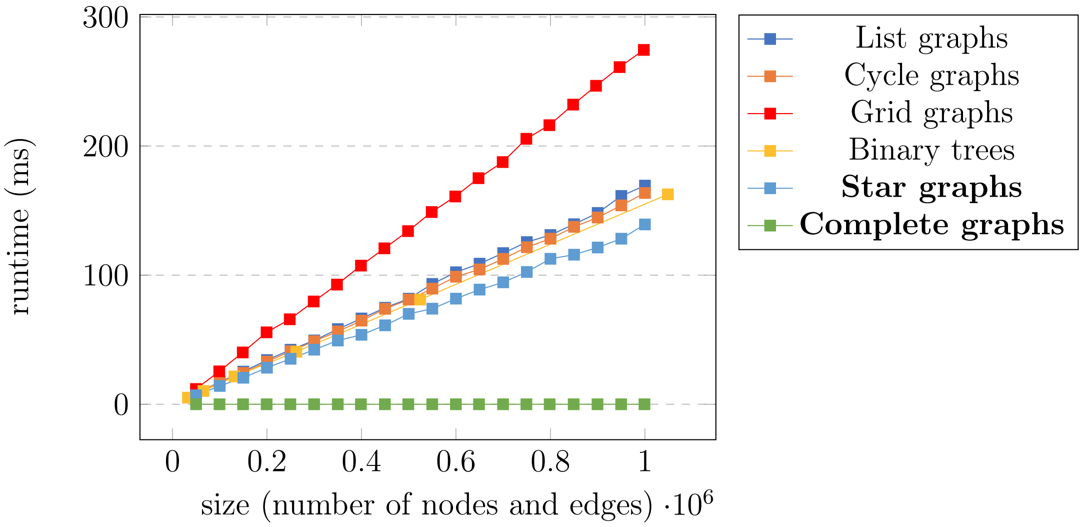


FIGURE 4.14: Measured performance of the program `2-colouring`. Unbounded-degree graph classes are boldened in the legend.

As evidenced, both unbounded-degree graph classes, complete graphs and binary trees, exhibit linear runtime performance in these tests. It is interesting to note that complete graphs exhibit almost constant runtime. We conjecture that since any complete graph K_n with $n \geq 3$ is not 2-colourable, the GP 2 program detects a violation early during execution. This consistent behaviour is primarily attributed to the deterministic nature of the compiler implementation. In principle, matches in GP 2 are non-deterministic, and it is conceivable that a GP 2 compiler strictly adhering to this non-determinism would visit every node in an arbitrary complete graph before encountering such a violation.

4.2.3 Two-Colouring Disconnected Graphs

As stated in Definition 4.2.1, input graphs are assumed to be connected. However, the program `2-colouring` can be modified to allow for disconnected graphs.

```
Main    = COLOUR!; if flag then fail
COLOUR  = try init then (DFS!; Check)
DFS     = FORWARD!; try back else break
FORWARD = next_edge;
          try  {colour_red, colour_blue, blue_red, red_blue}
          else (unroot; break)
Check   = try unroot else (set_flag; break)
```

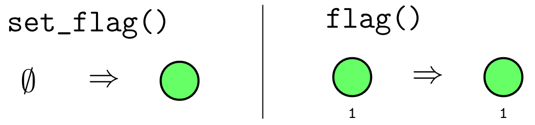


Figure 4.15: The program `2-colouring-disconnected`.

The program `2-colouring-disconnected` achieves the same purpose on disconnected input graphs. The main algorithm underlying the program is the same as `2-colouring`, except that it is contained within a loop. When the `fail` command is called within the procedure `COLOUR`, a flag node is created by `set_flag`, which will be subsequently found by the rule `flag`. The program also runs in linear time, as the rule `init` only requires one constant-time look up to the linked list of grey nodes to determine whether an unvisited connected component remains.

4.3 Checking for Acyclicity

Cycle detection is another foundational problem in graph algorithms. Several GP 2 programs were previously proposed to decide whether a given directed graph is acyclic (i.e. contains no directed cycle), however, they were either restricted to binary DAGs (directed acyclic graphs wherein each node is incident to at most two outgoing edges) or destructive, in that they partially or totally deleted the input graph [Bak15; CCP22].

```
Main  = (init; DFS!; try unroot else break)!; Check
DFS   = try next_edge then (try {move, ignore}
                            else (set_flag; break))
        else (try loop; try back else break)
Check = if flag then fail
```

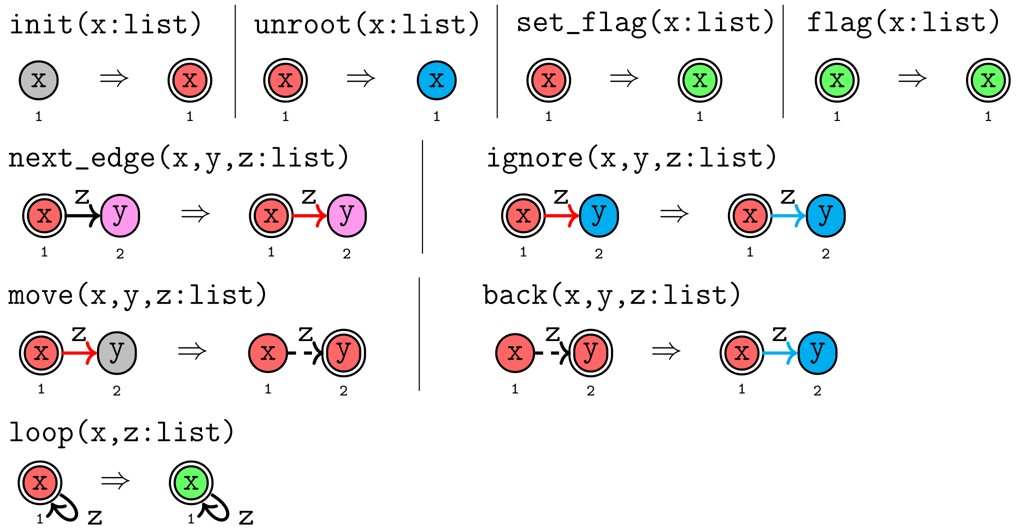


FIGURE 4.16: The program `is-dag`.

The program `is-dag`[4] from Figure 4.16 recognises acylic graphs while not being destructive; however, a cyclic input graph results in the invocation of the `fail` command, returning no output graph. Nonetheless, the program can easily be turned completely non-destructive by replacing the `fail` command in the procedure `Check` by a rule creating a distinct flag (such as an unmarked node), signalling the existence of a cycle. Figure 4.17 illustrates a sample execution of the program on an acyclic input graph, and Figure 4.18 on a cyclic one (`nx_dg`, `mv`, `bk` and `unrt` correspond to `next_edge`, `move`, `back` and `unroot`, respectively).

The program first calls a looping sequence in `Main`: `(init; DFS!; try unroot else break)!`. The rule `init` acts as an initialiser: it selects an arbitrary node to start a directed DFS from. Then, the looping procedure `DFS!` is invoked. The purpose of the procedure `DFS` is to move the root in a DFS fashion throughout the host graph, akin to `is-connected` and `2-colouring` (Figures 4.10 and 4.12). The procedure is structured around a `try-else` construct. It first tries to determine if there is any unprocessed edge (i.e. unmarked edge) incident with the root by invoking the rule `next_edge` (again, the magenta colour in the visual representations of the programs in this thesis denote the wildcard mark `any`). Should one be found, the rule marks the edge red so that it can be uniquely processed by the rest of the procedure. If none is found, the root can no longer move forward and the `else`-statement (on the next line) is invoked

---

[4] The concrete syntax of the program available at: https://gist.github.com/ismaili-ziad/959f26e188b210821d08eb5ed2965404.

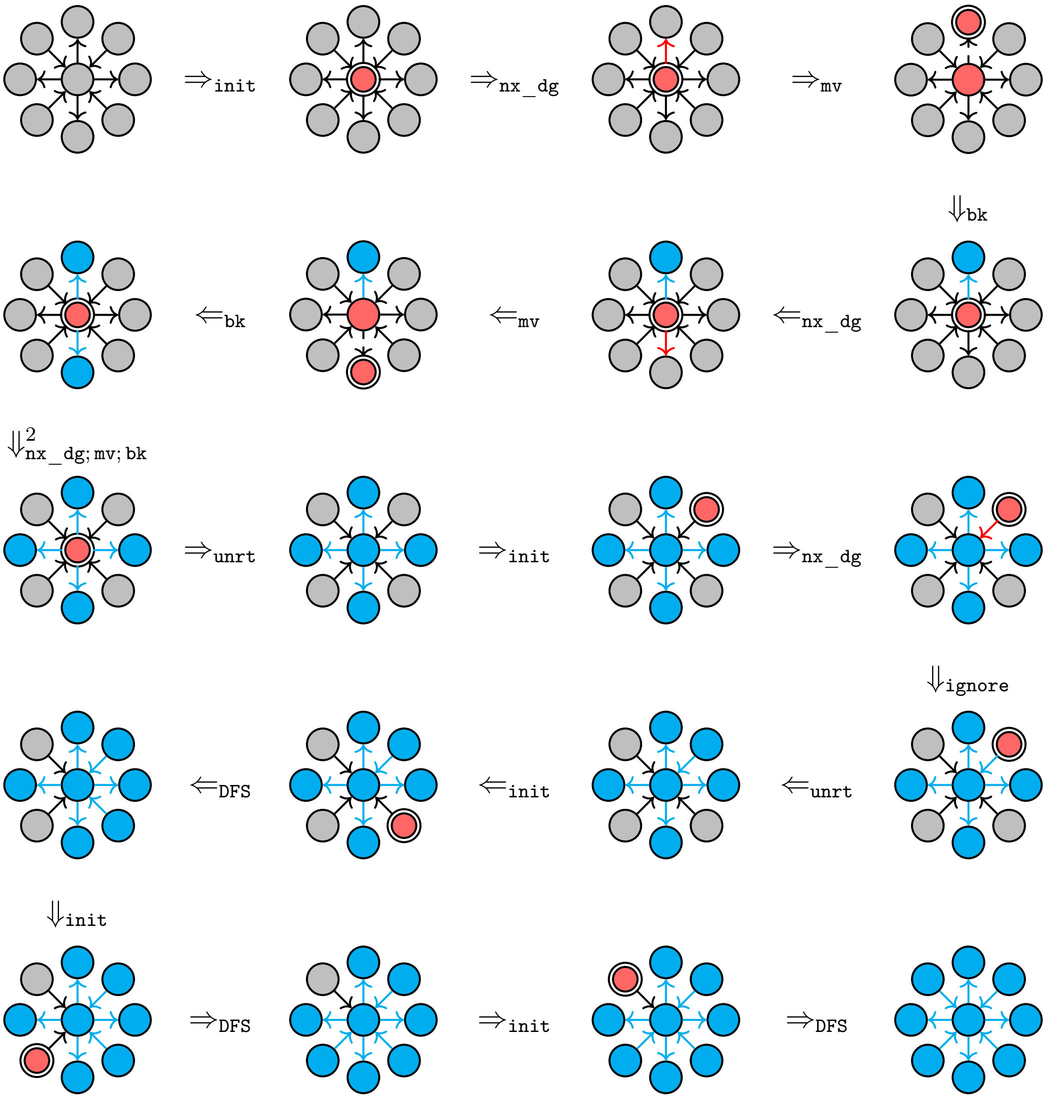


FIGURE 4.17: Sample execution of `is-dag` on an acyclic graph.

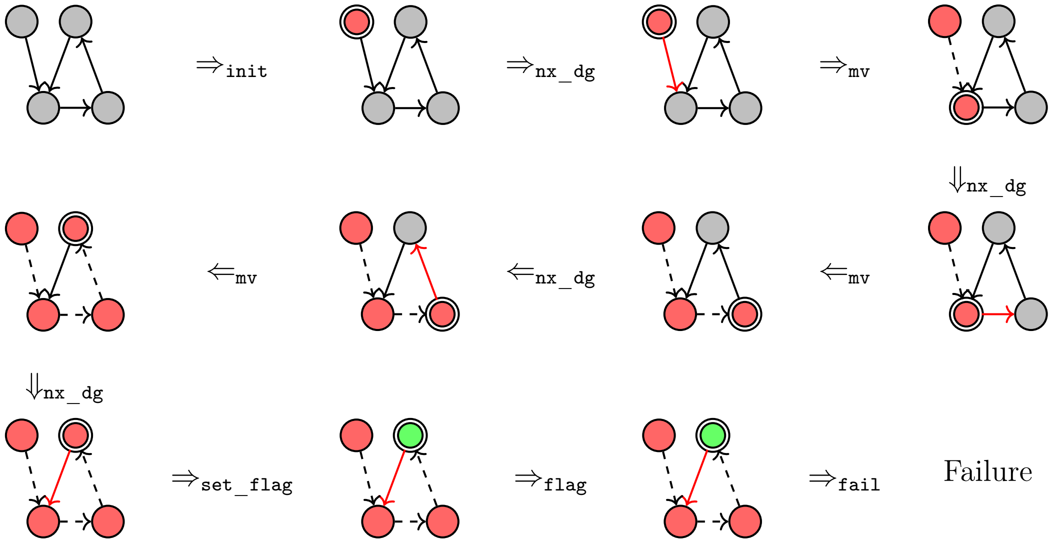


FIGURE 4.18: Sample execution of `is-dag` on a cyclic graph.

instead. Back to `next_edge`; after a successful application of the rule, there are three possible scenarios: the root is adjacent to (1) a grey-marked node, (2) a blue-marked node, or (3) a red-marked node. If the root is adjacent to a grey-marked node, the rule `move` moves the root forward to that node, marks it red and dashes the traversed edge. Dashed edges represent the ongoing path followed by the directed DFS. If the root is adjacent to a blue-marked node, the edge is ignored by the rule `ignore` marking it blue, so that it does not match with `next_edge` any further. If the root is adjacent to a red-marked node, no rule in the rule set `{move, ignore}` applies, hence invoking the rule `set_flag`, which marks the rooted node green, indicating the existence of a cycle. Prior to applying `next_edge`, if no unmarked edge is incident with the root, the latter rule fails, and `(try loop; try back else break)` is invoked. The rule `loop` is first tried to ensure that the root does not admit of a loop. If a loop is found, the rule turns the rooted node green, akin to `set_flag`. The rule `back` implements the *pop* operation by backtracking the root via an incoming dashed edge. If no incoming dashed edge is found, it implies that the rooted node is the only element of the stack; hence, the `break` command is called, terminating the looping procedure `DFS!`.

Following the termination of `DFS!`, the rule `unroot` is tried in `Main` to ensure that no flag was set during the execution of `DFS` and to unroot the sole red node of the stack. The only rules remarking (i.e. changing the marks of) rooted nodes in `DFS` are `set_flag` and `loop`, which are applied if and only if a cycle was found. Therefore, a failure to apply `unroot` entails the existence of a cycle. Given that input graphs can be disconnected and the DFS is directed, all nodes are not necessarily visited after an initial DFS from an arbitrary node. Hence, the rule `init` is called again following a successful application of `unroot`. The looping procedure terminates when `init` no longer applies, indicating the visitation of all nodes, or when `unroot` fails, indicating the discovery of a cycle.

Finally, the procedure `Check` is invoked, checking whether a flag exists in the host graph. Should one exist, a cycle was found and the command `fail` is called, failing the program. Otherwise, the program terminates and returns the host graph, isomorphic to the input graph up to marks.

4.3.1 Proof of Correctness

Definition 4.3.1 (Input Graph)**.** An *input graph*, in the context of the acyclicity problem, is an arbitrarily-labelled GP 2 host graph such that:

- every node is marked grey,
- every node is non-rooted, and
- every edge is unmarked.

Proposition 4.3.1. Throughout the execution of `is-dag` on an input graph, the following invariant holds: there is at most one root node in the host graph.

Proof. Upon inspection of the rules, `init` and `unroot` are the only rules adding or deleting root nodes in the host graph, both invoked in the looping sequence `(init; DFS!; try unroot else break)!`.

The input graph initially contains no root nodes. On the first invocation of the parenthetical looping sequence in `Main`, `init` adds one root node to the host graph. The looping procedure `DFS!` does not add nor delete root nodes. The rule `unroot` deletes the node added by `init` should it apply, or exits the loop. ☐

Proposition 4.3.2. Let G be a non-empty graph where all nodes are either grey or blue, at least one node is grey, all edges outgoing from blue nodes are marked blue, and all other edges are unmarked. Let v be the grey node marked blue and rooted by the rule `init` on G. Throughout the execution of `DFS!` on G, the following invariant holds: there exists exactly one path of red nodes such that v is the starting node, a root node (marked either red or green) is the endpoint, and all edges along the path are dashed.

Proof. The invariant holds in G, i.e. before `BFS!` is invoked. Now, assume the invariant holds at some stage of execution. Clearly, the rule `move` extends the dashed path by adding one edge and one red node, and `back` reduces the dashed path by removing one edge. These two rules are the only one dashing edges in the host graph, and no other rule deletes or adds root nodes (Proposition 4.3.1). Furthermore, `set_flag` and `loop` are the only rules altering the mark of the root node, and mark it green. Therefore, the invariant is maintained throughout the execution. ☐

Proposition 4.3.3. Let G be a non-empty graph where all nodes are either grey or blue, at least one node is grey, all edges outgoing from blue nodes are marked blue, and all other edges are unmarked. Let v be the grey node marked blue and rooted by the rule `init` on G. Upon the execution of `DFS!` on G, either `set_flag` or `loop` is applied if and only if there exists some directed cycle where all nodes in the cycle are reachable from v.

Proof. As established in Proposition 4.3.2, there exists exactly one path of red nodes starting from v, where the endpoint is marked either green or red. Initially, this path consists of exactly one root node marked red, v.

Observe that the rule `move` moves the root to a node directly reachable from the current root while leaving the previously rooted node marked red. The only rule that turns a red node blue is `back`, which corresponds to the *pop* operation in the DFS. Once a node is turned blue, it will no longer be revisited (this can be verified by inspecting the marks of the rules). The nodes that remain red are precisely those ancestral to the current rooted endpoint in the DFS.

If the rules `move` or `ignore` fail to apply, it implies that the second node (`node 2`) in these rules is already marked red. This condition indicates the presence of an edge from the current endpoint to a node already on the path, hence the existence of a directed cycle. Consequently, the rule `set_flag` is applied, and the looping procedure `DFS!` terminates.

Now, assume that some node u, reachable from v, has a looping edge. In this case, the rule `loop` applies before the invocation of `back`. The rule `loop` is called when the endpoint of the current path is u, which will eventually occur since all nodes reachable from v will be rooted during the DFS traversal, provided there is no edge-cycle (i.e. a cycle that is not a looping edge). ☐

Lemma 4.3.1 (Termination of `is-dag`)**.** The program `is-dag` terminates on any input graph.

Proof. An argument analogous to Lemma 4.1.1 can be made with respect to the program `is-dag`. □

Theorem 4.3.1 (Correctness of `is-dag`)**.** The program `is-dag` is totally correct with respect to the following specifications:

Input: An input graph.
Output: The program fails if and only if the input is cyclic.

Proof. Termination follows from Lemma 4.3.1. Let G be the input graph. If G is empty, `init` fails to match, `flag` fails to match, and the program terminates, outputting the empty graph and satisfying the specifications. Suppose now that G consists of at least one node. We then split the remainder of this proof into two cases and define the parenthetical sequence `(init; DFS!; try unroot else break)` as `R`.

Case 1. *G is acyclic.* Observe that `R` terminates under two conditions: (1) `init` fails to apply, or (2) `unroot` fails to apply.

The rule `init` fails to apply if and only if no grey node exists in the host graph, which indicates that all nodes have been visited in the DFS implemented by `DFS!`. By Proposition 4.3.3, the inapplicability of `init` implies that no directed cycle was found during any execution of `DFS!`, as `init` cannot be invoked after `unroot` fails to match. Consequently, `R` terminates, and the rule `flag` in `Check` fails to match. Furthermore, the rule `unroot` can never fail to apply in this context because the only rules capable of turning the root node green in `DFS!` are `set_flag` and `loop`. Both of these rules apply if and only if G contains a cycle, which it does not. Therefore, `unroot` will always succeed, leaving no green root for `flag` to apply successfully.

Case 2. *G is cyclic.* Upon the application of `init` to some node v in G, if a cycle is reachable from v, then either the rule `set_flag` or `loop` will eventually apply during the subsequent execution of `DFS!`. Consequently, `R` will terminate, and the rule `flag` will apply, causing the program to terminate with the execution of the `fail` command. If no cycle is reachable from v, however, then a cycle must exist within the set of nodes that are unreachable from v. These nodes will be visited in subsequent executions of `R` on G.

□

4.3.2 Proof of Complexity

Let us examine the time complexity of `is-dag`.

Proposition 4.3.4. Throughout the execution of `is-dag`, there is at most one red edge in the host graph.

Proof. Upon inspection of the rules, `next_edge` is the only rule that marks an edge red in the host graph (again, note that the program is structure-preserving). It is easy to see that when `next_edge` is applied, either `move` or `ignore` has to apply, or both fail to apply, triggering the looping sequence to terminate. Therefore, it follows that the number of red edges in the host graph is bounded to 1. ☐

Theorem 4.3.2 (Complexity of `is-dag`)**.** On any class of input graphs, the program `is-dag` terminates in time linear to the size of the input graph.

Proof. We first show that, for every rule, there is at most one matching attempt with respect to the complexity assumptions of the updated compiler (Figure 4.9).

An analogous argument to the proof of Theorem 4.1.2 can be made with respect to the rules `init`, `next_edge` (Proposition 4.3.4), `ignore`, `move`, and `back`, and the proof of Theorem 4.2.2 with respect to the rule `unroot`. Since there is at most one root node in the host graph, the rules `flag`, `loop` and `set_flag` only require at most one matching attempt.

Similarly, analogous arguments can be made to prove that the rules `next_edge`, `ignore` and `move` are called at most a number of times linear to the size of the graph. It is easy to see that the rule `init` can be called at most n times, where n is the number of nodes in the input graph (which does not vary). The rule `set_flag` is only called once as its application results in the termination of `(init; DFS!; try unroot else break)!`. The rule `loop` precedes the invocation of `back` and is therefore called as many times as the latter, thus a number of times linear to the size of the input graph. Finally, the rule `flag` is only called once in `Check`. Therefore, the program `is-dag` runs in time linear to the size of the input graph. ☐

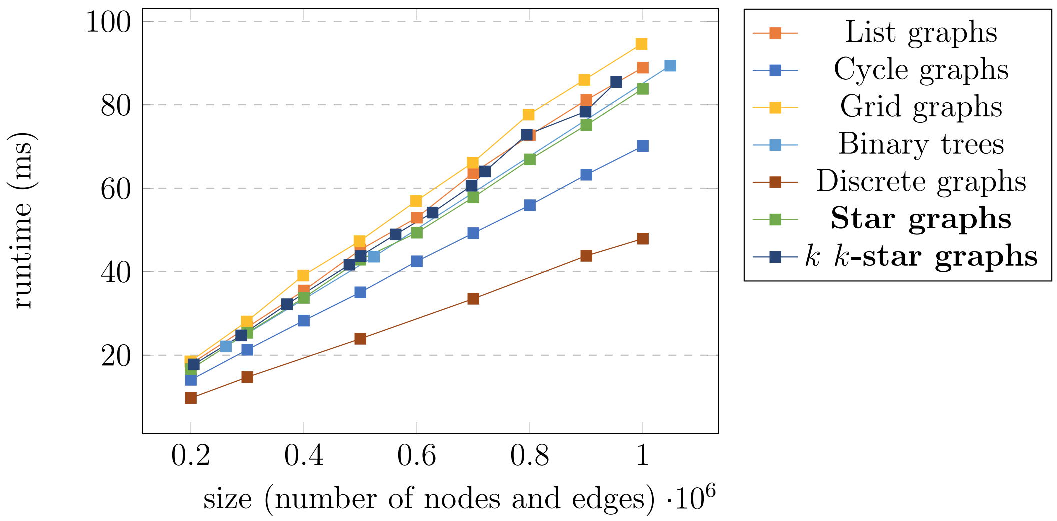


FIGURE 4.19: Measured performance of the program `is-dag`. Unbounded-degree graph classes are boldened in the legend.

Empirical timings are showcased in Figure 4.19 for various graph classes of bounded (Figures 4.1, 4.2, 4.4, 4.6 and 4.7) and unbounded degree (Figures and 4.3 and 4.8).

4.4 The Bellman-Ford Shortest Path Algorithm

In this section, we present the first implementation of a single-source shortest-path graph algorithm in GP 2 with a time complexity comparable to that of conventional imperative programming languages: $\mathcal{O}(nm)$, where n and m denote the numbers of nodes and edges, respectively. It computes the shortest distances from one node to all other nodes in the graph. The Bellman-Ford algorithm has been chosen for its simplistic elegance and its support for negative-value weights, in contrast to other popular alternatives, such as Dijkstra's shortest-path algorithm, which only accepts weights of positive value.

We will first examine a pseudo-code representation of the Bellman-Ford algorithm.

Algorithm 1 Bellman-Ford Algorithm

```
 1: Input: A weighted digraph $G = (V, E)$ of $n$ nodes and a source node $s \in V$
 2: Set $\texttt{dist}[v] = \infty$ for all $v$ in $V$
 3: Set $\texttt{dist}[s] = 0$
 4: for $n - 1$ iterations do
 5:     for each edge $e$ in $E$ do
 6:         $\texttt{dist}[t(e)] = min(\texttt{dist}[s(e)] + w(e),\ \texttt{dist}[t(e)])$
 7:     end for
 8: end for
 9: for each edge $e$ in $E$ do
10:     if $\texttt{dist}[s(e)] + w(e) < \texttt{dist}[t(e)]$ then
11:         Return FALSE
12:     end if
13: end for
14: Return TRUE
```

Algorithm 1, adapted from [SS03], provides a high-level overview of the Bellman-Ford algorithm. The chief idea is to iteratively decrease the upper bound of all distances $\texttt{dist}[\cdot]$ in the graph until no more upper bounds are left to be improved (hence reaching the optimal solution) or until the threshold of $n - 1$ iterations is reached, where n is the number of nodes in the graph. The following gives a descriptive account of each section of the code.

Lines 1 to 3. The algorithm begins by initialising an array $\texttt{dist}[\cdot]$. $\texttt{dist}[v]$ defines the lowest possible accumulated cost to go to v from the source node s in G. The upper bound distance for each node is initialised to ∞. Naturally, the distance from the source s to itself is 0.

Lines 4 to 8. This stage is called the *relaxation phase*. At each iteration, the algorithm iterates through all the edges in E and updates their upper bound whenever it finds a shorter distance. Let us consider the edge $e = (u,\ v)$. At the i^{th} iteration ($1 \leq i \leq n - 1$), the algorithm checks whether $\texttt{dist}[v]$ from the $(i - 1)^{\text{th}}$ iteration (we can consider the initialisation phase to be the 0^{th} iteration) can be further improved if there exists a shorter path to v via u with a weighted cost of $\texttt{dist}[u] + w(e)$. That process is repeated $n - 1$ times.

Lines 9 to 13. This section of the code detects negative cycles. The previous phase guarantees the computation of all shortest paths from s to all other nodes v in G, given that no negative cycle exists. If a negative cycle exists, it follows that there always exists a more optimal solution. The algorithm iterates through all edges and verifies whether a relaxation can still decrease at least one upper bound (which should have all been previously minimised). Should that be possible, this implies that a negative cycle exists and hence triggers the code to return `FALSE`.

Line 14. If the algorithm did not terminate during the negative cycle detection, it then successfully returns `TRUE` and exits. Alternatively, the algorithm could return the array `dist`$[\cdot]$.

The program `bellman-ford`[5] (Figure 4.20) takes a weighted graph, where edges are labelled with integers, and calculates the shortest distances from a single root node. It appends an integer label to each node, which represents its shortest distance from the root.

The program first creates a *counter* node labelled with the integer $n-1$, where n is the number of nodes in the input graph. The node is marked green, a mark unique to that node to allow for fast, constant-time access. The rule `set_counter` initialises this process by creating a green node labelled 0 and marking the root node blue to indicate it has been counted. The rule also unroots the node, creates a dashed edge from the counter to store the location of the original root, and appends the value 0 to the previously rooted node's label to denote that its distance to itself is 0.

The rule `count` is then applied iteratively to increment the counter for each grey node in the graph. As nodes are counted, they are marked blue to prevent duplicate counting. Additionally, the rule appends the string `f` to each node's label. Here, `f` serves as a representation of infinity, in practice commonly represented by a large positive integer in Bellman-Ford implementations, but is implemented as a string to avoid dependency on the integer size limitations of the GP 2 compiler. Define $l(v)$ as the value appended to the list label of node v by `set_counter` or `count`.

The sequence `(decrement; Relax!; Clean!)` implements the relaxation phase of the Bellman-Ford algorithm. The rule `decrement` decrements the value in the counter by 1 should that value be non-zero; this is to ensure that the relaxation phase happens exactly $n-1$ times. Initially, all nodes incident to at least one edge are marked blue in the host graph, no root exists, and all edges are unmarked. A single execution of `Relax` accomplishes the following:

1. The rule `root1` selects a blue node and roots it. At this stage, there is exactly one root in the host graph.
2. The rule `no_deg` checks whether the root node is isolated (notice that the node is not in the interface). If it is, it unmarks and unroots it to prevent

---

[5] The concrete syntax of the program available at: https://gist.github.com/ismaili-ziad/adc54b05aaff9ef6b3c57e7ea8a68543.

```
Main  = try set_counter
        then (count!; (decrement; Relax!; Clean!)!; Final)
Relax = root1; try no_deg
               else ((unmarked_edge; try {unvisited, reduce};
                    finish)!; unroot1)
Clean = root2; unmark_edge!; unroot2
Final = (root1; (unmarked_edge; if reduce
                                then set_flag; finish)!;
                                unroot1)!;
        if flag then fail; delete_counter; no_deg_inv!
```

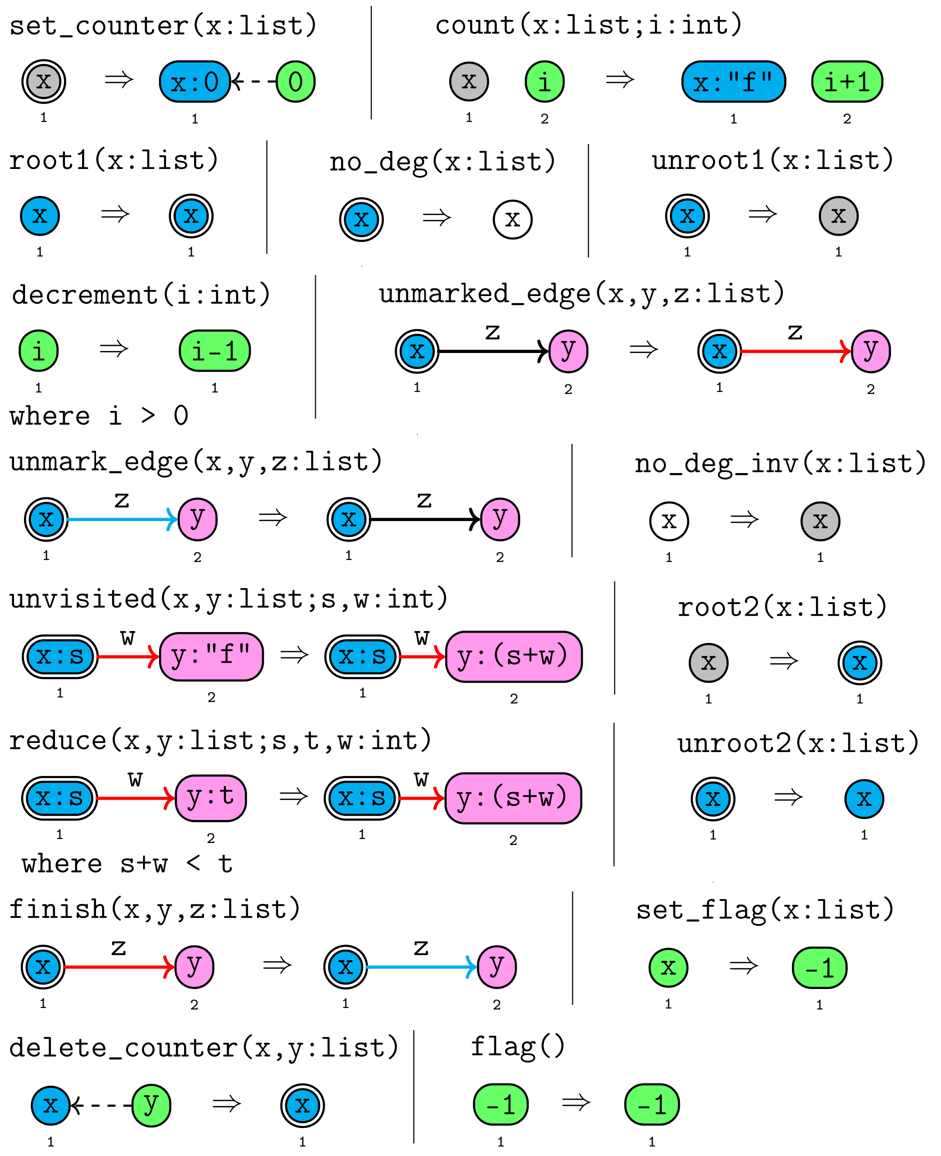


Figure 4.20: The program `bellman-ford`.

the procedure `Relax` from considering it again in a future iteration. Isolated nodes do not have outgoing edges, so they need not be considered in the relaxation phase.

3. In case the node is not isolated, the sequence `(unmarked_edge; try {unvisited, reduce})` is invoked as long as possible. `unmarked_edge` selects an unmarked edge outgoing from the root node and marks it red; let u and v be the source and target of the edge, respectively, and w the weight of the edge; then two possible cases arise:
    (a) $l(u)$ is an integer value (i.e. not the string `f`). If $l(v) =$ `f`, then the rule `unvisited` applies and sets $l(v)$ to $l(u) + w$. Otherwise, if $l(u) + w$ is strictly lesser than $l(v)$, the rule `reduce` applies and sets $l(v)$ to $l(u) + w$; else, none of the rules in the rule set `{unvisited, reduce}` applies.
    (b) $l(u)$ is the string `f`. None of the rules in the rule set `{unvisited, reduce}` applies.
4. The rule `finish` marks the selected edge blue to denote that it has been processed, so that it is no longer considered for relaxation until all other unmarked edges have been relaxed.
5. The rule `unroot1` unroots the selected root node and unmarks it, so that it is no longer considered as a source node for relaxation until all other grey nodes have been considered. After the application of this rule, there is no root node in the host graph.

Following an execution of `Relax!`, all edges in the host graph are marked blue (due to `finish`), and all non-isolated nodes (i.e. nodes whose degrees are greater than 0) are marked grey (due to `unroot1`). The looping procedure `Clean!` prepares the host graph for another round of edge relaxations by unmarking all edges and turning all grey nodes blue.

The procedure `Final` checks that no negative-weight cycle reachable from the source exists in the host graph by running a modified version of the procedure `Relax!` once more. Note that the counter node is labelled 0 at this stage. Should the rule `reduce` be applicable, the counter node's label is set to -1 by the rule `set_flag`.

Finally, the program checks the applicablity of the rule `flag`, which can only apply if and only if `set_flag` had been applied. If `flag` is applicable, it entails that a negative cycle was found, and the program fails. Otherwise, no negative cycle exists in the host graph and `no_deg_inv!` marks all unmarked nodes (i.e. isolated nodes) grey, as they were in the input graph.

We will now examine the correctness of the program `bellman-ford`.

4.4.1 Proof of Correctness

Definition 4.4.1 (Input Graph). An *input graph*, in the context of the program `bellman-ford`, is an arbitrarily-labelled GP 2 host graph such that:

- every node is marked grey,
- if the graph is non-empty, exactly one node is rooted,

- looping edges are not allowed, and
- every edge is unmarked and labelled with a list label consisting of one integer.

The input graph may contain negative cycles reachable from the root as the program `bellman-ford` is designed to detect them.

Proposition 4.4.1. Let G, I and H be graphs such that G is a non-empty input graph, and $G \Rightarrow_{\texttt{set\_counter}} I \Rightarrow_{\texttt{count!}} H$. The graph H is obtained by performing the following transformations on G:

1. All nodes are marked blue.
2. The integer 0 is appended to the list label of the root node.
3. The string `f` is appended to the list labels of other nodes (i.e. non-root nodes).
4. A new green-marked node is added, labelled with the integer $n-1$, where n is the number of nodes in G (prior to the addition of the node).
5. A new dashed-marked edge is added, with the green-marked node as the source and the root node in G as the target.
6. The root node is unrooted.

Proof. Let us first look at the application of the rule `set_counter`; that is, $G \Rightarrow_{\texttt{set\_counter}} I$. Initially, all nodes in G are grey and one node is rooted. Thus, the application of the rule `set_counter` marks the root node blue, appends 0 to its list label, creates a green node labelled 0 and generates a dashed edge from that newly created green node pointing at the previously rooted node. Now, let us examine the application of `count!`; that is, $I \Rightarrow_{\texttt{count!}} H$. I consists of $n+1$ nodes and m edges, where n and m are the numbers of nodes and edges in G, with one green node, one blue node and $n-1$ grey nodes. The rule `count` marks one grey node blue, appends the string `f` to its list label, and increments the counter by 1. Since the rule is applied as many times as there are grey nodes in I, the counter is labelled $n-1$ in H. Hence, H consists of one green node labelled 0, n blue nodes, m unmarked edges retaining their source and target nodes (notice that none of the rules involves edges), and one dashed edge from the green node to the node rooted in G. In addition, the node rooted in G has 0 appended to its list label in H, and all other nodes have the string `f` appended to their list labels. □

Lemma 4.4.1 (Termination of `Relax!`). The looping procedure `Relax!` terminates.

Proof. Let us show that the loop `(unmarked_edge; try unvisited, reduce; finish)!` terminates first. To show termination, consider a measure $\#(X)$ consisting of the number of unmarked edges in the host graph X. The rule `unmarked_edge` is invoked at the beginning of the loop, and if it fails to match, the loop breaks. Clearly, an application of `unmarked_edge` reduces the measure #. Since the number of edges is finite and no rule in the loop unmarks an edge, the rule eventually fails and the loop terminates.

Coming back to `Relax!`, now consider $\#(X)$ to be a measure consisting of the number of blue nodes in the host graph X. Clearly, an application of `root1` reduces the measure #. Since there is a finite number of nodes in the graph and no other rule in the procedure marks a node blue, the rule eventually fails and `Relax!` terminates. □

Lemma 4.4.2 (Termination of `Clean!`)**.** The looping procedure `Clean!` terminates.

Proof. Define $\#(X)$ as a measure consisting of the number of grey nodes in the host graph X. An application of `root2` clearly reduces that measure, and since there is a finite number of nodes in the host graph and no rule in `Clean` marks a node grey, the rule eventually fails and the loop terminates. □

Proposition 4.4.2. Upon the execution of the program `bellman-ford` on an input graph G, the looping sequence `(decrement; Relax!; Clean!)!` terminates after exactly $n-1$ iterations, where n is the number of nodes in G.

Proof. Termination of `Relax!` and `Count!` follow from Lemmata 4.4.1 and 4.4.2.

Recall from Proposition 4.4.1 that the host graph contains exactly one green node labelled $n-1$ after the execution of the loop `count!`. The looping sequence terminates if and only if the rule `decrement` fails to apply. However, since the rule `decrement` is the only rule in the sequence to involve the counter node, then the sequence must be invoked exactly $n-1$ times before `decrement` can no longer apply and the sequence terminates. □

Proposition 4.4.3. On a host graph following the application of `decrement` during the execution of `bellman-ford` on an input graph, the looping procedure `Relax!` relaxes each edge in the host graph exactly once.

Proof. Initially, all non-isolated nodes (beside the counter node) are blue; recall that isolated nodes need not be considered in the relaxation phase as they do not admit of any edge incident to them. As established in the proof of Lemma 4.4.1, `Relax!` terminates if and only if `root1` fails to apply. This can only be the case if there is no blue node in the host graph, since either `no_deg` or `unroot1` applies following an application of `unroot1`.

Once a blue node is rooted by `root1`, either `no_deg` applies and `Relax!` moves on to its next iteration, or it does not apply and the looping sequence `(unmarked_edge; try {unvisited, reduce}; finish)!` is executed instead. Given that all edges outgoing from non-counter nodes are unmarked initially, the rule `unmarked_edge` applies for any edge outgoing from the root node following the application of `root1` on that node. After such a rule is applied, if the root node's distance written on its list label is infinity (i.e. the string `f`), then the edge cannot be used to reduce the distance inscribed on the target of the edge and none of the rules in the rule set `{unvisited, reduce}` applies, and the program subsequently applies `finish` to mark the edge blue, to denote that it has been processed in the iteration of `Relax!`. Otherwise, the last variable on the list label of the root is an integer; in that case, either the edge is used to reduce the distance of the target node via the application of either `unvisited` in case the distance of the target node is still infinity, or the application of `reduce` should the weight added to the distance of the root be strictly smaller than the distance inscribed on the target of the edge, or none of those applies (the edge cannot be used to improve the distance on the target node) and the program applies `finish` to mark the edge blue to denote it has been processed in the iteration of `Relax!`.

The loop `(unmarked_edge; try {unvisited, reduce}; finish)!` terminates if and only if `unmarked_edge` is inapplicable (observe that `finish` will always apply following an application of `unmarked_edge`). Therefore, the loop will terminate when all outgoing edges are processed and turned blue by `finish`. Since the loop `Relax!` terminates once all blue nodes are processed by `root1`, all unmarked edges are incident to some blue node in the host graph and `(unmarked_edge; try {unvisited, reduce}; finish)!` terminates once all unmarked outgoing edges are processed by `unmarked_edge` and relaxed, then the looping procedure `Relax!` relaxes all edges in the host graph.

Finally, observe that no rule in the procedure `Relax` changes the mark of a blue edge; therefore, every edge is relaxed exactly once. □

Lemma 4.4.3 (Path Size)**.** *Let G be an input graph where s the root node in G, no negative-weight cycle is reachable from s, and t any node in G. Let P be the shortest path of edges from s to t such that the sum of the weights of the edges in the path is minimal among all paths from s to t. Then, the number of edges in P is at most $n-1$, where n is the number of nodes in G.*

Proof. For the sake of contradiction, assume that P contains more than $n-1$ edges. Since G contains n nodes, this implies that P must revisit at least one node, forming a cycle in the path. Let this cycle be denoted as C.

If the total weight of the edges in C is positive, removing the cycle would result in a path with a smaller total weight, contradicting the assumption that the sum of the weights of the edges in P is minimal among all paths from s to t. If the total weight of the edges in C is zero, removing the cycle does not change the total weight of the path. In this case, we can remove C and obtain a shorter path (in terms of the number of edges) with the same weight, contradicting the minimality of P as a shortest path. Finally, if the total weight of the edges in C is negative, then P is not well-defined as a shortest path because traversing C repeatedly would decrease the total weight of the path indefinitely. In this case, G would contain a negative-weight cycle, and the shortest path from s to t would not exist.

In all cases, we arrive at a contradiction. Therefore, P cannot contain more than $n-1$ edges. Since the number of edges in any simple path (i.e. a path that does not contain cycles) in G is at most $n-1$, the shortest path P from s to t must also satisfy this condition. □

For the remainder of the proof, define $l(v)$ as the value appended to the list label of node v by `set_counter` or `count` in the program `bellman-ford`.

Lemma 4.4.4 (Computation of Shortest Distances)**.** After k iterations of the loop `(decrement; Relax!; Clean!)!`, for every non-counter node v in the host graph reachable from the root node r in the input graph by a path of at most k edges, $l(v)$ is set to the shortest weighted distance from r to v by a path of at most k edges.

Proof. Let us proceed by induction on the number of iterations of the looping sequence `(decrement; Relax!; Clean!)!`.

When $k = 0$ (base case), the program set $l(r)$ to 0, and for all other nodes $v \neq r$, $l(v)$ was assigned the value `f`, representing infinity. At this stage, no edges were considered by the procedure `Relax`, so the shortest path to r from r, trivially of length 0, is correctly set to 0, and the shortest path to all other nodes remains undefined (i.e. infinity, the string `f`). Thus, the lemma holds for $k = 0$.

Now, assume the property holds for some k; we show it still holds for $k + 1$. Consider the $(k + 1)^{\text{th}}$ iteration of `(decrement; Relax!; Clean!)!`. During this iteration, the procedure `Relax!` processes all edges in the host graph (Proposition 4.4.3). Indeed, for each edge (u, v) with weight w, the program checks whether the distance to v can be improved by relaxing the edge (u, v) from a node u whose current shortest distance $l(u)$ has already been computed in prior iterations (by the induction hypothesis). The relaxation phase `Relax!` updates[6] $l(v)$ as follows:

$$l(v) \leftarrow \min\big(l(v), l(u) + w\big).$$

If there exists a path from r to v consisting of at most $k + 1$ edges, it must either:

1. pass through a node u such that the shortest path to u uses at most k edges, followed by the edge (u, v); or
2. be a path already computed during the first k iterations, using at most k edges.

Therefore, during the $(k+1)^{\text{th}}$ iteration, the looping procedure `Relax!` correctly updates $l(v)$ to the shortest distance from r to v using at most $k+1$ edges. □

Proposition 4.4.4. Upon the execution of `bellman-ford` on an input graph G, the rule `set_flag` is applied if and only if G contains a negative-weight cycle reachable from the root node.

Proof. Let r be the root node in G. From Lemma 4.4.3, it is established that the shortest path of edges from r to some arbitrary non-counter node v in the host graph such that the sum of the weights of the edges in the path is minimal among all paths from r to v consists of at most $n - 1$ edges (where n is the number of nodes in G), provided no negative-weight cycle reachable from r exists. Lemma 4.4.4 states that k iterations of the loop `(decrement; Relax!; Clean!)!`, called prior to `Final`, compute the smallest weighted distances from

---

[6] $\min\big(l(v), l(u) + w\big) = l(u) + w$ if $l(v)$ is not an integer.

r to any non-counter node v in the host graph by a path of at most k edges. Furthermore, Proposition 4.4.2 establishes that this loop terminates after $n-1$ iterations. Therefore, upon termination of `(decrement; Relax!; Clean!)!`, the program has computed the smallest distances from r to all other non-counter nodes in the host graph by a path of at most $n-1$ edges.

The procedure `Final` executes one iteration of `Relax`, during which only the rule `reduce` is considered for relaxation. If the rule `reduce` is applicable during this iteration, it implies that there exists some node u with $l(u)$ set to the shortest distance from r to u (by a path of at most $n-1$ edges), and an edge (u, v) with weight w such that $l(u) + w < l(v)$. In this case, the rule `reduce` would update $l(v)$ to $l(u) + w$.

For such a relaxation to occur after $n-1$ iterations, there must exist a path from r to v consisting of more than $n-1$ edges that has a smaller total weight than any path of at most $n-1$ edges. This is only possible if there exists a negative-weight cycle reachable from r. Indeed, by repeatedly traversing the negative-weight cycle, the total path weight can decrease indefinitely, which would for a smaller value of $l(v)$ to be computed.

When the rule `reduce` applies, the rule `set_flag` subsequently sets the counter node's label to -1. This serves as an indicator that a negative-weight cycle was detected. Finally, conversely, if no negative-weight cycle is reachable from r, then no edge relaxation can occur during the `Final` procedure, and the rule `set_flag` is never applied. □

Theorem 4.4.1 (Correctness of `bellman-ford`)**.** The program `bellman-ford` is totally correct with respect to the following specifications:

Input: An input graph.

Output: The program fails if and only if the input graph contains a negative-weight cycle reachable from the root. Otherwise, it produces a graph isomorphic to the input graph up to marks, such that:

- nodes unreachable from the root have the string `f` appended to their list label, and
- all other nodes have an integer appended to their list label denoting the smallest weighted distance from the root.

Proof. Termination easily follows from Lemmata 4.4.1 and 4.4.2, and Proposition 4.4.2. An argument analogous to the proof of Lemma 4.4.1 can be made with respect to the termination of `Final`.

Let G be the input graph. The theorem is trivially true if G were empty, since `set_counter` would not apply and the program would immediately terminate without invoking the `fail` command. Let us then assume that G contains at least one node and split the remainder of this proof into two cases.

Case 1. *G contains a negative-weight cycle reachable from the source.* Upon the termination of `count!`, the counter node is labelled $n-1$, where n is the number of nodes in G. By Proposition 4.4.2, the rule `decrement` is applied exactly $n-1$. Therefore, upon termination of the loop `(decrement; Relax!; Clean!)!`, the counter node is labelled 0. As per Proposition 4.4.4, `set_flag` will apply during the execution of `Final`, and given that `set_flag` is the only rule tempering with the list label of the counter node, an application of `flag` would entail that `set_flag` was applied, and a negative-weight cycle reachable from the root was found. The program would subsequently invoke the `fail` command and fail, satisfying the specifications of the program.

Case 2. *G does not contain a negative-weight cycle reachable from the source.* Analogously to the previous case, the counter node is labelled $n-1$ upon termination of the loop `(decrement; Relax!; Clean!)!`. Given that no negative-weight cycle reachable from the source node exists in G, the rule `set_flag` is never applied and the `fail` command is never invoked. The rule `delete_counter` returns the root to the source node and deletes the counter (observe that no rule beside `set_counter` and `delete_counter` involves a dashed edge). Finally, as all shortest distances from the source can be achieved by a path of at most $n-1$ edges (Lemma 4.4.3), the loop `(decrement; Relax!; Clean!)!` terminates after $n-1$ iterations (Proposition 4.4.2), and k iterations of the loop label the shortest weighted distances from the root to all other nodes by a path of at most k edges (Lemma 4.4.4), all nodes reachable from the source have an integer appended to their list label denoting the smallest weighted distance from the root. Nodes that are not reachable from the source retained the string `f`.

□

4.4.2 Proof of Complexity

Let us now analyse the time complexity of the program `bellman-ford`.

Proposition 4.4.5. Every rule in the program `bellman-ford` matches or fails to match in constant time with respect to the complexity assumptions of the updated compiler (Figure 4.9).

Proof. If the input graph is empty, the rule `set_counter` fails to match with a node immediately. Otherwise, a single root node exists and is matched by `set_counter` in constant time, as root nodes have direct access.

The rule `count` matches node `1` in constant time since the list label of the node can be arbitrary, and node `2` in constant time as well since there exists at most one green node in the host graph. Node `1` of `decrement`, `set_flag` and `flag` is also matched in constant time due to the bound on the number of green nodes.

The rule `root1` matches any blue node in the host graph of arbitrary list label, and thus in constant time. An analogous argument can be made for `root2`. Since there is exactly one root node after an application of `root1`, `no_deg` and `unroot1` match in constant time. An analogous argument can be made for `unroot2` with respect to `root2`.

The rule `unmarked_edge` matches node `1` in constant time due to the bound on the number of roots in the host graph, and matches any unmarked edge of arbitrary list label outgoing from the root in constant time. The rules `finish`, `unvisited` and `reduce` match node `1` and the red edge in constant time, since there is at most one red edge throughout the execution of the program `bellman-ford`. The rule `unmark_edge` matches node `1` and the blue edge in constant time, given that the latter can have an arbitrary list label.

Finally, `delete_counter` matches node `1` and the dashed edge in constant time, as there is at most one dashed edge throughout the execution of the program. And the rule `no_deg_inv` matches in constant time given that the list label of the node can be arbitrary. ☐

Lemma 4.4.5 (Handshaking Lemma)**.** For any graph G, the sum of the degrees of all nodes is equal to $2m$, where m is the number of edges in G.

Proof. Each edge in the graph contributes exactly 1 to the degree of its source node and 1 to the degree of its target node. Therefore, each edge increases the total sum of degrees by 2. Since there are m edges in the graph, the sum of the degrees of all vertices is $2m$. ☐

Proposition 4.4.6. For any graph G, let n' be the number of non-isolated nodes (i.e. nodes that are incident to no edge) in G. Then, n' is at most $2m$, where m is the number of edges in G.

Proof. Define $\deg v$ to be the degree of node v. Let V be the set of nodes in G, and $C = G - I$ where I is the set of isolated nodes in G. Clearly, there are as many edges in C as in G. Considering that C does not contain isolated nodes (i.e. nodes v such that $\deg v = 0$), every node $v \in V$ satisfies $1 \leq \deg v$. Thus, $\sum_{v \in V} \deg v \geq n'$. Therefore, by the handshaking lemma (Lemma 4.4.5), $2m \geq n'$. ☐

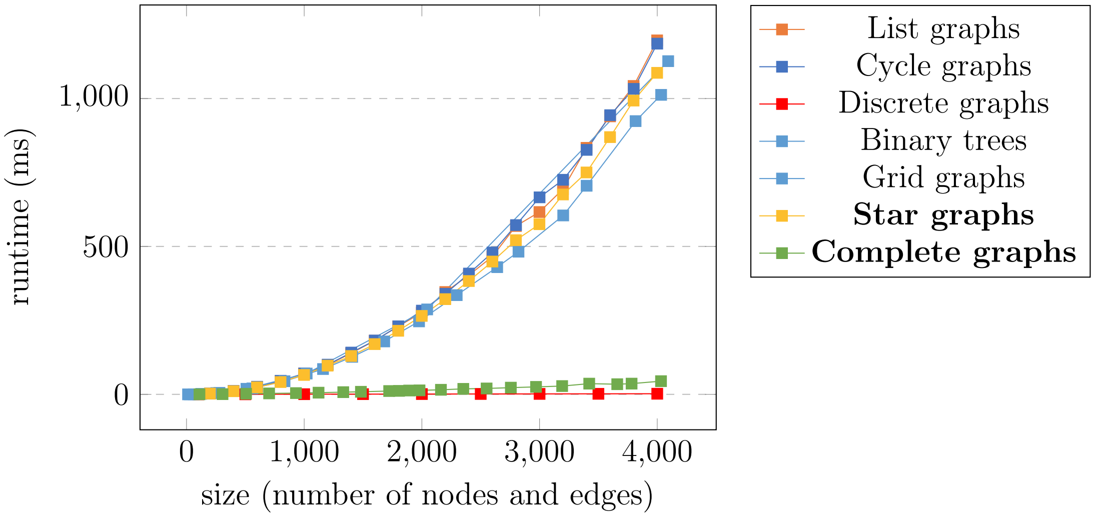


FIGURE 4.21: Measured performance of the program `bellman-ford`. Unbounded-degree graph classes are boldened in the legend.

Theorem 4.4.2 (Complexity of `bellman-ford`). On any class of input graphs, the program `bellman-ford` terminates in time $\mathcal{O}(nm)$, where n and m are the numbers of nodes and edges in the input graph, respectively.

Proof. We established in Proposition 4.4.5 that, for every rule, there is at most one matching attempt with respect to the complexity assumptions of the updated compiler (Figure 4.9). Let us now examine how many times each rule is called.

Clearly, the rules `set_counter`, `delete_counter` and `flag` are called at most once. Following an application of `set_counter`, there are $n-1$ grey nodes in the host graph, where n is the number of nodes in the input graph. The loop terminates when there is no grey node left, thus it is called n times. Proposition 4.4.2 states that the looping sequence `(decrement; Relax!; Clean!)` terminates after $n-1$ iterations, thus the rule `decrement` is called n times. Let that sequence be referred to as `R`. Prior to the very first iteration of `R`, during the execution of the program, isolated nodes are not unmarked. Following the iteration, every node that does not have an incidence to an edge is unmarked. Therefore, the rule `root1` is called $n+1$ times in the first iteration of `R` in the execution of the program, and $n'+1$ times on each iteration afterward, where n' is the number of non-isolated non-counter nodes in the host graph. The rule `no_deg` is called after each application of `root1`, thus n times on the first iteration of `R`, and n' times afterward. The loop `(unmarked_edge; try unvisited, reduce; finish)!` is called as many times as there are unmarked edges in the host graph and non-isolated nodes, thus $m+n'$ times where m is the number of edges in the input graph. The rule `unroot1` is called n' times, that is, as many times as there are non-isolated nodes.

In each execution of `Clean!`, the rule `root2` is called $n'+1$ times, and the rule `unroot2`, n' times. The rule `unmark_edge` is called $m+n'$ times (m successful applications and n' failed attempts to terminate the loop on each non-isolated node). Finally, the rule `root1` in `Final` is called at most $n'+1$ times, `unmarked_edge` is called at most $m+n'$ times, `reduce` at most m times and `finish`, at most m times.

By Proposition 4.4.6, n' is at most $2m$. Therefore, on all iterations of `(decrement; Relax!; Clean!)!` but the first one, a number of calls at most linear to the number of edges are called. On the first iteration, a number of calls linear to the size of the graph are called.

Therefore, the overall time complexity of the program `bellman-ford` is $\mathcal{O}(nm)$. □

We supplement the proof with empirical measurements presented in Figure 4.21. The low runtime of complete graphs, compared with the other graph classes, is due to the number of edges being quadratic in the number of nodes, and hence the runtime bound nm grows at a slower rate. For discrete graphs, given that there are no edges to relax, each iteration takes constant time.

Chapter 5

Conclusion and Future Work

In this thesis, we have demonstrated through several case studies how to modify the GP 2 compiler and implement efficient graph algorithms to achieve linear runtime, even for input graphs with unbounded node degrees or disconnected components. Addressing these challenges has been an open problem since the publication of the first paper on rooted graph transformation [PB12]. Until now, only specific graph reduction programs that destroy their input graphs could be designed to run in linear time on graph classes with unbounded-degree nodes or disconnected graphs [CCP22].

Our approach involves two key ingredients: enhancing the graph data structure generated by the GP 2 compiler, as detailed in Chapter 3, and developing programming techniques that leverage this improved graph representation, as extensively shown in the case studies of Chapter 4.

Previously, the graph data structure in the C program generated by the compiler stored lists of incoming and outgoing edges for each node. The matching algorithm relied on searching these lists to quickly locate edges corresponding to those in the left-hand side of a rule. However, when edges had different marks, these searches became linear-time operations, compromising constant-time rule matching as ideally desired. The new data structure addresses this limitation by enabling constant-time matching of incident edges with specific marks. This change is exploited in rules like `next_edge` in the `2-colouring` and `is-connected` programs, which assign a unique mark to an edge, and performs operations on it.

Additionally, the previous compiler implementation stored all host-graph nodes in a single linked list. When nodes had different marks, searching this list could require linear time. Our new data structure enables constant-time retrieval of both nodes and edges with specific marks and orientations (for edges). This feature of the new compiler is crucial for programs like `is-dag`, which determines whether a graph is acyclic. For instance, finding the next unvisited node for a depth-first search now requires constant time, unlike the previous implementation, which unnecessarily traversed multiple nodes to identify one with an appropriate mark.

Future Work

As avenue for future work, we speculate that virtually all linear-time graph algorithms based on depth-first or breadth-first search can be implemented as

GP 2 programs running in linear time.

The implementation of a GP 2 library of advanced data structures, such as priority queues, Fibonacci heaps, or AVL trees, to support programmers in constructing GP 2 versions of conventional graph algorithms that match the time complexity achievable in the imperative setting would be greatly advantageous. For instance, common implementations of Dijkstra's algorithm make use of heap trees, which can be embedded into a GP 2 program.

Bibliography

[Agr+06] Aditya Agrawal et al. “The design of a language for model transformations”. In: *Software & Systems Modeling* 5.3 (2006), pp. 261–288. DOI: 10.1007/s10270-006-0027-7.

[Bak15] Christopher Bak. “GP 2: Efficient Implementation of a Graph Programming Language”. PhD thesis. Department of Computer Science, University of York, UK, 2015. URL: https://etheses.whiterose.ac.uk/12586/.

[BFS03a] Gil Benkö, Christoph Flamm, and Peter F. Stadler. “A Graph-Based Toy Model of Chemistry”. In: *Journal of Chemical Information and Computer Sciences* 43.4 (2003). PMID: 12870897, pp. 1085–1093. DOI: 10.1021/ci0200570. eprint: https://doi.org/10.1021/ci0200570. URL: https://doi.org/10.1021/ci0200570.

[BFS03b] Gil Benkö, Christoph Flamm, and Peter Stadler. “Generic Properties of Chemical Networks: Artificial Chemistry Based on Graph Rewriting”. In: vol. 2801. Sept. 2003, pp. 1–11. ISBN: 978-3-540-20057-4. DOI: 10.1007/978-3-540-39432-7_2.

[BGP18] Guillaume Bonfante, Bruno Guillaume, and Guy Perrier. *Application of Graph Rewriting to Natural Language Processing*. Vol. 1. Logic, Linguistics and Computer Science Set. ISTE Wiley, Apr. 2018, p. 272. URL: https://inria.hal.science/hal-01814386.

[BP12] Christopher Bak and Detlef Plump. “Rooted Graph Programs”. In: *Proc. 7th International Workshop on Graph Based Tools (GraBaTs 2012)*. Vol. 54. Electronic Communications of the EASST. 2012. DOI: 10.14279/tuj.eceasst.54.780.

[BP16] Christopher Bak and Detlef Plump. “Compiling Graph Programs to C”. In: *Proc. 9th International Conference on Graph Transformation (ICGT 2016)*. Vol. 9761. Lecture Notes in Computer Science. Springer, 2016, pp. 102–117. DOI: 10.1007/978-3-319-40530-8_7.

[CCP22] Graham Campbell, Brian Courtehoute, and Detlef Plump. “Fast rule-based graph programs”. In: *Science of Computer Programming* 214 (2022), p. 102727. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2021.102727.

[Cor+09] Thomas H. Cormen et al. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[Cou23] Brian Courtehoute. "Time and space complexity of rule-based graph programs". PhD thesis. University of York, 2023. URL: https://etheses.whiterose.ac.uk/33407/.

[CRP20] Graham Campbell, Jack Romö, and Detlef Plump. "The Improved GP 2 Compiler". In: (2020). URL: https://arxiv.org/abs/2010.03993.

[CRPP91] Andrea Corradini, Francesca Rossi, and Francesco Parisi Presicce. "Logic Programming as Hypergraph Rewriting." In: Apr. 1991, pp. 275–295. ISBN: 978-3-540-53982-7. DOI: 10.1007/3-540-53982-4_16.

[Dör95] Heiko Dörr. *Efficient graph rewriting and its implementation*. Springer, 1995.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. "Graph-grammars: An algebraic approach". In: *14th Annual symposium on switching and automata theory (swat 1973)*. IEEE. 1973, pp. 167–180.

[FH00] H. Fahmy and R.C. Holt. "Using graph rewriting to specify software architectural transformations". In: *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. 2000, pp. 187–196. DOI: 10.1109/ASE.2000.873663.

[FKP19] Maribel Fernández, Hélène Kirchner, and Bruno Pinaud. "Strategic port graph rewriting: an interactive modelling framework". In: *Mathematical Structures in Computer Science* 29.5 (2019), pp. 615–662. DOI: 10.1017/S0960129518000270.

[FM07] Alessandro Folli and Tom Mens. "Refactoring of UML models using AGG". In: *ECEASST* 8 (Jan. 2007). DOI: 10.14279/tuj.eceasst.8.112.

[Gha+12] Amir Ghamarian et al. "Modelling and analysis using GROOVE". In: *International Journal on Software Tools for Technology Transfer* 14.1 (2012), pp. 15–40. DOI: 10.1007/s10009-011-0186-x.

[HP02] Annegret Habel and Detlef Plump. "Relabelling in Graph Transformation". In: *Graph Transformation*. Ed. by Andrea Corradini et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 135–147. ISBN: 978-3-540-45832-6.

[IAP24] Ziad Ismaili Alaoui and Detlef Plump. "Linear-Time Graph Programs for Unbounded-Degree Graphs". In: *Graph Transformation*. Springer Nature Switzerland, 2024, 3–20. ISBN: 9783031642852. DOI: 10.1007/978-3-031-64285-2_1. URL: http://dx.doi.org/10.1007/978-3-031-64285-2_1.

[IAP25] Ziad Ismaili Alaoui and Detlef Plump. "Linear-Time Graph Programs without Preconditions". In: Proceedings of the Fourteenth and Fifteenth International Workshop on *Graph Computation Models,* Leicester, UK and Enschede, the Netherlands, 18 July 2023 and 9 July 2024. Ed. by Jörg Endrullis et al. Vol. 417. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2025, pp. 39–54. DOI: `10.4204/EPTCS.417.3`.

[JBK10] Edgar Jakumeit, Sebastian Buchwald, and Moritz Kroll. "GrGen.NET – The expressive, convenient and fast graph rewrite system". In: *International Journal on Software Tools for Technology Transfer* 12.3–4 (2010), pp. 263–271. DOI: `10.1007/s10009-010-0148-8`.

[Kol+22] Lothar Kolbeck et al. "Graph Rewriting Techniques in Engineering Design". In: *Frontiers in Built Environment* 7 (2022). ISSN: 2297-3362. DOI: `10.3389/fbuil.2021.815153`.

[MP08] Greg Manning and Detlef Plump. "The GP Programming System". In: *ECEASST* 10 (Jan. 2008). DOI: `10.14279/tuj.eceasst.10.150`.

[PB12] Detlef Plump and Christopher Bak. "Rooted Graph Programs". In: *Electronic Communications of the EASST* 54 (2012). DOI: `10.14279/tuj.eceasst.54.780`. URL: `https://eceasst.org/index.php/eceasst/article/view/2412`.

[PE93] Marinus Plasmeijer and Marko Eekelen. *Functional Programming and Parallel Graph Rewriting.* Jan. 1993. ISBN: 0-201-41663-8.

[Plu12] Detlef Plump. "The Design of GP 2". In: (2012). DOI: `10.4204/EPTCS.82.1`.

[Plu17] Detlef Plump. "From imperative to rule-based graph programs". In: *Journal of Logical and Algebraic Methods in Programming* 88 (2017), pp. 154–173. ISSN: 2352-2208. DOI: `https://doi.org/10.1016/j.jlamp.2016.12.001`.

[Ski20] Steven S. Skiena. *The Algorithm Design Manual.* third. Springer, 2020. DOI: `10.1007/978-3-030-54256-6`.

[SS03] Robert Sedgewick and Michael Schidlowsky. *Algorithms in Java, Part 5: Graph Algorithms.* 3rd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0201361213.

[Str+17] Daniel Strüber et al. "Henshin: A Usability-Focused Framework for EMF Model Transformation Development". In: *Proc. 10th International Conference on Graph Transformation (ICGT 2017).* Vol. 10373. Lecture Notes in Computer Science. Springer, 2017, pp. 196–208. DOI: `10.1007/978-3-319-61470-0_12`.