Evolving comprehensible and scalable solvers using CGP for solving some real-world inspired problems

Patricia Ryser-Welch

PhD

University of York

Electronic Engineering

July 2017

Abstract

My original contribution to knowledge is the application of Cartesian Genetic Programming to design some scalable and human-understandable metaheuristics automatically; those find some suitable solutions for real-world NP-hard and discrete problems. This technique is thought to possess the ability to raise the generality of a problem-solving process, allowing some supervised machine learning tasks and being able to evolve non-deterministic algorithms.

Two extensions of Cartesian Genetic Programming are presented. Iterative Cartesian Genetic Programming can encode loops and nested loop with their termination criteria, making susceptible to evolutionary modification the whole programming construct. This newly developed extension and its application to metaheuristics are demonstrated to discover effective solvers for NP-hard and discrete problems. This thesis also extends Cartesian Genetic Programming and Iterative Cartesian Genetic Programming to adapt a hyper-heuristic reproductive operator at the same time of exploring the automatic design space. It is demonstrated the exploration of an automated design space can be improved when specific types of active and non-active genes are mutated.

A series of rigorous empirical investigations demonstrate that lowering the comprehension barrier of automatically designed algorithms can help communicating and identifying an effective and ineffective pattern of primitives. The complete evolution of loops and nested loops without imposing a hard limit on the number of recursive calls is shown to broaden the automatic design space. Finally, it is argued the capability of a learning objective function to assess the scalable potential of a generated algorithm can be beneficial to a generative hyper-heuristic.

Contents

Abstract **3**

List of Figures **vi**

List of Tables **xii**

Acknowledgements **xxi**

Declaration **xxiii**

1 Introduction **1**

1.1 Thesis aims and contributions 2

1.2 Plan of thesis 5

2 Optimisation of algorithms **7**

2.1 Basic principles 8

2.2 The problem domain 10

2.2.1 The problem search space 10

2.2.2 The problem encoding scheme 11

2.2.3 The problem evaluation process 11

2.2.4 Problem-specific operators 12

2.2.5 Problem parameters 12

2.2.6 Discussion 13

2.3 The algorithm domain 14

2.3.1 The algorithm search space 14

2.3.2 The algorithm encoding scheme 16

2.3.3 The learning objective function 17

2.3.4 The algorithm parameters 18

2.3.5 The algorithm understandability metrics 20

2.3.6 Discussion 21

2.4 Algorithm optimisation processes 23

2.4.1 Predicting the performance of algorithms 24

2.4.2 Automating parameter settings 26
2.4.3 Selection of operators 27
2.4.4 Generation of algorithms 31
2.4.5 Discussion 35
2.5 Conclusion 39

3 Three problem domains **41**
3.1 Common features 43
3.1.1 Population operators 44
3.1.2 Termination criteria 45
3.1.3 Summary 48
3.2 The Mimicry Problem 49
3.2.1 The chosen encoding scheme 50
3.2.2 Fitness evaluation 51
3.2.3 Problem parameters 52
3.2.4 Problem operators 53
3.2.5 Summary 57
3.3 The Traveling Salesman Problem 58
3.3.1 The chosen encoding scheme 60
3.3.2 Fitness evaluation 61
3.3.3 Parameters 63
3.3.4 Problem operators 64
3.3.5 Summary 69
3.4 The Nurse Rostering Problem 70
3.4.1 The chosen encoding scheme 73
3.4.2 Fitness evaluation 76
3.4.3 Problem Operators 80
3.4.4 Summary 84
3.5 Discussion and conclusion 85

4 Graph-Based GP **87**
4.1 Review of graph-based genetic programming 88
4.1.1 Parallel distributed genetic programming 89

4.1.2 Linear-graph genetic programming 90
4.1.3 Graph structure program evolution 92
4.1.4 Parallel Algorithm Discovery and Orchestration 93
4.1.5 Cartesian Genetic Programming 94
4.1.6 Implicit-context CGP 97
4.1.7 Adaptive Cartesian Harmony Search 98
4.1.8 Discussion 99
4.2 CGP hyper-heuristics 101
4.2.1 Cartesian Genetic Programming 101
4.2.2 Iterative Cartesian Genetic Programming 104
4.2.3 Autoconstructive Cartesian Genetic Programming 108
4.3 Conclusion 117

5 Evolving metaheuristics 119
5.1 Introduction 119
5.2 Learning objective function 120
5.3 Evolving the body of a loop 121
5.3.1 Validation 123
5.4 Iterative Cartesian Genetic Programming: the full evolution of loops . 127
5.4.1 Validation of the learnt iterative metaheuristics 129
5.5 Discussion and conclusion 133

6 Improved learning objective process 135
6.1 Introduction 136
6.2 Problem domain 139
6.2.1 Traveling salesman problem 139
6.2.2 Mimicry problem 141
6.2.3 Nurse rostering problem 142
6.3 Evolution of the body of a loop 145
6.3.1 Discovery of Traveling Salesman Problem solvers 146
6.3.2 Performance 148
6.3.3 Discovery of Mimicry problem solvers 152
6.3.4 Discovery of nurse rostering problem solvers 156

6.4 The full evolution of loops 162
6.4.1 Discovery of iterative Travelling salesman solvers 163
6.4.2 Discovery of iterative mimicry solvers 165
6.4.3 Discovery of iterative nurse rostering problem solvers 169
6.4.4 Performance and comparison 171
6.5 Discussion and conclusion 171

7 Evolving hyper-heuristic reproductive operators 173
7.1 Introduction 173
7.2 Experiments 174
7.2.1 Discovering sequential and iterative mimicry solvers 176
7.2.2 Discovering sequential and iterative traveling salesman solvers 176
7.2.3 Genetically improving some CGP mutation operators 176
7.2.4 Effect of the online generative hyper-heuristics 179
7.2.5 Comparison to an offline learning process 180
7.3 Validation of a learnt CGP mutation 182
7.3.1 Performance of discovered NRP solvers 183
7.3.2 Effect of the learnt CGP mutation operators 185
7.4 Discussion and conclusion 186

8 Critical analysis 187
8.1 Scalable patterns of primitives 188
8.1.1 The Traveling Salesman Problem 188
8.1.2 The mimicry problem 192
8.1.3 The nurse rostering problem 196
8.2 Automatic design of metaheuristics 200
8.2.1 Templates and directed graphs 201
8.2.2 Effect of the learning objective functions 204
8.2.3 Effectiveness of the learning 207
8.3 Comprehensibility metrics 208
8.3.1 Problem-specific solvers 209
8.3.2 Other forms of GP 212
8.3.3 Effect on human understandability metrics 214

8.3.4 Comparison with other techniques 216
8.3.5 Discussion 221
8.4 Conclusion 222

9 Conclusion **223**
9.1 Recommendations 225
9.2 Future work 226

Appendix A: Algorithms **229**

Appendix B: Statistical results **253**

Statistical results **253**

Abbreviations **311**

Glossary **313**

References **319**

List of Figures

2.1 A decomposition of the algorithm search space 15
2.2 Evolutionary fitness landscape with a three and two dimensional representation model [87] 19
2.3 A comparison of hyper-heuristics, metaheuristics and heuristics [31] . 37
2.4 Classification of meta-learning methods as proposed by Pappa et al. [250] 38
2.5 Classification of hyper-heuristic techniques as proprosed by Pappa et al. [250] 38

3.1 A process showing the mechanism of a loop 46
3.2 An optimum solution of a the mimicry problem, for an instance of 10 bits 49
3.3 Solutions of the mimicry problems for an instance of 10 bits. A Hamming distance is given with the fitness value of each solutions. The bits in bold in the imitator are dissimilar from the the prototype. . . . 51
3.4 Crossovers techniques applied to the imitators of the two solutions . . 54
3.5 Mutation techniques applied to the imitators of a solution 56
3.6 Possible solutions for the Icosian game [252] 58
3.7 An optimum solution of a TSP instance made of 5 cities. 59
3.8 Examples of TSP solutions for a 5-city instance. 62
3.9 Examples of TSP solutions for a 5-city instance. 66
3.10 Examples of TSP solutions for a 5-city instance. 67
3.11 An example of a 2-Opt Local Search [153] 69
3.12 An example of a 3-Opt Local Search [153] 69
3.13 A formulation of work patterns using a linear programming problem as suggested by [84] 72
3.14 A formulation of work patterns using for the nurse rostering problem [3] 72
3.15 A visual representation of the nurse rostering problem [57] 73
3.16 A description of shifts [48] 78
3.17 *New swap* techniques used by the *NewSwaprLocalSearch* [83] 82

3.18 Horizontal swap used by the *HorizontalSwapLocalSearch* [83] 83
3.19 Vertical swap used by the *VerticalSwapLocalSearch* [83] 83

4.1 An example of a program encoded in grid [255] 89
4.2 An example of a linear gp individual as provided by [160] 91
4.3 An example of a linear gp individual as provided by [160] 91
4.4 An GRAPE program with its data set as given by [282] 92
4.5 A PADO example program [309] 93
4.6 A graphical representation of a CGP graph [226] 95
4.7 An example of how a ICGP graph is interpreted as provided by [293] . 98
4.8 An example of how a CGP graphs provided by [99] 99
4.9 A solver expressed with its active nodes 102
4.10 A solver expressed with its active nodes 104
4.11 : autoconstructive CGP graphs. The top individual encodes an algorithm with directed acyclic graph and the bottom individual with iterative CGP graph. Both individual encodes a mutation operators with an iterative CGP graph. All the branching genes are represented with blue arrows. 109

5.1 CGP graphs representing the TSP solvers B as described in algorithms 5.3 123
5.2 A comparison of the the solvers TSP-[A-C] and TSP-[R-T] during the search for an optimum tour for the learning benchmark pr439. 126
5.3 A statistical comparison of solvers TSP-A, TSP-B and TSP-C for the instance *eg7146* 127
5.4 CGP graphs representing the TSP solvers D and described in algorithms 5.4 129
5.5 A comparison of the solvers TSP-[D-E] and TSP-[U-W] during the search for an optimum tour for the learning benchmark pr439. 131
5.6 A statistical comparisons of TSP-A, TSP-B, TSP-C,TSP-D and TSP-E over a 100 runs with 6,000 problem evaluations. The mean and standard deviation are represented with a diamond shape. 132

6.1 Results of a game with a target with three challenges 137

6.2 A statistical comparisons of TSP-[F-H] over a 100 runs with 6,000 problem evaluations. The mean and standard deviation are represented with a diamond shape. 148
6.3 The outline of histograms showing the statistic distribution of traveling salesman problem solvers obtained with **a two-operator exhaustive search**. 149
6.4 The outline of histograms showing the statistic distribution of traveling salesman problem solvers obtained with **a three-operator exhaustive search**. 150
6.5 The outline of histograms showing the probability distribution of traveling salesman solvers obtained with **our offline non-iterative optimisation process**. 150
6.6 The outline of histograms showing the probability distribution of mimicry solvers obtained with **our offline non-iterative optimisation process**. The learning instances are used with a maximum number of problem evaluations of 1,500. 153
6.7 The outline of histograms showing the probability distribution of mimicry solvers obtained with **a four-operator exhaustive search** 156
6.8 A statistical comparison of solvers NRP-A, NRP-B and NRP-C for the instance BCV-3.46.1 159
6.9 A statistical comparison of solvers NRP-A, NRP-B and NRP-C for the instance G-Post 159
6.10 A statistical comparison of solvers NRP-A, NRP-B and NRP-C for the instance BCV-1.8.4 160
6.11 The outline of histograms showing the statistic distribution of the nurse rostering problem solvers obtained with **a two-operator exhaustive search**. The learning instances are used with a maximum number of problem evaluations of 40. 161
6.12 A statistical comparison of solvers TSP-I, TSP-J and TSP-K for the instance eg7146 164

6.13 A comparison of the solvers MC-[D-F] and MC-M during the search for a perfect imitator for a 800-bit benchmark. 1,500 problem evaluations were used. 166
6.14 A statistical comparison of solvers MC-C, MC-D and MC-E for the 3000-bit instance 167
6.15 A statistical comparison of solvers MC-C, MC-D and MC-E for the 5000-bit instance 168
6.16 A statistical comparison of solvers MC-C, MC-D and MC-E for the 10000-bit instance 168

7.1 Autoconstructive CGP graphs. The top individual encodes an algorithm with directed acyclic graph and the bottom individual with iterative CGP graph. Both individual encodes a mutation operators with an iterative CGP graph. All the branching genes are represented with blue arrows. 175
7.2 A statistical comparison of solvers NRP-E, NRP-F and NRP-G for the instance BCV-1.8.4 185
7.3 A statistical comparison of solvers NRP-H, NRP-I and NRP-J for the instance BCV-1.8.4 185

8.1 A graphical representation of the expected fitness of for the algorithms *TSP-A to TSP-E* and the metaheuristics published by [246, 319]. The instances ranges between 38 to 7663 cities 189
8.2 A graphical representation of the expected fitness of for the algorithms *TSP-F to TSP-Q* and the metaheuristics published by [246, 319]. The instances ranges between 38 to 7663 cities 190
8.3 A graphical representation of the expected fitness of for the algorithms *TSP-A to TSP-E* and the metaheuristics published by [246, 319]. The instances ranges between 9,152 to 33,708 cities 190
8.4 A graphical representation of the expected fitness of for the algorithms *TSP-F to TSP-Q* and the metaheuristics published by [246, 319]. The instances ranges between 9,152 to 33,708 cities 191

8.5 A graphical representation of the expected fitness of each algorithms for the instances ranging between 100 to 30,000 bits. 193
8.6 A graphical representation of the expected fitness of each algorithms for the instances ranging between 100 to 100,000 bits. 195
8.7 A graphical representation of the best expected fitness obtained by solvers *NRP-A - NRP-J*. The limit approximately varies between $[-0.8..2]$. 197
8.8 A graphical representation of the best expected fitness obtained by solvers *NRP-A - NRP-J*. The limit approximately varies between $[0..35]$. 197
8.9 A graphical representation of the best expected fitness obtained by solvers *NRP-A - NRP-J*. The limit approximately varies between $[0..1,320]$. 198
8.10 An hyper-heuristic searching the multiple areas of the algorithm search space. Each search algorithm has potentially a different set of problems associated to it [257]. 200
8.11 The metaheuristics design space 201
8.12 An example of published algorithm generated by a generative tree-based hyper-heuristics. 212

List of Tables

3.1 Problem statement of the mimicry problem 49
3.2 Mimicry operators with their opcode and the number of evaluations used. 57
3.3 TSP Problem statement 59
3.4 Traveling Salesman operators with their opcode and the number of evaluations used. 70
3.5 Nurse rostering operators with their opcode and the number of evaluations used. 84

4.1 a list encoding the label, the coordinates of the nodes, and the horizontal displacement for example given in figure 4.1 89
4.2 This table summarises the function set of each subspecies of a reproductive mechanism. The operators formatted in italic are only applied to the iterative algorithm. 117
4.3 This table summarises the condition set of each subspecies of reproductive operators 117

5.1 Parameters of the Classic CGP 122
5.2 The function set made of TSP-specific and population operators . . . 123
5.3 State of populations p and t at generation 7 during a validation run. The solver TSP-B was used with the validation instance d1291. 124
5.4 State of populations p and t at generation 7 during a validation run. The solver TSP-T was used with the validation instance d1291. 125
5.5 Parameters of the Iterative CGP for all the tests 128
5.6 Function set: List of TSP heuristics used as primitives. 129
5.7 Condition set: Boolean primitives chosen for the stopping criterion. . 129
5.8 State of populations p and t when 2,990 algorithm evaluations have been used during a validation run. The solver TSP-T was used with the validation instance d1291. 131

6.1 Function set: List of TSP heuristics used as primitives. 139

6.2 Parameters of the metaheuristics for all the test 140
6.3 Parameters of the metaheuristics for all the learning test 141
6.4 Mimicry operators with their opcode and the number of evaluations used. 142
6.5 Nurse rostering operators with their opcode and the number of evaluations used. 143
6.6 Parameters of the metaheuristics for all the test 143
6.7 Parameters of our CGP hyper-heuristics 146
6.8 A comparison of the tours likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs. 147
6.9 Median of tours obtained in Chesc 2011, automatic design of selective hyper-heuristics [275], automatically designed selective-hyper-heuristics [120] and our experiments. The table reports the median tours using arelative error to the known optima. 151
6.10 A comparison of the tours likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs. 154
6.11 State of the populations p and t at generation 1 during a learning run. The solver MC-M was used with the learning instance 800. 155
6.12 State of the populations p and t at generation 1 during a learning run. The solver MC-A was used with the learning instance 800. 155
6.13 A comparison of the rosters likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs. 157
6.14 State of the populations p and t at generation X during a validation run. The solver the solver NRP-A was used with the learning instance Ikegami-2Shift-DATA1. 158
6.15 State of the populations p and t at generation X during a validation run. The solver the solver NRP-K was used with the learning instance Ikegami-2Shift-DATA1. 158

6.16 A comparison of the averages of rosters obtained by [21] and our experiments. The table shows the relative error of the results. 161
6.17 Iterative CGP parameters applied in these experiments. The parameters in bold have been refined and differ from our experiments in chapter 5 162
6.18 Condition set: Boolean primitives chosen for the stopping criterion. The conditions formatted in bold are different from our previous experiments. 162
6.19 A comparison of the tours likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs. 163
6.20 A comparison of the imitators likely to be obtained during a learning run (i.e. by the learning objective function) and those obtained by 100 independent validation runs. 167
6.21 Condition set: Boolean primitives chosen for the stopping criterion. . 169
6.22 A comparison of the rosters likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs. 170

7.1 Reproductive operators parameters 175
7.2 Some examples of genetically improved CGP mutation altering iterative CGP graphs. 178
7.3 A comparison of the rosters likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs. 184

8.1 Software metrics for the traditional metaheuristics as expressed by Luke et al.[205] 209
8.2 Software metrics applied to the generated mimicry solvers and Herdy's evolution strategy [146] 210
8.4 Software metrics applied to the generated NRP solvers 210

8.3 Software metrics applied to the generated TSP solvers, some metaheuristic written by human-activity, and also some solvers generated with tree-base GP 211
8.5 Summary of metaheuristics understandability metrics evolved with a non-graph-based form of genetic programming 213

B.1 Statistical comparison for some mimicry solvers generated in chapters [5-7] 256
B.2 Statistical comparison for some mimicry solvers generated in chapters [5-7] 257
B.3 Statistical comparison of imitators obtained by Herdy [146] and the generated solvers MC-[A-L] 258
B.4 Statistical comparison of imitators obtained by generated solver MC-A and the generated solvers MC-[B-L] 259
B.5 Statistical comparison of imitators obtained by generated solver MC-B and the generated solvers MC-[C-L] 260
B.6 Statistical comparison of imitators obtained by generated solver MC-C and the generated solvers MC-[D-L] 261
B.7 Statistical comparison of imitators obtained by generated solver MC-D and the generated solvers MC-[E-L] 262
B.8 Statistical comparison of imitators obtained by generated solvers MC-[E-F] and the generated solvers MC-[G-L] 263
B.9 Statistical comparison of imitators obtained by generated solvers MC-[G-I] and the generated solvers MC-[J-L] 264
B.10 Statistical comparison of imitators obtained by generated solvers MC-[J-K] and the generated solver MC-L] 265
B.11 Statistical comparison of some imitators obtained by some mimicry solvers. 266
B.12 Statistical comparison of imitators obtained by Herdy [146] and the generated solver MC-A, and the generated solvers MC-A, MC-E, MC-L, MC-M 267
B.13 Statistical comparison of imitators obtained the generated solver MC-D, and the generated solvers MC-E, MC-L, MC-M 267

B.14 Statistical comparison of tours obtained by solvers TSP[A-J] for the instances u2152,usa13509, d18512, dj38, q194 and zi929 269
B.15 Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for the instances u2152,usa13509, d18512, dj38, q194 and zi929 270
B.16 Statistical comparison of tours obtained by solvers TSP[A-J] for the instances lu980,rw1621,nu3496, ca4663, tz6117, eg7146. 271
B.17 Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for the instances lu980,rw1621,nu3496, ca4663, tz6117, eg7146. 272
B.18 Statistical comparison of tours obtained by solvers TSP[A-J] for the instances ym7663,ei8246,ar9152, ja9847, gr9882, and kz9976. 273
B.19 Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for the instances ym7663,ei8246,ar9152, ja9847, gr9882, and kz9976. 274
B.20 Statistical comparison of tours obtained by solvers TSP[A-J] for some instances with a greater number 10,000 cities. 275
B.21 Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for some instances with a greater number 10,000 cities. 276
B.22 Statistical comparison of tours obtained by Ulder [319] and Ozcan [247] and the generated solvers TSP-[A-E] 277
B.23 Statistical comparison of tours obtained by Ulder [319] and the generated solvers TSP-[F-Q] 278
B.24 Statistical comparison of tours obtained by Ozcan [247] and the generated solvers TSP-[F-Q] 279
B.25 Statistical comparison of tours obtained by the generated solvers TSP-[A-E] 280
B.26 Statistical comparison of tours obtained by the generated solver TSP-A and the generated solvers TSP-[F-Q] 281
B.27 Statistical comparison of tours obtained by the generated solver TSP-B and the generated solvers TSP-[F-Q] 282

B.28 Statistical comparison of tours obtained by the generated solver TSP-C and the generated solvers TSP-[F-Q] 283
B.29 Statistical comparison of tours obtained by the generated solver TSP-D and the generated solvers TSP-[F-Q] 284
B.30 Statistical comparison of tours obtained by the generated solver TSP-E and the generated solvers TSP-[F-Q] 285
B.31 Statistical comparison of tours obtained by the generated solver TSP-F and the generated solvers TSP-[G-Q] 286
B.32 Statistical comparison of tours obtained by the generated solver TSP-G and the generated solvers TSP-[H-Q] 287
B.33 Statistical comparison of tours obtained by the generated solver TSP-H and the generated solvers TSP-[I-Q] 288
B.34 Statistical comparison of tours obtained by the generated solver TSP-I and the generated solvers TSP-[J-Q] 289
B.35 Statistical comparison of tours obtained by the generated solver TSP-J and the generated solvers TSP-[K-Q] 290
B.36 Statistical comparison of tours obtained by the generated solvers TSP-[K-L] and the generated solvers TSP-[M-Q] 291
B.37 Statistical comparison of tours obtained by the generated solvers TSP-[M-Q] 292
B.38 Definition of a nurse rostering instances, with the number of nurses, types of shift and days. 293
B.39 Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-1.8.1, BCV-1.8.2, BCV-1.8.3, BCV-1.8.4 and BCV-2.46.1. 294
B.40 Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-3.46.1, BCV-3.46.2, BCV-4.13.2, BCV-5.4.1. . 295
B.41 Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-6.13.1, BCV-6.12.2, BCV-7.10.1, BCV-8.13.1 and BCV-8.13.2. 296

B.42 Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-A.12.1, BCV-A.12.2, Instance 2, Instance 3, and Instance 4. 297
B.43 Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances Instance 5, Instance 6, Instance 7, Instance 9 and Instance 10. 298
B.44 Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances ORTECO1, ORTECO2, G-Post, G-Post-B, and Ikegami-3Shift-Data.1. 299
B.45 Statistical comparison of the generated solver NRP-A and the generated solvers NRP[A-J]. 300
B.46 Statistical comparison of the generated solver NRP-A and the generated solvers NRP[A-J]. 301
B.47 Statistical comparison of the generated solver NRP-B and the generated solvers NRP[C-J]. 302
B.48 Statistical comparison of the generated solver NRP-B and the generated solvers NRP[C-J]. 303
B.49 Statistical comparison of the generated solver NRP-C and the generated solvers NRP[D-J]. 304
B.50 Statistical comparison of the generated solver NRP-C and the generated solvers NRP[D-J]. 305
B.51 Statistical comparison of the generated solvers NRP-[D-E] and the generated solvers NRP[F-J]. 306
B.52 Statistical comparison of the generated solvers NRP-[D-E] and the generated solvers NRP[F-J]. 307
B.53 Statistical comparison of the generated solvers NRP[F-J]. 308
B.54 Statistical comparison of the generated solvers NRP[F-J]. 309
B.55 List of abbreviations 311

Acknowledgements

I will be forever grateful to Dr. Julian Miller who first encouraged me to apply for this Ph.D. project. I would like to thank him for all the time he has dedicated so that this work could be achieved. Without his support, mentorship, and guidance, the findings presented in this document would not have been completed.

I would like to thank Dr. Martin Trefzler for his patience and leadership skills. His expertise in evolutionary computations has been invaluable in the design of many empirical methods. Without his support, mentorship, and guidance many analytical aspects may not have been considered.

Without Dr. Jerry Swan, a detailed framework and some software metrics would not have been used in the context of generative hyper-heuristic. I would like to thank him for his persistent approach to identify some areas that needed to be refined. Without his expertise in machine learning in general, certain aspects would not have been discussed.

The N8 HPC is provided and funded by the N8 consortium and EPSRC (Grant No. EP/K000225/1), the Centre is coordinated by the Universities of Leeds and Manchester. Without their computer facilities, many of our experiments would have been able to run successfully.

I am indebted to their support and outstanding computer services they provide. I would like to thank Dr. Gabriela Ochoa and Dr. Timothy Curtois for sharing some of their software libraries. Without their kindness, many of our experiments and findings would not have been possible. I would like to thank Dr. Kyle Harrington for his valuable advice on autoconstructive techniques.

Finally, I would like to thank my family for their support and extreme patience. My husband Paul, my children Paola and Thomas have provided the most loving support. I believe they will not miss the laptop accompanying their partner and mum everywhere she goes.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

The research presented in this thesis features in a number of the author's publications listed below:

- Ryser-Welch, Patricia, and Julian F. Miller. "A review of hyper-heuristic frameworks." Proceedings of the Evo20 Workshop, AISB. Vol. 2014. 2014.
- Ryser-Welch, Patricia, and Julian F. Miller. "Plug-and-Play hyper-heuristics: an extended formulation." Self-Adaptive and Self-Organizing Systems (SASO), 2014 IEEE Eighth International Conference on. IEEE, 2014.
- Ryser-Welch, Patricia, Julian F. Miller, and Shahriar Asta. "Generating human-readable algorithms for the traveling salesman problem using hyper-heuristics." Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation. ACM, 2015.
- Ryser-Welch, Patricia, Julian F. Miller, and Shariar Asta. "Evolutionary Cross-domain Hyper-Heuristics." Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation. ACM, 2015.
- Ryser-Welch, Patricia, et al. "Iterative Cartesian Genetic Programming: Creating General Algorithms for Solving Travelling Salesman Problems." European Conference on Genetic Programming. Springer, Cham, 2016.
- Ryser-Welch, Patricia, and Julian F. Miller. "PPSN 2016 Tutorial: A Graph-based GP and Cartesian Genetic Programming."

Chapter 1. Introduction

Contents

1.1 Thesis aims and contributions **2**

1.2 Plan of thesis **5**

Designing effective algorithms for solving computational problems is a time-consuming and challenging task. A comprehension of a problem characteristics should contribute in planning and order some operations in sequences, to form an algorithm [159].

The algorithms should be expressed using a carefully chosen encoding scheme. Suitably expressive algorithms may never terminate or have over-long computations. It is, therefore, useful to consider an algorithm search-space consisting of feasible and infeasible algorithms. Various forms of constraints have many times prevented these unwanted occurrences [174, 244, 199, 180, 344, 182, 285]. Only suitable algorithms could be generated and assessed, preventing an excessive use of valuable resources during an algorithm search [298, 270, 224].

Non-deterministic methods should guarantee to return an algorithm, but it may not be optimum. These algorithms compensate certain operators weaknesses with the strengths of others. An algorithm search should sample a wide range without becoming impractical. Enumerating every possible combination of operators may not always be possible. When the numbers of primitives increases, the number of potential combinations exponentially grow.

Genetic Programming (GP) is a systematic and domain-independent method for computers to solve problems automatically [172] . The evolution identifies the steps that need to be carried out to find a solution. Some domain knowledge, a data structure, and an EA can often efficiently produce human-competitive results [158, 173, 224, 175, 76]. Circuit design, image processing, polymers, medicine, chemistry, mathematics, biology, and optimisation are some examples of applications that have benefited from the various form of GP. Some evolutionary algorithms (EAs) can also be generated. However, some human-competitive results have yet to be consistently observed and studied.

The hypothesis of this thesis is, therefore; an automated design of algorithms can be used to discover human-understandable and human-competitive algorithms, which are scalable and effective for a chosen problem.

1.1. Thesis aims and contributions

These discussions stimulate the following question.

Does the added complexity of using an automatic design process together with an imposed syntactic algorithm structure bring the desirable qualities of scalability, compactness and human-understandability?

Based on the described motivations and material presented in the literature review; a number of objectives are proposed.

1. To explore whether Cartesian Genetic Programming (CGP) can be an effective generative hyper-heuristics to evolve metaheuristics that solve computationally hard problems.
2. To investigate whether CGP can automate the decision to particular sequences employed in a metaheuristic; the latter is evolved to solve a problem.
3. To extend Recurrent CGP technique to be capable of evolving complete iterative constructs.

4. To extend the CGP technique to be capable of implementing an autoconstructive mechanism.

5. To apply the developed iterative and autoconstructive CGP to the generation of metaheuristics that find solutions to computationally hard problems.

6. To investigate how the developed autoconstructive CGP can affect the scalability and the compactness of metaheuristics.

7. To apply some software complexity metrics to seek whether a graph-based hyper-heuristic can improve human-understandability.

8. To analyse the solutions of computationally hard problems obtained by some generated metaheuristics, to identify some effective patterns of problem-specific operators that find suitable solutions to computationally hard problems.

Throughout this thesis, several substantial contributions are made the generative hyper-heuristics, CGP and the broader field of machine learning. The most significant contributions are now summarised.

1. A significant proportion of this thesis is dedicated to providing empirical evidence of the advantages of using a CGP generative hyper-heuristics. The following benefits are shown and discussed:

   (a) A coefficient of variation can provide a substantial benefit to automating the decision to particular sequences employed in a metaheuristic. Results show an effective pattern of problem-specific operators can be generated using a reduced amount of computer resources. Generative hyper-heuristics solely rely on the free-lunch-theorem to evaluate generated metaheuristics and widely overlook this significant advantage.

   (b) The ability to generate effective metaheuristics that find suitable solutions for computationally hard problems. Results presented indicates mutating active and non-coding genes continually during an automated design process represents a significant advantage over methods randomly selecting genes during reproduction.

2. This thesis presents Iterative CGP, a significant extension to Cartesian Genetic Programming. It enables the formation of iterative sub-programs and the encoding termination of those. In this thesis, Iterative CGP is shown to be capable of efficiently generating some metaheuristics that find some near-optimum to problems inspired by real-life problems. From these applications, presented results demonstrate that standard and iterative CGP could perform better for the nurse-rostering problem, but particularly well for the traveling salesman problem. Additionally, Iterative CGP is demonstrated to outperform not only standard CGP for some NP-hard problems but also some selective and generative hyper-heuristic techniques.

3. This thesis presents autoconstructive CGP, a significant extension to standard and iterative Cartesian Genetic Programming. Firstly, experiments demonstrate to improve a CGP-reproductive operator during the evolution of metaheuristics genetically. Some CGP-reproductive operators were obtained and tested to generate some metaheuristics for an unseen problem. Secondly, as with standard and iterative Cartesian Genetic Programming, results demonstrate to discover some effective metaheuristics for some computationally hard problems.

4. This thesis thoroughly analyses the solutions obtained by some problem-specific metaheuristics. Some parametric and non-parametric statistics show some human-competitive results are observed and studied. We believe these results are significant and in line with GP research in areas such as digital circuits.

5. This thesis provides an extensive list of generated solvers. Some metaheuristics have been translated from their CGP form to an imperative pseudo-code. A detailed statistical analysis shows a validation process has used some known and unknown distinct learning set of instances to validate these solvers. It is also demonstrated some parametric statistics have helped establishing some patterns of primitives that are likely to scale well. Some non-parametric statistical tests identify too some patterns of primitives that can achieve the same performance. A rigorous assessment by inspection has validated some features and patterns of primitives can impact a metaheuristic positively.

6. This thesis rigorously assesses the level of difficulty to understand some metaheuristics. It is shown some generated metaheuristics with standard, iterative and autoconstructive CGP can score similarly as some metaheuristics written by human-activity. Results presented indicate CGP-generated metaheuristics can be expressed with a similar vocabulary than problem-specific and published metaheuristics. Secondly, it is shown that other forms of genetic programming can generate shorter metaheuristics a smaller vocabulary. Published metaheuristics may seem more unfamiliar to a reader with expert knowledge in metaheuristics.

1.2. Plan of thesis

This thesis has been planned in three main sections; a detailed literature review, the details of our experiments and then a critical analysis and a conclusion.

Chapter 2 provides a detailed literature review of optimisation of algorithms. A general framework for generative hyper-heuristics is introduced and used in in subsequent chapters.

Chapter 3 reviews and discusses the problems we have chosen for our experiments.

Chapter 4 reviews graph-based GP and presents the graph-based hyper-heuristics applied in our experiments

Chapter 5 reports and discusses the results obtained by using standard CGP and iterative CGP. Objectives [1-3] are mainly explored.

Chapter 6 describes a learning objective process using a coefficient of variation. The results are reported. Objectives [1-3, 8] are mainly explored.

Chapter 7 mainly explores the objectives [5-8]. The experiments completed with autoconstructive CGP are described and their results discussed.

Chapter 8 critically analyses the impact brought to the algorithm search by improving some elements incrementally. Scalable patterns of primitives are suggested, and the comprehensibility of some generated solvers are analysed mathematically. All objectives are considered.

Chapter 9 concludes this documents and suggests some further research arising from this work.

Appendix A lists all the solvers obtained by our experiments in an imperative pseudo-code.

Appendix B provides all the results of our thorough statistical analysis.

Abbreviations lists the abbreviations appearing in this thesis.

Glossary provides a glossary keywords used in this document.

Chapter 2. Optimisation of algorithms

Contents

2.1 Basic principles 8

2.2 The problem domain 10

2.2.1 The problem search space 10

2.2.2 The problem encoding scheme 11

2.2.3 The problem evaluation process 11

2.2.4 Problem-specific operators 12

2.2.5 Problem parameters 12

2.2.6 Discussion 13

2.3 The algorithm domain 14

2.3.1 The algorithm search space 14

2.3.2 The algorithm encoding scheme 16

2.3.3 The learning objective function 17

2.3.4 The algorithm parameters 18

2.3.5 The algorithm understandability metrics 20

2.3.6 Discussion 21

2.4 Algorithm optimisation processes 23

2.4.1 Predicting the performance of algorithms 24

2.4.2 Automating parameter settings 26

2.4.3 Selection of operators 27

2.4.4 Generation of algorithms 31

2.4.5 Discussion 35

2.5 Conclusion 39

Optimised algorithms are assumed to run more efficiently and therefore reduce the execution time while finding some equally or better solutions than non-optimised algorithms. Various methods can achieve the process of optimising algorithms. This chapter positions our work in the active field of research, by reviewing extensively techniques attempting to accomplish this significant goal. Some functional mathematical expressions describe the elements of some standard and shared components. The subsequent chapters can, therefore, refer to these formally defined some essential features to discuss our experiments and their results.

2.1. Basic principles

Many methods postulate some algorithms may perform better for a set of instances than others. Newell et al. [237, 238] aimed at grouping some deterministic algorithms that could exhibit some abilities to solve specific mathematical problems. Their early attempt in artificial intelligence has led to the development of a technique that constructs some computer programs (i.e., *"The General Problem Solving I"*). Friedberg [104] proposed *"the program of a stored-program computer be gradually improved by a learning procedure which tries many programs and chooses, from the instructions that may occupy a given location, the one most often associated with a successful result."*

Rice et al. [267] introduce *"The algorithm selection problem"*; a method that maps the algorithm performance to certain instances to predict their performance with some unknown instances. This problem is well studied in the research field of selective meta-learning and algorithm configuration. Examples of such systems include the concept of *"Programming by optimisation"* or the generation of parallel portfolio of algorithms [148, 195].

Some similar focus has been reported with non-deterministic algorithms. Wolpert et al. [345, 346] prove if a search algorithm or a supervised machine learning method may work well for a problem, it may not work for another one. The automation of parameter settings and the evolution of EAs have both aimed at addressing the outcome of this seminal paper.

Burke et al [47] provides a generalisation of a technique referred as *"selective hyper-heuristic"* (SEL-HH). Introduced by Cowling et al. [78], this optimisation technique selects some problem-specific operators randomly to search for some problem solutions.

Finally, Spector et al. [302] extend genetic programming (GP) by co-evolving some GP genetic operators and some evolved algorithms. An algorithm search constructs some problem-specific algorithms and some GP genetic operators at the same time, without any human involvements.

Each research community studies distinctive perspectives and approaches. Some literature predicts the performance of algorithms, and some other automate the parameter settings of algorithms. Other select problem-specific operators, while other communities generate some algorithms. Notwithstanding the wide area of research, some highly-cohesive frameworks exist. Their architecture often rely on a *problem domain*, an *algorithm domain* and some *optimisation processes*. The loose coupling between a problem and some problem-solving techniques is often perceived to increase the generality of a technique [302, 78, 346, 148, 195, 237, 238, 47].

2.2. The problem domain

Problem-solving techniques often formulate the task at hand with a *problem space* distinct characteristics [288]. Many algorithm optimisation techniques can refer to this critical element with a different name. However, in selective and generative methods it is often called *the problem domain.* Therefore we will use this terminology to conform with the literature.

Each of our chosen problem domains is not only hard to solve but also have unique and problem-specific features. By their nature, their operators, problem search space, encoding scheme and evaluation process are very dissimilar.

2.2.1 The problem search space

A problem statement defines abstractly some conditions indicating whether a goal is reached. In mathematics, a *problem* represents the objective(s) to be met. More concrete elements are added with an $instance$; this input is useful for judging a *solution complexity* [20]. The latter suggests the amount of work required to find a solution.

Some instances can become larger and then increase the size of some possible solutions and a problem solution search space (see expression 2.1). The number of computations may increase as well. All possible solutions are held in a large set referred as the *problem search space*; it is then divided into subsets for every possible solution of each instance (see expressions 2.1 and 2.2) [27, 233]. An *"optimum solution"* is considered to be the best known solution or the global optimum (see expression 2.3).

$$problem(instance) : Instance \mapsto Solutions \tag{2.1}$$

$$\forall\, Solution \in Instance : \{Solutions \in ProblemSearchSpace\} \tag{2.2}$$

$$Optimum : Solution = Best\, known\, solution \tag{2.3}$$

2.2.2 The problem encoding scheme

A specific and specialised format organises some data to represent a problem solution (see expression 2.4). The *problem encoding scheme* could be very simple or more intricate. Many suitable data structures may be available. Still, a chosen problem encoding scheme should efficiently represent a problem statement. Otherwise, their problem solution may not be understood and analysed easily.

$$problem\,encoding\,scheme : (data \times data\,structure) \mapsto Solution \tag{2.4}$$

2.2.3 The problem evaluation process

A problem evaluation process assesses whether a solution meets the objective described in a problem statement. Often a solution is mapped to a real value (see expression 2.5). Each problem solution can then be analysed by an automated system or a person; numerical values can be analysed by statistical methods.

Solutions found by non-deterministic methods are also unlikely to be optimal or perfect. With that in mind, the problem fitness evaluation process could also approximate the discrepancy between a known minima and a solution fitness value (see expression 2.6). A so-called relative error naturally provides a metric indicating whether a solution is an optimum ($RelError = 0$), a near-optimum ($RelError > 0$), a new know optimum ($RelError < 0$), and an inadmissible solution ($RelError \to \infty$).

$$ProblemEvaluation(aSolution) : Solution \mapsto \mathbb{R} \tag{2.5}$$

$$RelError(aValue, knownMin) : (\mathbb{R} \times \mathbb{R}) \mapsto \mathbb{R} \tag{2.6}$$

2.2.4 Problem-specific operators

An operator should transform a problem solution from a current state to another state. The new answer could be in a near region or a different part of the problem search space. The quality of a new solution could, therefore, be affected positively or negatively or even remained the same.

Each operator should provide a unique functionality. their arity n could vary from $0 \leq n \leq \infty$ and one solution should be at least returned (i.e. $1 \leq m \leq \infty$); expression 2.7 formally defines the general signature of operators. A problem domain is likely to have more than one operator. Therefore a list of problem-specific operators is made available (i.e $ListOfOp$ see expression 2.8). Sections 2.4 and 2.5 discusses how this list contribute an algorithm optimisation process and an algorithm domain.

$$Op(arg_1, arg_2, ..., arg_n) : Solution^n \mapsto Solution^m \tag{2.7}$$

$$ListOfOp : \{Op_1 ... Op_{max}\} \tag{2.8}$$

2.2.5 Problem parameters

This type of metadata provides some specific information related to a problem domain. The parameters would influence the problem search space and the operator's performance.

To simplify our model, we assume that the problem domain has at least one parameter; an instance (see equations 2.1, 2.9 and 2.11). The operators may also rely on some parameters to achieve their tasks. Because it is a challenging task to predict the number of parameters required, we prefer expressing this variable between 1 and infinity (see expression 2.10).

$$ProblemParam : Instance \cup \{pp_1, ..., pp_{last}\} \quad (2.9)$$

$$1 \leq last \leq \infty \quad (2.10)$$

$$setProblemParam(pp_1, ..., pp_{last}) : parameters^{last} \mapsto ProblemParam \quad (2.11)$$

2.2.6 Discussion

We are not pretending this decomposition of a problem domain offers a panacea. It can be argued a search space, a problem encoding scheme, some problem operators, some parameters, and an evaluation process provide a certain completeness to the concept of a problem within an optimisation or discrete context. We would acknowledge this point and would welcome a comparison against another decomposition more suitable for another approach.

The definition of a problem search space is comprehensive enough for this work. A unique problem statement with the description of instances is later discussed (see chapter 3). The problem-solving methods used in this thesis are not-deterministic; often the problem search space is referred as a fitness landscape.

Some non-deterministic operators can randomly find solutions of lesser, equal or better quality. This type of operators will be applied instead of mathematical operations (i.e. $+, -, \%, \div, \times$). The general concept of *mutation*, *recombination*, *ruin-and-recreate* and *local search* will be adapted to some various encoding schemes (i.e. binary strings, directed acyclic graphs and table). These operators will rely on some probabilistic parameters.

A problem-solving method should explore the problem search space, whether it is automated or not. For example, *"Mimicry solvers"* should survey many possible solutions of the *mimicry search space* and perhaps find an optimum solution; similarly *"TSP solvers"* and *"NRP solvers"* should consider solutions from their own search space (i.e. the *"TSP search space"* and the *"NRP search space"*). In the context of this work, we consider non-deterministic algorithms as problem-solving techniques. The next section describes the elements of an algorithm domain.

2.3. The algorithm domain

Solving a problem requires performing a series of actions, with the hope of producing an optimum solution. In computer science and mathematics, the so-called algorithms are sequences of operators that are also performed in a particular order to achieve a well-defined goal.

Some variety of algorithms can be identified by the type of operators they apply. Mathematical operations can form mathematical expressions, organising operators and variables of a programming language can compose a program and also digital gates can represent a digital circuit. These tools can solve a problem with different outcomes; some of them may find some appropriate solutions and some others may not. This situation could lead in trying and assessing many algorithms, before identifying an effective algorithm [292, 43, 288, 338, 234, 342, 62, 67, 154, 339, 220, 278, 159].

2.3.1 The algorithm search space

When a finite list of operators exists, then the algorithm search space should represent every possible distinct algorithm. Every step should correspond to a valid operator that can be mapped to a given problem domain (see expressions 2.12 and 2.13).

The algorithm search space should only "be aware" of an operator list provided by a chosen problem domain. Consequently the same signature as $ListOfOp$ is also adopted (see expression 2.8 defined in section 2.2.4). This level of abstraction should make completely transparent the type of algorithms represented (i.e., mathematical expressions, digital circuits or computer programs), making the algorithm domain more general and loosely coupled to the problem domain.

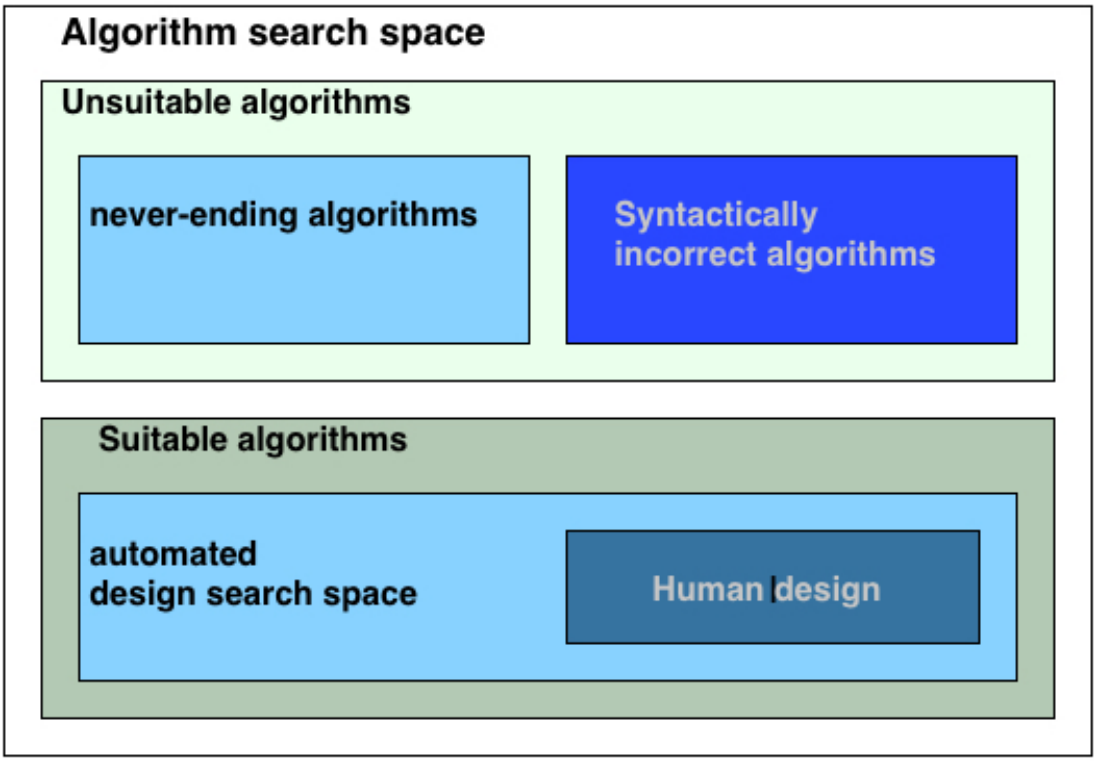
$$\forall A : (step_1 ... step_{last}) \; where \; step \in ListOfOp \; and \; 1 \leq last \leq \infty \tag{2.12}$$

$$Algorithm : \{A \in AlgorithmSearchSpace\} \tag{2.13}$$

$$Algorithm \; Search \; space : UnsuitableAlg \cup SuitableAlg \tag{2.14}$$

It is assumed the human design space is part of the automated design space (see figure 2.1). Some early assumptions can restrict the resulting combination of operators. Detailed study of solutions obtained from some sample problems and personal inspections based on experience can lead to premature commitment to a specific design; some alternatives solvers can then be eliminated or abandoned at an early stage [159, 148].

Figure 2.1: A decomposition of the algorithm search space



Theoretically, automating some aspects of the algorithm search could prevent this situation occurring; perhaps some suitable and syntactically correct algorithms could be found outside the human design space. As a practical necessity, imposing some constraints on the loops of evolved programs prevent unending iterations. Some grammatical rules or templates can make this possible by stating the elements that remain unchanged and the part of the program that is evolved [174, 244, 199, 180, 344, 182, 285].

2.3.2 The algorithm encoding scheme

A data structure encodes sequences of operators; it should represent the operations acting upon the data (see expression 2.15) [140]. During the execution of an algorithm, each step can apply an operator on some give solutions (see expression 2.16); a unique operator code (i.e. *OpCode*) represent an function.

An execution process can then decode each step to obtain a problem solution (see expression 2.17). Algorithm 2.1 illustrates how a finite sequence of operators can be decoded using the functions $ApplyOp$ and $ExecAlg$.

$$Alg.\,encoding\,scheme : data \times data\,structure \mapsto Algorithm \tag{2.15}$$

$$ApplyOp(aStep, someSolutions) : (Op \times Solution^{n}) \mapsto Solution^{m} \tag{2.16}$$

$$ExecAlg(anAlgorithm, anInstance) : (A \times Instance) \mapsto Solution \tag{2.17}$$

Algorithm 2.1. A general decoding process that sequentially applies each step of an algorithm

```
1: function EXECALG(anAlgorithm, anInstance)
2:     ProblemSolution ← InitialiseSolution(anInstance)
3:     for CurrentStep ∈ anAlgorithm do
4:         ProblemSolutions ← ApplyOp(CurrentStep, ProblemSolutions)
5:         CurrentStep ← nextStep
6:     end for
7:     return ProblemSolution
8: end function
```

An algorithm encoding scheme specifies the order of execution. An algorithm can, therefore, be executed a number of times; each run becomes independent and could return each time a different problem solution.

2.3.3 The learning objective function

This evaluation process assesses the performance of a given algorithm, by mapping its ability of solving a problem against a numerical value (i.e an *algorithm fitness value* defined by expressions 2.18 and 2.19). Also referred as *learning objective function*, this process should predict whether an algorithm could find optimum or near-optimum solutions of unseen instances.

$$AlgEvaluation(anAlgorithm, Instances) : (A \times Instance^n) \mapsto \mathbb{R} \quad (2.18)$$

$$AlgFitVal = AlgEvaluation(anAlgorithm, Instances) \quad (2.19)$$

$$RunAlg(anAlg, anInstance, Runs) : (A \times Instance \times IN) \mapsto IR^m \quad (2.20)$$

A learning objective function should include at least three steps to compute an algorithm fitness value :

1. A given algorithm is decoded and find some solutions for some given instances. Those are passed by the parameter $Instances$ in algorithm 2.2. The function $ExecAlg$ defined in expression 2.17 performs this task. This step maps $(A \times Instance^n) \mapsto S^m$.

2. The problem solutions obtained in step 1 are then evaluated (see line 5 of algorithm 2.2). A specific problem evaluation process is applied; those were defined by expressions 2.5 and 2.6. This second stage maps $S^m \mapsto \mathbb{R}^{\mathrm{m}}$.

3. These problem fitness values can then be statistically analysed to compute an algorithm fitness value (see expression 2.18 and line 8 of algorithm 2.2). This final step maps $\mathbb{R}^{\mathrm{m}} \mapsto \mathbb{R}$.

Algorithm 2.2. Run several times an algorithms and evaluates the problem solutions.

```
1: function RUNALG(anAlgorithm, anInstance, Runs)
2:     someResults ← sizeOf(Runs)
3:     for aRun ∈ Runs do
4:         aSolution ← execAlg(anAlgorithm, anInstance)
5:         aProbFitVal ← ProblemEvaluation(aSolution))
6:         Results[aRun] ← RelError(aProbFitVal, anInstance.Optimum)
7:     end for
8:     return ApplyStatistics(Results)
9: end function
```

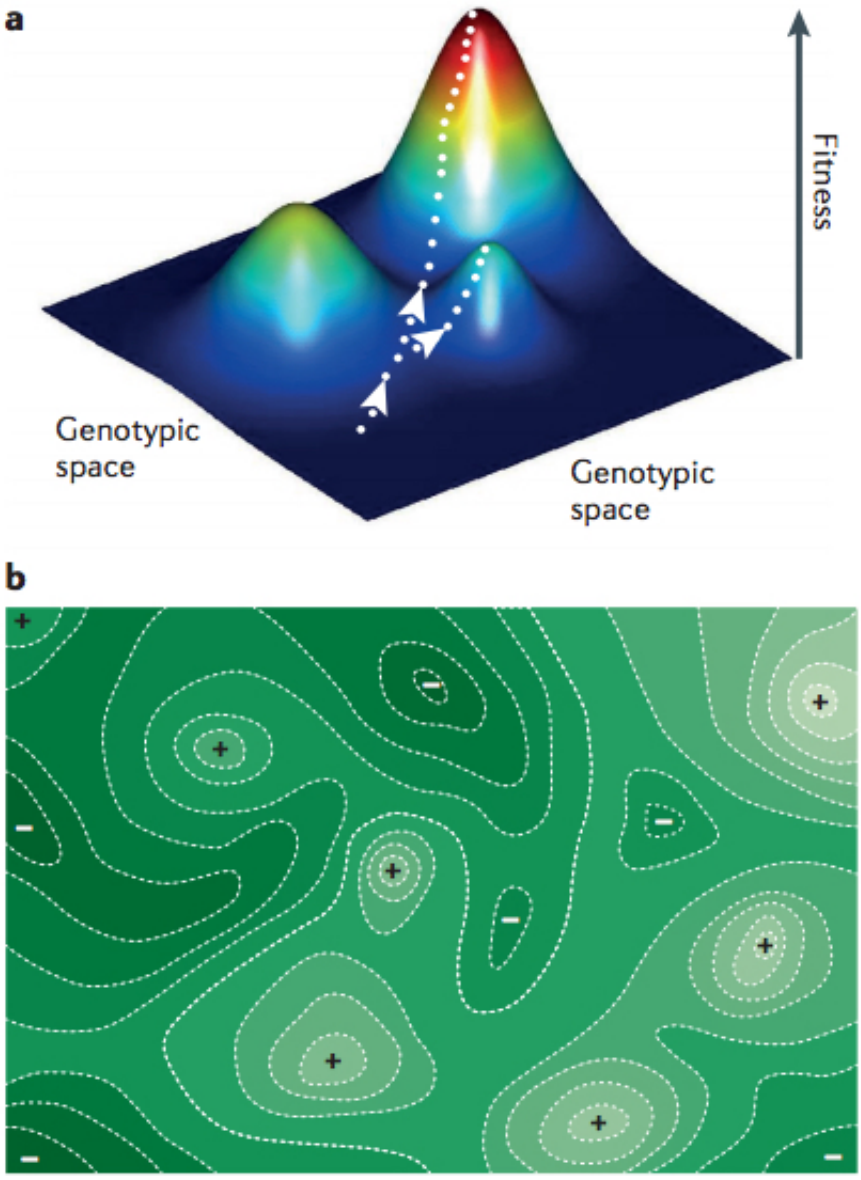
In theoritical biology, every possible genotype is assigned a fitness value. A description of how frequently one genotype is reached from another can now be visualised in a mountainous landscape (see Figure 2.2 part a) or a two-dimensional space (see figure 2.2 part b) [87, 243, 157, 351]. For the remaining of the thesis, we will assuming an excellent solver should be minimising its problem solutions; therefore any learning objective function should reward any effective metaheuristics with a low value [168].

2.3.4 The algorithm parameters

The algorithm domain parameters describe the algorithm characteristics. Therefore they are part of the algorithm domain. The algorithms parameters should also embed in the problem parameters; otherwise, they may not be set before any algorithms are executed. For example, the functions $RunAlg$ and $AlgEvaluation$ defined respectively in expressions 2.17 and 2.18 have the instance of a problem as an argument, which is a compulsory parameter in the problem domain (see equation 2.1).

The set of algorithm parameters has two important characteristics. First, the minimum number of algorithm parameters should be greater than or equal the number of parameters of a given problem domain (see equation 2.22). Secondly, all the algorithms parameters should at least include all the parameters of a given problem domain. In equations 2.22 and 2.9, the problem domain parameters are referred as pp and then the algorithm parameters ap. The function $SetAlgorithmParam$ defined in expressions 2.23 respects these two conditions; it includes both sets of parameters.

Figure 2.2: Evolutionary fitness landscape with a three and two dimensional representation model [87]



$$last : Size(ProblemParam) \leq last \leq \infty \tag{2.21}$$

$$\forall AlgParam \in ProblemParam \cup (\{\} \cup \{ap_1, ..., ap_{last}\}) \tag{2.22}$$

$$SetAlgParam(pp_1, ..., ap_{last}) : parameter^{last} \mapsto AlgorithmParam \tag{2.23}$$

2.3.5 The algorithm understandability metrics

Newell et al. [237] highlights the importance of choosing the vocabulary of a program. They argue a program would not be able to operate within a task environment otherwise. We include some understandability metrics to capture and quantify its effect on its understandability. The comprehension of an algorithm is often subjective and understudied. Nonetheless, it is usually considered a small value is more favourable to represent the barrier of understanding has been lowered [127].

In our framework, some metrics quantify the elements affecting the effort to understand an algorithm; those include the vocabulary and the length of an algorithm. Halstead et al [127] defines that each distinct symbol chosen to express an algorithm is part of its *vocabulary*. Those may include some constants and some variables an algorithm relies on to represent some values (i.e. *noOperand*). The operators (i.e arithmetical, logical and assignment), functions and keywords are also part of this metric; they are referred as *noOp* (see expression 2.24). The *Length* sums every occurrence of these operators and operands appearing in an algorithm (see expression 2.25).

$$Vocabulary \leftarrow NoOp + NoOperand \tag{2.24}$$

$$Length \leftarrow TotOp + TotOperand \tag{2.25}$$

The effort to understand an algorithm is shown in expression 2.26. For two algorithms of the same length, a larger vocabulary increases the effort metric. Adding a new operand or a new operator not only affects the vocabulary metric, but also influences the level of error-proneness (i.e. $\left(\frac{NoOp \times TotOperand}{2 \times noOperand}\right)$). Halstead et al. [127] reflected that many variables or constants could require more mental resources to working out the data they represent. Also, increase the distinct functions and operators can quickly require effort to understand how they transform the data. At the same time, more error can be introduced.

We also embed a cyclometric complexity, so that we can compute the number of the independent paths within our algorithms. This metric represents each line of an algorithm as a node graph. Some edges model the path between each node. Expression 2.27 compute the number of independent path using McCabe expression [215].

$$Effort \leftarrow \frac{(NoOp \times TotOperand \times Length) log_2(Vocabulary)}{2 \times noOperand} \tag{2.26}$$

$$NoIndependentPaths \leftarrow NoEdges - NoNodes + 2 \tag{2.27}$$

2.3.6 Discussion

We have decomposed the algorithm domain into its search space, understandability metrics, evaluation process, parameters as well as execution process. Our work focuses on metaheuristics, and perhaps this view may have guided the composition of this domain in this manner. We would be happy to compare and validate this model to a greater context.

An evaluation process, an encoding scheme and also some parameters are common to both domains. Each of these is part of a different component and represent an algorithm or a problem. For example, a *problem evaluation process* assesses the problem solutions, but *a learning objective process* evaluates the quality of an algorithm. A *problem encoding scheme* encodes solutions of a problem and an *algorithm encoding scheme* represents an algorithm. The *problem parameters* only affect the problem domain and the *algorithm parameters* influences the algorithm.

Both domains have exchanged some information and ”services” with each other, so they can efficiently provide their purpose. A list of operators, some instances, some problem parameters and some problem fitness values are some essential information obtained from the problem domain and used by the algorithm domain. The latter provides solutions to the problem domain resulting from the execution of an algorithm.

The definition of all sub-components should be general enough to ensure the two domains remain independent from each other. Otherwise, an algorithm domain and a problem domain needs to be written again for each type of problem and algorithms used to find solutions.

Instead of having some “algorithm operators”, the algorithm domain had an algorithm execution process. We believe “algorithm operators” are a form of algorithm optimisation; they can act on the various elements of an algorithm.

In the next section, we will review how the algorithm domain elements have been optimised in the vast literature of machine learning. We will be discussing the prediction of performance, the automation of parameter settings, the selection of operators and the generations of algorithms.

2.4. Algorithm optimisation processes

Algorithms can find solutions to problems in many different ways, but some of them work better than others. Often the quality of these solutions, the length or cost used by this process can offer a measure of the efficiency and effectiveness of these algorithms. While effective algorithms are likely to be more successful in finding the desired outcomes (i.e., optimum or near-optimum solutions), efficient ones can achieve the same results with less effort or resources.

Optimising algorithms can achieve this ultimate goal, but a manual process can take a considerable amount of time. Automating the most repetitive parts of such methods can hasten the whole process. Ultimately, when Moore's law limits are reached, a much shorter period than human activities will then be required to process information [349]. So automating the optimisation of certain aspects of algorithms could potentially not only improve their performance, but also the research community could advance their knowledge base in much faster pace.

To the best of our knowledge, automating the optimisation of algorithms has focused on four main areas. Its simplest form, *"The algorithm selection problem"* maps known algorithms to specific instances of a problem, to predict their performance on unseen instances [267]. Then, parameter settings can automatically adapt the algorithms parameters to the problem to be solved [148, 14, 98]. *"Selective hyper-heuristics" (SEL-HH)* repetitively chooses operators from a given finite list and applies them to the current state of the search [78]. However, the most ambitious form of optimisation is to automate the creation of computer programs, so that *"computers are programming themselves"* [231].

2.4.1 Predicting the performance of algorithms

Predicting the performance of algorithms on specific instances can require a lot of effort, time and attention. Algorithm fitness evaluations completed on a set of training problems can become useful to predict the performance on unseen instances. Some algorithms are mapped to a problem domain, by executing solvers to every instance of the training set (see expression 2.28).

In this context, the algorithm evaluation process has been adapted to focus specifically with one instance and one known algorithm at a time (see expression 2.29), to allow a three-dimensional Euclidean space being constructed with the tuple $< algorithm, instance, algorithmperformancemeasure >$ as dimensions (those were defined respectively in expressions 2.13, 2.2, 2.29). The performance measure can then be maximised for either an algorithm or an instance assisting in predicting the performance of an algorithm on unknown instances (see equations 2.31) [267, 230, 296, 82, 331, 41].

$$\forall instances\, of\, a\, training\, set : ExecuteAlg(AnAlgorithm, AnInstance) \quad (2.28)$$

$$AlgorithmEvaluation(anAlgorithm, anInstance) : (A \times I) \mapsto \mathbb{R} \quad (2.29)$$

$$performance\, measure :< A, I, AlgorithmEvaluation(A, I) > \quad (2.30)$$

$$\underset{A \epsilon Algorithm}{\arg\max} : ||AlgorithmEvaluation(A, I)|| \quad (2.31)$$

$$\underset{I \epsilon Instance}{\arg\max} : ||AlgorithmEvaluation(A, I)|| \quad (2.32)$$

2.4.1.1 Previous and recent work

The *"Algorithm Selection Problem"* (ASP) [267] has yet to attract a lot of interest. Nonetheless, the prediction of some algorithms' performance was successful for some NP-hard problems (i.e., scheduling and boolean satisfiability problem). Broad libraries of instances and algorithms were assembled to model algorithm runtimes, using statistical regression [352, 109, 276, 18, 155].

Some portfolios have also been applied to discover new knowledge about some specific problem and the algorithm domain [186, 185, 187]. These methods often can be limited to one problem domain. The algorithms also tend to be deterministic and consequently with large instances such techniques can become ineffective. Nonetheless, the use of non-deterministic-algorithm portfolios has overcome this issue [361, 360, 359].

The prediction accuracy has improved by adding some features to these frameworks. Online algorithm portfolio has used a form of reinforcement and machine learning. Some selection mechanisms have successfully reduced the execution time [106, 34, 108, 123, 289, 12, 313, 143, 315]. A filtering system has also improved the number of instances solved across well-established benchmarks of boolean satisfiability problem; they identify the likeliness to negatively or positively affect the computer solutions. Some well-known online entertainment providers have adopted these solvers [320, 229, 266, 40].

Others have taken the advantages of evaluating the algorithms in parallel to achieve this same aim [151, 147, 194]. The most recent advancement has incrementally added some parameters and shares the configuration space across the parallel processes [196]. Some parameter setting features have been added and will be discussed in the next subsection.

The strengths and weaknesses of different algorithms have also been studied to influence an algorithm design process. Graphical representation of the performance measure space is examined and discussed in details. This type of optimisation often classify the instances in term of level of challenge and often evolve them to study the behaviours of an algorithm. Such works have positively brought more understanding the characteristics of the algorithms and instances of the traveling salesman problem and the timetabling problem [294, 295, 262, 297, 81]. To conclude this section, Kotthof should provide a more comprehensive and detailed review of the algorithm selection problem and its future development [170].

2.4.2 Automating parameter settings

The relationship between the parameters of a problem and an algorithm domain was discussed in section 2.3.4. The performance of an algorithm could be affected positively or negatively by some parameter settings. Finding its optimised tuning could take time and resources too [148], despite being a straightforward process. Any of these parameters can then be tuned by hand or by an automated process, without any differentiation between both methods. This section will be reviewing the automatic parameter setting during the problem search.

The first stage often initialises some parameters. This process can be implemented using some deterministic or stochastic mechanisms (see expression 2.33). In the subsequent stages, at least one parameter is set at a time (see expression 2.34). Expression 2.35 retrieves the value of a given parameter.

$$InitialiseParam() : \{\} \mapsto AlgParam \quad (2.33)$$

$$SetAlgParam(aParam, aValue) : (AlgParam \times Value) \rightarrow AlgParam \quad (2.34)$$

$$GetAlgParamValue(aParam) : AlgParam \rightarrow Value \quad (2.35)$$

2.4.2.1 Previous and recent work

Self-adaptive metaheuristics should adapt some strategic parameters during their search. For a long time, Evolution Strategies have influenced each individual step size with self-adaptation [33, 24, 341, 36, 128, 152, 146, 130] . The mutation rate adjusts itself to the need of the search but can be quite slow [130]. A covariance matrix adaptation has also been used for this purpose, with a much quicker response. [129, 132, 131, 36, 19, 156, 269, 209].

In recent years, [93, 94, 17] has extended this concept to iterated local search and memetic algorithms; the perturbation brought by local search adapts itself to prevent staying in local optima for many generations. Crossover and mutation rates are also adjusted during the search [189, 204]. Unlike an evolution strategy (ES) there are yet some explicit self-adaptive methods to be identified.

Self-adaptive metaheuristics are often considered as a form of parameter control. In contrast, a parameter tuning technique searches for suitable parameters values, which remains fixed during the run [98]. Such methods have mainly been applied to many metaheuristics [291, 97, 10, 80, 290]. Still, a minority of software engineering communities have studied the benefits such techniques could bring to their research [148, 162]. The most recent development has adjusted the parameter of a hybrid metaheuristic [314]. The parameter tuning outcome can positively guide the design of algorithms. By exploring a wider range of parameters in a small space of time, the optimised parameters value should bring more knowledge about parameters setting, some instances and a pattern of operators.

2.4.3 Selection of operators

Operators randomly chosen are repetitively (1) concatenated and (2) applied on solutions. These new solutions are produced at time t. When $t = 0$, problem solutions are randomly created, mapping an empty set to one with at least one problem solution (see expressions 2.2, 2.1, 2.36 and 2.37). The consecutive actions often select repetitively operators from a list (see equation 2.41), before applying it to the current set of problem solution (i.e. equations 2.8, 2.16, 2.38, 2.39, 2.40).

$$t \in [0] : InitProbSolution(Instance) \quad (2.36)$$

$$InitProbSolution() : (\{\} \times Instance) \mapsto Solution_0 \quad (2.37)$$

$$\forall t \in [1, TimeLimit] : Solution_t = SelectOp(ListOfOp, Solution_{t-1}) \quad (2.38)$$

$$SelectOp(ListOfOp, Sol_t) : ApplyOp(Choose(listOfOp), Solutions_t) \quad (2.39)$$

$$SelectOp(ListOfOp, Sol_t) : (ListOfOp \times Solutions^n) \mapsto Solutions^m \quad (2.40)$$

$$Choose(listOfOp) : ListOfOp \mapsto Op \quad (2.41)$$

The distinction between a problem domain, an algorithm domain, and an algorithm optimisation process is often unclear. *The algorithm* chooses some problem operators using some programming construct (i.e., selection and iteration); it becomes the algorithm optimisation process itself. The functions $ApplyOp(Choose(ListOfOp), Solution^n)$ (see expressions 2.17 and 2.41) bridges both components; connecting directly to a chosen problem domain. Despite being aware of the strengths and weaknesses of each operator, it could be difficult to distinguish between the combinations of specific operators that can make a positive impact on the search. Consequently, some added analytical tools may be required to compensate this weakness.

It would be fair to argue this process generates some algorithms. However, a very long list of operators selected through a learning selective algorithm would need to be extracted. These algorithms may be syntactically correct, but they would be those could be extremely long sequences of operators which may (or may not) have some logical patterns. These algorithms can be very different from the ones programmers are used to and would program themselves. They would be very challenging to code again with a high-level programming language. It would not only take a very long time, but also no control flow would be employed.

2.4.3.1 Previous work

The idea behind selective hyper-heuristics arose to compensate the strengths and weaknesses of operators, to find more effective problem solutions. Denzinger et al. [89] have introduced this concept 1996 by to prove mathematical theorems automatically. It was then formulated by Cowling et al. [79] three years later approximately.

It is worth noting some research communities can shorten the term *"selective hyper-heuristics"* to "hyper-heuristics". Hyper-heuristic also includes the generation of (meta-)heuristics, which is discussed in the next subsection. For the remaining of the thesis, the terminology hyper-heuristics will encompass both disciplines, and we will differentiate both types adequately.

SEL-HH has been applied to various problems with NP-hard computational complexity; scheduling, packing, constraint satisfaction and routing problems have been studied. Burke et al. [52] provides a complete survey of such research and the results of the cross-domain heuristic search challenges can offer more information [1]. It is noticeable in the literature, very little or no comparison with the state-of-the-art outside the field is provided; making challenging to position precisely the efficiency of such methods. However, SEL-HH frameworks can offer some many useful benchmarks.

Some frameworks (i.e., the *Hyper-heuristics Flexible Framework* [242], *parHyflex* [324] and *hMod* [321]) treat all the stochastics operators, and benchmarks in a black-box. Very little domain knowledge of the problem domain is required; this counter-intuitive aspect has led to the development of "cross-domain" hyper-heuristic algorithms. With such tools, explaining the logical flow of instructions could become a challenging task [270]. For the exception of *Hyperion* [306], the architecture itself prevent identifying the operators that have contributed to finding the best solutions. As a result, this aspect is often missing from the literature. For all that, the ability to solve significant real-world problems could also be limited.

[1] These websites can be found in http://www.asap.cs.nott.ac.uk/external/chesc2011/ and http://www.hyflex.org/chesc2014/

2.4.3.2 Recent work

Following the development of these frameworks, the optimisation process has applied some grammatical rules to guide the selection of operators, so that some patterns of stochastic operators (i.e., mutation, crossover, ruin-and-recreate, local search) can be guided more efficiently by a learning process [265, 212, 214, 13, 305].

Many other researchers have used some ”metaheuristic” patterns again to call some types of operators; those include an EA, a “harmony search” and an “ant colony” as an inspiration [88, 207, 31, 23, 115, 190]. Some problem-specific operators have been stretched to whole metaheuristics; various EAs are randomly selected instead of some genetic operators [120].

The solutions obtained are often compared with those found by another selective hyper-heuristics. We argue it would be valuable to compare against other optimisation techniques and the state-of-the-art for each problem. Otherwise, the real contribution of this method may not be fully appreciated.

Some new trends area of research has started to emerge. For Chen et al. [70] has published an analysis of the selected operators to solve routing problems. Their results can then be used again in designing new algorithms. Also, the original concepts behind selective hyper-heuristic have been revisited to solve more effectively the algorithm selection problem. The operators have become whole EAs that are applied in parallel to solve a problem. The best solutions found are selected, before the next iteration [118, 120, 121, 119, 37]. The latest advancement clusters some operators to improve the selection process [300, 356, 228]. On innovative technique has hybridized a selective and generative hyper-heuristics [286].

2.4.4 Generation of algorithms

This optimisation process should produce and assess some complete algorithms. In the first step, a set of algorithms should be randomly created using a list of operators provided by the problem domain (see equations 2.8, 2.42, 2.43). In the subsequent generations, new sequences of operators are repetitively generated. This is formally defined in expressions 2.44 and 2.45.

$$t \in [0] : Alg_0 = InitialiseAlg(aListOfOp) \quad (2.42)$$

$$InitialiseAlg(ListOp) : (ListOfOp \times \{\}) \mapsto \{Algorithm\} \quad (2.43)$$

$$\forall t \in [1, TimeLimit] : Alg_t = GenerateAlg(aListOfOp, Alg_{t-1}) \quad (2.44)$$

$$GenerateAlg(\{Algorithm\}) : (ListOp \times \{Algorithm\}) \mapsto \{Algorithm\} \quad (2.45)$$

$$\underset{A \in Algorithm}{\arg\max} \; : ||AlgEvaluation(A)|| \quad (2.46)$$

Similarly to predicting the performance and the automating of parameters setting of algorithms, this form of optimisation also uses again many functionalities provided by the problem and algorithm domains. The evaluation of algorithms, a list of operator, and a sequence of steps provides the provide the key elements required by the generation process (see expressions 2.8, 2.12 and 2.18). As a result, ordering some operators effectively to generate some sequences becomes fully independent from the problem domain.

2.4.4.1 Generations of sequential algorithms

The idea of program synthesis is not new; it has originated demonstrate some form of artificial intelligence [159, 238, 237]. An algorithm encoding scheme chosen can represent a neural network or a metaheuristic. Nonetheless, sequences of operators need to be followed to find some solutions [250]. Neural networks, decision trees and induction rules have been evolved using some genetic algorithms (GA) and a form of GP [355, 357, 260, 90, 7, 318, 249, 330, 30, 29]. The more recent development in this area has composed some first-order logical rules for knowledge base benchmarks dataset [354]. Some effort has also been made with other techniques such as Bayesian networks and other stochastic methods [193, 235, 122].

In his highly acclaimed book, Koza [172] explains how computers could be programmed with a GA and a tree-based encoding scheme. This idea was adopted not only to discover mathematical formulae and circuits [179, 56, 53, 59, 224, 75, 279], but also evolve EAs with various variants of genetic programming [245, 244, 161, 308, 199, 105, 25, 349].

Another methodology generates some algorithms using a self-assembly approach; the components autonomously assemble operators based on a statistical analysis of the configurations [312, 311, 188]. Implicit CGP uses a unique set of signature for some types of operators to assemble some classifiers [293].

2.4.4.2 Generations of iterative algorithms

Koza [174, 180] also attempted to raise interest in automatically designing iterative algorithms, but this area of research had remained quite inactive in the GP community. This type of algorithm repeats some sub-sequences instructions when a specific condition is met. A mechanism encodes an initialisation and an update step, a termination condition, as well as the body of a loop.

The evolution of indefinite loops can be prevented by applying various forms of constraints. Some syntactic rules can govern the structure of the algorithms and a maximum number of times the body of a loop can be executed is often limited [180, 344, 182, 283, 74, 344, 69].

Earlier work have also considered the body of a loop as an automatically defined function [39, 333, 332, 309, 310]. Sometimes these iterations may only repetitively apply one operator, which can be too restrictive. These mechanisms aim at keeping control of the halting problem.

2.4.4.3 Generations of EAs

The automatic design of EAs has adopted another approach. A template guarantees (1) a population of solutions is initialised, (2) some individuals for reproduction is selected and (3) a valid stopping criterion is applied. The remaining part of the EA is automatically designed by the evolution.

This approach has successfully improved the quality of problem-solutions found for various problem including function optimisation and the “royal road” problems [308, 244, 199]. The most recent advancement has evolved only one mathematical expression of an algorithm [241, 286]; solutions for small instances of the traveling salesman and vehicle routing problems have been found.

The automatic design of evolutionary operators is also an active field of research. The solutions quality for the problems including function optimisation, timetabling, and the TSP have been improved in comparison to other techniques. The evolutionary operator's order remains unchanged (i.e., crossover then the mutation is applied); GP generates the code of at least one operator; a selection mechanism of an individual for reproduction was evolved [350, 271] as well as some genetic operators [350, 348, 26, 308]. Similarly to parameters tuning, a generated genetic operator remains unaltered each time an algorithm is executed.

Some algorithm generation operators can also be evolved while exploring an algorithm search space. A generation operator adapts itself to the needs of the algorithm search. PushGp, AutoDoG and Self-Modifying CGP are examples are examples of these techniques [302, 141, 138, 303].

2.4.4.4 Hybridisation of generative techniques

During the completion of this work, some grammatical evolution has led to some exciting developments; the grammar structure has been extended to explicitly consider the features of the grammar being used [200, 201, 144]. Some other explore the mapping between problems, genotype size and genetic operator [217] and other have generated some parallel programs [72, 73].

Some hybrid techniques have also extended a generative optimisation process with some other capabilities. Some generated algorithms performance is predicted too. A first phase generates many EAs then an algorithm portfolio techniques matches the performances of those against some inputs [227].

Some recent development has automated the design of selective hyper-heuristic algorithms. Some grammatical evolution techniques generate some algorithms that randomly select some problem-specific operators [275]. The results have improved in comparison to a more traditional selective hyper-heuristics.

2.4.5 Discussion

The optimisation of algorithms involves a rather large variety of research communities. We have discussed four different types of optimising algorithms; those have been introduced from its simplest form its most complicated methods. Two trends appear across the field. The first approach adapts some elements of an algorithm during the search.

For the exception of [250, 82, 31, 98], these two trends are not often differentiated in the literature, making it difficult to compare their results against the state-of-the-art or with another form of optimisation. Also, the lack of consensus about the idea of termination criteria (i.e., maximum runtime, maximum of evaluations or generations) can bring another difficulty to efficiently compare the results with other approaches.

2.4.5.1 Self-adaptation

Some elements of a solver can adapt themselves while searching the problem fitness landscape. Selecting some problem-specific operators selection, controlling some algorithm parameters, and generating some reproductive operators during the algorithm search adopted such concept.

Many of these methods treat the optimisation processes as a black-box, preventing the identification of interesting features that human may have yet thought about. Also, the operators and parameter settings that have positively contributed to finding good problem solutions may not be recognized easily.

2.4.5.2 Training solvers

Performance prediction, parameter tuning and algorithm generation use information gained from a training set that can be valuable to solve some unseen instances. Many research tends to focus on one problem (or family of problems), making it challenging to establish their real level of generality.

Often the literature focuses on the problem solutions obtained and an engineering process; more theoretical aspect could be discussed, and some scientific fundamentals may be discovered. It would be also beneficial if some research community could discuss how measuring success may be achieved consistently. Similar suggestions have been made in the field of metaheuristics [298, 112].

2.4.5.3 Learning mechanisms

Meta-learning and hyper-heuristics may share more common features than many of their practitioners may admit. The architecture of their frameworks and intentions are very similar. In both fields, a problem domain and a learning mechanism (i.e., an optimisation method) are applied to accomplish comparable desired outcomes; their frameworks separate a problem domain and an optimisation process. However, this decomposition was suggested in earlier research in the use of heuristic to synthesise some programs automatically [159, 238, 237].

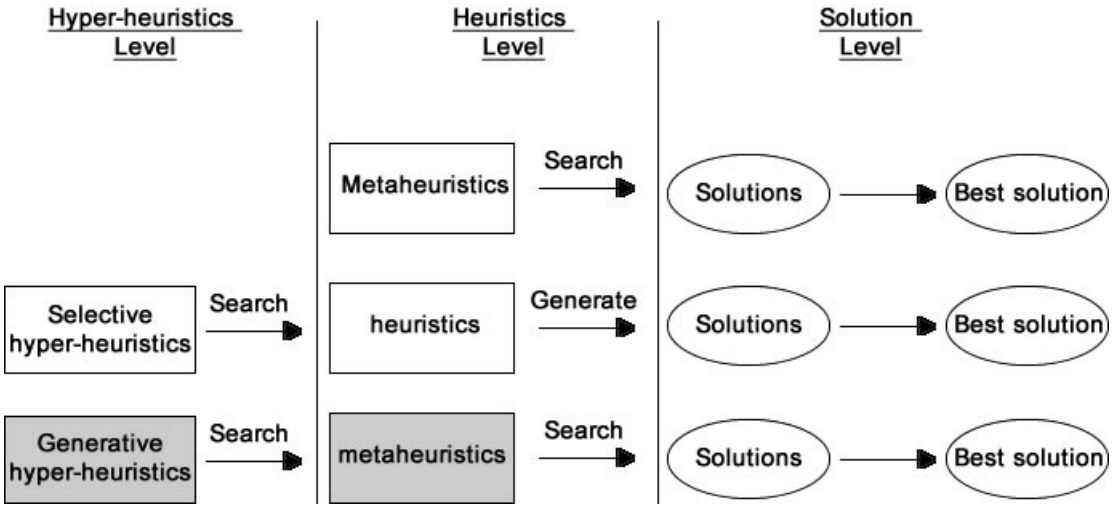
In meta-learning, the learning at a *meta level* aims at obtaining information about the performance of the algorithm. A *base level* is concerned about a task to achieve [331]. In hyper-heuristics, a learning mechanism should operate in the search space of heuristics and improve their performance; this is often referred as the *hyper-heuristic level.* A *problem domain* should contain all the information related to a computational search problem and some stochastic operators that can be used to find an optima [52, 65, 78].

Similar intentions are also achieved by various research communities. Meta-learning has focused on improving algorithms performance through experience; those can be deterministic or stochastic methods. Besides, *hyper-heuristics* selects or generates heuristics to find solutions of hard computational problems.

The terms *"metaheuristics"* and *"heuristics"* are often used interchangeably in the literature; it can either refer to a stochastic operator or an EAs. Sorenson et al. [299] have recently suggested that in a metaheuristic can be an algorithm or a framework. We refer a metaheuristic as an algorithm that can find some solutions by searching a problem fitness landscape.

To combat this lack of consensus that can bring some confusion, Barros et al. [31] disambiguates adequately the intent of a heuristic and a metaheuristic in the context of hyper-heuristics. Metaheuristics can be generated by a form of generative hyper-heuristics (in grey in figure 2.3) and hopefully, their sequences of stochastic operators can be studied. A selective hyper-heuristics searches the space of heuristics to generate a problem solution. These two types of hyper-heuristics must not be confused as they have different purposes; one generates algorithms and the others one select some operators.

Figure 2.3: A comparison of hyper-heuristics, metaheuristics and heuristics [31]



In some cases, mathematical operators have been considered as heuristics [203, 241, 287] instead of operators. We argue then that hyper-heuristics is a form of meta-learning that specialises in the selection of heuristics and the generation of meta-heuristics. In fact, Pappa et al [250] refer to these two fields as “meta-learning/hyper-heuristics”. Their classification of meta-learning and hyper-heuristics methods includes the ”selection” and ”generation” sub-categories (see figures 2.4 and 2.5). However, meta-learning has a broader scope than hyper-heuristics.

Figure 2.4: Classification of meta-learning methods as proposed by Pappa et al. [250]

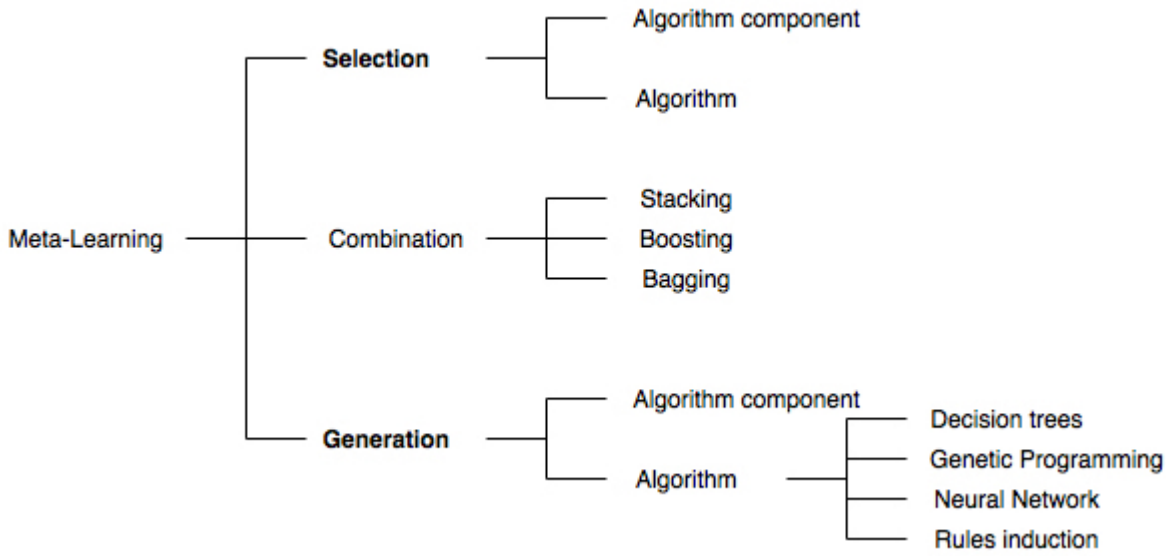
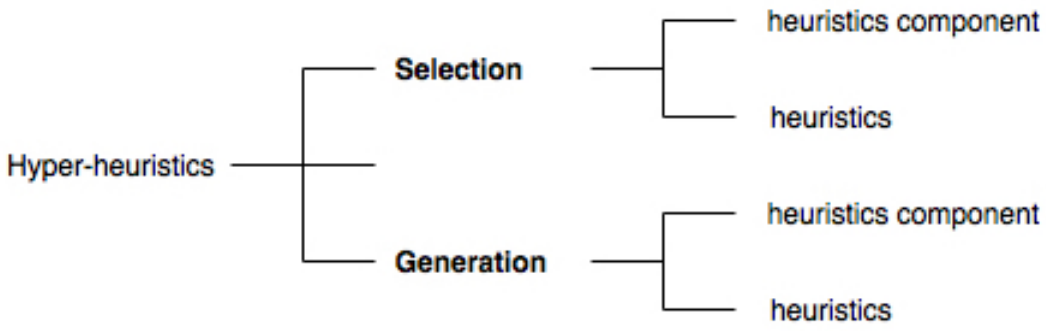


Figure 2.5: Classification of hyper-heuristic techniques as proprosed by Pappa et al. [250]



2.5. Conclusion

Some important concepts were introduced, and we will be referring to this chapter in the remaining sections of this document. We are not pretending this decomposition offers an answer to all questions, but it defines and differentiates all the features of this vast field of research suitably.

We will be generating some metaheuristics using a graph-based algorithm encoding scheme and an evolution strategy (ES); our focus will be on a generative hyper-heuristics (as highlighted in gray in figure 2.3). The next chapter introduces our chosen problem domains.

Chapter 3. Three problem domains

Contents

3.1 Common features 43
3.1.1 Population operators 44
3.1.2 Termination criteria 45
3.1.3 Summary 48
3.2 The Mimicry Problem 49
3.2.1 The chosen encoding scheme 50
3.2.2 Fitness evaluation 51
3.2.3 Problem parameters 52
3.2.4 Problem operators 53
3.2.5 Summary 57
3.3 The Traveling Salesman Problem 58
3.3.1 The chosen encoding scheme 60
3.3.2 Fitness evaluation 61
3.3.3 Parameters 63
3.3.4 Problem operators 64
3.3.5 Summary 69
3.4 The Nurse Rostering Problem 70
3.4.1 The chosen encoding scheme 73
3.4.2 Fitness evaluation 76
3.4.3 Problem Operators 80
3.4.4 Summary 84
3.5 Discussion and conclusion 85

The previous chapter models some algorithm optimisation processes independently from a given problem domain. Selective and generative hyper-heuristics have adopted this level of separation between a search method (i.e., a learning mechanism) and the problem domain [52, 78, 31, 53].

A majority of optimisation processes specialises in one problem or a family of problems. Consequently, we have chosen three different problems to demonstrate the generality of our techniques. Those are not only hard to solve, but also have unique features. Two NP-hard combinatorial problems have many real-life instances and benchmarks available. All these problems have commonly defined optimality concerning one objective function, which suits well many deterministic and non-deterministic methodologies. When the size of these instances grows large, it becomes unfeasible to find a solution through an exhaustive search. On that basis, a metaheuristic can find a solution in some reasonable amount of time, but near optimum may only be obtained.

We will introduce a discrete optimisation problem generalising the *"OnesMax"* problem. The so-called 'Mimicry problem' aims to find a bitstring that is identical to a fixed target bitstring[146]. Then we discuss our second problem; the traveling salesman problem finds the shortest tour that visits a collection of cities [16] . Finally, we discuss a scheduling problems that should assign shift to nurses in an optimal way [50].

For purpose of clarification, the common features of the three problem domains are discussed in the next section.

3.1. Common features

The description of our three problem domains uses the general characteristics and features introduced in section 2.2. These elements are general enough to communicate the concepts and ideas behind each problem. Each problem is introduced in term of *encoding scheme*, *problem evaluation process*, and *problem operators*. We will also specify a problem statement and provide a list of operators at the end of sections [3.2 - 3.4].

In this work, e will be using some well-known operators; such as mutation, crossover, ruin-and-recreate and local searches to generate some hybrids metaheuristics. *Mutation operators* alter some *"genes"* to bring some diversity to the search. These changes can be positive or negative. *Crossover operators* takes some genes from two parents (i.e., solutions) to produce an offspring. *Ruin-and-Recreate operators* often mutates some genes of an individual and then improves the solution. Finally, *Local-search operators* find some candidate solutions close to an individual in the problem search space.

In generative and selective hyper-heuristics, every operator often returns one solution. Some selective hyper-heuristic frameworks manage a population of solutions in a vector [242, 324, 306]. Returning one solution adapts the operators to this data structure. In generative hyper-heuristics, a consistent signature for every operator can be used more effectively with a form of genetic programming [245, 244]. For that reason, crossover operators often produce one offspring instead of two.

Some categories of non-deterministic operators may be detrimental or inapplicable to a particular problem domain. Therefore only crossover and mutation operators are commonly discussed across our three problem domains. Ruin-and-recreate and Local-search operators are only used when it was identified favourable. Nonetheless, each problem domain frequently includes some population operators as well as some termination criteria. .

3.1.1 Population operators

Our metaheuristics should sample some candidate solutions referred as *"populations p and t"*. Individuals of p can survive for more than one generation, but individuals of t can be much shorter. Their ephemeral life-span terminates when a metaheuristic selects some new individuals for reproduction.

The individuals of t are "tweaked" by some problem-specific operators; new solutions replace the previous ones. All these operations are defined in expressions [3.1 - 3.5].

$$InitPopulation(ProblemParam, \mu, \lambda) : \mapsto Solution^{\mu+\lambda} \quad (3.1)$$

$$Restart(ProblemParam, \mu) : \mapsto Solution^{\mu} \quad (3.2)$$

$$SelectElitism(p) : Solution^{\mu} \mapsto Solution^{\lambda} \quad (3.3)$$

$$ReplaceLeastFit(p, t) : Solution^{\mu} \times Solutions^{\lambda} \mapsto Solutions^{\mu} \quad (3.4)$$

$$ReplaceRandom(p, t) : Solution^{\mu} \times Solutions^{\lambda} \mapsto Solutions^{\mu} \quad (3.5)$$

InitPopulation randomly generates a population p and t. The unique problem parameters and encoding scheme specify the solutions characteristics generated by this function. The number of parents is defined by The operator parameter μ defines the number of parents p and the number of temporary solutions (i.e., t by λ).

SelectElitism selects the best individuals for reproduction. This selection operator should help to descend towards a global minimum more efficiently, without guiding the search into a local optimum.

ReplaceLeastFit identifies the individual of p with the highest fitness value and replaces it if a solution has a lower fitness value. We will be seeking to miminise the solutions of every problem. This operator should maintain a population with the best solutions. However, we are aware this feature may contribute to reaching local optima without leaving it. We hope our suite of problem-specific operators should perturb enough the solutions to prevent this situation.

ReplaceRandom selects individuals of p randomly and replaces them with some individuals of t. This operator applies no criterion. As a result, a much poorer quality solution can replace an individual of p of better quality.

RestartPopulation initialises again randomly all the individuals of p. This operator can help moving forward the search within the fitness landscape, especially when it remains in a local optimum.

3.1.2 Termination criteria

Metaheuristics may stop when it has found an ideal solution, or it has run out of time. In this thesis, optimum solutions are likely to have a solution fitness value set to zero; all our problem fitness evaluation return a relative value (see sections 3.2.2, 3.3.2 and 3.4.2). When a solution fitness value becomes negative, it indicates a lower optimum solution may have been found.

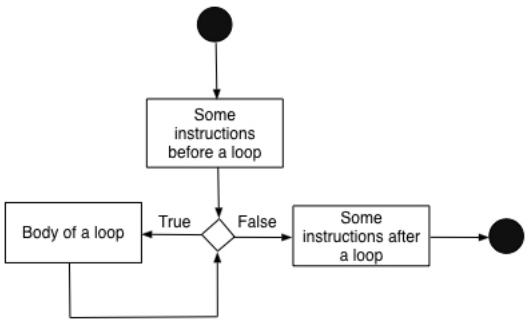
A budget of evaluations replaces the concept of time; the execution time can, therefore, vary for metaheuristics, as different instances are attempted to be solved. We anticipate more time may be required to find some solutions for the most challenging instances. A "budget of evaluations" should compare the performance of our generated algorithms fairly and consistently across the problem domains and their instances.

Some of our experiments evolve not only the order of operators but also iterations. A maximum number of evaluations $MaxEval$ should bound the execution of metaheuristics, preventing issues of program termination occurring.

In selective hyper-heuristics, it is not uncommon each time an operator alters a solution the new individual is evaluated [242, 324, 306]. Our generative hyper-heuristics will use this technique again and increase an evaluation counter called $EvalCount$.

For clarification, we will assume a body of the loop is executed when expressions [3.6 - 3.12] are evaluated as true. Otherwise, the loop terminates. Our reasonably sized set of termination criteria guarantees each expression can terminate a loop without any other mechanism. These termination criteria have been documented in [205].

Figure 3.1: A process showing the mechanism of a loop



3.1.2.1 Number of evaluations only

This category of termination criteria relies only on the number of evaluations. Some of the expressions rely upon only on a sample of the evaluation.

$$EvalCount \leq MaxEval : (IN \times IN) \mapsto IB \quad (3.6)$$

$$EvalCount \leq Limit : (IN \times IN) \mapsto IB \quad (3.7)$$

$$EvalCount \leq \frac{MaxEval}{2} : (IN \times IN) \mapsto IB \quad (3.8)$$

$$EvalCount > \frac{MaxEval}{2} \, and \, EvalCount \leq MaxEval : (IN \times IN) \mapsto IB \quad (3.9)$$

EvalCount $\leq$ MaxEval stops a loop as soon as all the evaluations have been used.

EvalCount $\leq$ Limit ends a loop when a randomly set number of evaluations has been reached. A variable referred as $Limit$ must respect the inequality $EvalCount \leq Limit \leq MaxEval$.

EvalCount $\leq \frac{MaxEval}{2}$ terminates a loop when half of the evaluations have been used; the value of $Limit = \frac{MaxEval}{2}$. It is hoped some stage evolution may be generated.

EvalCount $> \frac{MaxEval}{2}$ **and EvalCount** $\leq$ **MaxEval** executes the body of a loop when half of the evaluations have been used. This type of condition should help generating some stage evolution.

3.1.2.2 Number of evaluations and fitness value

The number of evaluations and the quality of the best individual are the two criteria that can determine whether a loop should terminate (see expressions [3.10- 3.14]). In our termination criteria, we consider a search has reached the highest mountain when no distance exists between the known optima and the best individual of p. Therefore we use the equality $p.fitness = 0$, where the variable ***p.fitness*** returns the fitness of the best individual of p (see section 3.1.1).

$$EvalCount \leq MaxEval \, or \, p.fitness > 0 : (IB \times IN \times IN) \mapsto IB \quad (3.10)$$

$$EvalCount \leq MaxEval \, or \, p.fitness > goal : (IB \times IN \times IN) \mapsto IB \quad (3.11)$$

$$EvalCount \leq Limit \, or \, p.fitness > goal : (IB \times IN \times IN) \mapsto IB \quad (3.12)$$

$$EvalCount \leq MaxEval \, or \, IsBetter(noEval) : (IB \times IN \times IN) \mapsto IB \quad (3.13)$$

$$EvalCount \leq MaxEval \, or \, IsBetter(1) : (IB \times IN \times IN) \mapsto IB \quad (3.14)$$

EvalCount $\leq$ **MaxEval or p.fitness** $>$ **0** either terminates a loop when all the evaluations have been used or when an optimum solution is found.

EvalCount $\leq$ **MaxEval or p.fitness** $>$ **goal** ends a loop when a near-optimum (i.e $goal = 0.05$) has been reached or all the evaluations have been used.

EvalCount $\leq$ **Limit or p.fitness** $>$ **goal** ends a loop when a near-optimum (i.e $goal = 0.05$) has been reached or the problem evaluations used have reached a limit randomly set. A variable referred as $Limit$ must respect the inequality $EvalCount \leq Limit \leq MaxEval$,.

EvalCount $\leq$ MaxEval or IsBetter(noEval) stops the execution of a loop, when a known optimum is found or when the search remains for too long in a local optimum [319]. A parameter referred as *Probation period* defines the minimum number of evaluations before the gradient of p is assessed.

EvalCount $\leq$ MaxEval or IsBetter(1) terminates the execution of a loop when the population p has not improved over one generation. It is a special case of the previous termination criterion.

3.1.2.3 Random Walk

The idea behind simulated annealing is to search randomly the fitness landscape, before applying a hillclimber; in the literature it is refferred as a *"walk"*. A probability $Prob = e^{\frac{best(p)-best(t)}{EvalCount}}$ is calculated before compared against a random number generated at each generation (i.e R); $Prob \in [0,1]$ and $R \in [0,1]$). $Prob$ and R help evaluating an acceptance criteria. When it is true, $R \leq Prob$ or the newly generated solution is better than its parent. This *acceptance criteria* is used in a selection instead of a loop [205].

Expression 3.15 is inspired by this acceptance criteria. The condition 3.15 stops when one of the three states is met; (1) when all the evaluations have been used, (2) when an optimum solution is found, or (3) the end of a random walk has been reached.

$$EvalCount \leq MaxEval \, or \, p.fitness > 0 \, or \, Walk() : (IB \times IB \times IN \times IN) \mapsto IB \tag{3.15}$$

3.1.3 Summary

The features introduced in section 2.2 are not only being used to define every problem domain. Also, some *population operators* and some *termination criterion* have been specified, so that the search space of hybrids metaheuristics can be widened.

Problem-specific operators return only one solution; a problem-specific operator evaluates this solution each time it is called. It has been designed in this manner to suitably support our learning mechanism.

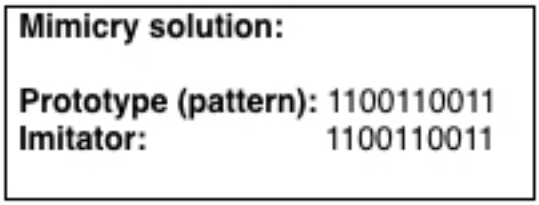
3.2. The Mimicry Problem

In a natural environment, some creatures share common patterns with others species that are distasteful to their predators [281]. As a discrete problem, the mimicry problem minimises the numbers of different bits between two bitstrings; a full problem statement and an example of an optimum solution are given below.

Table 3.1: Problem statement of the mimicry problem

Problem statement
The purpose of the Mimicry problem is to imitate a pattern of bits, so that two bitstrings become identical [349, 146].

Figure 3.2: An optimum solution of a the mimicry problem, for an instance of 10 bits



An optimum solution should have two identical bitstrings, as illustrated in figure 3.2. OnesMax problem is a particular instance of the mimicry, with its pattern restricted to 1s. In comparison, the mimicry problem sets more complex pattern using 0s and 1s. Some of them can be generated randomly, some others by hand; however, a prototype remains the same during a run. Also, OnesMax' fitness evaluation tends to sum all the 1s, and an optimum solution fitness value is equivalent to the length of the bits. The Mimicry evaluation often uses a form of Hamming distance; this is discussed in section 3.2.2.

The Mimicry has yet to attract the same attention as OnesMax. Optimum solutions for the OnesMax problem tends to be shorter than 2000 bits[347, 316, 206] and for the Mimicry problem a quarter of that amount [349, 146]. Our interest lies in finding solutions for much more substantial instances using generated solvers.

As a black-box problem, no direct comparison is allowed by any solvers or operators applied on a solution. Only meta-knowledge can indicate how similar or dissimilar are both bitstrings. This constraint raises the level of challenge and should make it relevant to the research community and our research.

3.2.1 The chosen encoding scheme

A pair of bitstrings encodes a mimicry solution (see figure 3.2 and expressions 3.16, 3.17, 3.18); a *prototype* stores a pattern of bits that is imitated by an *imitator*. Both bitstrings have the same fixed number of bits and are randomly generated during the initialisation process.

A prototype remains unchanged during the lifespan of the solution and passes from one solution to another during one run. The mimicry-specific operators only alter the imitator.

$$Prototype : \{p_1, ..., p_L\}, p \in \{0, 1\} \tag{3.16}$$

$$Imitator : \{i_1, ..., i_L\}, i \in \{0, 1\} \tag{3.17}$$

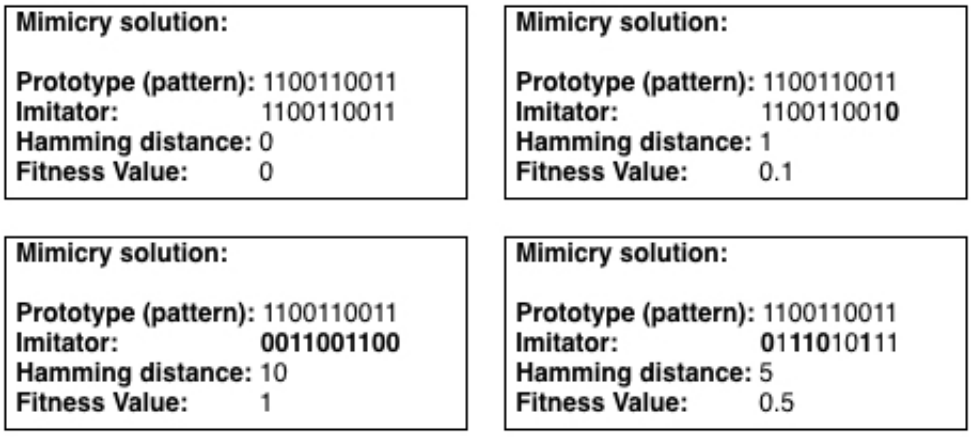
$$Solution : \{I, P\} \tag{3.18}$$

3.2.2 Fitness evaluation

Our purpose is to minimise the Hamming distance between a prototype and an imitator. The total number of bits that differs between the two bitstrings is computed. The difference between each bit (i.e. $|p_i - i_i|$) is 0 when the bit are similar or otherwise 1 (see equation 3.19).

The fitness value is generalised in every instance. The evaluation process divides the Hamming distance by the length of the substrings. Every solution has consequently a fitness value in the range of $[0, 1]$, as illustrated in figure 3.3. An optimum solution has a Hamming distance of 0, while the least suitable solution has a value of 1. In this solution, the imitator is in bold to indicates every bit differs from the prototype. The two other solutions provide examples with varying fitness values.

Figure 3.3: Solutions of the mimicry problems for an instance of 10 bits. A Hamming distance is given with the fitness value of each solutions. The bits in bold in the imitator are dissimilar from the the prototype.



$$HammingDistance(aSolution) : \sum_{i=0}^{Length} |p_i - i_i| \quad (3.19)$$

$$ProblemEvaluation(aSolution) : HammingDistance(Length)/Length \quad (3.20)$$

$$ProblemEvaluation(aSolution) : Solution \mapsto IR \mapsto [0, 1] \quad (3.21)$$

It is also worth noting, a Hamming distance of 5 bits for an instance of 1000 bits should have a higher quality than for an instance of 10 bits. Using expression 3.20, the first solution (i.e. 1000-bit instance) would score less than 1% (0.005) and for the 10-bit solution 0.5. As a result, some relevant and meaningful information about the quality of a solution is suitably included in the fitness value.

3.2.3 Problem parameters

All the mimicry parameters are described below.

$$Instance : Length \in IN \quad (3.22)$$

$$Prototype \in [0,1]^{Length} \quad (3.23)$$

$$MutRate \in [0,1] \quad (3.24)$$

$$AdaptiveMutRate \in [0,1] \quad (3.25)$$

$$ProblemParam = \{Instance, Prototype, AdaptiveMutRate, MutRate\} \quad (3.26)$$

An instance or Length is characterised by the number of bits composing the prototype and the imitator of a solution. Referred as the *Length*, this parameter defines the instance of this problem as a natural number (i.e. $[1..\infty]$); this is formally defined in equation 3.22. For example, an instance sets to 10 then its solutions have 10 bits, and an instance sets to 1,000 generates solutions with 1,000 bits. Optimum solutions for the second instance are more difficult to find compared against the first one. As the length increases, it becomes challenging to imitate perfectly an imitator to its prototype.

A prototype sets the pattern of bits to be imitated by each solution.

Mutation rates set defines the number of bits that are mutated by some mutation operators. *MutationRate* remains unchanged during the search (see equation 3.24). The adaptive mutation rate is adapted to the search needs. This process is discussed in the next section. Both rates must have a value between 0 and 1, where 0 indicates no mutation and 1 should mutate a whole imitator.

3.2.4 Problem operators

In an evolutionary context, bitstring operators should generate a new mimicry solution from at least one individual. Our chosen non-deterministic operators (i.e., crossover and mutation) manipulates the imitator to produce only one solution. This solution has the same prototype but should have a different imitator.

3.2.4.1 Crossover operators

Figure 3.4 and equations 3.27, 3.28, 3.29 formally define several traditionnal crossover techniques; those are considered as valid recombination operators with a bitstring encoding scheme [4].

$$CrossoverOnePoint(Solution1, Solution2) : (Solution \times Solution) \mapsto Solution \tag{3.27}$$

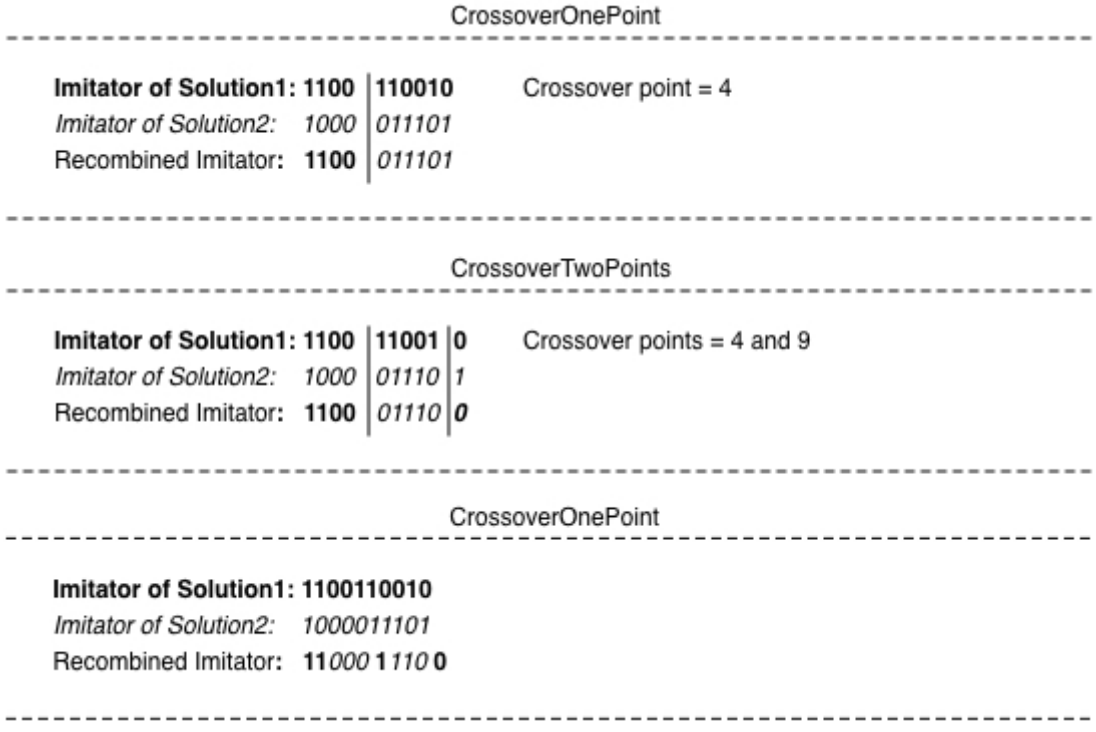
$$CrossoverTwoPoints(Solution1, Solution2) : (Solution \times Solution) \mapsto Solution \tag{3.28}$$

$$CrossoverUniform(Solution1, Solution2) : (Solution \times Solution) \mapsto Solution \tag{3.29}$$

All these operators create only one offspring, to reduce the level of disruption that a crossover can bring to one solution. A crossover can either be destructive or constructive. For that reason, Herdy et al. [146] have successfully adopted crossover operators that produce only one solution. However, we may be trading preservation with a lower survival rate for individually created with such crossover [301].

CrossoverOnePoint splits into two parts the imitator of two solutions. The first part of the imitator of Solution1 is recombined with the second part of the imitator Solution2. A crossover point is randomly selected so that $1 \leq point \leq Length$.

Figure 3.4: Crossovers techniques applied to the imitators of the two solutions



CrossoverTwoPoints uses two crossover points instead of one. Every bit of the imitator of Solution1 before the first crossover point is copied to the new imitator. The middle section of the imitator of Solution2 is recombined to the new bitstring, before adding the last part of the imitator of the Solution1 again.

CrossoverUniform copies bits randomly from the imitator of Solution1 or Solution2 randomly.

3.2.4.2 Mutation operators

A few randomly chosen bits are changed from 0 to 1 and from 1 to 0. This type of operations brings some diversity that can improve or weaken the quality of a solution. It would be undesirable to flip a corrected bit of an imitator; an error would introduce. With a small instance, the error could be corrected easily. However, we anticipate the probability to correct the erroneous bit decrease as the length increases.

With that in mind, we have chosen the mutation operator used by the state-of-the-art [146], and one that adapts the mutation rate during to the optimisation run. We have also devised some mutation operators that maintain the quality of a solution or improves it; these "hill-climbers" have been labelled with the two letters "HC". Those may limit search space but should help to find a near-optimum or optimum solutions. Such operators are trading quality against using more fitness evaluations.

All these operators are illustrated in figure 3.5, defined in equations 3.30, 3.31, 3.32, 3.33, 3.34. Their behaviour is described below.

$$MutateOneBit(aSolution) : Solution \mapsto Solution \quad (3.30)$$

$$MutateUniformVariableRate(aSolution) : Solution \mapsto Solution \quad (3.31)$$

$$MutateOneBitHC(aSolution) : Solution \mapsto Solution \quad (3.32)$$

$$MutateUniformHC(aSolution) : Solution \mapsto Solution \quad (3.33)$$

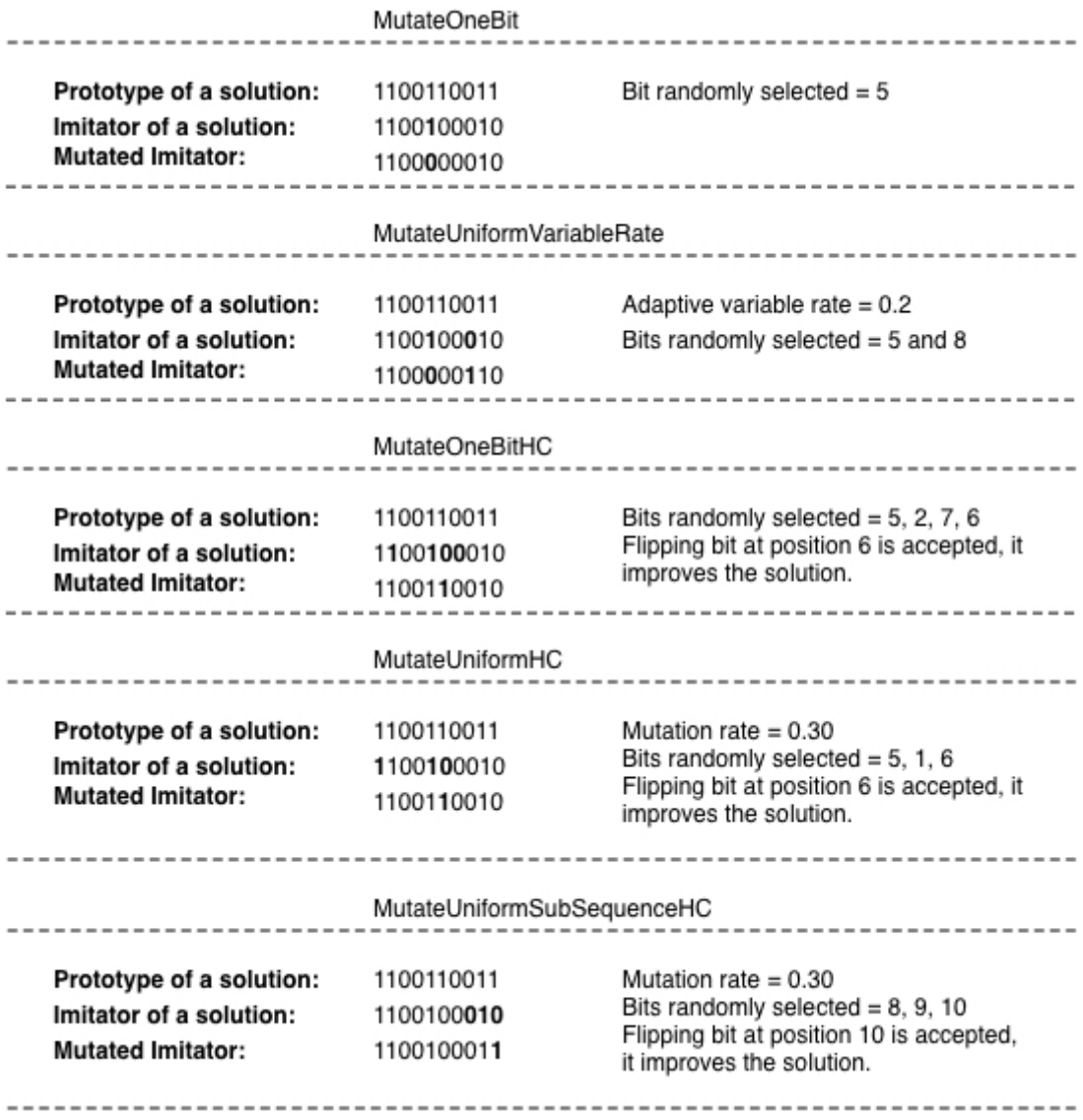
$$MutateUniformSubSequenceHC(aSolution) : Solution \mapsto Solution \quad (3.34)$$

MutateOneBit selects randomly one bit of an imitator and flips it; 0 becomes 1, and 1 becomes 0. The state-of-the-art [146] used this operator.

MutateUniformVariableRate uses an adaptive mutation rate and adjusts over time this parameter in response to the state of the search obtained from the optimisation run.

The *"One-Fifth rule"* has been implemented as suggested by Rechenberg in 1973 [150]. The operator is increasing the adaptive mutation rate if a $\frac{1}{5}$ of the offsprings are fitter than their parents. It is decreased when less than $\frac{1}{5}$ of the children are fitter than the parents, but remain unchanged when $\frac{1}{5}$ of the offsprings have a better fitness value than the parents.

Figure 3.5: Mutation techniques applied to the imitators of a solution



Our adjustments use a step of $\frac{1}{Length}$ so that the increase and decrease in the adaptive mutation rate has a relation to the instance. Short instances are likely to recover more easily when an error is introduced. Still, the probability to correct such mistake becomes reduced as the length increases. This operator is added for completeness, despite being discouraged by the state-of-the-art [146].

MutateOneBitHC applies the *mutateOneBit* operator repetitively. This process stops when its fitness has improved (i.e., one different bit has been flipped) or four attempts have been made.

MutateUniformHC also applies repetitively the *MutateOneBit* operator. In this case, the operator stops when a maximum number of flips has been applied. Only flips that have made a positive impact on the solution are kept. The number of flips is computed from the *mutation rate* parameter and the number of bits in the instance; i.e. $noFlip = mutationRate \times Length$.

MutateUniformSubSequenceHC flips all the bits of a sub-string. When an error is introduced, then the alteration is revoked. For consistency, the maximum number of flips is calculated using the same formulae as the *MutateUniformHC* operator. As a result, any sub-strings position are defined as $[start, start + noFlip]$. A start position is randomly chosen within the range $[1, Length]$. When the end position exceeds the length of an instance (i.e., $start + noFlip > Lentgh$), then the sub-string is shortened as the like a bit of the bitstring becomes the end of the sub-sequence.

3.2.5 Summary

This section has introduced the mimicry problem domain. A pair of bitstring encodes a solution, but the operators listed in table 3.2 change only the bits from an imitator. Those form a complete set. Either the state-of-the-art or other contexts have applied these operators successfully.

Table 3.2: Mimicry operators with their opcode and the number of evaluations used.

OpCode	Operator(s)	Number of evaluations
0	CrossoverOnePoint()	1
1	CrossoverTwoPoints()	1
2	CrossoverUniform()	1
3	MutateOneBit()	1
4	MutateOneBitHC()	[1,4]
5	MutateUniformSubSequenceHC()	Length * MutationRate
6	MutateUniformHC()	Length * MutationRate
7	MutateUniformVariableRate	1

3.3. The Traveling Salesman Problem

For a long time, researchers have studied routing problems. In the 18th century, Euler generalised the *"Könisberg bridges"* problem, which is often regarded as the birth of Graph theory. 100 years later an extension to his formulation allowed solving the mail carrier and "commis-voyageur" problems. Since, this concept has successfully designed some circuits as well as many of forms of networks [44, 277, 16].

On numerous occasions, suitable routing problem solutions form some closed circuits; those passe only once on a bridge, on the street, a city or in a node of a graph and tend to start and end at the same point. Formally defined as *"Hamiltonian cycles"*, a graph $G = \{V, E\}$ is used where each vertex V is visited only once as a constraint.

It is believed the traveling salesman problem (TSP) originated from the Icosian game developed by the mathematician W. R. Hamilton more than 150 years ago. The purpose was to devise a route a traveller that needed to visit 20 different cities, without visiting any of them twice. Solutions start and end in the same city (i.e., those are considered as Hamiltonian tours). Otherwise, they are thought to be Hamiltonian paths (see figure 3.6).

In the 20th century, Karl Menger set the task of finding the shortest path connecting each vertex whose pairwise distances are known. [252, 44, 277, 16]. This deceptively simple goal of the "traveling salesman problem" stated in table 3.3 includes all these elements.

Figure 3.6: Possible solutions for the Icosian game [252]

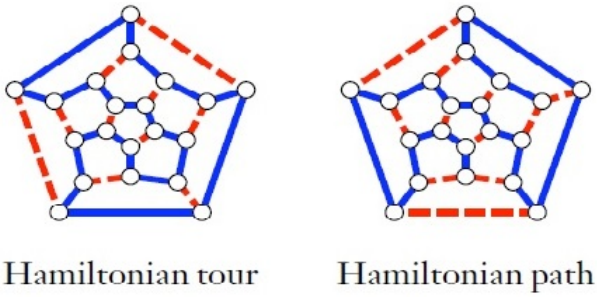


Table 3.3: TSP Problem statement

Problem statement
Given a set of cities along with the cost of travel between each pair of them, traveling salesman problem, or TSP for short, is to find the cheapest way of visiting all the cities and returning to the starting point [16].

Figure 3.7: An optimum solution of a TSP instance made of 5 cities.

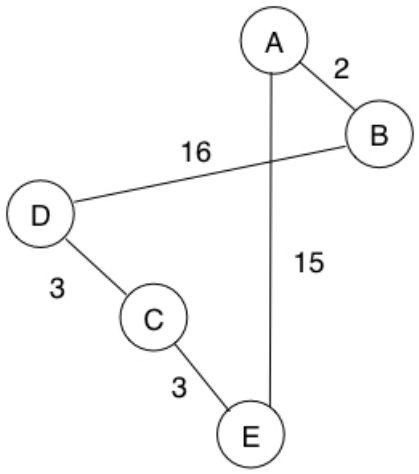


Figure 3.7 has five cities and its distance is 24; we will discuss in subsection 3.3.2 reasons why this Hamiltonian cycle is one optimum solution. At the time of writing, no general method can find optimum solutions for every instance of the TSP. The state-of-art includes Concorde, an integer programming system [15], *Lin-Kernighan heuristics* a local search heuristics [145] and an edge-assembly crossover [236]. These three methods are considered to be the state-of-the-art and have found many of the optima of the benchmarks available on TSPLIB [1] and the national TSP instances [2].

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html
[2] http://www.math.uwaterloo.ca/tsp/world/index.html

Evolutionary algorithms have found some known optima for instances up to a few hundreds of cities [116, 240, 181, 86, 28, 102, 163, 113, 219, 117]. When metaheuristics couples an EAs with a local search instances up to 1000 cities can be solved. However, their performance can decrease with the more substantial tours [232, 177, 124]. With a Memetic Algorithm, the genetic operators generate the genetic code of a TSP solution, but it remains unchanged during its life. Then, the local search mechanism performs individual learning by refining the quality of the TSP solutions; this brings an element of cultural evolution during a TSP-solution lifespan.

Iterated Local Search can "jump" from local optima to a nearby one. A tour is altered by a mutation operator to escape the local optima. These changes need to be big enough to prevent the search "falling" again in the same local optima. Readers who wish to read further about Iterated Local Search, in general, will find [198, 176] very informative.

3.3.1 The chosen encoding scheme

Formally, a graph encodes a tour (see equation 3.35), where a node represents a city and an edge distances between each pair (i.e., a Euclidean vector). An array of integer often encodes this graphical representation; the sequence of cities are arranged in a certain order to describe a tour and referred as *permutations*. The first city indicates the start of a tour, then the subsequent cells the cities that need to be visited. The last and first cities are paired to form a cycle. Each city is given a unique index that is used to form cyclic permutations; as formally defined in equations 3.36, 3.37, 3.40.

Figure 3.7 illustrates city A becomes index 1, city B index 2 up to city E index 5 and the tour is written again as this cyclic permutation $\{1, 2, 4, 3, 5\}$. Several examples of TSP solutions is given in 3.8, with two feasible solutions and one infeasible.

The pairing of cities defines the direction of each edge (i.e., one city to the subsequent one). The edge weight represents the distance to travel from one city to another one. Distance matrices or a list of Euclidean coordinates can provide these distances [126]. We will be adopting the latter to comply with the standard of TSPLIB defined by Reinelt [264]. Equations 3.38 and 3.39 formally defines formulae to find the weight of each edge.

$$G = \{V, E\} \quad (3.35)$$

$$Indices : 1..NoOfCities \quad (3.36)$$

$$Tour : \{i_1, .., i_{NoOfCities}\} \quad (3.37)$$

$$EuclideanDistance(x_i, y_i, x_j, y_j) : round(\sqrt{(}\,(x_i - x_j)^2 - (y_i - y_j)^2)) \quad (3.38)$$

$$EuclideanDistance(x_i, y_i, x_j, y_j) : (IR \times IR \times IR \times IR) \mapsto IN^+ \quad (3.39)$$

$$Solution : \{Tour, ListOfEuclideanCoordinates\} \quad (3.40)$$

3.3.2 Fitness evaluation

The problem statement (see table 3.3) suggests the best tour consist of the shortest possible Hamiltonian cycle for a distinctive graph. The weight of every edge connecting the cities are added together to calculate the tour length (see equations 3.41 and 3.42).

The function $nextCity(index)$ retrieves the following city in a permutation to calculate the weight of an edge. When it is repetitively called from the first to the last town (i.e., a cycle), then the tour length can be computed.

$$Length(city_1, city_2) : EuclideanDistance(x_{city_1}, y_{city_1}, x_{city_2}, y_{city_2}) \quad (3.41)$$

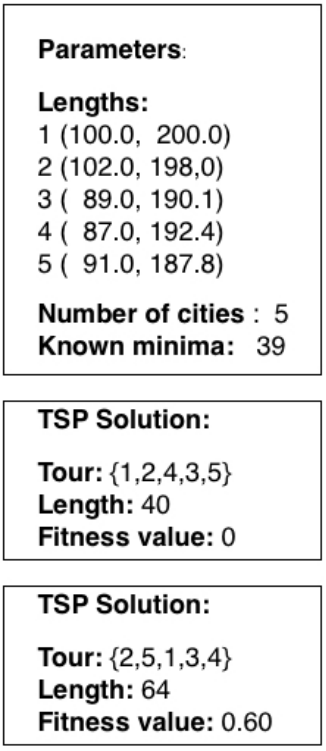
$$LengthOfAtour : \sum_{i=1}^{NoOfCities} Length(i, nextCity(i)) \quad (3.42)$$

$$ProblemEvaluation(aSolution) : \frac{TourLength - Mimina(Instance)}{Minima(Instance)} \quad (3.43)$$

$$ProblemEvaluation(aSolution) : Solution \mapsto IR \mapsto [-\infty, \infty] \quad (3.44)$$

Known optima can vary in length from one instance to another. For example the instance *dj38* for the country Djibouti has 38 cities and an minima of 6,656 km, and Canada (instance *ca4663*) short tour is 1,290,319 km [77]. A relative error provides a more consistent fitness value and also compare naturally against the state-of-the-art. For these reasons, the problem evaluation function could return a value between $-\infty$ and ∞.

Figure 3.8: Examples of TSP solutions for a 5-city instance.



3.3.3 Parameters

The parameters influencing the instance and TSP-specific operators are listed in expressions [3.45 - 3.50].

$$Instance : Name \in String \tag{3.45}$$

$$NoCities \in IN^{+} \tag{3.46}$$

$$Data : (indexOfACity, X - coor, Y - coord)^{NoOfCities} \tag{3.47}$$

$$Depth \in [0, 1] \tag{3.48}$$

$$Intensity \in [0, 1] \tag{3.49}$$

$$ProblemParam : \{Instance, NoCities, Data, Depth, Intensity\} \tag{3.50}$$

Each instance has a unique name composed of some letters and a number. The latter defines some cities, and the two letters produce a unique identifier (i.e., dj38). However, some instances may have the same amount of cities (see equation 3.45). The latter should be positive (see expression 3.46). We believe Hamiltonian cycles become interesting with a minimum of 4 vertices; with a lower number of nodes only one tour exists. Our experiments will find solutions for instances ranging from 38 to more than ten thousand cities.

We have adopted a list of Euclidean coordinates (i.e., $Data$ in expression 3.47) consistent with the TSPLIB format described in [264]. These tuples provides the X and the Y coordinates of any cities needed to calculate the Euclidean distance (see equations 3.38, 3.41, 3.42, 3.47).

The remaining parameters can affect positively or negatively the performance of some operators. The intensity of the mutation defines the number of cities to be shuffled in a permutation and the depth the number of iteration used in a local search. Both parameters are formally defined in equations 3.48 and 3.49. The next section introduces in detailed all the TSP-specific operators.

3.3.4 Problem operators

Permutations can be altered using a variety of deterministic and non-deterministic operators. We have chosen some popular crossover, mutation, and local search operators specialised for exploring the TSP solution search space [86, 113, 117, 219, 163, 102, 28, 145, 133].

We adopted the approach suggested by the Automated Scheduling, Optimisation, and Planning (ASAP) group, during the development the Hyflex framework [242]. Our TSP problem domain includes some heuristics divided into three subsets; crossover, mutation, and local search operators. All of them only produce one TSP candidate-solution and evaluate the length of its associated tour.

3.3.4.1 Crossover operators

With some population-based metaheuristics, crossover operators tend to probe a much larger portion of the TSP-solution search space and bring some diversification to a population of solutions. Crossover operators often produce two-offsprings, from two parents, but they can also produce only one offspring. The function signature uses the latter (see equations 3.51, 3.52, 3.53), bringing a certain consistency with the other categories of operators.

Applying a crossover operator onto two permutations is likely to produce some infeasible tours. A mechanism that guarantees to transform two Hamiltonian cycles into one prevents the creation of Hamiltonian paths. The research community has adopted this specialism for a long time, instead of letting the evolution finding the Hamiltonian cycles naturally.

$$OX(Sol1, Sol2) : (Solution \times Solution) \mapsto Solution \tag{3.51}$$

$$PMX(Sol1, Sol2) : (Solution \times Solution) \mapsto Solution \tag{3.52}$$

$$VR(Sol1, Sol2) : (Solution \times Solution) \mapsto Solution \tag{3.53}$$

$$SEC(Sol1, Sol2) : (Solution \times Solution) \mapsto Solution \tag{3.54}$$

Order Based Crossover (OX) chooses a sub-tour in one parent and imposes the relative order of the cities of the other parent [86]. In figure 3.9, the relative order of the sub-tour $\{2, 4, 3\}$ of Tour no 1 has been assigned to the second tour (i.e., Tour no 2).

Partially-Mapped Crossover (PMX) copies an arbitrarily chosen sub-tour from the first parent into the second parent, before applying minimal changes to construct a valid tour [113, 117]. Figure 3.9 illustrates the sub-tour $\{2, 4, 3\}$ of the first parent has been copied to the recombined tour. To form a Hamiltonian cycle, the city no 2 has been swapped with city no 5 and the city no 1 with no 4.

Voting Recombination Crossover (VR) uses a randomized Boolean voting mechanism to decide from which parents each city is copied from [219]. Our simulation in figure 3.9, has selected position 1, 3 and 5 to choose a city from the first parent and the remaining one from the second individual.

Subtour-Exchange Crossover (SEC) preserves randomly selected sub-tours from both parents to construct one new offspring. Some minor adjustments are applied to build a feasible tour [163]. The recombined tour in figure 3.9 is composed of the first two cities of the first parents and the remaining ones from the second tour. It worth noting, the city no 1 correctly appear only once in the solution, and it has been replaced by city no 5, to prevent an invalid tour.

Figure 3.9: Examples of TSP solutions for a 5-city instance.

Order-Based Crossover

Tour no 1:	**{1,2,4,3,5}**	Start position: 2
Tour no 2:	*{2,5,1,3,4}*	End position: 4
Recombined Tour:	{*2,5,1,***4,3**}	sub-tour {2,4,3}

Partially-mapped Crossover

Tour no 1:	**{1,2,4,3,5}**	Start position: 2
Tour no 2:	*{2,5,1,3,4}*	End position: 4
Recombined Tour:	{*5,***2,4,3,***1*}	

Voting Recombination Crossover

Tour no 1:	**{1,2,4,3,5}**
Tour no 2:	*{2,5,1,3,4}*
Recombined Tour:	{**1**,*2*,**4**,*5*,**3**}

Subtour-Exchanged Crossover

Tour no 1:	**{1,2,4,3,5}**	Dividing position: 2
Tour no 2:	*{2,5,1,3,4}*	
Recombined Tour:	{**1**,**2**,*5,3,4*}	

3.3.4.2 Mutation operators

The cities' order of a permutation are altered with the hope of refining and reducing the tour length. Swapping two cities, inversing sub-tours, rearranging the whole permutation or part of it can suitably produce a new tour from an existing one. These unary operations should transform a Hamiltonian cycle to another one without any added specialism; no city or sub-tours are interchanged between solutions.

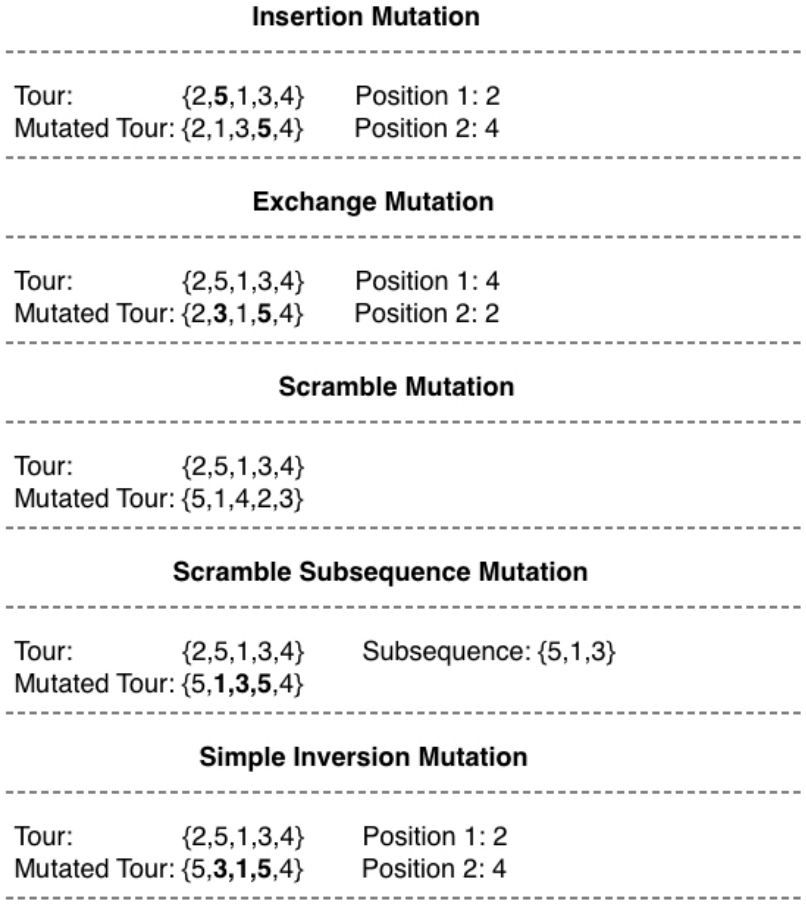
$$InsertionMutation(aSolution) : Solution \mapsto Solution \tag{3.55}$$

$$ExchangeMutation(aSolution) : Solution \mapsto Solution \tag{3.56}$$

$$ScrambleMutation(aSolution) : Solution \mapsto Solution \tag{3.57}$$

$$SimpleInversionMutation(aSolution) : Solution \mapsto Solution \tag{3.58}$$

Figure 3.10: Examples of TSP solutions for a 5-city instance.



Insertion Mutation (IM) moves a randomly chosen city in a tour to randomly selected place [102]. In figure 3.10, the city no 5 has moved to the fourth position.

Exchange Mutation (EM) swaps two randomly selected cities. The cities 3 and 5 were swapped in figure 3.10 [28].

Scramble Mutation (SM) rearranges a random sub-tour of cities [86]. Hyflex applies this mutation operator on a sub-tour and the whole tour. Examples of both operators are given in figure 3.10.

Simple Inversion Mutation (SIM) implements a 2-opt heuristics, that is discussed in the next section. In our example given in figure 3.10, the order of the subsequence $/5, 1, 3/$ has been inverse. In this instance, the outcome produces the same tour as the exchange mutation. It is worth noting, with longer permutations, it is less likely to occurs.

3.3.4.3 Local search operators

Local searches iteratively move from one permutation to a neighbour tour. Traditionally those are often referred as *k-opt heuristics*. Some edges between two pair of cities are reconnected differently to obtain a new shorter tour. Examples of such moves are illustrated in figures 3.11 and 3.12.

Unlike the mimicry problem (see section 3.2), the effect of some alterations can be locally assessed, without evaluating the whole solution. The local search operators provided by Hyflex analyses the possible (if not all) the k-opt moves before applying the best move [145, 133]. It is, therefore, suitable to count one evaluation for each local search, as only once full evaluation of a newly created tour is computed at the end of the process.

Our chosen local search operators implement these well-documented local searches. Those are formally define in equations 3.59, 3.60, 3.61. And described below. Instead of providing some sample solutions, we have preferred to provide some example moves. These operations have a higher level of sophistication that cannot be fully demonstrated in a simple example.

$$2\_OptLocalSearch(aSolution) : Solution \mapsto Solution \tag{3.59}$$

$$Best2\_OptLocalSearch(aSolution) : Solution \mapsto Solution \tag{3.60}$$

$$3\_OptLocalSearch(aSolution) : Solution \mapsto Solution \tag{3.61}$$

2_OptLocalSearch stops the search as soon as a shorter tour is found, by a 2-opt exchange [133].

Best2_Opt-LocalSearch relies on ranking all edges with each nodes ordered by increasing length. At a node, all the possible 2-opt-moves are examined by enumeration. The best move is then applied to the solution [133].

3_OptLocalSearch() deletes 3 edges before reconnecting them with optimum reconnection. All the possible connections are considered [145].

Figure 3.11: An example of a 2-Opt Local Search [153]

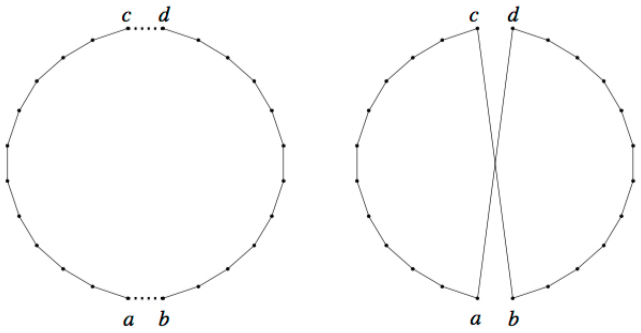
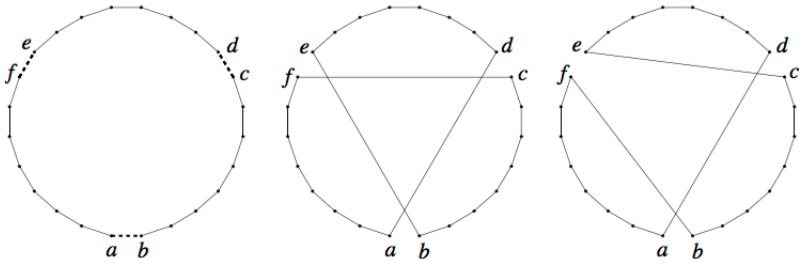


Figure 3.12: An example of a 3-Opt Local Search [153]



3.3.5 Summary

The traveling salesman problem is a more complicated problem than the mimicry problem. Therefore its problem domain uses an encoding scheme with several elements; a graph defined by some nodes, edges and some weights. Its $ProblemEvaluation$ function relies on Euclidean distances and known minima to compute a relative error. Those are unique for each instance. Some instances have a short tour length and some others very long ones. A relative error offers a consistent approach to assess a solution.

Table 3.4 lists all the operators specific to the traveling salesman problem domain. Heuristic techniques that have found some tours efficiently and that are well-documented in the literature have been included in our set of TSP-specific operators.

Table 3.4: Traveling Salesman operators with their opcode and the number of evaluations used.

OpCode	Operator(s)	Number of evaluations
0	InsertionMutation()	λ
1	ExchangeMutation()	λ
2	ScrambleWholeTourMutation()	λ
3	ScrambleSubTourMutation()	λ
4	SimpleInversionMutation()	λ
6	2-OptLocalSearch()	λ
7	Best2-OptLocalSearch()	λ
8	3-OptLocalSearch()	λ
9	OrderBasedCrossover()	λ
10	PartiallyMapCrossover ()	λ
11	VotingRecombinationCrossover()	λ
12	SubtourExchangeCrossover()	λ

3.4. The Nurse Rostering Problem

The primary goal of the nurse rostering problem (NRP) is to arrange some non-overlapping shifts for some nurses over a well-defined period so that the cost of the workforce is minimised. This problem is part of specific real-life problems under the umbrella referred as *personnel scheduling problem* [83, 3].

In 1954, Dantzig [84] and Edie [96] were the first mathematician to simplify the *Port of authority toll booth* personnel, to reduce the delays at the bridges without increasing the number of employees. Their suggested mathematical models were rather simple, but efficient. Dantzig [84] chose to model the working pattern with a square matrix. Each row represents the pattern of a shift and each column a period of work. A binary value flags whether an employee is scheduled to work at a given time (see figure 3.13).

The number of periods needed for a work pattern is represented by w_i (see expression 3.62). The variable b_t defined in expression 3.63 is the number of toll gates required for a period. The number of periods available for one day is expressed by expression 3.64 and stored in N. In this formulation, w_0 is the number of unused periods and those needs to be maximised (see expression 3.65).

$$NoPeriodsInAWorkPattern : w_{aPattern} = TotalPeriods(x, aPattern) \quad (3.62)$$

$$NoGatesOpenedInAPeriod : b_{aPeriod} = TotalGates(x, aPeriod) \quad (3.63)$$

$$NoPeriodsAvailableInADay : N = TotalPeriodAvailable(x) \quad (3.64)$$

$$NoPeriodsUnused : w_0 = Max(TotalUnusedPeriods(x)) \quad (3.65)$$

Personnel scheduling problems include many different industries and services. Keeping a company's cost as low as possible can determine its competitive strength. One factor often controls its workforce efficiency [323]. The *nurse rostering problem* has a complex set of constraints that are hard to comply; it is even more difficult to find an optimum. Figure 3.14 shows a roster with 8 nurses over a period of 29 days schedules the periods of relief, the day and night shifts for each nurse.

Detailed reviews and surveys of mathematical and artificial intelligence methods are provided by [100, 68, 263, 50]. Integer programming and non-deterministic methods have successfully found solutions to many instances of the nurse rostering problem [46, 68, 71, 246, 46, 322, 125, 22, 111, 340].

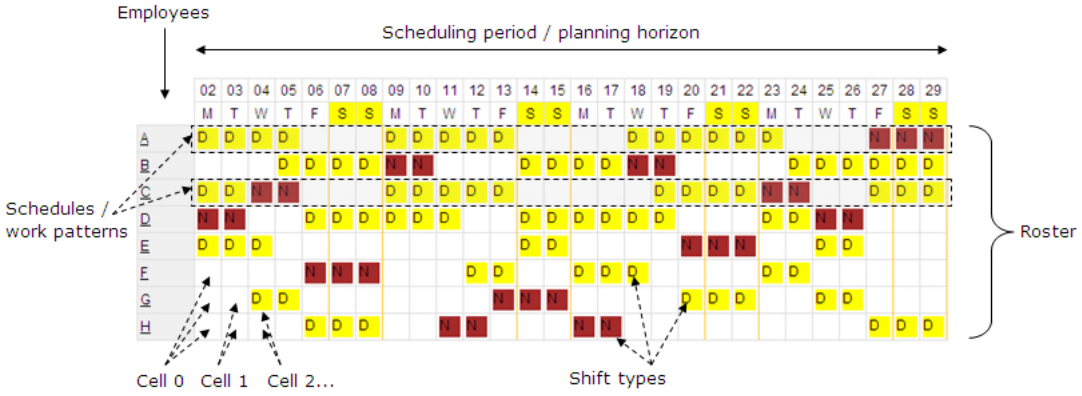
Classical genetic algorithm and some metaheuristics often handle the conflict between the objectives and constraints ineffectively. These paradigms have therefore been specialised; some extra features have been added, such as ranking or repairing the constraints violations [208, 8, 9, 35].

More recently, selective hyper-heuristics have made some good advancements [167, 57, 95, 66]. To the best of our knowledge, NRP solvers have yet to be discovered within the context of generative hyper-heuristic.

Figure 3.13: A formulation of work patterns using a linear programming problem as suggested by [84]

Pattern	Period t						Total
	(1)	(2)	(3)	(4)	(5)	(6)	
1.1	x_{11}	—	x_{13}	x_{14}	—	x_{16}	w_1
1.2	x_{21}	—	x_{23}	x_{24}	—	x_{26}	w_2
2.1	x_{31}	x_{32}	—	x_{34}	x_{35}	—	w_3
2.2	x_{41}	x_{42}	—	x_{44}	x_{45}	—	w_4
2.3	x_{51}	x_{52}	—	x_{54}	x_{55}	—	w_5
3.1	—	x_{62}	—	x_{64}	x_{65}	—	w_6
—	—	—	—	—	—	—	w_0
Total	b_1	b_2	b_3	b_4	b_5	b_6	N

Figure 3.14: A formulation of work patterns using for the nurse rostering problem [3]

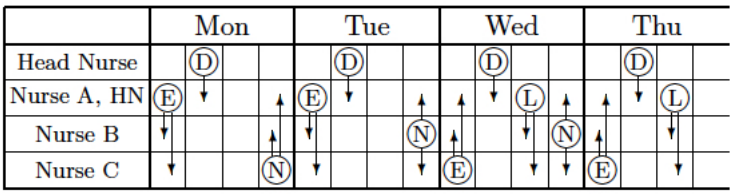


3.4.1 The chosen encoding scheme

A roster plans nurses shifts over a period of time. A minimal set of decision variables, some domain values and a complex set of constraints are encoded in one solution (see expression 3.66); these three elements are common to constraint-satisfaction problem [272]. We will discuss these three elements in the following paragraphs.

A table helps to visualise the binary programming. Expressions 3.67 and 3.68 defines the variable and domain of the nurse rostering problem. For example, $x_{nurse,day,shiftType}$ indicates whether a nurse has been scheduled to cover a shift on a certain day. In Figure 3.15 $x_{HeadNurse,Mon,D}$ is set to 1 and $x_{HeadNurse,Mon,N}$ to 0 . Figure 3.14 has simplified the schedules by showing only one column per day. As a result, when no shift is scheduled for a nurse on a certain day, the cell remains empty.

Figure 3.15: A visual representation of the nurse rostering problem [57]



$$Solution : \langle Variable, Domain, Constraints \rangle \tag{3.66}$$

$$Variable : x_{nurse,day,shiftType} \tag{3.67}$$

$$Domain : \{0, 1\} \tag{3.68}$$

The constraints have threefold. First, the feasibility of a roster is defined. Hard constraints are assumed to be met all the time, to guarantee that only infeasible and acceptable rosters are found. Burke et al. [57] have demonstrated that metaheuristics initially generating and creating feasible rosters is highly beneficial. Secondly, the soft constraints or *objectives* improve the quality and accuracy of feasible rosters. [110, 48] have relaxed some hard constraints to objectives to assess more accurately the quality of a roster. The evaluation process is discussed in section 3.4.2. Thirdly, the objectives can differentiate the cover needs from the soft constraints concerned about the nurses' satisfaction. These two aspects play an important role to the efficiency of the wards. All these constraints are often defined with more complex mathematics [83, 57, 51]. We prefer expressing them in a more general manner using a functional language (see expressions [3.70 - 3.79]).

The hard constraint $\bar{g}$ prescribes the number of shifts per day (see expression 3.70). The constraint must be applied to every nurse to ensure a roster is feasible. Nonetheless, a *coverage objective* ensures the preferred number of nurses are working during each shift (see expression 3.71).

$$Constraints : \left\langle \bar{g}, \bar{h} \right\rangle \tag{3.69}$$

$$\bar{g}_0(x) = TotalDailyShift(aNurse, aDay) \leq maxDailyShift \tag{3.70}$$

$$\bar{h}_0(x) = \begin{cases} 1 & if\, TotalShift(aType, aDay) \leq MinCover(aType, aDay) \\ 0 & otherwise \end{cases} \tag{3.71}$$

The intent of some nurses working objectives is to optimise the nurses' satisfaction with their work schedules. It is highly preferable to prevent some undesirable succession of shifts, to limit to a reasonable amount of consecutive working days and to plan a minimum of days off. The nurses need to sufficiently rest between shift and do not exceed working their contractable hours. These objectives are defined in expressions 3.72, 3.73, 3.74 and 3.75.

$$\bar{h}_1(x) = \begin{cases} 1 & if\, TotalWeeklyHours(aNurse) \geq MaxHours(aNurse) \\ 0 & otherwise \end{cases} \tag{3.72}$$

$$\bar{h}_2(x) = \begin{cases} 1 & if\, hasUndesirableShiftSuccession(aNurse) = true \\ 0 & otherwise \end{cases} \tag{3.73}$$

$$\bar{h}_3(x) = \begin{cases} 1 & if\, TotalConsecD(aNurse) \notin ConsecDays(aNurse) \\ 0 & otherwise \end{cases} \tag{3.74}$$

$$\bar{h}_4(x) = \begin{cases} 1 & if\, TotalConsecDOff(aNurse) < MinConsecDayOff \\ 0 & otherwise \end{cases} \tag{3.75}$$

A reasonable life-work balance must be instilled. Weekends should be either day off or working days (see expression 3.76). A maximum number of working weekends should also be respected for a period of planning (see expression 3.77).

$$\bar{h}_5(x) = \begin{cases} 1 & if\, TotalWeekendShift(aNurse) \notin NoWEShift \\ 0 & otherwise \end{cases} \tag{3.76}$$

$$\bar{h}_6(x) = \begin{cases} 1 & if\, TotalWorkWE(aNurse) \geq MaxWorkingWE \\ 0 & otherwise \end{cases} \tag{3.77}$$

Finally, a maximum number of nights shift should be limited to an allowed consecutive number of working days. Expressions 3.78 and 3.79 define this highly desirable outcome.

$$\bar{h}_8(x) = \begin{cases} 1 & if\, TotalConsecShift(aNurse, N) \notin ConsecDays(N) \\ 0 & otherwise \end{cases} \tag{3.78}$$

$$\bar{h}_9(x) = \begin{cases} 1 & if\, TotalWeekShift(aNurse, N) < NoWeekShift(N) \\ 0 & otherwise \end{cases} \tag{3.79}$$

3.4.2 Fitness evaluation

By the nature of the problem, the evaluation of a roster provides a cost, which becomes large when some constraints are not met. Otherwise, its value is low. This cost can, therefore, indicate whether a solution is suitable or not.

The soft constraints, introduced in the previous section, have therefore been transformed to weighted objectives with high values. The cost of a roster has become a weighted sum of all the objectives (see expression 3.80). The fitness value can adequately inform a solver whether its search is finding optimum, near-optimum or inadequate solutions. This process is independent of the varying level of information provided by the parameters.

Some instances have a known minimum with a value of 0. Our problem evaluation function has been adapted so that a relative error can be returned too. We are now adding the main features describing the problem (i.e. the number of nurses, the number of days in the planning period and the number of types of shift). Those are given in expressions 3.81 and 3.82.

$$Cost : \sum_{n=1}^{LastObjective} weight_n \times Constraints_n(x) \quad (3.80)$$

$$ProblemEvaluation(aSolution) : \frac{cost - Minima(Instance)}{Nurses + Period + ShiftTypes} \quad (3.81)$$

$$ProblemEvaluation(aSolution) : Solution \mapsto IR \mapsto [-\infty, \infty] \quad (3.82)$$

The problem parameters are very different than the ones used for the two previous problems. It has been defined a set of three different types of operators, to communicate more clearly the numerous information required by the problem domain. Expression 3.83 groups the nurse rostering problem parameters with three subsets; (1) the instance, (2) the constraints and (3) the operators. These three set of parameters are discussed below.

$$ProblemParam : \{Instance, Constraints, Operators\} \quad (3.83)$$

3.4.2.1 Instance parameters

An instance has specified the type of shifts, the length of a scheduling period and the number of employees (see expressions [3.84 - 3.87]). They contribute to defining the structure of the schedules; i.e. the number of rows, columns and the possible values of a cell. It is assumed the instance name is unique. However, the remaining parameters may be the same in several instances, as sometimes the constraints may only be different.

$$Nurses \in IN^+ \quad (3.84)$$

$$LengthPeriod \in IN^+ \quad (3.85)$$

$$ShiftTypes \in IN^+ \quad (3.86)$$

$$Instance : \{Name, Nurses, LengthPeriod, ShiftTypes\} \quad (3.87)$$

3.4.2.2 Constraints parameters

For each nurse, some weekly contractable hours needs to be specified. A range of numbers defines Suitable numbers of consecutive working days and weekend shifts. A nurse can only work a maximum number of weekends. All these parameters are expressed in expressions 3.88 and 3.89. A maximum of daily shift for every nurse is defined in expression 3.93.

The 24-hour cover needs to be scheduled. A shift type has a well-defined start and end time (see figure 3.16). It is undesirable for certain shift type to occur in succession. As a result, these patterns are defined in set referred as *Succession* and undesirable shifts are paired. A restriction on the maximum number of weekly occurrences and consecutive working days is desirable; therefore those are also specified in this set. Expressions 3.91 and 3.91 group these parameters together.

Each day of the week requires a minimum cover. A tuple in expression 3.92 specifies a minimum number of nurses needed for a type of shift on a specific day of the week (i.e. Monday to Sunday). The minimal cover required for a day is obtained using the function $MinCover(ShiftType, Day)$. An example of some shift type and their coverage is given in figure 3.16.

Figure 3.16: A description of shifts [48]

Shift	Start time	End Time	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Day (D)	08:00	17:00	3	3	3	3	3	2	2
Early (E)	07:00	16:00	3	3	3	3	3	2	2
Late (L)	14:00	23:00	3	3	3	3	3	2	2
Night (N)	23:00	07:00	1	1	1	1	1	1	1

A period of schedule often starts on Monday and it is referred to the value 1. Every Sunday is represented by a multiple of 7. Expression 3.94 defines a period of schedule as a series of days ranging between 1 to $LenghPeriod$.

$$Nurses : Nurse_1..Nurse_{NoOfNurses} \quad (3.88)$$

$$Nurse : \{MaxHours, ConsecDays, NoWEShifts, MaxWorkingWE\} \quad (3.89)$$

$$Shifts : ShiftType_1..ShiftType_{NoOfShiftTypes} \quad (3.90)$$

$$Shift : \{startTime, endTime, Succession, ConsecDays, WeeklyShift\} \quad (3.91)$$

$$Cover : \langle DayOfTheWeek, ShiftType, noOfNurses \rangle \quad (3.92)$$

$$DayShift = 1 \quad (3.93)$$

$$PeriodSchedule : Day_1..Day_{LengthOfPeriod} \quad (3.94)$$

$$Constraints : \{Nurses, Shifts, PeriodSchedule, Cover, DayShift\} \quad (3.95)$$

3.4.2.3 Operators parameters

Similarly to the traveling salesman problem, the parameters $Depth$ and $Intensity$ can affect positively or negatively the performance of some operators. The intensity is used to calculate the number of schedules changed by a non-deterministic operator. In this context, the depth affects the time some operators are applied on some rosters [83]. These two parameters are part of the Operators parameters and are defined in expressions [3.96-3.98].

$$Depth \in [0, 1] \quad (3.96)$$

$$Intensity \in [0, 1] \quad (3.97)$$

$$Operators : \{Intensity, Depth\} \quad (3.98)$$

3.4.3 Problem Operators

3.4.3.1 Crossover operators

The crossover operators should create a new roster using the best features of two parents. Shifts can be assigned and unassigned to obtain improved roster solutions hopefully. Expressions [3.99 - 3.101] defines the nurse rostering's crossover operators. Those are described below.

$$MultiEventCrossover : (Solution \times Solution) \mapsto Solution \tag{3.99}$$

$$ScatterSearchCrossover : (Solution \times Solution) \mapsto Solution \tag{3.100}$$

$$SimpleCrossover : (Solution \times Solution) \mapsto Solution \tag{3.101}$$

MultiEventsCrossover unassigns each shift temporarily to measure the changes in the problem fitness function. The largest increase in the cost identifies the best assignments. The best assignments for both are rosters then copied into the offspring. The number of best assignments are computed using the formulae $4 + round((1 - intensityOfMutation) * 16)$ [45, 83]

ScatterSearchCrossover uses first all the common assignments of both parents to create a new roster. Then it selects assignments alternately from each parent for the objectives that have yet to be met. [54]

SimpleCrossover creates a new roster by selecting only the common assignments of both parents.

3.4.3.2 Mutation operator

Similar to the TSP, a mutation operator often returns a solution that is worse than the original solution. It becomes useful to move the search from an optimum to another region of the problem solution space. Mutation operators have been successful at altering a roster when it is encoded as permutations [68].

The problem domain has only one mutation operator. Shifts are unassigned randomly to return a feasible roster. This number is proportional to the parameter *IntensityOfMutation* and it is computed by the formulae $IntensityOfMutation * 80.0D$ [83] (see expression 3.102);

$$UnassignedShiftsMutation : Solution \mapsto Solution \tag{3.102}$$

3.4.3.3 Local Search operators

These neighbourhood operators have been introduced by [49] for the nurse rostering problem. In our problem domain, we have adopted the five local searches well-documented by Curtois et al. [83]. All of these operators can be described as hill-climbers. They either lower the penalty cost for a roster or reverses the changes. All these operators are given in expressions [3.103 - 3.107] and described in details below.

$$NewSwapLocalSearch : Solution \mapsto Solution \tag{3.103}$$

$$HorizontalLocalSearch : Solution \mapsto Solution \tag{3.104}$$

$$VerticalSwapLocalSearch : Solution \mapsto Solution \tag{3.105}$$

$$VariableDepthLocalSearch : Solution \mapsto Solution \tag{3.106}$$

$$GreedyVariableDepthLocalSearch : Solution \mapsto Solution \tag{3.107}$$

HorizontalSwapLocalSearch repetitively swaps some blocks of adjacent days during the scheduling period. Only improving swaps are kept (i.e. the penalty for a nurse is reduced). Other swaps are reversed. Figure 3.18 provides an example.

VerticaSwapLocalSearch swaps repetitively some blocks of shifts between employees (see figure 3.19). Swaps deteriorating the roster are reversed; this occurs when the penalty for a day increases.

NewSwapLocalSearch may delete an existing shift or move a block of adjacent days (see figure 3.17). If the penalty of a nurse is reduced, then the move is kept. Otherwise, the changes are reversed. These steps are repeated while some new moves exist in the roster.

VariableDepthLocalSearch applies an ejection chain by alternating a deletion of an existing shift and moving a block of adjacent days. This technique was introduced by Burke et al. [49] and also by Yagiura with a job scheduling problem [353]. Only changes that lower the penalty for a nurse is kept. The length of time is limited to $Depth \times 5seconds$ [83].

GreedyVariableDepthLocalSearch is also a variant the *variable depth* technique introduced by [49]. It extends the *"VariableDepthLocalSearch"* operator with two features. First, a greedy algorithm generates an entire pattern of work for a nurse. Secondly, the weekend objectives must be satisfied. The length of time is limited to $Depth \times 5seconds$ [83].

Figure 3.17: *New swap* techniques used by the *NewSwaprLocalSearch* [83]

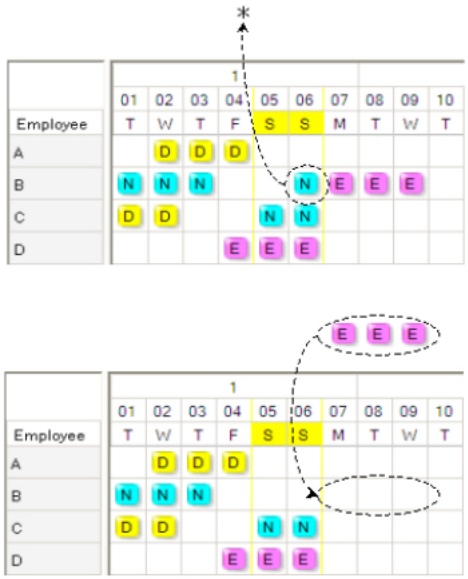


Figure 3.18: Horizontal swap used by the *HorizontalSwapLocalSearch* [83]

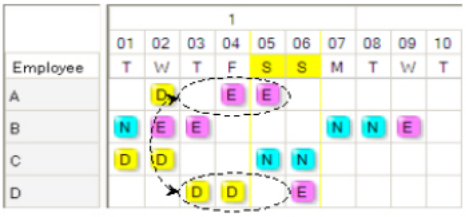
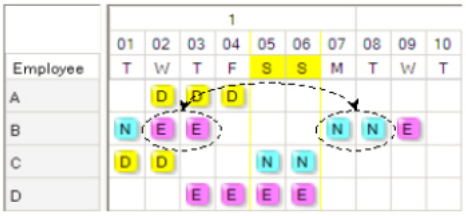


Figure 3.19: Vertical swap used by the *VerticalSwapLocalSearch* [83]



3.4.3.4 Ruin-and-Recreate operators

Ruin-and-Recreate operators first remove at least one shift of a roster causing the penalty cost to increase (i.e the *"Ruin"* phase). Then the *"Recreate"* phase attempt to repair the roster. Three ruin-and-recreate operators are included in our problem domain (see [3.108-3.110]).

$$SimpleGreedyRuinRecreate : Solution \mapsto Solution \tag{3.108}$$

$$SmallGreedyRuinAndRecreate : Solution \mapsto Solution \tag{3.109}$$

$$LargeGreedyRuinAndRecreate : Solution \mapsto Solution \tag{3.110}$$

The behaviour of these operators share a similar feature; the three of them repair a *ruined* roster by assigning the shift to a nurse that has the least increase in their penalty cost. For each operator, the number of un-assigned shifts varies from 1 schedule too much [48, 83]. These variations are discussed below.

SimpleGreedyRuinRecreate brings a small disruption by removing 1 schedule [83].

SmallGreedyRuinRecreate un-assigns all the shift for one or more randomly selected nurses. The number of nurses is calculated using the formulae given by Curtois et al [83]; $x = round(Intensity * 4) + 2$.

LargeGreedyRuinRecreate brings the largest level of disruption. The number of nurses is calculated using the expression $x = round(Intensity * NoOfNurses)$

3.4.4 Summary

The encoding scheme of the nurse rostering problem domain is the most complex of the three problems. It has many constraints that either lower or increase the problem fitness value. When these restrictions are not met, then a penalty is added to the cost.

Tuples indicate whether a nurse has been scheduled to work a shift on a specific day. The operators listed in table 3.5 changes the values of these tuples. Unlike the other two problem domains, the problem parameters needed to split into three categories; instance, constraints and operators. This classification helps in understanding their purpose and their effect on the problem domain.

Table 3.5: Nurse rostering operators with their opcode and the number of evaluations used.

OpCode	Operator(s)	Number of evaluations
0	newSwapLocalSearch()	λ
1	HorizontalSwapLocalSearch()	λ
2	VerticalSwapLocalSearch()	λ
3	VariableDepthLocalSearch()	λ
4	GreedyVariableDepthLocalSearch()	λ
5	SimpleGreedyRuinRecreate()	λ
6	SmallGreedyRuinRecreate()	λ
7	LargeGreedyRuinRecreate()	λ
8	MultiEventCrossover()	λ
9	ScatterSearchCrossover ()	λ
10	SimpleCrossover()	λ
12	UnassignedShiftMutation()	λ

3.5. Discussion and conclusion

Each problem represents its solutions with a distinct encoding scheme. Well-known data structures such as bitstrings, undirected weighted graphs, or triples rely on specific operators manipulating the data encoded in a solution. Various constraints may exist in the form of black-box optimisation, feasible and infeasible solutions or a set of preferences. It becomes more challenging to find suitable solutions when the number of these constraints increases. Any forms of algorithm optimisation should be able to withstand such changes too.

We have introduced three different problems regarding a problem statement, encoding scheme, specific parameters and operators. These three problems are not only hard to solve but also have unique features. The traveling salesman and the nurse rostering scheduling have not only real-life instances available, but they also are two NP-hard problems. The remaining problem (i.e. the mimicry problem) may be uncomplicated, but it can become very hard to solve when the number of bits increases. In the literature, instances often focus on less than 1,000 bits.

We will be generating some metaheuristics that find solutions for these three problems. A graph-based genetic programming uses the evolution to sample algorithms of varying length. Population and problem-specific operators are passed to the genetic programming as a function set. When iterations are evolved too, a stopping criterion is also given. Our methods are inspired by the model discussed in section 2.4.4 and discussed in our next section.

Chapter 4. Graph-Based GP

Contents

4.1 Review of graph-based genetic programming 88
4.1.1 Parallel distributed genetic programming 89
4.1.2 Linear-graph genetic programming 90
4.1.3 Graph structure program evolution 92
4.1.4 Parallel Algorithm Discovery and Orchestration 93
4.1.5 Cartesian Genetic Programming 94
4.1.6 Implicit-context CGP 97
4.1.7 Adaptive Cartesian Harmony Search 98
4.1.8 Discussion 99
4.2 CGP hyper-heuristics 101
4.2.1 Cartesian Genetic Programming 101
4.2.2 Iterative Cartesian Genetic Programming 104
4.2.3 Autoconstructive Cartesian Genetic Programming 108
4.3 Conclusion 117

Graphs represent a pair wise relationship between two objects. Euler introduced the concept of connecting vertices with some edges in 1736. Approximately a century later, K.G.C Von Staudt mentions for the first time the idea of trees; an undirected and acyclic graph [64]. Trees are suitable for encoding some hierarchical relationship between data; they have no cycle.

In 1952, the first application of a tree structure in computer science was to analyse mathematical expressions. Operating systems also organise files on some data storage with a tree. Each file is read recursively, and an element is only aware of its parent. Then, abstract syntax trees are generated by compilers to represent programmes, before creating some machine code. Later on in machine learning, mathematical expressions represented with trees considered as *"programs"*. The idea of evolving these trees with an EA gave birth to Beagle [103] and then to *"tree-based genetic programming"* [171].

Computer science has many uses of directed graphs. First, the connection between the elements of a computer network is represented by a directed graphs. Secondly, data flow and activity diagrams model the information through a system and how a process transformed the data. Thirdly, algorithms and processes can also express their sequences of instructions diagrammatically with some vertices and edges. Fourthly, McCabe et al. [215] have used again this idea to develop some metrics to measure the complexity of algorithms and computer programs. Lastly, in selective hyper-heuristics, a *graph-based hyper-heuristics* often refers to an algorithm that selects some problem-specific operators altering the structure of a graph. Such graph can encode some problem solutions of scheduling problems [92, 58].

4.1. Review of graph-based genetic programming

A technique referred as *"graph-based genetic programming"* has attempted to transform tree-structure into a hybrid graph with the use of some interactive outputs. Some sub-trees become redundant during the interpretation process; some special functions can disactivate some branches and active others[107]. In the remaining of this thesis graph-based genetic programming refers to a form of genetic programming that encodes a program with a directed graph.

4.1.1 Parallel distributed genetic programming

A matrix of active and inactive nodes encodes a program. Figure 4.1 encodes the mathematical expression $max\,(x * y, 3 + x * y)$ and the inactive nodes are represented with a dot. Each node is assigned a physical address with a row and column. The address $(0, 0)$ is always given the output node [255, 256]. The other nodes can either be a terminal or a function. During the decoding process, only the latter has a displacement attached to its node.

A depth-first mechanism differentiates the active nodes from the inactive nodes. Algorithm 4.1 retrieves first all the nodes connected to a node. From the displacement of some function and output nodes, the algorithm recursively calls the procedure $DecodeDirectedGraph$ and stops when the three terminals $x, y, 3$ are reached and their value saved in a hash-table. Then the recursive call stack applies, in turn, each operator encoded in the function nodes. Table 4.1 displays the list encoding a label, coordinates, and some horizontal displacement for the next connected nodes.

Figure 4.1: An example of a program encoded in grid [255]

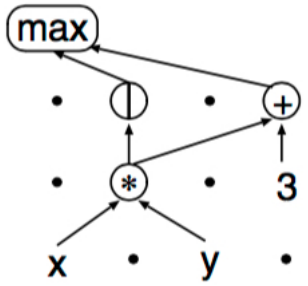


Table 4.1: a list encoding the label, the coordinates of the nodes, and the horizontal displacement for example given in figure 4.1

Label	**Coordinates**	**Displacement**
max	(0,0)	+1 +3
I	(0,1)	0
+	(3,1)	-2, 0
*	(1,2)	-1 + 1
3	(3,2)	
x	(0,3)	
y	(2,3)	

Algorithm 4.1. A feedforward mechanism used to decode the program in a grid [255]

```
1: procedure DECODEDIRECTEDGRAPH(NodeCoordinates, Values)
2:     NodesConnected ← DisplacementFromPreviousLayer()
3:     for currentNode ∈ NodesConnected do
4:         if CurrentNode == function then
5:             Values ← DecodeDirectedGraph(CurrentNode, Values)
6:             Values(NodeCoordinate) ← applyOperator(CurrentNode, Values)
7:         else if NodeType(NodeCoordinate) == Terminal then
8:             Values(NodeCoordinate) ← ValueOfTerminal
9:         end if
10:    end for
11:    return Values
12: end procedure
```

A genetic algorithm evolves these grids. The genetic code passed from one generation to another includes active and inactive nodes. As a result, some inactive nodes can be activated by the genetic operators at a later stage. One crossover swaps sub-graphs with some inactive nodes. Mutation operators can either activate some sub-graphs by mutating a link or insert a sub-graph to some terminals. Otherwise, new offspring are generated by swapping only active sub-graphs. This form of genetic programming has solved a variety of problems including regression, lawnmower, exclusive-or, even-parity, and finite state-automata induction problem.

4.1.2 Linear-graph genetic programming

Symbolic regression was also successfully solved by linear-graph genetic programming. A branching mechanism controls an execution path occurring between programs. A program node encodes a series of instructions (i.e a *linear program*), with some conditions that guide the connection to the next program node (see figure 4.3). In figure 4.2, four program nodes encode the mathematical expression $R_0 = (R_1 + 2)^2 - ((R_1 + 2)^2 \, mod \, 9)$.

An interpretation process starts at the root and executes the instructions on a set of registers. These values are assessed dynamically by the branching nodes, to decide the edge to use progress the program. In our example (see figure 4.2) the branching node should either connects to using the edges 0 or 1. In this instance, the minimum value of R_0 is 0 when $R_1 \in [-4...0]$. Therefore edge 0 is always going to be used.

Figure 4.2: An example of a linear gp individual as provided by [160]

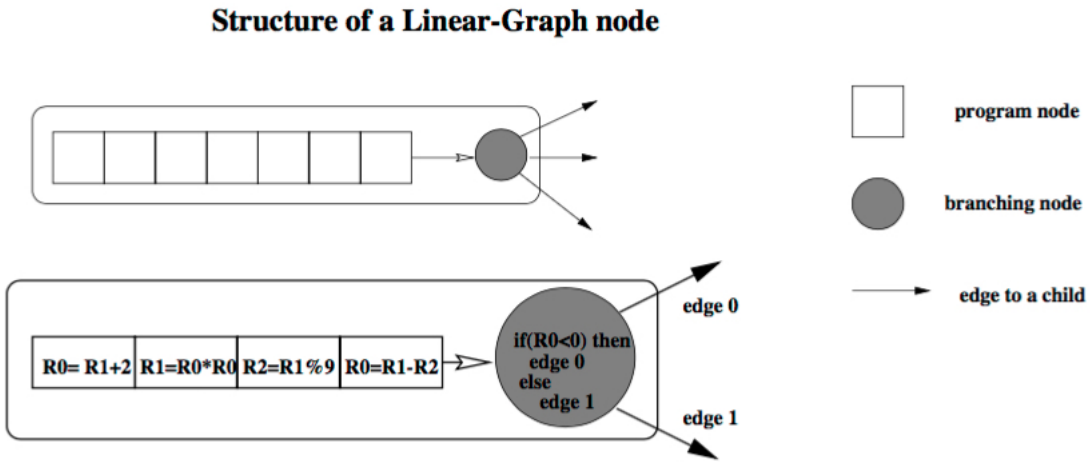
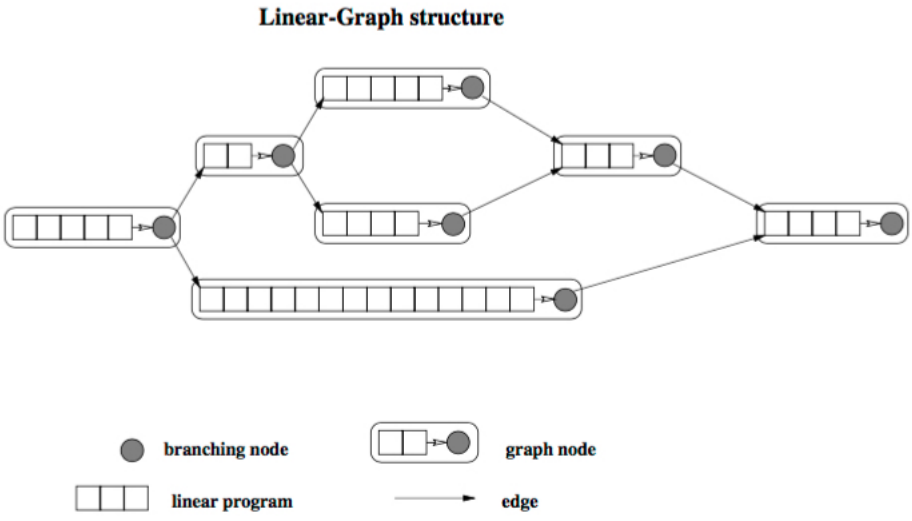


Figure 4.3: An example of a linear gp individual as provided by [160]

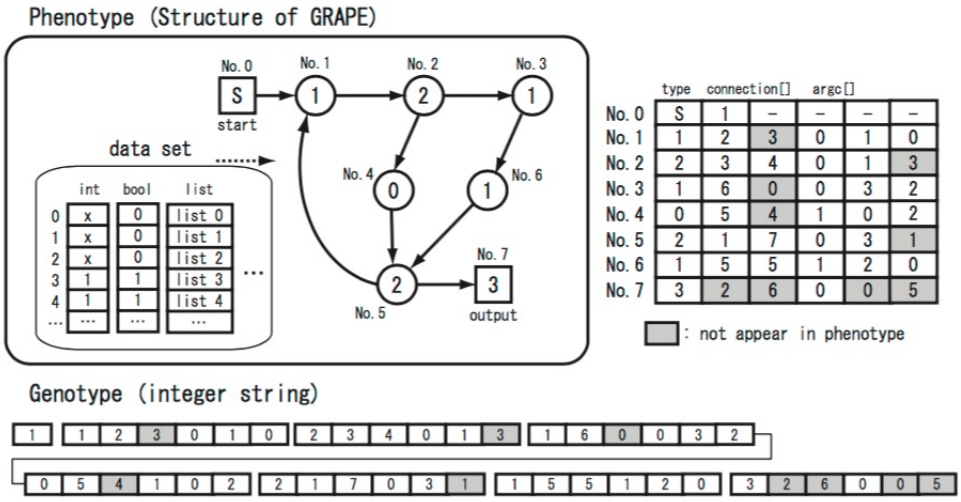


A genetic algorithm with a small population has been the most successful in evolving these type of programs [160]. A *"graph crossover"* exchanges sub-graphs from both parents, to form two new offsprings. A *"linear crossover"* exchanges equally sized-segments between vertices. A mutation operator can alter an element of the linear program, a branching function or the number of outgoing edges.

4.1.3 Graph structure program evolution

Graph-structure program evolution (GRAPE) models the flow of the data over a program. It emulates the registers in a microprocessor where the operations and addresses of the values are stored (see figure 4.4). The operators encoded in the nodes alter the variables and may use the constants stored in this dataset. These nodes are arbitrarily connected to each other. Not all the information of a node is active; a *node type* activates or deactivates some information stored in a fixed length string of integer values. In figure 4.4 the start node and the output node have the least information active. The other nodes may have some arguments or connection greyed out, as shown in figure 4.4.

Figure 4.4: An GRAPE program with its data set as given by [282]



The sequence of operations is defined by a feed-forward mechanism. Algorithm 4.2 establishes the nodes connected to the output and their order of execution. Then the sequence is executed and applied to the data. A genetic algorithm with uniform crossover and mutation evolves successfully programs that solve factorial, exponential equations, Fibonacci numbers and reversing a list [282, 284, 285].

Algorithm 4.2. A feedforward mechanism used to decode the program in a grid [282]

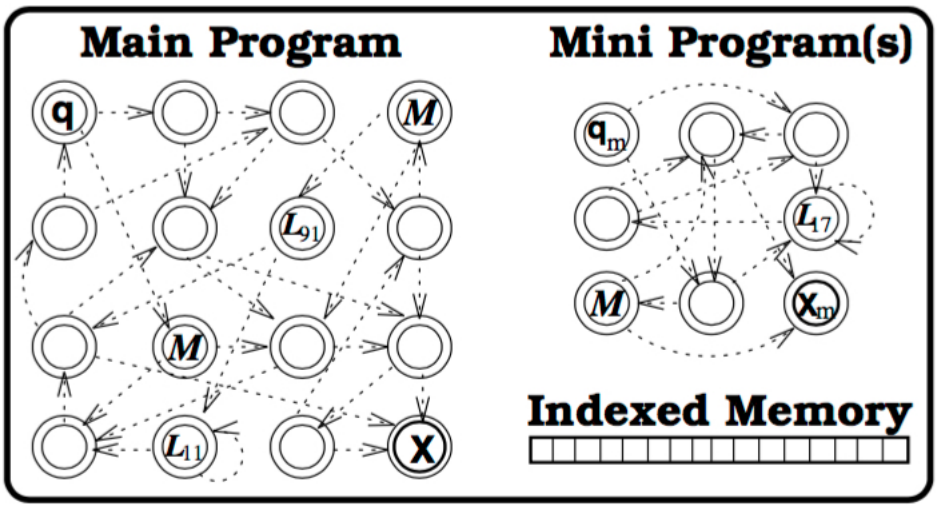
```
1: procedure DECODEDIRECTEDGRAPH
2:     Values(StartNode) ← resetValue()
3:     NodesConnected ← getNodesConnectedToTheOutput()
4:     for currentNode ∈ NodesConnected do
5:         Values(CurrentNode) ← applyOperator(CurrentNode)
6:     end for
7:     return Values(CurrentNode)
8: end procedure
```

4.1.4 Parallel Algorithm Discovery and Orchestration

PADO evolves some programs with an evolutionary strategy; these programs should find some suitable solutions for some challenging vision problems. Real-life objects should be detected in high resolution, noisy images of real-world objects [309, 309, 310].

Two arbitrary graphs encode two programs (i.e. the "main program" and the "mini program"). This technique considers programs as some code expressed in an imperative programming language. A path between the node **q** and the node **X** represents a program. A branch-decision function decides to which node to move to. This function would rely on the previous state of the program and the memory (see figure 4.5).

Figure 4.5: A PADO example program [309]



A mini program can be attached to any main programs; the node **M** can recursively call this sub-program. However, a fixed time prevents each program to run indefinitely. Finally the nodes L_{91} and L_{17} call a subroutine from a given library of programs.

A simple index memory can store some integer values. Those are used during the execution of a program. Although in theory those could be extended to any other data type and object, those had yet to be implemented in PADO.

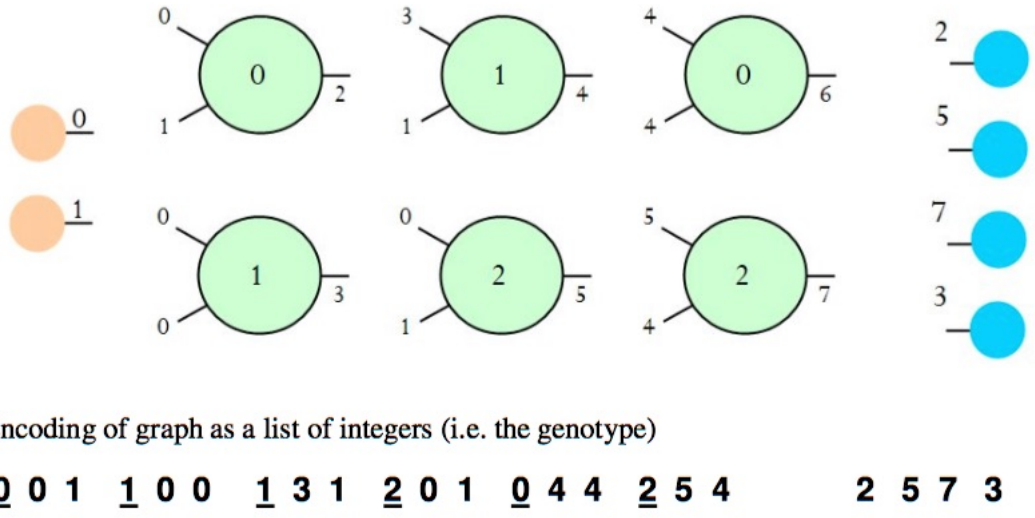
This complex graph-based genetic programming implements many features of a simple programming language. The function set contains a well-developed set of operators including reading and writes from an index memory and some mathematical operators. Also, many constraints have been implemented to ensure the programs to stop after a fixed execution time. Some others constrain the language primitives to a particular type of problems. It has yet to be extended to other NP-hard problems.

4.1.5 Cartesian Genetic Programming

Miller et al. [225] developed CGP (CGP) in 1999-2000. In its classic form, it uses a very simple integer address-based genetic representation of a program in the form of a directed graph. CGP represents a program using a grid of nodes; each node can be addressed using the Cartesian coordinates as addresses (i.e. a row and a column).

A string of integers encodes some programs. In figure 4.6, the program has two inputs (i.e. the orange circle labelled 0 and 1) that are connected mainly to the nodes of the first and second column. The program data inputs are given the absolute data addresses 0 to $n-1$, where $n-1$ is the number of program inputs. The number of nodes (i.e. length) is fixed. Each can connect to a previous node or a program input. Node inputs become restricted by a number of nodes they can link back.

Figure 4.6: A graphical representation of a CGP graph [226]



Each node contains a function; those are underlined in the list of integers and listed in a function "look-up table". The remaining node genes state where the node gets its data from and models the edges. For example, node 7 connects to the nodes 4 and 5 in figure 4.6. In classic CGP nodes either links to a previous node or a program input.

During the decoding process, only the nodes connected to an output are considered to be active. The remaining ones become inactive (i.e., node 6 in figure 4.6). Our example has four outputs (represented in blue in figure 4.6) that determine the sequences of operators for four different programs. Output no 2 points to node 2 creating a concise program with the operator 0 to execute. A longer program encodes the sequence of operators $1, 5, 4, 2$ (see the nodes numbered $3, 4, 5$ and 7). Those were interpreted using a feed-forward mechanism given in algorithm 4.3. This process identifies first the active nodes then executed them from left to right.

Active and inactive genes are passed from one generation to another. Inactive nodes can either be activated when the output points to a different node or a node input are mutated.

An evolutionary strategy can explore a wide distribution of offspring (see algorithm 4.4) [222]. An initial population of CGP graphs is randomly generated and evaluated before the best CGP-graph is promoted (see lines [1-2]). The remaining lines repetitively mutate the best CGP graph (i.e. μ) to produce and evaluate new offspring. A point mutation can randomly change the genes of coding and non-coding nodes.

The purpose of a function referred as $Promote$ has twofold. First, changes in inactive nodes are passed from one generation to another when the fitness of an offspring remains the same as the parent. Secondly, it replaces the parent μ with any CGP-graphs with a better fitness. Therefore, ineffective problem solvers can be tested, but do not survive more than one generation.

Algorithm 4.3. A feedforward mechanism used to decode the program in a grid [225]

```
1: procedure DECODEDIRECTEDGRAPH(OutputNo)
2:     NodesConnected ← IdentifyNodesConnectedToAnOutput(OutputNo)
3:     for currentNode ∈ NodesConnected do
4:         Values(CurrentNode) ← applyOperator(CurrentNode)
5:     end for
6:     return Values(CurrentNode)
7: end procedure
```

Algorithm 4.4. The $(\mu + \lambda)$ evolutionary strategy [225], where μ represents number of the parent population; it is often set to 1, but can have a greater size. $lambda$ is the number of the offspring.

```
1: CGP_offspring ← RandomlyGenerateIdividual(μ + λ)
2: CGP_parent ← Promote(CGP_offspring)
3: while Not solutionFound() or generation < Limit do
4:     for i ∈ [1..λ] do
5:         CGP_offspring[i] ← Mutate(CGP_parent)
6:         CGP_offspring[i] ← Evaluate(CGP_offspring[i])
7:     end for
8:     CGP_parent ← Promote(CGP_offspring)
9: end while
```

CGP has solved symbolic regression, lawnmover, if-and-only-if, classification problems [335, 334, 135, 99]. It has been successful in images filtering [134, 183]. Neural networks have also been evolved [165, 166, 317]. In addition, CGP has been very productive in synthesis of circuits [327, 328, 223, 224, 329, 164, 136, 280, 149, 210]

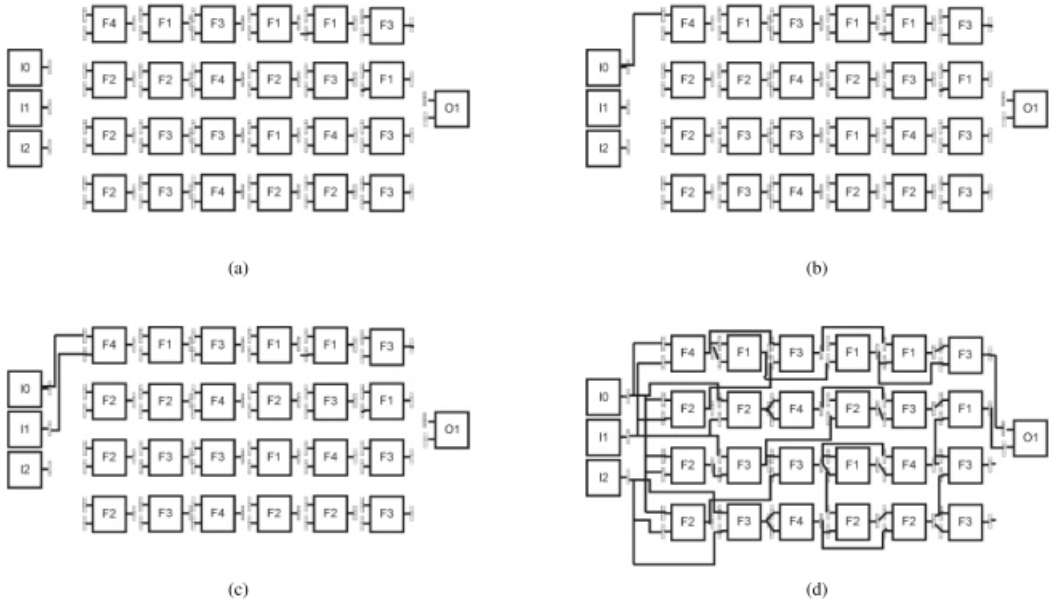
Various features have extended the classic CGP. In section 4.1.6, an implicit context has been added; the modified technique has been used in the diagnosis of some degenerative diseases. A coevolution mechanism has improved the design of circuits [149]. New genetic operators have been studied to explore new methods to improve CGP [218, 114]. Modules have been encoded in CGP graphs with some success [333, 336, 335, 334] and [135]. Finally, self-modifying CGP has added some functions that can modify an encoded program to refine the sequences of operators that solve the Fibonacci numbers, squares, regression, summing and parity [137, 139].

4.1.6 Implicit-context CGP

Implicit-context Cartesian genetic programming (ICGP) constructs some directed acyclic graphs by simulating substrate binding. In biology, molecules and enzyme bind together with to complete a chemical reaction. This process relies on both elements having a region that can fit together like two lego bricks [259].

ICGP relies on functionality profiles to filter inappropriate variations, to simulate the active site used by a substrate and an enzyme to bind. *"Formally, a functionality profile is a vector in a n r-dimensional space where each dimension corresponds to a function or terminal. This vector describes the relative occurrence of each function and terminal, weighted by depth, within an expression. In effect, it provides a means of representing and comparing (through vector difference) the functional behaviour of an expression"* [293]. A graph is interpreted bottom-up, by connecting a node with a previous node that matches its functionality profile. For example, in figure 4.7, the node with the Cartesian address $(1, 1)$ can bind with the first and second input[293, 61].

Figure 4.7: An example of how a ICGP graph is interpreted as provided by [293]

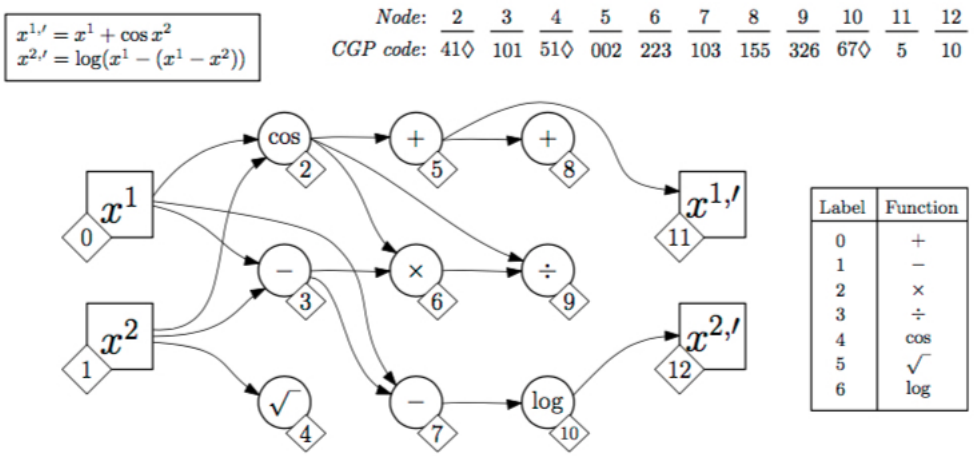


ICCGP has extended CGP, by representing some grids of nodes with a string of integers. Those are evolved with a genetic algorithm, instead of an evolutionary strategy; a uniform crossover and mutation produce some offspring. The primary application of an ICCGP has been in medical assessment of diagnosis of Alzheimer [178] as well as some classifiers [293, 61].

4.1.7 Adaptive Cartesian Harmony Search

During the completion of this thesis, a new extension of CGP has been used to evolve some classifiers. The datasets include some chemical analysis of plants, morphological features of plants, signals recorded from high-frequency antennas, and also engines sounds. *Adaptive Cartesian Harmony Search* (ACHS) evolves some CGP graph (see figure 4.8) with Harmony search [99]. This algorithm is very similar to the evolutionary strategy used by Miller et al. [225]. This observation is not surprising, as a Harmony search is a special case of an Evolutionary Strategy [248, 259]. This framework also estimates the predictive capability of intermediate solutions, to improve convergence.

Figure 4.8: An example of how a CGP graphs provided by [99]



4.1.8 Discussion

GP is more than a tree encoding programs that are evolved with a genetic algorithm. The research community is increasingly using a graph-based form. CGP has been one of the first technique developed in the late 1990s; other graph-based genetic programming techniques have yet to be applied to a wider range of problems.

The diversification of the nodes size was necessary to encode various information. Each technique has a unique purpose, which has focused on solving specific problems. The methods introduced in sections [4.1.1 - 4.1.4] mentions some specific areas of application, with the exception of symbolic regression.

CGP has adapted to various applications using an offline and online method of learning. The information held in the nodes has proven to be highly flexible, to add modules as well as mixed-data type variables have been encoded. An increasing number of researchers are interested in hybridising the technique to improve it and adapts to the needs of various applications. It is a good sign of the real potential of CGP.

Nonetheless, it is challenging to compare the real general performance of these techniques with another graph-based genetic programming. Very few literature has compared these methods under the same parameters applied to the problem domain. This issue was also raised by Poli et al. [258] with genetic programming in general. It is worth noting, that [221, 335] have found that CGP could find better solutions than a tree-based GP for Boolean and the lawn mowing problems.

CGP and PADO are often evolved by an evolution strategy with a small population of individuals. Other graph-based GP have used a genetic algorithm, with a larger population. These large populations may use a lot of resources, but more importantly, they have adapted their algorithm encoding scheme. For example, PDGP and LGP resemble the structure of a tree with a root as a starting point. The crossover has been adapted to exchange ”sub-graphs” instead of ”sub-trees”. Some crossover and mutation operators have specialised in mutating the content of the node or activate some sub-graphs. *ICCGP* and *GRAPE* have both adopted a uniform crossover and mutation; both of these techniques encodes directed graph with a bit string.

It is worth noting, a comparison of tree-based, graph-based, stack-based and grammatical genetic programming has been compared with the same method of evolution. The results and discussions have found the tree-based genetic programming was the best hyper-heuristics [142]. Their chosen method was a genetic algorithm with a large population, which has been proven very effective in tree-based GP. It would be interesting to repeat these experiments with an evolution strategy, a small population and a more significant nodes budget.

The reasons why such a simple evolutionary strategy works well is primarily due to the presence of non-coding genes, and the $1+4$ strategy cannot decrease the algorithm fitness, improving the quality of the algorithms. This phenomenon is also predominant in PDGP, GRAPE, PADO and ICCGP. This idea was introduced to tree-based genetic programming by [107] to attempt to transform a tree into a graph. Otherwise, tree-based genetic programming only stores coding genes.

CGP is mostly implemented with a point mutation (also referred as neutral mutation), but Goldman et al. [114] has found useful to mutate genes until one active gene is altered. Originally empirical studies completed by Miller [221] has found that recombination does not seem to add anything; this confirms some observations made by [310] for PADO. As a result, a $(1 + 4)$ evolution strategy has been adopted with CGP. Nonetheless, crossover might be useful if there are multiple programs with independent fitness assessment [336, 337].

Graph-based genetic programming has a lower number of publications compared against tree-based genetic programming. CGP has grown steadily and has been adopted by many research communities. We would hope in the future that may be some new forms may arise.

4.2. CGP hyper-heuristics

4.2.1 Cartesian Genetic Programming

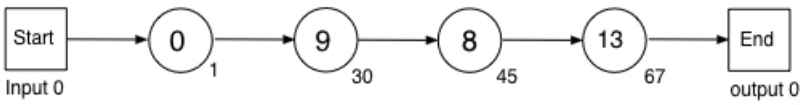
We have chosen a $(1 + 1)$ Evolutionary Strategy (see Algorithm 4.4) to search some algorithm search spaces. One-dimensional CGP graphs have a start, an end and a workflow; flow charts are represented and executed sequentially.

The number of coding nodes or operations can be anything from zero to the maximum number of nodes defined in a CGP graph. Only the nodes connected to an output node are considered to be part of an algorithm; the remaining nodes become inactive (i.e. non-coding genes).

4.2.1.1 Decoding the CGP graph

Figure 4.9 only shows all the active nodes connected to an output node. These have the indexes 1, 30, 45, and 67 and encode the sequence of instructions *"0-9-8-13"*; this CGP graph represents the *TSP Solver A* (i.e. algorithm A.19 that can be found in section 9.2).

Figure 4.9: A solver expressed with its active nodes



Templates can specify the "initialisation" step, the "update" step and the termination criterion of an iteration, leaving *the body of a loop* being only influenced by the evolution [174]. For example, algorithms 4.5 and 4.6 illustrate how a template can be adapted to a population-based non-deterministic algorithm, but other applications may use different templates.

Algorithm 4.5. A feedforward mechanism used to decode a CGP graph

```
1: procedure DECODEDIRECTEDACYCLICGRAPH(OutputNo)
2:     NodesConnected ← IdentifyNodesConnectedToAnOutput(OutputNo)
3:     CurrentNode ← GotoFirstNodeOfGraph
4:     numEvals ← 0
5:     while Not LastNodeOfGraph(CurrentNode) do
6:         Values(CurrentNode) ← applyOperator(CurrentNode)
7:         CurrentNode ← GoToNextNode()
8:         numEvals ← numEvals + 1
9:     end while
10:    return NumEvals
11: end procedure
```

Algorithm 4.6. An general algorithm template of a population-based metaheuristics

```
1:  function FINDSOLUTION(problemParam, μ, λ)
2:      p ← InitPopulation(problemParam, μ, λ)
3:      t ← SelectElitism(p)
4:      EvalCount ← 0
5:      while EvalCount ≤ MaxEvals or p.fitness > 0 do
6:          NumEvals ← 0
7:          NumEvals ← DecodeDirectedAcyclicGraph(OutputNo)
8:          EvalCount = EvalCount + NumEvals
9:      end while
10:     P ← ReplaceLeastFit(p,t)
11:     return Best(p)
12: end function
```

Lines 1-3 An initial population p is randomly generated and evaluated. Some individuals are also selected for reproduction and initialise a temporary population t.

Line 4 An "initialisation" step set the number of evaluations to 0.

Line 5 The loop is guaranteed to end. The condition terminates the loop when no more evaluations are available, or a known optimum solution is found.

Line 7 The evolved sequence of operators is applied to the populations of individuals t and p; those were introduced in section 3.1.1. The function *DecodeDirectedAcyclicGraph()* decodes the CGP graphs and apply the operators sequential. This subroutine also counts and returns the number of evaluations used .

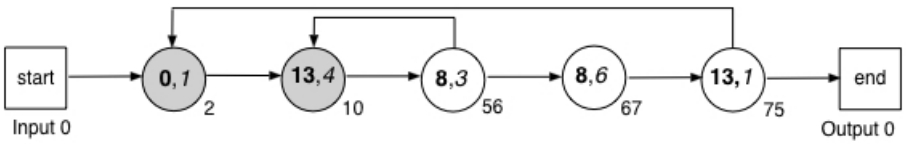
Line 8 The update step increases the number of evaluations used by the evolved sequence of instructions.

Lines 10-11 The best problem solution found by the metaheuristic is saved in p and returned when the metaheuristics stops.

4.2.2 Iterative Cartesian Genetic Programming

Cycles are formed with iterative CGP so that loops can be altered by the evolution and terminates without any hard limits. Directed *"cyclic"* graphs can now encode a stopping criterion, an iterative update step and the body of a loop; all these elements are made susceptible to the evolution. Consequently, each node now contains two new genes; we name them *Branching and Condition*. Figure 4.10 represents the iterative CGP graph of algorithm *TSP Solver J* given in section 9.2; this figure omits many branching connections and non-coding genes for clarity.

Figure 4.10: A solver expressed with its active nodes



Feed-forward connections are standard feed-forward CGP connection genes.

Branching connections can point to a previous node, a program input, or itself; in this case a node is referred as a *process node* and shown in white in figure 4.10. When the branching gene points to a suitable subsequent node a cycle is formed; then the node becomes a *decision node*. In figure 4.10 those are shaded in grey.

A level-back parameter determines how many nodes (before and after) a branching gene can connect to define the boundaries of the body of a loop and split a CGP graph into smaller sub-sequences.

Function genes are as in standard CGP and encode a primitive operation. In figure 4.10, these genes are formatted in bold.

Condition genes represent the stopping criteria of loops. A condition look-up table provides a set of Boolean primitives; these indicate whether a loop exits (and control subsequently moves to the next node following the last loop node) or continues to execute the next node inside the loop. In figure 4.10, these genes are formatted in italic font.

4.2.2.1 Decoding iterative CGP graphs

The clear distinction between "decision" and "process" nodes allows a decoding process to repetitively apply a sub-sequence of operators sequentially, under a distinct condition. The first and last problem-specific operation of a sub-sequence is determined by (1) the function gene of decision node and (2) the function gene encoded in the node pointed to by the branching gene of the decision node.

Algorithm 4.7 continues to decode process nodes the same way as CGP. The decision nodes can then guide the execution to an algorithm to the first operation of the body a loop or the next operation.

Algorithm 4.7. A feedforward mechanism used to decode an iterative CGP graph

```
1:  procedure DecodeDirectedCyclicGraph(OutputNo)
2:      NodesConnected ← IdentifyNodesConnectedToAnOutput(OutputNo)
3:      OrderedNodesConnected ← IdentifyBranchingNodes(NodesConnected)
4:      CurrentNode ← GotoFirstNodeOfGraph
5:      NumEvals ← 0
6:      while Not LastNodeOfGraph(CurrentNode) do
7:          if TypeOf(CurrentNode) = processNode then
8:              Values(CurrentNode) ← applyOperator(CurrentNode)
9:              NumEvals ← NumEvals + 1
10:         end if
11:         if TypeOf(CurrentNode) = DecisionNode then
12:             if IsTerminationCriteriaMet(CurrentNode) then
13:                 CurrentNode ← GoToEndOfTheLoop()
14:             else
15:                 Values(CurrentNode) ← applyOperator(CurrentNode)
16:                 NumEvals ← NumEvals + 1
17:             end if
18:         end if
19:         CurrentNode ← GoToNextNode()
20:     end while
21:     return NumEvals
22: end procedure
```

Lines 2-4 All the active nodes are identified by working backwards from an output node. In figure 4.10 the output 0 is used. The decision nodes are placed so that branching can happen during the decoding process; the decision node index is inserted after the last active node of a subsequence (i.e. the body of a loop). For example, in figure 4.10 all the active nodes are executed in the following order: $2, 10, 56, 10, 67, 75, 2$. The decision nodes index(i.e. 2 and 10) are repeated to indicate the starts and then end of the body of the nested loops.

Lines 5-17 The sequence of active nodes iteratively applies the operators and termination criteria.

Lines 6-8 The operator encoded in some *process nodes* are applied.

Lines 9-15 The termination criterion encoded in some *decision nodes* are decoded and applied. When the termination criteria are met, the execution of a given loop is stopped. The current node becomes the last node of the loop. Otherwise, the operator is applied in the same manner as a process node.

Line 16 The decoding process moves to next node.

As a result, the template has now become less restrictive. A population-based non-deterministic algorithm uses again the function *DecodeDirectedCyclicGraph()* but no loop is expressed (see algorithm 4.8). The first three lines and the last instruction of algorithm 4.8 are part of the template. An initial population is generated randomly before being evaluated, at least one individual is selected for reproduction.

Algorithm 4.8. A minimalistic template of a hybrid metaheuristic

```
1: function FINDSOLUTION(ProblemParam,μ,λ)
2:     p ← InitPopulation(ProblemParam,μ,λ)
3:     t ← SelectElitism(p)
4:     EvalCount ← 0
5:     NumEvals ← DecodeDirectedCyclicGraph(OutputNo)
6:     EvalCount ← NumEvals
7:     p ← replaceLeastFit(t, p)
8:     return Best(p)
9: end function
```

Similarly to algorithm 4.6, the best solution found the metaheuristic is saved in p and returned when the metaheuristics have stopped its run. Those are formatted in a black colour and normal font. The remaining instruction decodes the Iterative-CGP phenotype to an iterative algorithm.

4.2.2.2 Evolution of iterative CGP graphs

We use an $(1+1)$ Evolutionary Strategy again to search the algorithm space of iterative algorithms (see algorithm 4.4). The added branching and condition genes now need two basic grammatical rules to ensure that either only nested loops are created, or new iterations do not overlap.

The creation of an initial iterative CGP population and a point mutation operator rely on the following mechanisms.

1. When an iterative CGP graph does not encode any loops the value of any branching gene is free to point to any nodes and program inputs.
2. For any nodes inside an existing loop, their branching genes can only connect to a node with a higher index that is inside the current loop or any previous nodes and program inputs. In figure 4.10, the branching gene of nodes with an index greater than 2 can be valid if its value is lower than the index. It can also point to the right to a node with an index lower than 75.
3. For any nodes outside an existing loop, their branching genes can only connect to a node that is outside any existing sub-sequences. A valid value for the branching gene of node 1 can only point to the input or nodes greater than 75 or the program input in figure 4.10.

Genes continue to be randomly chosen. When a *"condition gene"* is selected, then a valid condition is randomly chosen from the condition look-up table. Also when a *"Branching gene"* is chosen, then a valid address is randomly selected. This part of the mutation verifies that either only nested loops are created, or new loops do not overlap; the two aforementioned basic grammatical rules are applied in this genetic operator.

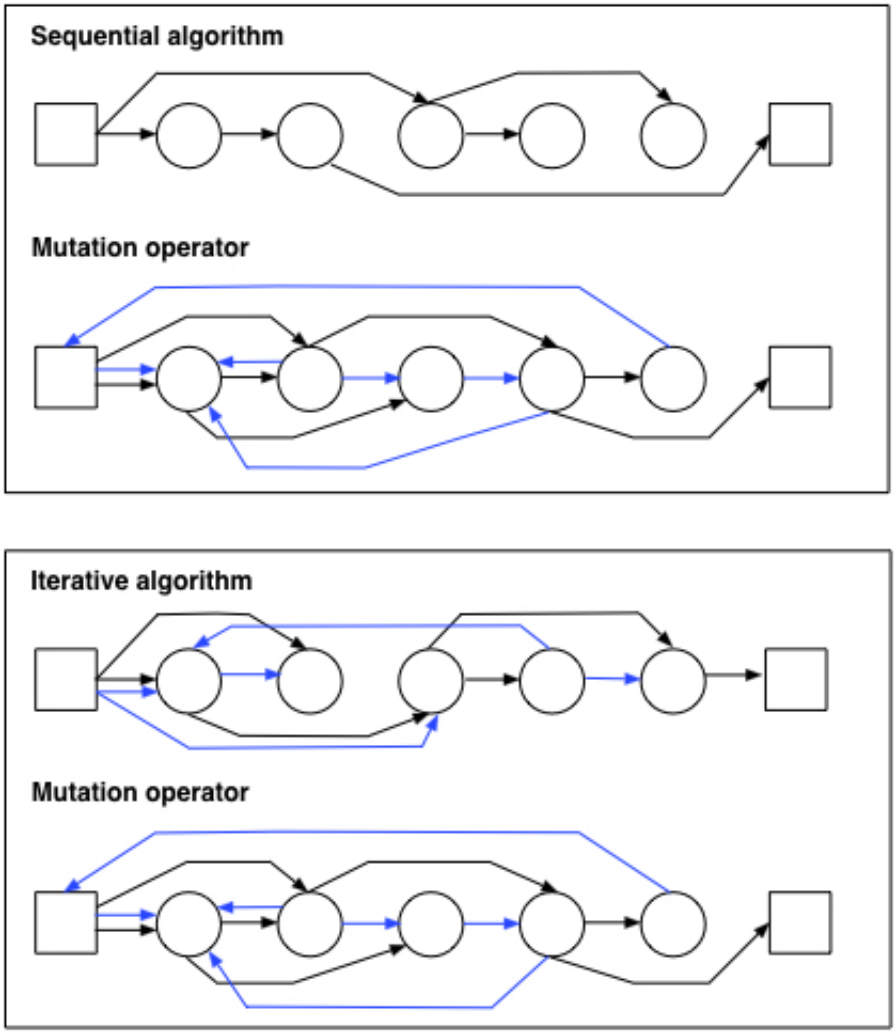
4.2.3 Autoconstructive Cartesian Genetic Programming

The mutation operators of the two previous hyper-heuristics discussed in sections 4.2.1 and 4.2.2 remain the same during the algorithm search. Autoconstructive CGP (Autoconstructive CGP) evolves algorithms and a hyper-heuristic reproductive mechanism; a sequence of operations that constructs the mutation should evolve during the algorithm search.

The equation $offspring' = offspring(reproductive\,operator)$ was introduced by Lee Spector in PushGP [302]. This autoconstructive form of genetic programming uses a tree to encode reproductive operators and stacks for the algorithms. To implement such ideas with Cartesian Genetic Programming, we have coupled the encoding scheme introduced in sections 4.2.1 and 4.2.2 with an iterative CGP graph. The latter stores a sequence of operations that represents a mutation operator (see Figure 4.11). Mutation operators are passed from one generation of algorithm individual to another. As result, the parent copies its reproductive mechanism unaltered to its offspring.

Some sequential and iterative algorithms, as well as the mutation operators, are decoded in the same manner than CGP and iterative CGP graph. An evolution strategy initialises, promotes and evaluates some problem-solvers using the same techniques as described in sections 4.2.1 and 4.2.2. The fitness value of an algorithm remains the fitness value of an autoconstructive CGP graph. Later in this section, we will discuss the evaluation process of a mutation operator.

Figure 4.11: : autoconstructive CGP graphs. The top individual encodes an algorithm with directed acyclic graph and the bottom individual with iterative CGP graph. Both individual encodes a mutation operators with an iterative CGP graph. All the branching genes are represented with blue arrows.



An evolutionary strategy co-evolves a population of algorithms and a population reproductive mechanisms the algorithm search. Algorithm 4.9 is general enough to evolve sequential or iterative algorithms. The first two steps initialise randomly and evaluate the individuals of a problem-solver population. The next line promotes the best of these algorithm offspring.

A similar process is then implemented for a population of mutation operators. The best reproductive mechanism is subsequently assigned to the newly promoted CGP graph (i.e. CGP_{parent}) (see lines 3-6 of algorithm 4.9.

New algorithms are then produced using an evolved mutation operator. These algorithm offspring are evaluated before being promoted; these steps are similar to the Evolutionary Strategy discussed in section 4.1.5 (see lines [7-8] and [18-21]).

Algorithm 4.9. The $(\mu+\lambda)$ evolutionary strategy [225] extended to co-evolve a hyper-heuristic reproductive operator and algorithms.

```
 1: CGP_offspring ← RandomlyGenerateIdividual(μ + λ)
 2: CGP_parent ← Promote(CGP_offspring)
 3: Mutation_Offspring ← InitialiseReproductiveOperators(μ_mutation, CGP_parent)
 4: Mutation_Parent ← PromoteReproductiveOperator(Mutation_Offspring)
 5: CGP_parent ← AssignMutationOperator(Mutation_Parent)
 6: NoGraphGenerated ← 0
 7: while not solutionFound() or generation < Limit do
 8:     for i ∈ [1..λ] do
 9:         if NoGraphGenerated = MaxGraphGenerated then
10:             NoGraphGenerated ← 0
11:             CGP_parent.Mutation ← EvalMutationOp(CGP_parent.Mutation)
12:             CGP_parent.Mutation ← EvolveMutationOp(CGP_parent)
13:         else
14:             NoGraphGenerated ← NoGraphGenerated + 1
15:         end if
16:         CGP_offspring[i] ← CGP_parent.Mutate()
17:         CGP_offspring[i] ← Evaluate(CGP_offspring[i])
18:     end for
19:     CGP_parent ← Promote(CGP_offspring)
20: end while
```

With this online learning hyper-heuristics, the reproductive mechanism is likely to change as the algorithm search progresses. A mutation operator must produce a certain number of algorithm offspring, before being evaluated and evolved. It is defined by an added parameter $MaxGraphGenerated$. Then the resulting reproductive mechanism is then assigned to the CGP parent (i.e. a sequential or iterative algorithm), and the algorithm search can resume. This is shown in line [9-16] of algorithm 4.9.

4.2.3.1 Evaluation of reproductive operators

An autoconstructive CGP individual acts as a host for a species of mutation operators. A reproductive mechanism does not only benefit from this "living environment", but also receives some information about its performance. A positive value rewards new metaheuristics that perform better than its parent. Otherwise a score of 0 is recorded in a list. Reproductive mechanisms that alter exclusively non-coding genes are penalised by the negative value (-1).

Once a certain number of algorithm offspring has been generated, the arithmetic mean of all these performances is computed by the function $EvaluateMutationOp()$. The resulting mutation fitness value can then be used to assess the quality of a reproductive mechanism.

4.2.3.2 Genetic improvement of reproductive operators

The function *EvolveMutationOp* can genetically improve the subspecies of hyper-heuristic reproductive operators (i.e the population $Mutation_{Offspring}$). Each type of reproductive mechanism attempts three times to evolve a new CGP offspring (see algorithm 4.10 lines [3 - 14]). New mutation operators are promoted if their performance is better than their parent. When a better improved reproductive operator is found then, it replaces the current CGP mutation operator.

The process terminates when a reproductive operator should perform better than the mutation used (at the current stage of the coevolution). After three attempts, the population of the three subspecies are reset. This mechanism offers another chance to find a new reproductive operator. A new operator is assigned to the CGP_{parent}, replacing the mutation operator encoded in the autoconstructive CGP graph, if it is likely to perform better.

Algorithm 4.10. This algorithms shows the steps used to generate a reproductive mechanism for sequential and iterative algorithms.

```
1: function EVOLVEMUTATIONOP(ByValue CGP_Parent)
2:     Attempt = 0
3:     while Attempt < 3 do
4:         for Parent ∈ [ActiveNodes,AnyNodes,Structure] do
5:             Offspring ← Mutate(Parent)
6:             Offspring ← EvaluateMutationOp(Offspring, CGP_Parent)
7:             Mutation_Offspring ← Promote(Offspring )
8:             if Parent is better than CGP_Parent.Mutation then
9:                 CGP_parent ← AssignMutationOperator(Parent)
10:                goto end
11:            end if
12:        end for
13:        Attempt ← Attempt + 1
14:    end while
15:    if Attempt = 3 then
16:        Mutation_Offspring ← InitialiseReproductiveOperators(μ_mutation, CGP_parent)
17:        Mutation_Parent ← PromoteReproductiveOperator(Mutation_Offspring)
18:        CGP_parent ← AssignMutationOperator(NewReproductiveOp)
19:    end if
20:    return CGP_parent.Mutation
21: end function
```

4.2.3.3 Function and termination set of reproductive operators

A comprehensive function set provides the operators that can change the genes of each node and the graph output; they are applied to the sequential or iterative algorithms. Some of these operators modify the coding genes common to both CGP hyper-heuristics (i.e. sequential and iterative). Expressions [4.1 - 4.5] either change a function, a feedforward connection of active and inactive nodes or the output of a CGP graph. It is worth noting, expressions 4.2, 4.4 and 4.5 are used in the point mutation discussed in section 4.1.5.

$$FlipFunctionOfActiveNode(ActiveNodeIndex) \quad (4.1)$$

$$FlipFunctionOfAnyNode(ANodeIndex) \quad (4.2)$$

$$FlipFeedForwardConnToActiveNode(ActiveNodeIndex) \quad (4.3)$$

$$FlipFeedForwardConnToAnyNode(ANodeIndex) \quad (4.4)$$

$$FlipAnOutput(OutputIndex) \quad (4.5)$$

FlipFunctionOfActiveNode() and FlipFunctionOfAnyNode() change the function genes of a randomly selected node. While *FlipFunctionOfActiveNode()* only selects active nodes, but *FlipFunctionOfAnyNode()* can choose any nodes from the graph.

FlipFeedforwardConnToAnActiveNode() and FlipFeedForwardConnToAnyNode() mutate one input forward of a randomly selected node. *FlipTheInputForward-OfANode()* can choose a node from the entire graph; the new input forward can point to any previous nodes or a graph input. On the other hand, *FlipTheInput-ForwardToAnActiveNode()* is restricted to select from the subset of active nodes in a graph. The new input can only points to a previous active node or a graph input.

FlipAnOutput() changes an output of a graph to a randomly selected node.

Some operations have specialised in altering the added genes of an iterative CGP node; those are a condition and a branching gene. The operations given in expressions [4.6 - 4.9] extends the function set so that iterative algorithms can be mutated with this online learning mechanism. Coding genes and non-coding genes can be mutated with an evolved mutation operator. A point mutation discussed in section 4.2.2 relies on the operators given in expressions 4.2, 4.4, 4.5, 4.7, and 4.9 to produce new CGP offspring.

$$FlipConditionOfActiveNode(ActiveNodeIndex) \quad (4.6)$$

$$FlipConditionOfAnyNode(ANodeIndex) \quad (4.7)$$

$$FlipBranchingGeneOfAnActiveNode(ActiveNodeIndex) \quad (4.8)$$

$$FlipBranchingGeneOfAnyNode(ANodeIndex) \quad (4.9)$$

$$(4.10)$$

FlipConditionOfActiveNode() andFlipConditionOfAnyNode() change the condition genes of a node randomly selected.*FlipConditionOfActiveNode()* is restricted to the active nodes of a CGP graph, but *FlipTheConditionOfANode()* can choose any node in the entire graph.

Both **FlipBranchingGenesToAnActiveNode() and FlipBranchinGeneANode()** mutate the branching gene of an iterative node with the grammar discussed in section 4.2.2. *FlipBranchingGenesToAnActiveNode()* changes coding genes to a valid active node. However, *FlipBranchingGenesOfANode()* is free to choose any nodes of a graphs and point to any suitable nodes.

The function set has also some other unusual operators. Those are expressed in expressions [4.11-4.16] can bring small changes or bring larger disruption to a graph. It is hoped the algorithm search space could be searched within a region or move to another part, with more control.

$$SwapFunctions() \quad (4.11)$$

$$ApplyAFunctionLocalSearch() \quad (4.12)$$

$$ApplyInputForwardLocalSearch() \quad (4.13)$$

$$ApplyConditionLocalSearch() \quad (4.14)$$

$$InitialiseActiveNode(ActiveNodeIndex) \quad (4.15)$$

$$InitialiseAnyNode(ANodeIndex) \quad (4.16)$$

SwapFunctions() randomly selects two active nodes and swap their function genes.

ApplyAFunctionLocalSearch() applies three times $FlipFunctionOfActiveNode()$ on the same active node. The function that brings the most beneficial changes to a CGP graph is kept. If no improvement to the algorithm fitness function occurs the changes are revoked.

ApplyConditionLocalSearch() applies three times the operator *FlipConditionOfActiveNode()* and keep the best changes that improves an iterative CGP graph. Otherwise, the change is revoked.

ApplyInputForwardLocalSearch() makes three attempts to change a randomly selected input forward of an active node; it uses again the operator FlipTheInputForwardToAnActiveNode()). Then the most favourable mutation flip is kept. If no improvement to the algorithm fitness function occurs the changes are revoked.

InitialiseActiveNode() and InitialiseAnyNode() change every gene of a randomly selected node. Only coding genes values are altered by *InitialiseActiveNode()*. However, *InitiaseANode()* can change of any nodes of a CGP graph. These two operators can operate on sequential and iterative algorithms.

The encoding scheme of the reproductive operator also relies on a condition set. Iterative CGP graphs offer more freedom to evolve the reproductive mechanism than directed acyclic graphs. Expressions [4.17 - 4.20] implements a ”for” loop. Each time the body of a loop is executed a counter is incremented by one. The remaining expressions (i.e. [4.21 - 4.23]) increments a counter each time a node has been altered instead. The loop stops when the correct proportion of nodes has been reached.

$$IsCounterLessThanTwo() \tag{4.17}$$

$$IsCounterLessThanFour() \tag{4.18}$$

$$IsCounterLessThanEight() \tag{4.19}$$

$$IsCounterLessThanTen() \tag{4.20}$$

$$HasLessThanATenthOfAGraph() \tag{4.21}$$

$$HasLessThanAQuarterOfAGraph() \tag{4.22}$$

$$HasLessThanAHalfOfAGraph() \tag{4.23}$$

Our list of operators and termination criteria that are used by the reproductive mechanism is undeniably large. We have therefore allocated these operators to a subspecies of generative mechanism that fit best with their purpose.

For example, the function set for the ”active-nodes” subspecies changes coding genes and the termination criteria brings only a few iterations (Tables 4.2 and 4.3). The small variations brought to the algorithms are likely to test and assess problem-solvers with similar operators in different orders. It is worth noting this type of reproductive operator may drive the algorithm search in a local optimum.

The function and condition set for the ”any-nodes” subspecies can also mutate some non-coding genes. As a result, the benefit of a neutral mutation can continue to be applied to the search. However, we would expect reproductive mechanism with a mixture of coding and non-coding genes being the more successful. Finally, the smallest function set has been given to the ”structure” subspecies. These three operators should bring the most disruption, moving the algorithm search to a new area of the algorithm search space. These function and condition sets are summarised in tables 4.2 and 4.3.

Table 4.2: This table summarises the function set of each subspecies of a reproductive mechanism. The operators formatted in italic are only applied to the iterative algorithm.

Active-Nodes	**Any-Nodes**	**Structure**
FlipFunctionOfActiveNode	FlipFunctionOfActiveNode	InitialiseActiveNode
FlipFeedForwardConnToActiveNode	FlipFeedForwardConnToActiveNode	InitialiseAnyNode
SwapFunctions	FlipFunctionOfAnyNode	FlipAnOutput
ApplyAFunctionLocalSearch	FlipFeedForwardConnToAnyNode	
ApplyInputForwardLocalSearch	*FlipConditionOfActiveNode*	
ApplyConditionLocalSearch	*FlipTheConditionOfANode*	
FlipConditionOfActiveNode	*FlipBranchingGeneOfActiveNode*	
	FlipBranchingGeneOfAnyNode	

Table 4.3: This table summarises the condition set of each subspecies of reproductive operators

Active-Nodes	**Any-Nodes**	**Structure**
IsCounterLessThanTwo	IsCounterLessThanFour	IsCounterLessThanTen
IsCounterLessThanFour	HasLessThanAQuarterOfAGraph	HasLessThanATenthOfAGraph
IsCounterLessThanEight	HasLessThanAHalfOfAGraph	

4.3. Conclusion

This chapter has reviewed quite comprehensively graph-based genetic programming techniques. Those have evolved directed acyclic graphs and directed graphs with a variety of EAs. CGP has been one of the first graph-based genetic programming techniques, and it remains still popular; this original technique is quite flexible to be extended for a different purpose.

Three CGP-based hyper-heuristics techniques have been described. Two of these techniques are some extensions from the original work from Miller et al. [221]. Our first extension has allowed the full evolution of iterations in algorithms; the technique can be used in a wider context than evolving metaheuristics. Our second extension is an online-learning hyper-heuristics that should evolve a reproductive mechanism during the algorithm search. Our next chapter will discuss the experiments we have conducted with these techniques to generate some problem-solvers for the three problem domain introduced in chapter 3.

Chapter 5. Evolving metaheuristics

Contents

5.1 Introduction 119

5.2 Learning objective function 120

5.3 Evolving the body of a loop 121

5.3.1 Validation 123

5.4 Iterative Cartesian Genetic Programming: the full evolution of loops 127

5.4.1 Validation of the learnt iterative metaheuristics 129

5.5 Discussion and conclusion 133

5.1. Introduction

Evolving the body of a loop was originally suggested by Koza et al. [174], when he raised the following question: *Is it possible to automate the decision about whether to employ the particular sequence of iterative steps in a computer program that is evolved by genetic programming to solve a problem?* [174].

Suitably expressive algorithms may never terminate or have over-long computations. When the algorithm design is automated, some forms of constraints prevent these unwanted occurrences as a practical necessity. Defining the elements that remain unchanged and the evolved part of a program can be achieved with some grammatical rules or templates [244, 199, 180, 344, 182, 285].

The evolution of some iterative or recursive structures has been made possible by relaxing some of these constraints. The evolution of some *"for loops"* is often restricted to a hard-coded and maximum number of times such constructs can be repeated [192, 343, 69, 191, 253, 6, 358, 253, 11]. Some iterative programs have also been generated with a form of graph-based genetic programming. The operation set often relies on arithmetical or boolean operators. At the time of writing, we are not aware of any direct applications to search problems. [282] .

The purpose of this chapter has therefore twofold. We investigate first the effectiveness of classic CGP in evolving the body of a loop. We then extend CGP exposing the iteration of a metaheuristic fully to the evolution. In the remainder of this thesis, the cluster *N8 HPC* [1] host all our experiments.

5.2. Learning objective function

The learning objective function we use in this chapter is given in Algorithm 5.1; its signature complies with the general definition provided in section 2.3.3 (i.e a function referred as *AlgEvaluation* in expression 2.18).

We penalise metaheuristics without any replacement operators with a very large algorithm fitness value (see line 3 of algorithm 5.1). This mechanism aims at decreasing the likelihood of such algorithms surviving to the next generation. Otherwise, some problem solutions are obtained (i.e., one for each given instances) and their arithmetic mean is returned.

[1] N8 HPC provided and funded by the N8 consortium and EPSRC (Grant No.EP/K000225/1), The Centre is coordinated by the Universities of Leeds and Manchester

Algorithm 5.1. Learning objective function

```
1: function ALGEVALUATION(anAlgorithm, Instances)
2:     if AnAlgorithm has no replacement operator then
3:         Fitness = ∞
4:     else
5:         for anInstance ∈ Instances do
6:             aResult ← RunAlg(anAlgorithm, anIntance, Runs = 1)
7:             Total ← Total + aResult
8:         end for
9:         Fitness ← Total / (Number of instances)
10:    end if
11:    return Fitness
12: end function
```

5.3. Evolving the body of a loop

We hope to generate some TSP solvers. Each generated metaheuristic is evaluated using the process described in algorithm 5.1. The three predetermined learning instances were chosen *pr299, pr439* and *rat783*; we hope they would assess suitably well the algorithm abilities. Metaheuristic such as *memetic algorithms* and *iterated local search* often apply a local search operator before searching iteratively the problem-solution space [205]. Consequently, the general template introduced in algorithm 4.6 has been extended: a $3\_OptLocalSearch()$ is applied to the TSP-candidates solutions of p before the loop (see algorithm 5.2). A maximum number of 500 evaluations is applied each time a generated metaheuristic is executed. Each time an operator is applied on a TSP individual, a problem evaluation is deducted. Both population p and t have two individuals. The problem-specific parameter Depth of search has been set to 0.89 and the intensity of mutation to 0.8.

A CGP hyper-heuristic evolves the body a loop (see line 6 of algorithm 5.2). The technique introduced in section 4.2.1 is applied with the parameters and function provided in table 5.1 and 5.2).

Algorithm 5.2. : The template of a hybrid metaheuristic makes the body of the loop susceptible to the evolution.

```
1: function FINDSOLUTION(ProblemDomain,μ,λ)
2:     p ← InitPopulation(ProblemDomain,μ,λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectElitism(p)
6:         NumEvals ← DecodeAcyclicGraph(OutputNo = 0)    ▷ Evolved part of the code
7:         EvalCount = EvalCount + NumEvals
8:     end while
9:     return Best(p)
10: end function
```

Table 5.1: Parameters of the Classic CGP

Parameter	**Value**
Length (no of nodes)	100
Levels-forward (no of nodes)	100
Program inputs	1
Program outputs	1
$\mu + \lambda$	1 + 1
Mutation Rate	0.05
Generations	1200
Hyper-heuristics evaluations:	1202
Number of Runs	250

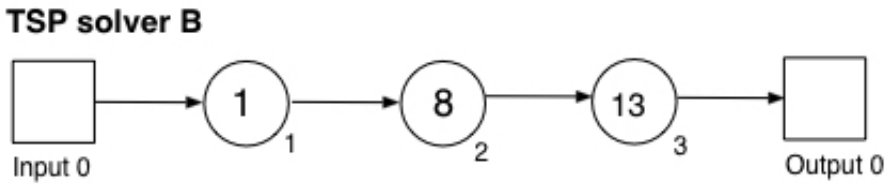
Table 5.2: The function set made of TSP-specific and population operators

opCode	Problem operators
0	t ← InsertionMutation(t)
1	t ← ExchangeMutation(t)
2	t ← ScrambleWholeTourMutation(t)
3	t ← ScrambleSubtourMutation(t)
4	t ← SimpleInversionMutation(t)
6	t ← 2_OptLocalSearch(t)
7	t ← Best2_OptLocalSearch(t)
8	t ← 3_OptLocalSearch(t)
9	t ← OrderBasedCrossover(t)
10	t ← PartiallyMapCrossover(t)
11	t ←VotingRecombinationCrossover(t)
12	t ← SubtourExchangeCrossover(t)
13	p ←ReplaceLeastFit(t,p)
14	p ←ReplaceRandom(t,p)
15	p ← RestartPopulation(p)

5.3.1 Validation

Some metaheuristics evolved in these experiments are given in algorithms [A.19, A.20, A.21, A.36, A.37, A.38]; those are referred as *TSP-[A-C]* and *TSP-[R-T]*. These sequences of instructions were translated from their CGP graphs to be hard-coded in three unique TSP solvers; an example is given in figure 5.1 and algorithm 5.3. These solvers were programmed with the programming language Java and use again all the primitives. For direct comparison, the metaheuristics due to Ozcan [247] and Ulder [319] were also coded in Java. All these algorithms can be found in section 9.2 in Appendix 9.2.

Figure 5.1: CGP graphs representing the TSP solvers B as described in algorithms 5.3



Algorithm 5.3. : **TSP Solver B**. The code formatted in black are part of the template shown in algorithm 4.8. The code in blue and italic fonts is the outcome of the decoding process.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectElitism(p)
6:         t ← ExchangeMutation(t)                    ▷ start generated code
7:         t ← 3 − OptLocalSearch(t)
8:         p ← ReplaceLeastFit(t, p)                  ▷ end generated code
9:     end while
10:    return Best(p)
11: end function
```

We were disappointed, but have found interesting, not every generated metaheuristic can find some suitable solutions. Some operators can increase the distance to a known optima, rather than shortening it. We had hoped our technique would balance both types of operators more effectively. Table 5.3 shows how the solver TSP-B relies on *ExchangeMutation* to disrupt the TSP solutions, then one local searche to shorten some tours *3_OptLocalSearch*.

Table 5.3: State of populations p and t at generation 7 during a validation run. The solver TSP-B was used with the validation instance d1291.

Operator	p_1	p_2	t_1	t_2
SelectElitism	1.394e-01 (=)	1.233e-01 (=)	1.394e-01 (≤)	1.233e-01 (≤)
ExchangeMutation	1.394e-01 (=)	1.233e-01 (=)	2.232e-01 (>)	2.236e-01 (>)
3_OptLocalSearch	1.394e-01 (=)	1.233e-01 (=)	1.110e-01 (<)	1.110e-01 (<)
ReplaceLeastFit	1.110e-01 (<)	1.110e-01 (<)	1.110e-01 (=)	1.110e-01 (=)

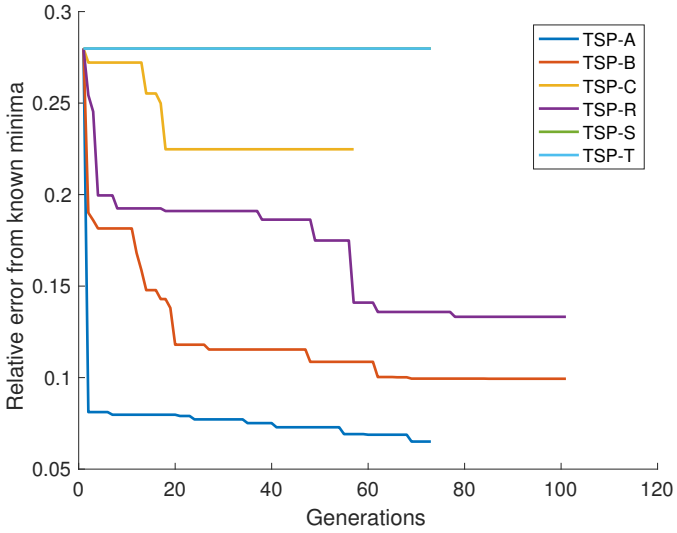
Some other algorithms have been unable to move away from a local optima (i.e. algorithms [A.37-A.38] in section 9.2 and graph 5.2). The body of the loop of solver TSP-T finds a near-optima using a *3-OptLocalSearch()*, then a *ScrambleSubtourMutation* moves away from the local optima. Some tour lengths (t_1 and t_2) are then shortened by a *2_OptLocalSearch* and a *Best2_OptLocalSearch()*. *OrderBasedCrossover* increases the tour length or preserves it. However, the offspring are much longer now than the parents. The latter remains therefore unchanged as the *ReplaceLeastFit* operator cannot replace the parents with the new offsprings; $f.fitness \geq p.fitness$. The same tours obtained at the start of a search are therefore selected over and over again without a shorter tour being obtained (see Table 5.4).

Table 5.4: State of populations p and t at generation 7 during a validation run. The solver TSP-T was used with the validation instance d1291.

Operator	p_1	p_2	t_1	t_2
SelectElitism	1.394e-01 (=)	1.233e-01 (=)	1.394e-01 ($\leq$)	1.233e-01 ($\leq$)
3-OptLocalSearch	1.394e-01 (=)	1.233e-01 (=)	1.112e-01 ($<$)	0.908e-02 ($<$)
ScrambleSubtourMut	1.394e-01 (=)	1.233e-01 (=)	1.230e+01 ($>$)	2.025e+01 ($>$)
2_OptLocalSearch	1.394e-01 (=)	1.233e-01 (=)	7.430e+00 ($<$)	6.794e+00 ($<$)
Best2_OptLocalSearch	1.394e-01 (=)	1.233e-01 (=)	6.59e+00 ($<$)	6.587e+00 ($<$)
OrderBasedCrossover	1.394e-01 (=)	1.233e-01 (=)	6.619e+00 ($<$)	6.587e+00 ($<$)
ReplaceLeastFit	1.394e-01 (=)	1.233e-01 (=)	6.619e+00 (=)	6.587e+00 (=)

Figures 5.2 illustrate how some generated metaheuristics can descent towards an optima during a learning and a validation run.

Figure 5.2: A comparison of the the solvers TSP-[A-C] and TSP-[R-T] during the search for an optimum tour for the learning benchmark pr439.



5.3.1.1 Performance

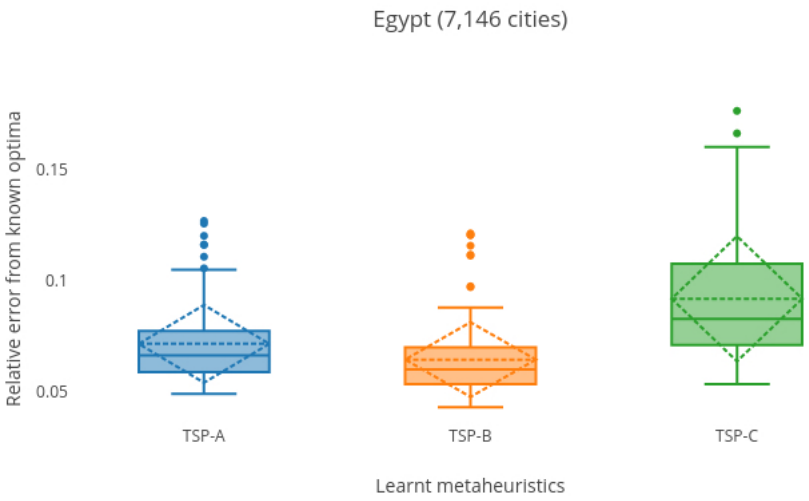
The solvers TSP-[A-C] have found some near-optimum ranging between 0 and 0.15 to a known optima. Those were first published in [273], with 3,000 evaluations and 20 independent runs. We have completed an additional 100 validation runs with a doubled number of problem evaluations, to deepen our understanding of these metaheuristics. More validation instances were also used ranging from 38 to 33,708 cities [2]. A detailed statistical analysis of the tours found for these instances is provided in section 9.2.

Except for the validation instance *dj38*, Ozcan[247] and Ulder [319] have consistently found some tours with an expected relative error greater or equal to 0.18. The automatically-designed metaheuristics have found better tours with an expected relative error lower or equal 0.10. These solvers have also a lower median than the ones humanly-written; the solver *TSP-B* has found the shortest tours. A Mann-Whitney U non-parametric test with a P-value set to 0.01 has confirmed the solver TSP-B is significantly better than solver TSP-A, TSP-C, Ozcan and Ulder's metaheuristics. Also, TSP-A is significantly better than TSP-C. A big effect is reported in tables B.25, B.23 and B.24. The A-measure is often greater than 0.71 for a majority of instances [304].

[2] http://www.math.uwaterloo.ca/tsp/world/countries.html

The distribution of the solutions obtained by these metaheuristics TSP [A-C] has a positive skew (i.e. $mean \geq median$). In figure 5.3 the standard deviation is shown in a diamond and the mean as a dotted line.

Figure 5.3: A statistical comparison of solvers TSP-A, TSP-B and TSP-C for the instance *eg7146*



5.4. Iterative Cartesian Genetic Programming: the full evolution of loops

We use the offline-learning generative hyper-heuristic introduced in section 4.2.2. These experiments should provide an insight into how hybrid metaheuristics can be discovered with an iterative Cartesian Genetic Programming.

The loops of our metaheuristics are fully evolved using *iterative CGP*. Our proposed method evolves merely a sequence of non-deterministic operators, but also repeated sub-sequences (or loops). Any iterations can terminate without any hard limits being implemented CGP.

We hope algorithms with several consecutive loops could be generated; often metaheuristics are designed without any nested loops. The iterative CGP hyper-heuristics settings (see table 5.5) therefore shows a number of nodes, mutation rate and hyper-heuristic evaluations have increased. The other parameters (i.e. program inputs and outputs, $\mu + \lambda$, the number of runs) have remained the same.

The method of evaluating a tour and the parameters remain unchanged from our previous experiments (see section 5.3). Tables 5.6 and 5.7 lists the function and condition set used in these experiments. The first three termination criteria rely on the number of problem evaluations used. Inspired Ulder [319], the termination criterion labelled with the terCode 4 relies on an additional parameter to function appropriately. Only when after 50 iterations no new shorter tour has been found in loop is exited.

Table 5.4 has shown some operators can disrupt too much a tour and it becomes too challenging to balance its effect with some others operators (see table 5.4). As well as *ScrambleSubtourMutation*, *ScrambleWholeTourMutation*, *PartiallyMapCrossover*, *VotingRecombinationCrossover* and *ReplaceRandom* can also affect negatively the TSP search. In the experiments reported in section 5.3, those appears the least in our generated metaheuristics (i.e. $\leq 5\%$) or none at all. Therefore those have been removed from the function set.

Table 5.5: Parameters of the Iterative CGP for all the tests

Parameter	**Value**
Length (no of nodes)	300
Levels-forward (no of nodes)	100
Levels-backs (no of nodes)	100
Program inputs	1
Program outputs	1
μ + λ	1 + 1
Mutation Rate	0.10
Hyper-heuristics evaluations:	1502
Runs	250

Table 5.6: Function set: List of TSP heuristics used as primitives.

Index	TSP heuristics
0	t ← InsertionMutation(t)
1	t ← ExchangeMutation(t)
4	t ← SimpleInversionMutation(t)
7	t ← Best2-OptLocalSearch(t)
8	t ← 3-OptLocalSearch(t)
9	t ← OrderBasedCrossover(t)
12	t ← SubtourExchangeCrossover(t)
13	p ← ReplaceLeastFit(p,t) and t ←SelectParents(p)
15	p ← RestartPopulation(p)

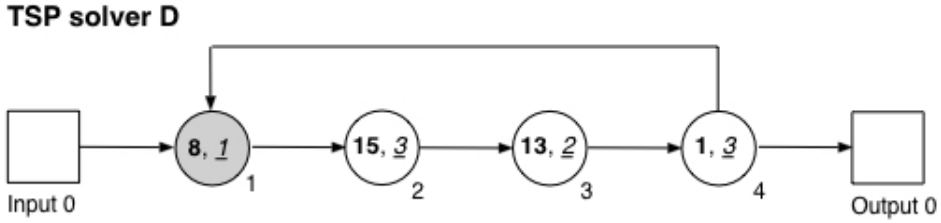
Table 5.7: Condition set: Boolean primitives chosen for the stopping criterion.

Index	Termination criteria
1	EvalCount $\leq$ MaxEval
2	EvalCount $\leq \frac{MaxEval}{2}$
3	EvalCount $> \frac{MaxEval}{2}$ and EvalCount $\leq$ MaxEvals
4	EvalCount $\leq$ MaxEval or IsBetter(noEval)

5.4.1 Validation of the learnt iterative metaheuristics

The solvers TSP-[D-E] and TSP-[U-W] were discovered (see algorithms [A.22-A.23] and [A.40-A.41] in section 9.2). An example of a translated iterative CGP graph into a programmed solver is provided in figure 5.4 and algorithm 5.4.

Figure 5.4: CGP graphs representing the TSP solvers D and described in algorithms 5.4



Algorithm 5.4. TSP Solver D - The code formatted in black is part of the template shown in algorithm 4.8. The code in blue and italic fonts is the outcome of the decoding process.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEval do                    ▷ start generated code
5:         t ← 3 − OptLocalSearch(t)
6:         p ← Restart(p)
7:         p ← ReplaceLeastFit(t, p)
8:         t ← SelectElitism(p)
9:         t ← ExchangeMutation(t)                     ▷ end generated code
10:    end while
11:    p ← ReplaceLeastFit(t, p)
12:    return Best(p)
13: end function
```

The solvers TSP-[D-E] have demonstrated some abilities to converge towards a known optima. However, solvers TSP-[U-W] have been unsuccessful to converge towards a suitable solution. Some of these metaheuristics ineffectively balance the operators that lengthen and shorten some tours within an iteration. The solver *TSP-U* (see algorithm A.40) has been unable to improve more than once the tour of population p, creating an abrupt drop during the learning run. Table 5.8 illustrates the operators *Best2_OptLocalSearch* and *SubtourExchangeCrossover* may leave unchanged the offsprings, until a *3_OptLocalSearch* operator is applied. However, the local search remains in local optima and as the search progresses, no better tour is found; resulting in creating only one improvement through the search (see solvers TSP-U in figure [5.5]).

Figure 5.5: A comparison of the solvers TSP-[D-E] and TSP-[U-W] during the search for an optimum tour for the learning benchmark pr439.

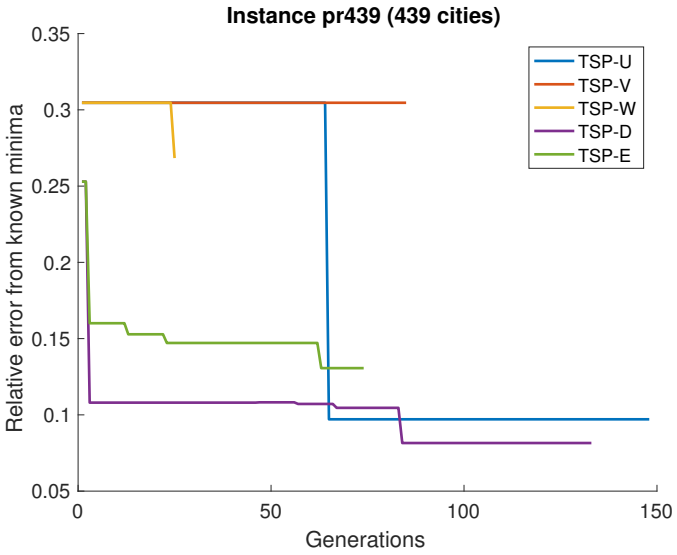


Table 5.8: State of populations p and t when 2,990 algorithm evaluations have been used during a validation run. The solver TSP-T was used with the validation instance d1291.

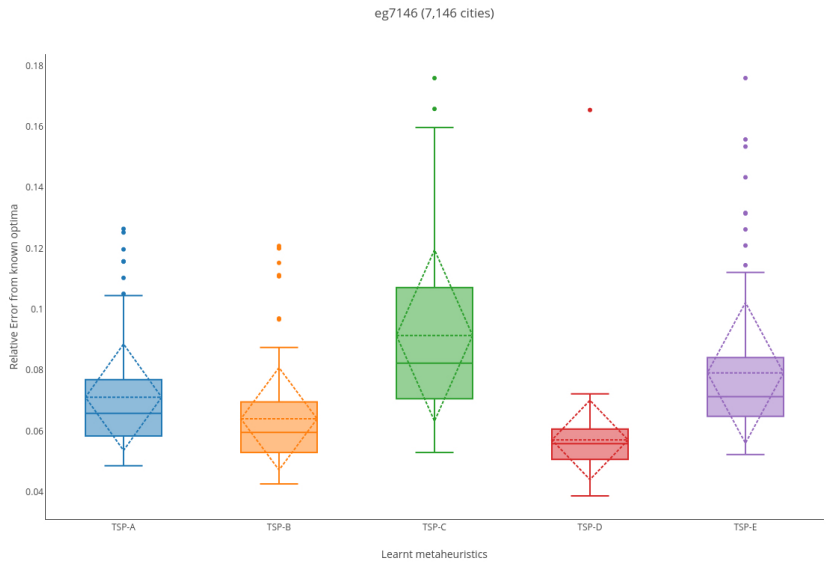
operator	p_1	p_2	t_1	t_2
Best2_OptLocalSearch	2.687e-01 ($\leq$)	2.537e-01 ($\leq$)	1.951e-01 ($\leq$)	1.951e-01 ($\leq$)
SubtourExchangeCrossover	2.687e-01 ($=$)	2.537e-01 ($=$)	1.951e-01($=$)	1.951e-01 ($=$)
Best2_OptLocalSearch	2.687e-01 ($=$)	2.537e-01 ($=$)	1.951e-01 ($=$)	1.951e-01($=$)
SubtourExchangeCrossover	2.687e-01 ($=$)	2.537e-01 ($=$)	1.951e-01 ($=$)	1.951e-01 ($=$)
Best2_OptLocalSearch	2.687e-01 ($=$)	2.537e-01 ($=$)	1.951e-01($=$)	1.951e-01 ($=$)
3_OptLocalSearch	2.687e-01 ($=$)	2.537e-01 ($=$)	1.778e-0 ($<$)	1.778e-01 ($<$)
ReplaceLeastFit	2.687e-01 ($=$)	2.537e-01 ($=$)	1.778e-01 ($=$)	1.778e-01 ($=$)
SelectElistism	1.778e-01 ($<$)	1.778e-01 ($<$)	1.778e-01 ($=$)	1.778e-01 ($=$)
3_OptLocalSearch	1.778e-01 ($=$)	1.778e-01 ($=$)	1.778e-01 ($=$)	1.778e-01 ($=$)
ReplaceLeastFit	1.778e-01 ($=$)	1.778e-01 ($=$)	1.778e-01 ($=$)	1.778e-01 ($=$)
SelectElitism	1.778e-01 ($=$)	1.778e-01($=$)	1.778e-01 ($=$)	1.778e-01 ($=$)

5.4.1.1 Performance

The solvers TSP-[D-E] were published [274]. Section 9.2 provides a detailed statistical analysis of the tours obtained by these solvers 9.2. Except for instances greater than 22,000 cities, these two solvers were able to find some shorter tours.

Most of the tours found by the solver *TSP-D* have a similar length; the standard deviation and interquartile range tend to be quite small (see tables of section 9.2). For example, the tours obtained for instance *eg7146* approximately vary by $[0.04]$ (see figure 5.6 and (see section 9.2)).

Figure 5.6: A statistical comparisons of TSP-A, TSP-B, TSP-C,TSP-D and TSP-E over a 100 runs with 6,000 problem evaluations. The mean and standard deviation are represented with a diamond shape.



5.5. Discussion and conclusion

Two offline-learning generative hyper-heuristics have evolved partially and fully the iterations of some metaheuristics, for the traveling salesman problem; the TSP is often used to test new methods. For a majority of the validation instances, some tours with an expected relative error to the known optima ranging between 0 and 0.10 have been consistently obtained. Our validations set included some benchmarks with a very large number of cities, but relatively short runs were able to find suitable tours.

We have translated the generated metaheuristics from their CGP-graph forms into a pseudo-code and demonstrating that those are compact and human-comprehensible. Some effective combinations of problem-specific operators were able to converge towards a known optima; the order of the TSP-specific operators have suitably perturbed some tours before improving them. On the other hand, ineffective patterns of primitives have remained in local optima.

Chapter 6. Improved learning objective process

Contents

6.1 Introduction 136

6.2 Problem domain 139

6.2.1 Traveling salesman problem 139

6.2.2 Mimicry problem 141

6.2.3 Nurse rostering problem 142

6.3 Evolution of the body of a loop 145

6.3.1 Discovery of Traveling Salesman Problem solvers 146

6.3.2 Performance 148

6.3.3 Discovery of Mimicry problem solvers 152

6.3.4 Discovery of nurse rostering problem solvers 156

6.4 The full evolution of loops 162

6.4.1 Discovery of iterative Travelling salesman solvers 163

6.4.2 Discovery of iterative mimicry solvers 165

6.4.3 Discovery of iterative nurse rostering problem solvers . . . 169

6.4.4 Performance and comparison 171

6.5 Discussion and conclusion 171

6.1. Introduction

Both finance and machine learning attempt to learn from a set of known data and make a prediction on an unknown set. These disciplines have benefited from applying some measures of dispersion and central tendency. In finance, a mean-variance analysis can predict the potential return of a portfolio. The possible loss and earning over a period of time should be balanced by a diversified portfolio [211]. In comparison, some supervised machine learning techniques should approximate a target function that maps some input variables to some output ones.

Sometimes the approximate target function becomes insensitive to small fluctuations in a learning set. Some other times it is unable to generalise at all on training or validation set of benchmarks. Some techniques have overcome these undesirable outcomes with the help of a coefficient of variation; a minimum level of quality that a target function should achieve during a learning run (i.e. a goal) has also been specified [184, 42, 251, 197].

This chapter explores whether an objective learning process inspired by a diversified learning set, a coefficient of variation and some achievable goals could improve the performance of our offline-learning generative hyper-heuristics.

We are proposing to balance the performances (effective and less effective) over a diversified set of learning instances. At least one instance is likely to be easy to solve, a second one brings more challenge to find some solutions and finally one that is known offer a high level of difficulty. The principles behind our improved learning objective process could be illustrated using some knives and a target. Figure 6.1 shows the performance of a knives-thrower after attempting a diversity of challenges.

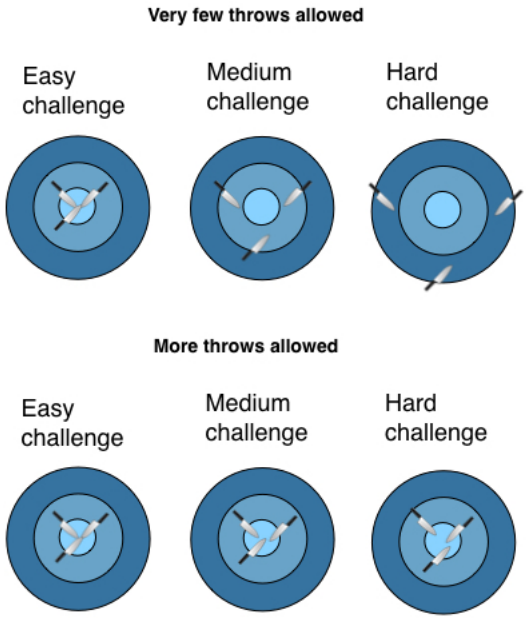
1. A knives-thrower was able to hit three times the centre when the player was throwing their pocket-knives from a very short distance away from the target. The central tendency and the dispersion for these results are set 0.00 (i.e. the results are known optimum).

2. The player then steps away further from the target; the central tendency and the dispersion are now greater, moving away from the centre and achieving near-optimum solutions.

3. The same pattern is repeated when the knives-thrower steps again further away and plays again. The knives have landed in the outer part of the target; the central tendency and the dispersion have increased too.

A successful knives-thrower should achieve a minimum requirement (see the top pattern illustrated in figure 6.1); otherwise, the player would only be able to hit the centre and miss other sections of the target. We would consider that the knives-thrower would be *overfitting* to the centre of a board. A player would be more sensitive to the fluctuation of some different challenges. When the number of throws increases, a successful knives-thrower should hit the centre of a target for the three targets.; the central tendency and dispersion should then decrease (see the bottom pattern of figure 6.1).

Figure 6.1: Results of a game with a target with three challenges



The improved learning objective function implements a similar incremental process as previously described.

Algorithm 6.1. The **Improved learning objective function** uses again the signature described in section 2.3.3;

Require: Instances must hold three instances referred as *easyInstance, mediumInstance, hardInstance*. Each of them should have a reasonable goal set for each step of the process.

```
1: function ALGEVALUATION(anAlgorithm, Instances).
2:     anInstance ← easyInstance
3:     if Algorithm has not improved initial population then          ▷ Phase 1
4:         return ∞
5:     end if
6:     for anInstance ∈ Instances do                                  ▷ Phase 2
7:         someResults[i] ← RunAlg(anAlgorithm, anInstance, Runs = 3)
8:         if Stats[anInstance].CentralTendency ≥ anInstance.goal then
9:             return ∞
10:        else
11:            Fitness = Fitness + Stats[anInstance].CoeffientOfVariation
12:        end if
13:    end for
14:    if The coefficient of variation has increased then             ▷ Phase 3
15:        return Fitness
16:    else
17:        return ∞
18:    end if
19: end function
```

Phase 1 assesses whether the generated metaheuristic improves its initial population. Learnt metaheuristics that fail this step stops the process at this phase. This undesirable feature affects the performance of the non-deterministic algorithm negatively.

Phase 2 collects some performance data incrementally. Compared to our previous learning objective function (see algorithm 5.1), the number of attempts has now increased from 1 to 3 for instance. If the central tendency of these results fails to meet an instance goal, then the algorithm stops. Otherwise, it repeats the same process with the next instance until there are none left.

Phase 3 penalises an algorithm that fails to demonstrate an ability to scale. A comparison of the coefficient of variations obtained from the independent runs for an easy and a hard instance should identify this ability.

6.2. Problem domain

The parameters and operators of three problem domain introduced in chapter 3 are used in our experiments.

6.2.1 Traveling salesman problem

Most of the traveling salesman problem parameters introduced in chapter 5.3 remain the same. From our observation made in our previous experiments, our function set has now a reduced number of operators; those listed in table 6.1.

Three different learning instances have been chosen to fit more suitably with the improved learning objective function. Using the results from chapter 5, we have chosen a goal of 0.00 for the instance *wi29* (i.e our "easy instance"). Then our chosen medium instance is *pr439* with a goal of 0.10, and finally, the most challenging instance is $d1291$ with a goal of 0.18. In our context, a goal indicates the distance away from a known minima. The domain knowledge gained from the results obtained from our previous experiments have helped us identify these expected performances.

Table 6.1: Function set: List of TSP heuristics used as primitives.

OpCode	Problem operators
0	t ← InsertionMutation(t)
1	t ← ExchangeMutation(t)
4	t ← SimpleInversionMutation(t)
7	t ← Best2-OptLocalSearch(t)
8	t ← 3-OptLocalSearch(t)
9	t ← OrderBasedCrossover(t)
12	t ← SubtourExchangeCrossover(t)
13	p ← ReplaceLeastFit(t,p)
	t ← SelectParents(p)

Table 6.2: Parameters of the metaheuristics for all the test

Parameter	**Value**
Offsprings t	2
Parents P	2
Maximum of evaluations	1002
Depth of Search	0.89
Intensity of mutation	0.8
Predetermined learning instances:	
- the easy instance with its goal	**wi29 (0.00)**
- the medium instance with its goal	**pr439 (0.10)**
- the hard instance with its goal	**d1291 (0.18)**

The template has been refined too. In line 3, a *3-OptLocalSearch()* operator is again applied. Other changes include executing a *population operator no 13* at the end of each iteration (see lines 12 and 13 of algorithm 6.2). These two lines replace the least fit parent with a better offspring, before selecting new individuals for reproduction in the temporary population t. The objective learning function is not testing for sequences of operators that include this combination of operations (i.e. operator 13). In chapter 5, metaheuristics that did not apply a replacement operator could not converge effectively towards the known minima.

Algorithm 6.2. : The template of a hybrid metaheuristic, with its core being evolved by an evolved Hyper-Heuristic algorithm.

```
function FINDSOLUTION(ProblemParam,μ,λ)
    p ← InitPopulation(ProblemParam,μ,λ)
    t ← SelectElitism(t)
    p ← 3-Opt-LocalSearch(p)
    while EvalCount ≤ MaxEvals or p.fitness > 0 do
                                          ▷ Evolved part of the code
        NumEvals ← DecodeAcyclicGraph(OutputNo = 0)
        p ← ReplaceLeastFit(t,p)
        t ← SelectElitism(p)
        EvalCount ← EvalCount + NumEvals
    end while
    return Best(p)
end function
```

6.2.2 Mimicry problem

The state-of-the-art [146] has influenced the settings of the metaheuristics' parameters. An evolution strategy *(1/1+1)* with 3072 generations have solved an instance with 500 bits. The evolution strategy recombines the genetic code of one parent and one offspring (see table 6.3).

Table 6.3: Parameters of the metaheuristics for all the learning test

Parameter	**Value**
Offsprings t	1
Parents P	1
Maximum of evaluations	1500
Mutation rate	0.001
Adaptive mutation rate	0.05
Predetermined learning instances:	
- the easy instance with its goal	300 (0.01)
- the medium instance with its goal	500 (0.05)
- the hard instance with its goal	800 (0.10)

The 500-bit instance is considered as a medium challenge with a goal of 0.05, and our easy instance has 300 bits and the most challenging one 800 bits. The goal has been set to some quite low values, as we hope to find near optima greater than 5,000 bits in our validation phase. All the mimicry-problem operators introduced in section 3.2 are included in our function set (see table 6.4). The termination criteria *IsBetter* has been adapted to the evolution strategy *(1/1+1)*. This condition terminates the execution of a loop when the populations p and t has not improved over one generation.

Each time a problem search starts, the mimicry problem domain requires generating a prototype randomly. Line 1 of algorithm 6.3 now applies the *InitPopulation* operator makes this process transparent (see section 6.2.2).

Table 6.4: Mimicry operators with their opcode and the number of evaluations used.

opCode	Operator(s)
0	t ← CrossoverOnePoint(t)
1	t ← CrossoverTwoPoints(t)
2	t ← CrossoverUniform(t)
3	t ← MutateOneBit(t)
4	t ← MutateOneBitHC(t)
5	t ← MutateUniformSubSequenceHC(t)
6	t ← MutateUniformHC(t)
7	t ← MutateUniformVariableRate(t)
13	p ← ReplaceLeastFit(t,p) t ← SelectParents(p)

Algorithm 6.3. : The template of a hybrid metaheuristic, with its core being evolved by an evolved Hyper-Heuristic algorithm.

function FINDSOLUTION(ProblemParam,μ,λ)
  p ← InitPopulation(ProblemParam,μ,λ)
  t ← SelectElitism(p)
  while EvalCount $\leq$ MaxEvals or p.fitness > 0 **do**
    ▷ Evolved part of the code
    $NumEvals \leftarrow DecodeAcyclicGraph(OutputNo = 0)$
    EvalCount = EvalCount + NumEvals
  end while
  return Best(p)
end function

6.2.3 Nurse rostering problem

Some initial experiments have highlighted that the disruption brought by some operators could be detrimental to the problem search. The alterations would deteriorate too much the quality of a roster so that it would be too challenging to correct to find a near-optima (or optimum). Their effect were tested and assessed by inspection. The function set given in table 6.5 has been reduced to local search, ruin-and-recreate, crossover and mutation operators that should help our learnt metaheuristics to move efficiently through the problem search space.

Table 6.5: Nurse rostering operators with their opcode and the number of evaluations used.

OpCode	**Operator(s)**
0	t ← NewSwapLocalSearch(t)
1	t ← HorizontalSwapLocalSearch(t)
3	t ← VariableDepthLocalSearch(t)
4	t ← GreedyVariableDepthLocalSearch(t)
5	t ← SimpleGreedyRuinRecreate(t)
6	t ← SmallGreedyRuinRecreate(t)
7	t ← LargeGreedyRuinRecreate(t)
11	t ← UnassignedShiftMutation(t)
13	p ← ReplaceLeastFit(t,p)
	t ← SelectParents(p)
15	p ← RestartPopulation()

The metaheuristics' parameters are given in tables 6.6. Our three learning instances have an increasing number of employees and type of shifts. Our easier instance *Instance1* schedules 8 nurses over a period of 14 days for 1 type of shifts. We consider the instance *BCV-4.13-1* as a medium challenge; rosters for an additional 5 nurses over 4 different of shifts for 29 days needs to be optimised. Our most challenging instances *Ikegami-2Shift-DATA1* more than double the number of nurses (i.e. 28) over 2 shifts over a period of 30 days.

Table 6.6: Parameters of the metaheuristics for all the test

Parameter	**Value**
Offsprings t	2
Parents P	2
Maximum of evaluations	40
Depth of Search	0.60
Intensity of mutation	0.60
Predetermined learning instances:	
- the easy instance with its goal	**Instance1 (0.00)**
- the medium instance with its goal	**BCV-4.13.1 (0.00)**
- the hard instance with its goal	**Ikegami-2Shift-DATA1 (0.15)**

We use the template described in algorithm 6.4 for the evolution of the body of a loop. The two population operations *replaceLeastFit* and *SelectParents* are applied to move the algorithm search forward more effectively. Iterative CGP-graphs also applies this feature at the end a subsequence. This has now been added in the decoded process for this problem (see algorithm 6.5).

Algorithm 6.4. : The template of a hybrid metaheuristics, with its core being evolved by an evolved Hyper-Heuristic algorithm.

```
function FINDSOLUTION(ProblemParam,μ,λ)
    p ← InitPopulation(ProblemParam,μ,λ)
    t ← SelectElitism(p)
    p ← GreedyVariableDepthLocalSearch(t)
    while EvalCount ≤ MaxEvals or p.fitness > 0 do
                                          ▷ Evolved part of the code
        NumEvals ← DecodeAcyclicGraph(OutputNo = 0)
        p ← ReplaceLeastFit(t,p)
        t ← SelectElitism(p)
        EvalCount = EvalCount + NumEvals
    end while
    return Best(p)
end function
```

Algorithm 6.5. A feedforward mechanism used to decode an iterative CGP graph

```
 1: procedure DECODECYCLICGRAPH(OutputNo)
 2:     NodesConnected ← IdentifyNodesConnectedToAnOutput(OutputNo)
 3:     OrderedNodesConnected ← IdentifyBranchingNodes(NodesConnected)
 4:     CurrentNode ← GotoFirstNodeOfGraph
 5:     while Not LastNodeOfGraph(CurrentNode) do
 6:         if TypeOf(CurrentNode) = processNode then
 7:             Values(CurrentNode) ← applyOperator(CurrentNode)
 8:         end if
 9:         if TypeOf(CurrentNode) = DecisionNode then
10:             if IsTerminationCriteriaMet(CurrentNode) then
11:                 Values(CurrentNode) ← applyOperator(13)
12:                 CurrentNode ← GoToEndOfTheLoop()
13:             else
14:                 Values(CurrentNode) ← applyOperator(CurrentNode)
15:             end if
16:         end if
17:         CurrentNode ← GoToNextNode()
18:     end while
19: end procedure
```

Algorithm 6.6. : The template of an hybrid metaheuristics, with its iteration(s) being fully evolved evolved by an Hyper-Heuristic algorithm.

```
function FINDSOLUTION(ProblemParam,μ,λ)
    p ← InitPopulation(ProblemParam,μ,λ)
    t ← SelectElitism(p)
    p ← GreedyVariableDepthLocalSearch(t)
                                        ▷ Evolved part of the code
    NumEvals ← DecodeCyclicGraph(OutputNo = 0)
    EvalCount ← NumEvals
    return Best(p)
end function
```

6.3. Evolution of the body of a loop

Our experiments apply the offline-learning generative hyper-heuristics introduced in section 4.2.1. We hope to investigate the effect of our new our new improved algorithm objective function (see algorithm 6.1) on the partial evolution of a loop.

Several parameters have been adjusted to reflect some of the findings of chapter 5; those have been formatted in bold in table 6.7. We hope longer algorithms can be evaluated as a larger algorithm search space can be explored. Perhaps more unusual metaheuristics performing more effectively can be discovered.

It is also hoped the effect on human understandability can also be further explored. Total freedom to connect forward the nodes remain our favoured choice. Lastly, we hope to reduce the number of learning runs, to save using some computer resources. Consequently, we have increased the total number of hyper-heuristics generations to 6000 (6002 hyper-heuristics evaluations in total).

Table 6.7: Parameters of our CGP hyper-heuristics

Parameter	**Value**
Length (no of nodes)	**200**
Levels-forward (no of nodes)	**200**
Program inputs	1
Program outputs	1
$\mu + \lambda$	1 + 1
Mutation Rate	**0.10**
Generations	**6000**
Hyper-heuristics evaluations:	**6002**

Some metaheuristics will be evolved for the mimicry, traveling salesman and nurse-rostering problem. The learnt metaheuristics will be translated from their directed acyclic graph form to the programming language Java. We also exhaustively enumerated the body of a loop over a period of 24 hours. In these experiments, we use the operators, parameters as and templates are given in section 6.2.

6.3.1 Discovery of Traveling Salesman Problem solvers

6.3.1.1 Effect of the improved learning objective function

Applying the improved learning objective function in conjunction to an increased number of hyper-heuristic evaluations have considerably lowered the number of hyper-heuristics evaluations; approximately 1.80e+06 algorithms evaluations were saved (i.e. the number of learning runs have been reduced from 250 to 20).

The improved learning objective function has assessed the metaheuristics more accurately. Metaheuristics demonstrating these prescribed behaviours could only be promoted by the hyper-heuristic, moving the search to a more desirable area of the algorithm search space. First, those needed to demonstrate an initial population could be improved; preventing some undesirable behaviours discussed in chapter 5.3. Secondly, the instance goals have contributed in identifying patterns of primitives that may not scale well. The discovered solvers, which have met the instance goals, have found the shortest tours with an increased number of runs and problem evaluations (table 6.8). The *TSP-[F-H]* and *TSP-[X-Y]* can be found in section 9.2 (see algorithms [A.24-A.26] and [A.42-A.43]).

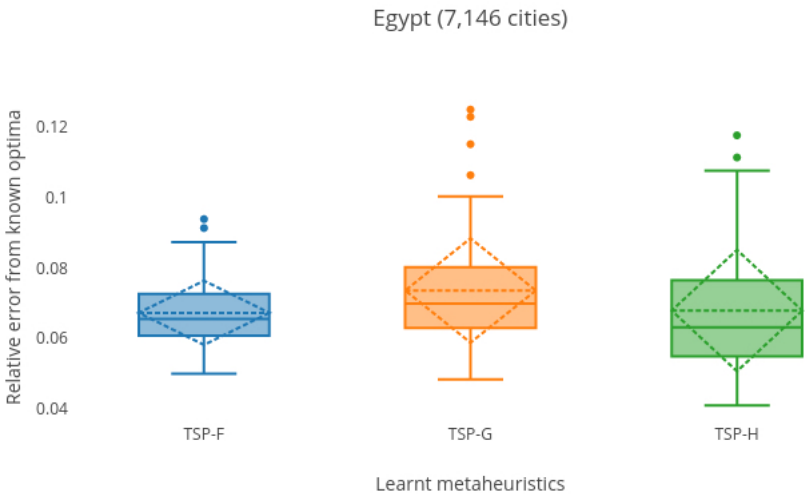
Table 6.8: A comparison of the tours likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs.

Algorithm	**Instance**	**Learning objective function** $NoRuns = 3$ $p.eval = 500$		**Goal**	**Validation** $NoRuns = 100$ $p.eval = 6000$	
		μ	σ	**met?**	μ	σ
TSP-F	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	**6.22e-02**	**1.81e-02**	yes	**3.82e-02**	**1.33e-02**
	d1291	1.39e-01	6.76e-02	yes	**1.03e-01**	**2.30e-02**
TSP-G	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	1.11e-01	3.49e-02	no	5.31e-02	2.04e-02
	d1291	**1.33e-01**	**3.82e-02**	yes	1.23e-01	2.78e-02
TSP-H	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	9.10e-02	2.50e-02	yes	4.55e-02	1.84e-02
	d1291	1.53e-01	5.98e-03	yes	1.13e-01	2.58e-02
TSP-X	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	1.50e-01	4.20e-02	no	5.89e-02	2.79e-02
	d1291	1.15e-01	7.06e-02	yes	1.61e-01	3.63e-02
TSP-Y	**wi29**	2.13e-02	3.01e-02	no	9.89e-03	1.50e-02
	pr439	9.91e-02	2.54e-03	yes	9.84e-02	3.68e-02
	d1291	1.29e-01	1.18e-02	yes	1.43e-01	2.05e-02

6.3.2 Performance

Section 9.2 provides a complete statistical analysis of the performance of solvers *TSP-[F-H]*. Figure 6.2 compares the tours obtained from these three solvers for the validation instance *eg7146*. The median and minima of solver TSP-H are the lowest, confirming the predicted performance of the improved learning objective function.

Figure 6.2: A statistical comparisons of TSP-[F-H] over a 100 runs with 6,000 problem evaluations. The mean and standard deviation are represented with a diamond shape.



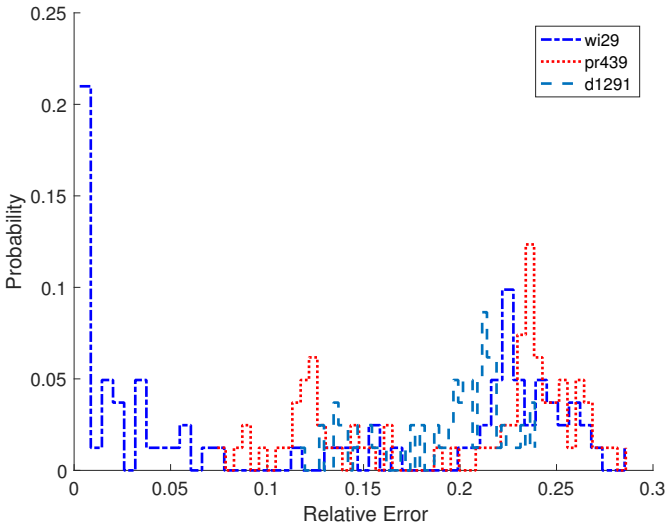
6.3.2.1 Comparison to an exhaustive search

An exhaustive search with up to 5 operators has been completed within a 24 hour period; employing more than five times the number of algorithm evaluations than our hyper-heuristics. For longer instances, the euclidean distance between each city could demand a lot of resources to identify their Cartesian coordinates. Some local-search operators rely on calculating euclidean distances in sub-tours, to establish whether the changes are shortening a tour; some of these operators can take a long time to run.

The outline of histograms represents the probability distribution of the tours obtained by traveling salesman solvers; these patterns of primitives were exhaustively enumerated with 2 or 3 operators using the metaheuristic parameter given in table 6.2. In both figures 6.3 and 6.4, less than 5% of metaheuristics would be able to meet the instance goal of *pr439* and *d1291*; this is approximately 3 metaheuristics out of 64 when 2 operators are applied and 25 out of 512 when 3 operators are applied.

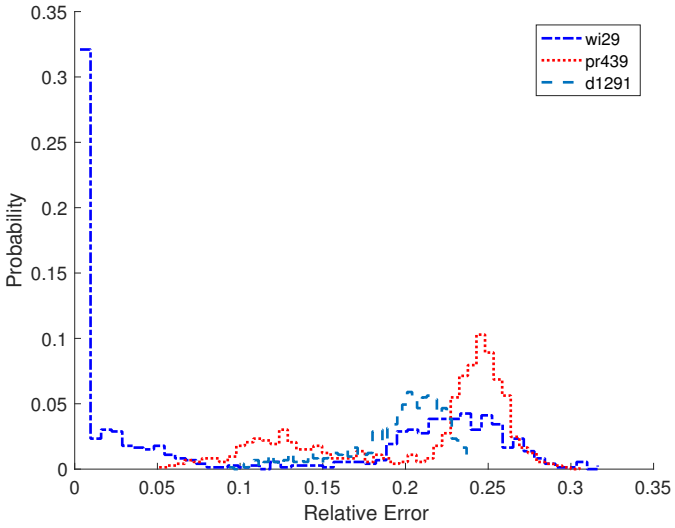
We can surmise the probability to design an effective metaheuristic applying a short number of operators can be quite low. When 2 operators have applied the probability to meet the instance goal for instances pr439 and d1291 can be less than $2.20e-03$. With 3 operators this probability slightly increases to $2.39e-03$.

Figure 6.3: The outline of histograms showing the statistic distribution of traveling salesman problem solvers obtained with **a two-operator exhaustive search**.



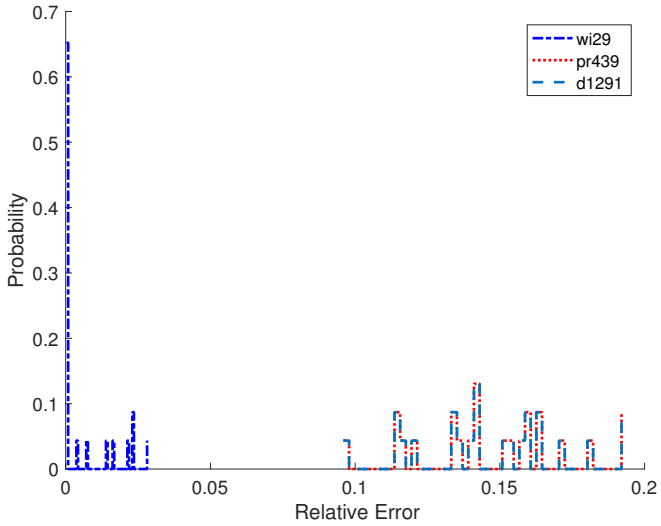
The probability distribution of traveling salesman solvers obtained by *CGP-designed metaheuristics* is much different (see table 6.5). First, its spread has been reduced to the approximate range $[0.00, 0.18]$; the exhaustive search was $[0.00, 0.28]$. The probability that CGP-designed metaheuristics find unsuitable tours for the instances *pr439* and *d1291* has been reduced; the peaks have moved to the left within the range $[0.1, 0.2]$.

Figure 6.4: The outline of histograms showing the statistic distribution of traveling salesman problem solvers obtained with **a three-operator exhaustive search**.



Unlike an exhaustive search, the patterns of primitives can vary between 0 and 200 operators (the length of CGP graph). The output a CGP graph and the feed-forward genes would vary the length of the encoded metaheuristic during the search. Therefore the improved learning objective function has assessed more varied patterns of primitives than an exhaustive search. This algorithm search space would have been much larger; less problem domain knowledge would have been provided by the programmer. More patterns of TSP-operators can, therefore, be explored; the solvers promoted would have been guided towards favourable areas of the algorithm search.

Figure 6.5: The outline of histograms showing the probability distribution of traveling salesman solvers obtained with **our offline non-iterative optimisation process**.



6.3.2.2 Comparison to selective hyper-heuristics

Our CGP hyper-heuristic has obtained some solvers, that can find better solutions than a selective hyper-heuristics. Table 6.9 compares the tours obtained in our experiments and two selective hyper-heuristics techniques. The tours obtained by the solvers TSP-[F-H] can be much shorter for the benchmark *usa13509* than those obtained a selective hyper-heuristic method. Automating the design of a selective hyper-heuristic or a metaheuristic can bring some scalability.

Table 6.9: Median of tours obtained in Chesc 2011, automatic design of selective hyper-heuristics [275], automatically designed selective-hyper-heuristics [120] and our experiments. The table reports the median tours using arelative error to the known optima.

Instance	**Selective hyper-heuristics**	**automatically designed selective hyper-heuristics**	**TSP-F**	**TSP-G**	**TSP-H**
pr299	**8.10e-05**	**8.10e-05**	6.35e-03	2.55e-03	4.23e-03
rat575	**5.53e-03**	**5.53e-03**	9.44e-03	7.52e-03	6.20e-03
u2152	**3.70e-02**	4.40e-02	4.75e-02	5.17e-02	4.09e-02
usa13509	9.43e+00	6.43e-02	5.54e-02	5.90e-02	**5.40e-02**

6.3.2.3 Comparison to a tree-based generative hyper-heuristic

[241] evolves deterministic algorithm that constructs a tour iteratively. This very different approach encodes within a template the algorithm (see algorithm 6.7). A tree-based GP generates a mathematical equation to replace a Euclidean distance as a metric to iteratively choose the order of cities. A small function set made of mathematical operators has been used. The terminals (i.e variables) represent the Cartesian coordinates of two cities (i.e. $y1$, $y2$, $x1$, $x2$) and the distance between the cities $c1$ and $c2$ (i.e M). Also, the number of unvisited cities (n) and the tour length k are also used.

Algorithm 6.7 shows only one expression has been made susceptible to the evolution. A mechanism prunes the branches exceeding a certain size; those are replaced by randomly selecting some variables (i.e. terminal). This technique relies on some input made by the programmer.

Algorithm 6.7. : TSP solver Ntombela. "Tour construction" algorithm using an evolved mathematical expression as reported by Ntombela et al [241]

```
procedure CREATETOUR(Cities[])
    startCity ← RandomlySelect (Cities[])
    firstCity ← startCity
    Tour ← EmptyList
    while UnallocatedCities(Cities[], Tour) do
        for City in Cities[] do                          ▷ Calculate a value for each city
            City.Value ← (y1/sqrt(M)) + x1 + 1 + 2 × k + y2          ▷ Evolved Part
        end for
        Cities[] ← SortAscendinglyByValues(Cities[])
        FirstCity ← SelectFirstCity(Cities[])
        Tour ← AddToTheEnd(FirstCity)
    end while
    return Tour
end procedure
```

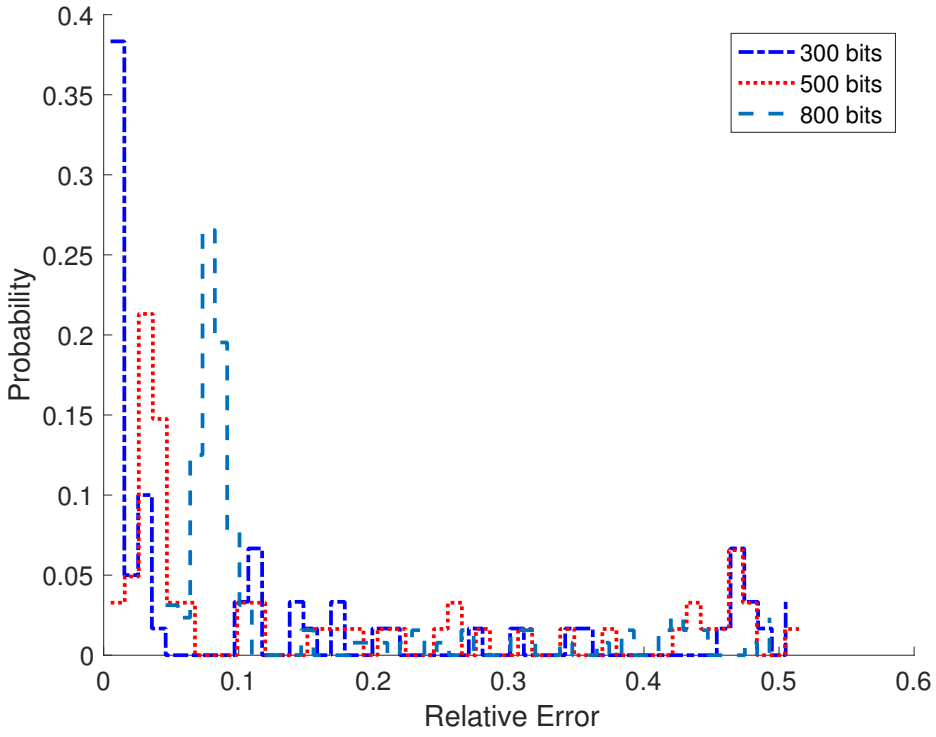
The performance of this techniques has been validated with instances ranging between 48 and 237 cities. The relative to the known minima often of the best tour obtained for these instances ranges between $[0.15, 0.20]$. It is worth noting, the benchmarks of our validation set use a greater number of cities and the gap to the known minima is often lower for most instances (see section 9.2).

6.3.3 Discovery of Mimicry problem solvers

6.3.3.1 Effect of the improved learning objective function

The metaheuristics referred as *MC-[A-C] and MC-[K-L]* were discovered from 20 learning runs; those are available in section 9.2 (see algorithms [A.2-A.4], A.12 and A.13. The probability distribution to find a suitable solution with a generated metaheuristics is quite high (see in figure 6.6). Many CGP-designed metaheuristics have found imitators that are near the instance goal. The distribution spread ranges between 0.0 and 0.5, suggesting that some runs could not find suitable patterns of primitives.

Figure 6.6: The outline of histograms showing the probability distribution of mimicry solvers obtained with **our offline non-iterative optimisation process**. The learning instances are used with a maximum number of problem evaluations of 1,500.



The solvers MC-A and MC-B have fully met the learning objective function targets; their algorithm fitness value is respectively 1.39e-01 and 1.56e+00. We would expect that the objective learning function would have found some suitable solutions for the three learning instances and some validation runs (see table 6.10). The three remaining metaheuristics (i.e. MC-C, MC-M and MC-N) have scored a high algorithm fitness value; at least one the instance goal has not been met. From these observations, we surmise the objective learning function has assessed well the generated metaheuristics.

Table 6.10: A comparison of the tours likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs.

Algorithm	**Instance**	**Learning objective function** $NoRuns = 3$ $p.eval = 1,500$ μ	σ	**Goal met?**	**Validation** $NoRuns = 100$ $p.eval = 20,000$ μ	σ
MC-A	**300**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	500	**3.00e-02**	**2.80e-03**	yes	**0.00e+00**	**0.00e+00**
	800	**7.66e-02**	**3.80e-03**	yes	**0.00e+00**	**0.00e+00**
MC-B	**300**	3.33e-03	0.00e+00	yes	**0.00e+00**	**0.00e+00**
	500	3.20e-02	6.40e-03	yes	2.00e-05	2.00e-04
	800	7.87e-02	2.59e-02	yes	6.25e-05	2.74e-04
MC-C	**300**	4.44e-03	1.57e-03	yes	**0.00e+00**	**0.00e+00**
	500	5.00e-01	3.33e-03	no	3.00e-04	7.72e-04
	800	5.13e-01	1.27e-02	no	2.88e-03	2.15e-03
MC-M	**300**	4.44e-01	1.95e-02	no	4.23e-01	4.63-02
	500	3.96e-01	2.40e-02	no	4.30e-01	4.60-02
	800	4.13e-01	5.55e-02	no	4.90e-01	4.39-02
MC-N	**300**	4.45e-01	1.34e-02	no	4.47e-01	3.69-02
	500	4.38e-01	1.74e-02	no	4.38e-01	3.51-02
	800	4.57e-01	2.33e-02	no	4.41e-01	3.66-02

Not all the combinations of operators have been effective. Solvers MC-[M-N] both combine a crossover operator with some hill-climber mutations; however, no replacement operator is applied. The metaheuristic MC-M recombines the parent solution with the offspring (i.e. CrossoverTwoPoints), increasing the number of incorrect bits (see table 6.11), after applying a MutationUniformSubSequenceHC(). Then three mutations are applied. Some of them improve the solutions, and some other do not. This pattern is repeated until the problem search stops; returning a poor imitator as the outcome.

Some other combinations have shown to be more efficient At each generation; four mutation operators can correct some bits; a *MutationOneBitHC* and *MutationUniformHC*. Therefore, the metaheuristic can find an optimum solution.

Table 6.11: State of the populations p and t at generation 1 during a learning run. The solver MC-M was used with the learning instance 800.

Problem-specific operator	p_1	t_1
MutationUniformSugSequenceHC	5.00e-01 (=)	4.80e-01 ($\leq$)
CrossoverTwoPoints	5.00e-01 (=)	5.06e-01 (>)
MutationVariableRate	5.00e-01 (=)	5.10e-01 (>)
MutationSubSequenceHC	5.00e-01 (=)	5.02e-01 (<)
MutationSubSequenceHC	5.00e-01 (=)	5.02e-01 (=)

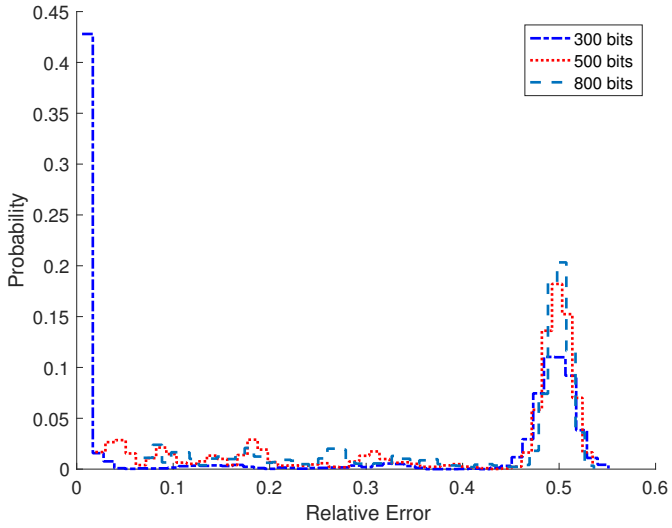
Table 6.12: State of the populations p and t at generation 1 during a learning run. The solver MC-A was used with the learning instance 800.

Operator	p_1	t_1
MutationOneBitHC	5.00e-01 (=)	4.76e-01 ($\leq$)
MutationUniformHC	5.00e-01 (=)	4.76e-01 (=)
MutationUniformHC	5.00e-01 (=)	4.76e-01 (=)
MutationUniformHC	5.00e-01 (=)	4.74e-01 (<)

A 24-hour run was able to enumerate successfully up to 6 operators. A minority of metaheuristics can find some suitable solutions for the 500-bit-long and 800-bit-long learning instances, using the parameters given in table 6.3. Those would apply four operators in the body of their loop. For example, in figure 6.7 less than 5% of the metaheuristics would meet the goal of the learning instances mentioned above (i.e. 0.05 and 0.10); it is approximately 200 metaheuristics.

The probability to design metaheuristics with the body of its loop applying 4 operators that meet these instances goal is less than $2.50e-03$. The assumption that this fixed number of operators needs to be made would rely on some well-developed domain knowledge. A CGP hyper-heuristic can generate algorithms of varying length; less domain knowledge is input by the programmer. The probability distribution shown in figure 6.6 is very different. It is skewed to left; the improved learning objective function has contributed in differentiating effective metaheuristics from ineffective ones; guiding the search to some favourable regions of the metaheuristic search space.

Figure 6.7: The outline of histograms showing the probability distribution of mimicry solvers obtained with **a four-operator exhaustive search** .



6.3.4 Discovery of nurse rostering problem solvers

6.3.4.1 Effect of the improved learning objective function

The NPC computer cluster[1] was also used for this series of experiments. Each hyper-heuristic evaluation has approximately been computed in 86.4 seconds; to complete a full run each hyper-heuristic evaluation could use a maximum of 7.2 seconds. Only a 12th of the hyper-heuristics evaluations were applied (i.e. 500). A list of constraints can use a lot of computer resources to compute a roster fitness evaluation and identify the best changes in a roster brought by an operator. Often the computer resources available can limit the number of hyper-heuristic evaluations. For these reasons, a reduced number of learning runs was completed over a period of 12-hours; in total 10 learning runs was attempted.

[1]N8 HPC provided and funded by the N8 consortium and EPSRC (Grant No.EP/K000225/1). The Centre is coordinated by the Universities of Leeds and Manchester

The solvers *NRP-[A-C]* and *NRP-L* (See algorithms [A.47 - A.49 and A.57] given in section 9.2) were discovered. A comparison of the rosters obtained by these metaheuristics during a learning run and some validation run suggests the improved learning objective function has been effective for the nurse rostering problem. In table 6.13, the solver NRP-A has not met all the instances goals; it has not found optimum roster during the validation.

Table 6.13: A comparison of the rosters likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs.

Algorithm	**Instance**	**Learning objective function** $NoRuns = 3$ $p.eval = 40$		**Goal met?**	**Validation** $NoRuns = 100$ $p.eval = 3,000$	
		μ	σ		μ	σ
NRP-A	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	2.83e-03	9.09e-03
	Ikegami	1.89e-01	6.94e-02	no	3.58e-02	2.92e-02
NRP-B	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	Ikegami	9.44e-02	6.74e-02	yes	**0.00e+00**	**0.00e+00**
NRP-C	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	Ikegami	6.11e-02	3.85e-02	yes	**0.00e+00**	**0.00e+00**
NRP-K	**Instance1**	2.35e+02	2.54e+00	no	3.34e+02	5.12e+01
	BCV-4.13.1	4.59e+02	7.90e+01	no	5.01e+02	9.01e+01
	Ikegami	2.13e+01	6.02e+00	no	4.45e+01	8.04e+00

Some generated metaheuristics can improve the roster and reduce the cost efficiently; table 6.14 illustrates the effect of each operator solver NRP-A on the search in one iteration. On the other hand, table 6.15 applies a very poor sequence of problem-specific operators. The cost of a roster decreases well until MultiEventCrossover increases dramatically the cost. The disruption cannot be efficiently corrected and would find an optimum roster for the more straightforward instance (i.e. instance 1).

Table 6.14: State of the populations p and t at generation X during a validation run. The solver the solver NRP-A was used with the learning instance Ikegami-2Shift-DATA1.

Operator	p_1	p_2	t_1	t_2
SmallGreedyRuinRec.	2.78e+01 (=)	2.48e+01 (=)	4.33e-01 ($\geq$)	4.33e-01 ($\geq$)
SimpleGreedyRuinRec.	2.78e+01 (=)	2.48e+01 (=)	4.16e-01 (<)	4.16e-01 (=)
ReplaceLeastFit	4.166e-01 (<)	4.16e-01 (<)	4.16e-01 (=)	4.16e-01 (=)
SelectElitism	4.166e-01 (=)	4.16e-01 (=)	4.16e-01 (=)	4.16e-01 (=)
GreedyVariableDepthLS	4.16e-01 (=)	4.16e-01 (=)	3.50e-01 (<)	2.50e-01 (<)
ReplaceLeastFit	3.50e-01 (<)	2.50e-01 (<)	3.50e-01 (=)	2.50e-01 (=)
SelectElitism	3.50e-01 (=)	2.50e-01 (=)	3.50e-01 (=)	2.50e-01 (=)

Table 6.15: State of the populations p and t at generation X during a validation run. The solver the solver NRP-K was used with the learning instance Ikegami-2Shift-DATA1.

Operator	p_1	p_2	t_1	t_2
VariableDepthLocalSearch	2.78e+01 (=)	2.48e+01 (=)	3.00e-01 ($\leq$)	3.16e-01 ($\leq$)
NewSwapLocalSearch	2.78e+01 (=)	2.48e+01 (=)	3.00e-01 (=)	3.00e-01 (<)
RestartPopulation	2.78e+01 (=)	2.48e+01 (=)	3.00e-01 (=)	3.00e-01 (=)
MultiEventCrossover	2.78e+01 (=)	2.48e+01 (=)	1.62e+03 (>)	1.62e+03 (>)
SmallGreedyRuinRecreate	2.78e+01 (=)	2.48e+01 (=)	1.43e+03 (<)	1.42e+03 (<)

6.3.4.2 Performance of the metaheuristics

The solvers have found some rosters of suitable quality over 100 independent runs with 3,000 evaluations. NRP-[B-C] have found the more suitable rosters; their distribution is more compact, and their skewness is undefined. The distribution of solver NRP-A can have a long tail to the right and is often positively skewed. Nonetheless, for certain instances, the median is the same for the solvers NRP-A, NRP-B and NRP-C. Section 9.2 provide all these results.

Figure 6.8: A statistical comparison of solvers NRP-A, NRP-B and NRP-C for the instance BCV-3.46.1

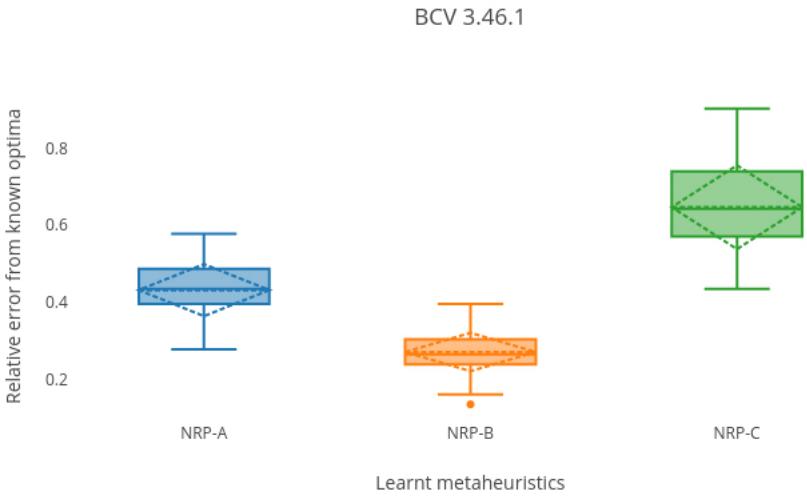


Figure 6.9: A statistical comparison of solvers NRP-A, NRP-B and NRP-C for the instance G-Post

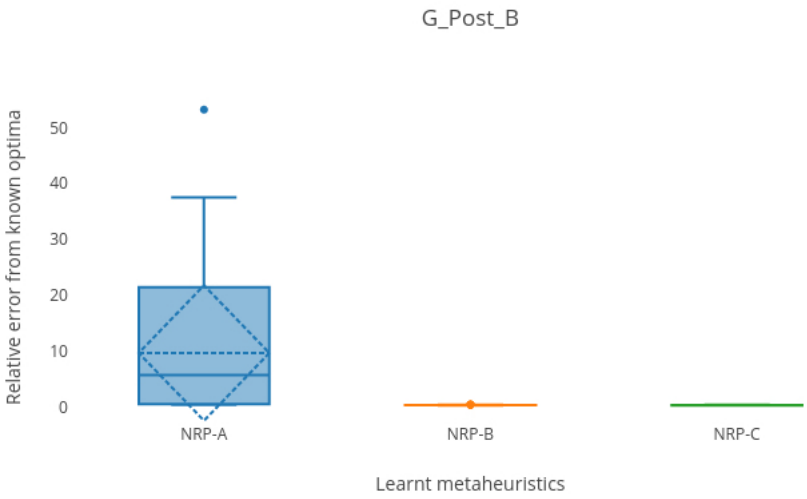
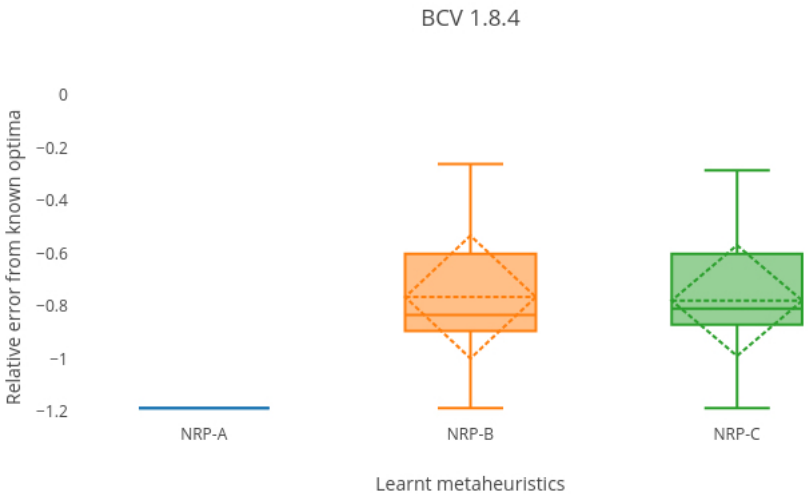


Figure 6.10: A statistical comparison of solvers NRP-A, NRP-B and NRP-C for the instance BCV-1.8.4



6.3.4.3 Comparison to an exhaustive search

Within a 24-hour run, a two-operator exhaustive search was completed. Every two-operator combination was able to find an optimum roster, for the benchmark referred as *Instance1*. The probability distribution to find a suitable solution for the other learning instances is given in figure 6.11. Our CGP hyper-heuristic would need more hyper-heuristics generations before it could find shorter combinations and perhaps more effective. For this reason, we have preferred not to provide a probability distribution of the solutions found by our generated metaheuristics.

6.3.4.4 Comparison to selective hyper-heuristics

The current state-of-the-art of the nurse-rostering problem is often considered as a form of selective hyper-heuristics or a form of integer programming. Metaheuristics are often ineffective as solutions can become infeasible easily. However, the solvers NRP-[B-C] have found better rosters than the state-of-the-art (see table 6.16).

Figure 6.11: The outline of histograms showing the statistic distribution of the nurse rostering problem solvers obtained with **a two-operator exhaustive search**. The learning instances are used with a maximum number of problem evaluations of 40.

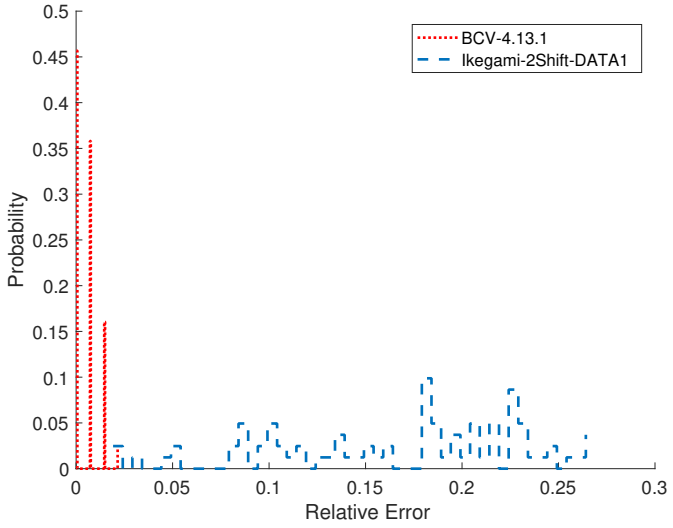


Table 6.16: A comparison of the averages of rosters obtained by [21] and our experiments. The table shows the relative error of the results.

Instance	**Selective hyper-heuristics** [21]	**NRP-A**	**NRP-B**	**NRP-C**
BCV-3.46.1	4.10e-01	4.25e-01	2.65e-01	6.40e-01
BCV-A.12.1	7.28e+00	6.42e+01	3.42e+00	4.06e+00
BCV-A.12.2	5.77e+00	6.32e+01	2.96e+00	3.45e+00
Ikegami 3Shift-data1	3.03e-01	3.92e-01	1.85e-01	1.28e-01
ORTEC01	1.34e+00	9.88e-01	6.92e-01	8.60e-01
ORTEC02	1.09e+00	2.03e+01	2.50e-01	9.00e-01

6.4. The full evolution of loops

We evolve the complete iterations of metaheuristics with the iterative Cartesian Genetic programming. Section 4.2.2 introduces this offline generative hyper-heuristics. For this series of experiments, our improved objective algorithm (see algorithm 6.1) will evaluate. The number of nodes has been reduced to match the length applied in our previous section (see the parameters formatted in bold in table 6.17).

Our condition set has been extended too; a wider range of termination criteria used in a variety of metaheuristics has been added (see the condition in bold in table 6.18 and section 3.1.2). No change has been made to the iterative template given in algorithm 4.8 in section 5.4.

Table 6.17: Iterative CGP parameters applied in these experiments. The parameters in bold have been refined and differ from our experiments in chapter 5

Parameter	**Value**
Length (no of nodes)	**200**
Levels-forward (no of nodes)	**200**
Levels-backs (no of nodes)	**200**
Program inputs	1
Program outputs	1
$\mu + \lambda$	1 + 1
Mutation Rate	0.10
Hyper-heuristics evaluations:	6002
Runs	**20**

Table 6.18: Condition set: Boolean primitives chosen for the stopping criterion. The conditions formatted in bold are different from our previous experiments.

TerCode	**Termination criteria**
1	EvalCount $\leq$ MaxEval
4	EvalCount $\leq$ MaxEval or IsBetter(p,noEval)
5	**EvalCount $\leq$ MaxEval or p.fitness $>$ 0**
6	**EvalCount $\leq$ Limit**
7	**EvalCount $\leq$ MaxEval or IsBetter(1)**
8	**EvalCount $\leq$ MaxEval or p.fitness $>$ 0 or Walk()**
9	**EvalCount $\leq$ MaxEval or p.fitness $>$ goal**
10	**EvalCount $\leq$ Limit or p.fitness $>$ goal**

6.4.1 Discovery of iterative Travelling salesman solvers

6.4.1.1 Effect of the improved learning objective function

The solvers TSP-[I-K] and TSP-[Z] were discovered over 20 runs (see algorithms [A.27 - A.29] and A.44 in section 9.2). The solvers TSP-A and TSP-B have been discovered again. Some validation runs have been completed with other combinations of TSP-specific operators; we wanted to explore the performance of a broader range of metaheuristics.

Iterative CGP was able to suitably balance some disruptive operators, local searches and the termination criterion. Those were have met the instance goals of the objective learning function. For example, solver TSP-Z applies a sequence of 12 operators in the body of a loop. This has been detrimental in finding suitable tours for the instance *pr439*, during the learning run. This metaheuristic applies many crossover operators; section 5.3 demonstrated the effect of some crossover operators might not bring any benefits.

Table 6.19: A comparison of the tours likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs.

Algorithm	**Instance**	**Learning objective function** $NoRuns = 3$ $p.eval = 500$		**Goal**	**Validation** $NoRuns = 100$ $p.eval = 6,000$	
		μ	σ	**met?**	μ	σ
TSP-I	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	**8.13e-02**	**2.20e-02**	yes	**4.21e-02**	**1.63e-02**
	d1291	1.41e-01	3.76e-02	yes	**1.09e-01**	**2.63e-02**
TSP-J	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	8.17e-02	3.74e-02	yes	4.46e-02	1.81e-02
	d1291	1.39e-01	2.62e-02	yes	1.12e-01	2.59e-02
TSP-K	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	9.54e-02	7.61e-03	yes	4.47e-02	1.97e-02
	d1291	**1.26e-01**	**2.97e-02**	yes	**1.09e-01**	**2.58e-02**
TSP-Z	**wi29**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	pr439	1.34.e-01	4.71e-02	no	6.90e-02	2.10e-02
	d1291	1.38e-01	3.38e-02	yes	1.59e-01	3.05e-02

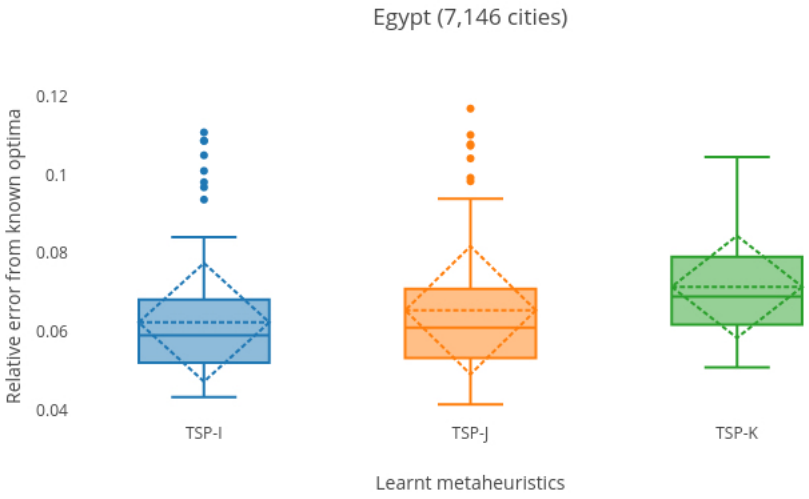
6.4.1.2 Performance and comparison

TSP-[I-K] have found the best tours in our learning set (see table 6.19). Those have therefore solved all the instances of our validation set. Detailed statistical results can be found in section 9.2.

Some large tours can be found by solver TSP-I and *TSP-J*; their distributions becomes skew more to the right (see figure 6.12). However, these two metaheuristics have found tours with similar median as the ones obtained by the best solvers discovered so far (i.e *TSP-B* , *TSP-D* and *TSP-H*) (see section section 9.2).

The solver *TSP-K* distribution is more compact. Similarly to solver *TSP-E*, fewer problem evaluations have found some suitable tours. Both metaheuristics apply a termination condition that reduces the problem search. The solver *TSP-K* has often found some better than the solver *TSP-E* (see table tab:TSP14 in section 9.2). These two metaheuristics would become useful to use when the level of accuracy is less important than the computing resources available.

Figure 6.12: A statistical comparison of solvers TSP-I, TSP-J and TSP-K for the instance eg7146



6.4.1.3 Comparison to a tree-based hyper-heuristics

During the completion of this work, Loloya et al. [203] have automatically designed a tour construction algorithm with a tree-based genetic programming. A very minimal template and function sets were also used. A branch encoded the header of a while loop, a selection or statements. Terminals could either add cities to a tour or modifies the tour. The latter includes a SimpleInversionMutation and a 2-Opt-Local-Search. Those were restricted to return integer values so that selection criteria could be expressed as a comparison (i.e. $= 1$ is true, and $= 0$ is false). One termination criterion is used (i.e. the current city is greater than a tour length).

These algorithms are quite compact too (see algorithms [A.45-A.46] in section 9.2). Large trees are penalised by a learning objective function. A programmer needs specifying a maximum number of branches and terminals., to prevent bloating.

It is disappointing no result obtained for a specific instance is reported; some benchmarks have been grouped together instead. The relative errors to the known optima ranges between 4.86e-02 and 6.44e-02. In section 9.2, the tours within the same range of cities vary between a perfect solution to a gap of 2.45e-02.

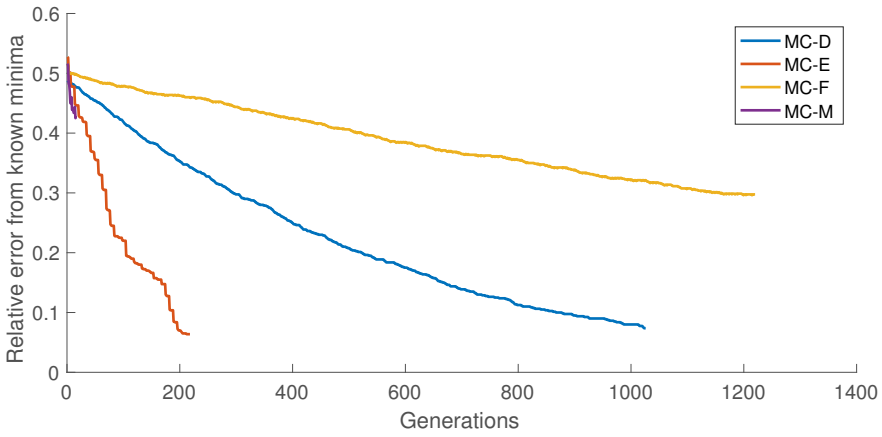
6.4.2 Discovery of iterative mimicry solvers

6.4.2.1 Effect of the improved objective learning function

The solvers *MC-[D-F] and MC-M* were discovered over 60 learning runs; those can be found in section 9.2 (see algorithms [A.5 - A.7] and A.14).

It was observed the solver *MC-M* would not meet some of the instances goals (see table 6.20). Randomly setting the number of problem evaluations can prevent searching the problem fitness landscape. The metaheuristic can find some imitators with a variable length (i.e. very short or quite long). For example, the number of problem evaluations would have been very small in figure 6.13); the curve for the solver *MC-M* is very short. Also, the search can be slowed when correct bits are flipped by the primitives in the body of a loop. As a result, this combination of problem-specific operators and termination criteria appears to be less effective (see figure 6.13).

Figure 6.13: A comparison of the solvers MC-[D-F] and MC-M during the search for a perfect imitator for a 800-bit benchmark. 1,500 problem evaluations were used.



6.4.2.2 Performance and comparison

A detailed statistical analysis of the imitators obtained by the solvers *MC-[D-F]* is given in section 9.2. These metaheuristics can find some imitators with the same median; the A-measure is very close to 0.5.

Some distribution may be affected differently by some outliers. For example, an undefined skewness is exhibited for solver *MC-F* in figures [6.14-6.16]); its median and arithmetical mean are very close to each other. The arithmetical mean of solvers *MC-D* and *MC-E* can either be lowered or increased by some outliers.

Table 6.20: A comparison of the imitators likely to be obtained during a learning run (i.e. by the learning objective function) and those obtained by 100 independent validation runs.

Algorithm	**Instance**	**Learning objective function** $NoRuns = 3$ $p.eval = 1,500$ μ	σ	**Goal met?**	**Validation** $NoRuns = 100$ $p.eval = 20,000$ μ	σ
MC-D	**300**	1.11e-03	1.92e-03	yes	**0.00e+00**	**0.00e+00**
	500	2.53e-02	1.62e-02	yes	**0.00e+00**	**0.00e+00**
	800	**7.29e-02**	**9.21e-03**	yes	**0.00e+00**	**0.00e+00**
MC-E	**300**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	500	2.80e-02	2.00e-03	yes	**0.00e+00**	**0.00e+00**
	800	7.79e-02	3.61e-03	yes	**0.00e+00**	**0.00e+00**
MC-F	**300**	1.11e-03	1.92e-03	yes	**0.00e+00**	**0.00e+00**
	500	**2.07e-02**	**6.11e-03**	yes	**0.00e+00**	**0.00e+00**
	800	7.38e-02	2.17e-03	yes	**0.00e+00**	**0.00e+00**
MC-O	**300**	3.91e-01	6.85e-03	no	4.43e-01	5.33e-02
	500	3.36e-01	2.89e-03	no	4.53e-01	2.93e-02
	800	3.48e-01	4.34e-03	no	2.22e-01	0.17e-01

Figure 6.14: A statistical comparison of solvers MC-C, MC-D and MC-E for the 3000-bit instance

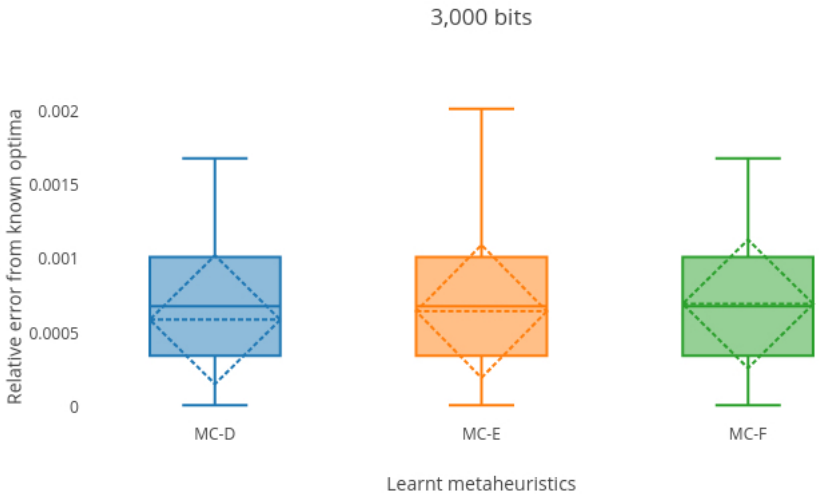


Figure 6.15: A statistical comparison of solvers MC-C, MC-D and MC-E for the 5000-bit instance

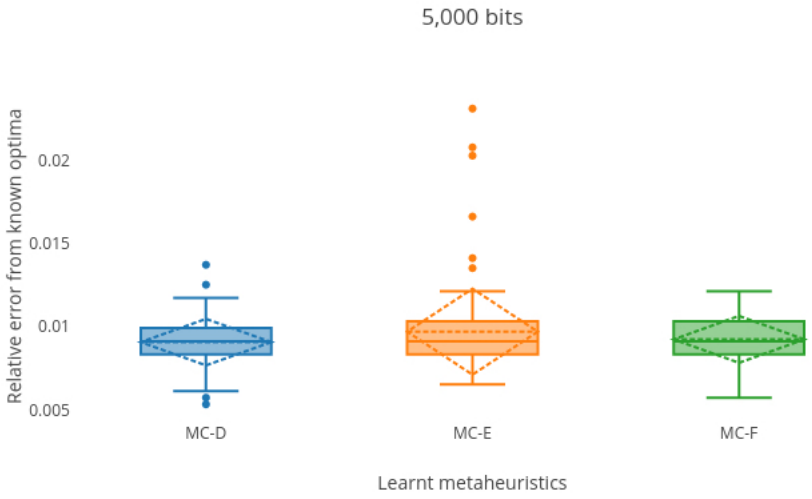
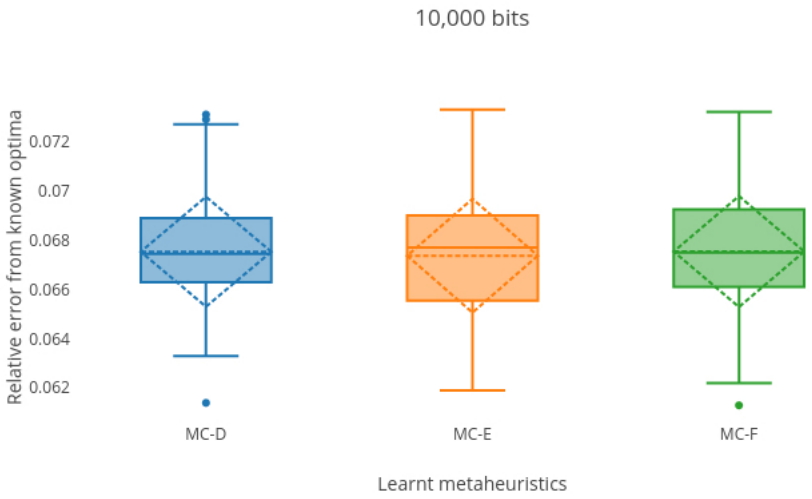


Figure 6.16: A statistical comparison of solvers MC-C, MC-D and MC-E for the 10000-bit instance



For instance with less than 5,000 bits, the iterative solvers have generally found better imitators than the solvers *MC-[A-B]*. The median of imitators found by solvers solvers *MC-[D-F]* are often lower than ones found by the solvers *MC-[A-B]* (see tables B.4 and B.5). With larger instances, the medians becomes the same. Evolve full iterations have been brought a positive effect to the algorithm search of mimicry solvers.

6.4.3 Discovery of iterative nurse rostering problem solvers

6.4.3.1 Effect of the improved objective learning function

20 learning runs were achieved in these experiments. Each of them was able to complete 10 times more evaluation than reported in section 6.3.4. The generated metaheuristics often could not improve the initial population; the learning objective process, therefore, stopped at phase 1. The learning algorithm has found too challenging to combine suitably a termination criteria with sequences of nurse-rostering operators. The short number of hyper-heuristic generations prevent testing and assessing many generated metaheuristics. Reducing the condition set to four conditions has had a little positive impact (see table 6.21).

Table 6.21: Condition set: Boolean primitives chosen for the stopping criterion.

Index	Termination criteria
1	EvalCount $\leq$ MaxEval
4	EvalCount $\leq$ MaxEval or IsBetter(noEval)
8	**EvalCount $\leq$ MaxEval or p.fitness $>$ 0 or Walk()**
9	**EvalCount $\leq$ MaxEval or p.fitness $>$ goal**

Despite experiencing these difficulties, we have obtained the solver *NRP-D* (see algorithm 6.8). This algorithm has met most of the learning instance goal but was able to find some optimum rosters consistently during the validation runs (see table 6.22).

Algorithm 6.8. : **NRP solver D** (NRP-D)

```
function FindSolution(ProblemParam,μ, λ)
    p ← InitPopulation(ProblemParam, μ, λ)
    t ← SelectElitism(p)
    while EvalCount ≤ MaxEvals do                    ▷ start generated code
        t ← VariableDepthLocalSearch(t)
        p ← ReplaceLeastFit(t, p)
        t ← SelectElitism(t)
        t ← SimpleGreedyRuinRecreate(t)
        t ← SmallGreedyRuinRecreate(t)
        p ← ReplaceLeastFit(t, p)
        t ← SelectElitism(t)
    end while                                        ▷ end generated code
    return Best(p)
end function
```

Table 6.22: A comparison of the rosters likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs.

Algorithm	**Instance**	**Learning objective function** $NoRuns = 3$ $p.eval = 40$		**Goal met?**	**Validation** $NoRuns = 100$ $p.eval = 3,000$	
		μ	σ		μ	σ
NRP-D	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	Ikegami	2.20e-01	1.84e-01	no	**0.00e+00**	**0.00e+00**

6.4.4 Performance and comparison

A detailed statistical analysis is available in section 9.2. For many instances, the iterative solver has the same median as the solver NRP-A, NRP-B and NRP-C. For some others a medium to a significant effect exists; some better or worse rosters have been found (see tables [B.45 - B.50]). It is therefore inconclusive whether the full evolution of loop has been beneficial to the algorithm search of nurse-rostering solvers.

6.5. Discussion and conclusion

Some solvers have been obtained successfully for more NP-hard or discrete problems (i.e. traveling saleman, the mimicry and nurse-rostering problems). Those have found suitable problem solutions; some of them are optimum or near to the known optima. New best solutions were also found. The problem solutions obtained from our CGP-designed hyper-heuristics are often better quality when compared to solutions obtained from a selective and tree-based generative hyper-heuristics.

Increasing four times the number of hyper-heuristic generations has extended the algorithm search efficiently. Most of the generated metaheuristics were obtained approximately after 2,400 hyper-evaluations. A minority of runs have promoted a generated metaheuristics on the last iteration; resulting in a long plateau. Nonetheless, we are pleased the number of learning runs has decreased by at least 92% and computing resources were economised. A reduction of approximately 180,000 hyper-heuristic evaluations was made.

Next chapter focuses on evolving the hyper-heuristic reproductive operator during the algorithm search.

Chapter 7. Evolving hyper-heuristic reproductive operators

Contents

7.1 Introduction 173

7.2 Experiments 174

7.2.1 Discovering sequential and iterative mimicry solvers 176

7.2.2 Discovering sequential and iterative traveling salesman solvers 176

7.2.3 Genetically improving some CGP mutation operators 176

7.2.4 Effect of the online generative hyper-heuristics 179

7.2.5 Comparison to an offline learning process 180

7.3 Validation of a learnt CGP mutation 182

7.3.1 Performance of discovered NRP solvers 183

7.3.2 Effect of the learnt CGP mutation operators 185

7.4 Discussion and conclusion 186

7.1. Introduction

An autoconstructive evolution evolves algorithms and a reproductive mechanism; sequences of operations that constructs hyper-heuristic reproductive operators should evolve during the algorithm search [302]. In section 2.4.4 this type of reproductive mechanism was referred as *GenerateAlg*. Autoconstructive CGP co-evolves a population of algorithms and CGP mutation operators; the latter are genetically improved during the algorithm search. This innovative CGP technique was introduced in section 4.2.3.

In chapter 4 new directed acyclic graphs and directed graphs were generated by altering some active and inactive genes. Some graph-based GP techniques, such as PADO, PDGP and CGP have included a concept of neutral mutation in their reproductive operators. Darwin [85] has described this phenomenon as *"Variations neither useful nor injurious would not be affected by natural selection, and would be left either a fluctuating element, as perhaps we see in certain polymorphic species, or would ultimately become fixed, owing to the nature of the organism and the nature of the conditions"*.

In previous chapters, CGP has randomly selected some active and inactive genes. It is possible some hyper-heuristic generations may only mutate non-active genes; those may be activated at a later stage of the algorithm search. Goldman et al. [114] has recently overcome this occurring using an independent-mutation-rate CGP; A CGP mutation alters randomly selected genes until on active genes is changed.

This chapter focuses on genetically improving some hyper-heuristic reproductive operators used during the algorithm search. We hope to extract these hyper-heuristic generative operators and code them with an imperative programming language. We aspire to discover some alternative hyper-heuristic reproductive mechanism that may improve the generation of CGP graphs.

7.2. Experiments

The online learning algorithm introduced in section 4.2.3 in used in these experiments; Autoconstructive CGP will be evolving partially and fully the iterations of some meta-heuristics. The hyper-heuristic parameters, learning objective function, templates and problem domains applied in chapter 6 have remained unchanged.

An iterative CGP-graph represents a reproductive operator for both types of solvers (figure 7.1). Additionally, table 7.1 provides the parameters used to encode the hyper-heuristic reproductive operators. Those have fewer nodes and a high mutation rate. Therefore, the number of possible paths has now been reduced to 1.17e+12 [2]. The possible number of CGP mutation operator can be larger. However, this estimation does not take into account the function and condition sets described in section 4.2.3.

Figure 7.1: Autoconstructive CGP graphs. The top individual encodes an algorithm with directed acyclic graph and the bottom individual with iterative CGP graph. Both individual encodes a mutation operators with an iterative CGP graph. All the branching genes are represented with blue arrows.

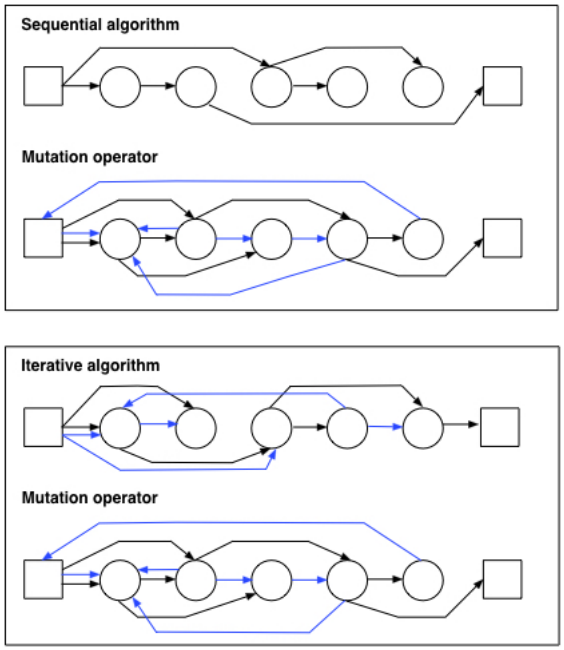


Table 7.1: Reproductive operators parameters

Parameter	**Value**
Length (no of nodes)	10
Levels-forward (no of nodes)	10
Levels-feedback (no of nodes)	10
Program inputs	1
Program outputs	1
Mutation Rate	0.20
Length of probation period	10 hyper-heuristics generations

7.2.1 Discovering sequential and iterative mimicry solvers

The sequential solvers (*MC-[G-I]*) and the iterative solvers (*MC-[J-L]*) were discovered (see algorithms [A.8 - A.13] in section 9.2).

A total of 80 learning runs were completed; 40 for each type of solvers. Often, the imitators obtained by these generated metaheuristics have the same or a lower median than the ones obtained by our previous experiments (see a detailed statistical analysis given section 9.2).

7.2.2 Discovering sequential and iterative traveling salesman solvers

20 learning runs were completed for each type of solvers. The sequential solvers *TSP-[L-M]* and the iterative solvers *TSP-[O-Q]* were obtained. The solver TSP-I has found the best expected tours by approximately a $\frac{1}{1000}$, but the tour distribution obtained with solvers TSP-M and TSP-Q appears to be more compact. Section 9.2 provides a detailed statistical analysis of the tours obtained by these metaheuristics. All the algorithms are given in section 9.2).

7.2.3 Genetically improving some CGP mutation operators

The *"Active-Node CGP mutation"* has successfully been genetically improved. This CGP mutation changes some active function and condition genes (see algorithms 7.1 and 7.2). The CGP mutation called *any-nodes* has been left unaltered during the algorithm search. When this CGP mutation operator is applied, then the algorithm search should progress to another area of the algorithm search space. The number of active nodes may change; this can lengthen or shorten a solver. Several active functions and condition genes are also altered to produce an algorithm with a different order of problem-specific operators (see algorithms 7.3 and 7.4).

Algorithm 7.1. : The *Active-Nodes* CGP mutation operator that can be applied on directed acyclic graphs.

```
1: function HYPERACTIVENODEMUTATION
2:     SwapFunctionBetweenTwoNodes()
3:     for i ∈ [0..2] do
4:         FlipTheFunctionOfAnActiveNode()
5:     end for
6: end function
```

Algorithm 7.2. : The *Active-Nodes* CGP mutation operator that can be applied on directed graphs.

```
1: function HYPERACTIVENODEMUTATION
2:     SwapFunctionBetweenTwoNodes()
3:     for i ∈ [0..1] do
4:         FlipTheFunctionOfAnActiveNode()
5:         FlipTheConditionOfAnActiveNode()
6:     end for
7: end function
```

The genetic improvement process has often reduced the number of alterations made to a CGP graph. The iteration initially coded was usually removed reducing a CGP mutation operator to an algorithm without any iterations. Some small changes would re-order some problem-specific operators (see CGP mutation A in table 7.2). The *CGP mutation B* is very likely to shorten or lengthen solver by pointing to another active node or a graph input.

Other genetic improvements have brought more changes to a solver. *CGP Mutation C* would change the order of the problem-specific operators and change a condition genes of an active node. This alteration may switch to a header of a loop if the randomly selected nodes encode a looping header in a solver. Finally, the *CGP Mutation D* searches locally for the best moves to improve a solver.

Algorithm 7.3. : The *Any-Nodes* CGP mutation operator that can be applied on directed graphs.

```
1: function HYPERLEARNTMUTATION
2:     for i ∈ [0..9] do
3:         FlipFeedForwardConnToAnActiveNode()
4:         FlipFunctionofActiveNode()
5:         FlipFeedForwardConnToAnyNodes()
6:     end for
7: end function
```

Table 7.2: Some examples of genetically improved CGP mutation altering iterative CGP graphs.

	CGP mutation	**Description**
A	SwapsFunctionsBetweenTwoActiveNodes()	Two randomly selected active function genes are swapped.
B	FlipTheInputForwardToAnActiveNode()	An active and randomly selected feed-forward connection genes is changed to point to another active node or a graph input.
C	SwapsFunctionsBetweenTwoActiveNodes() FlipTheConditionOfAnActiveNode	Two randomly selected active function genes are swapped. An active condition gene is changed.
D	For i $\in [0..3]$ InputForwardLocalSearch() End For	An active node is selected. Three attempts to point to another active node or a graph input are made. If one of the flips improves the solver, then the alteration is kept. Otherwise the CGP graph remains the same.This local search is repeatively applied 4 times.

Algorithm 7.4. : The *Any-Nodes* CGP mutation operator that can be applied on directed graphs.

```
1: function HYPERLEARNTMUTATION
2:     for i ∈ [0..9] do
3:         FlipFeedForwardConnToAnActiveNode()
4:         FlipFunctionofActiveNode()
5:         FlipConditionofActiveNode()
6:         FlipFeedForwardConnToAnyNodes()
7:         FlipBranchingGeneToAnyNodes()
8:     end for
9: end function
```

Algorithm 7.5. : The *Structure* CGP mutation operator that can be applied on directed graphs and directed acyclic graphs.

```
1: function HYPERLEARNTMUTATION
2:     for i ∈ [0..(Length/4)] do
3:         ChangeAnOutputOfAGraph()
4:         InitialiseANode()
5:         InitialiseAnActiveNode()
6:     end for
7: end function
```

The *Structure CGP mutation* has rarely been applied and it has not been genetically improved. This CGP mutation could be compared to ruining and recreating part of a CGP-graph. We had hoped this CGP mutation might reset the algorithm search if no other options (i.e. the *Any-Node* and *Active-Node* CGP mutation) would move the algorithm search forward. In the light of this result, we may now consider removing this option in the future.

7.2.4 Effect of the online generative hyper-heuristics

The algorithm evaluations required to evolve the hyper-heuristic reproductive operators are not considered as algorithm evaluations [302]; this may need to be reviewed in the future. It is therefore undeniable more CGP graphs have been evaluated by the co-evolution learning algorithm (see section 4.2.3.2). An additional $1,800$ algorithm evaluations may have been used. A genetic improvement process would have evolved 600 CGP mutation operators during a learning run (i.e using the experiments parameters stated in tables 6.17, 6.7 and 7.1).

We have observed a pattern of CGP mutations during some learning runs. An "active-node CGP mutation" would be applied and genetically improved several times in a row. Then an "any-node CGP mutation" would be identified and chosen to move the search forward. Sometimes this type of CGP mutation would be applied to many generations of CGP individuals. From these observations, we surmise the CGP mutation operators given in algorithms 7.3 and 7.4 have been used to most.

The mutation rate would indirectly vary during the algorithm search, ranging between 1 and 30 genes. Some of the genetically improved CGP mutation operators would mutate only one gene (see table 7.2). Some "any-nodes CGP mutation" much more. The genetic improvement process may increase or decrease the mutation rate each time it evolves the population of CGP-mutation operators.

The co-evolution of both problem solvers and CGP mutation operators have discovered some generated metaheuristics again. The best TSP solvers obtained from our previous experiments (i.e TSP-B and TSP-I) were re-discovered several times. Those were not chosen to explore a greater variety of TSP solvers.

More compact effective mimicry solvers have been obtained. For example, the solver *MC-A* has been reduced to two lines in solver *MC-G*. Also the solver *MC-L* shortens the solvers *MC-D*. These pairs share the same medians of imitators for most of the complete validation set.

7.2.5 Comparison to an offline learning process

The mutation rate has remained the same during the algorithm search; the neutral mutation applied in our offline generative hyper-heuristic would alter a constant number of genes during a learning run. An evolution strategy would only evolve a population of problem solvers.

Active and non-active genes are randomly selected. Unlike our online generative hyper-heuristic, active genes are not guaranteed to be altered each time a CGP offspring is produced. CGP graphs with identical active genes would pass to the next generation some new genetic material (see section 4.1.5 and algorithm 4.4). Yet an algorithm evaluation would be used to compute the algorithm fitness value of a known algorithm.

Goldman et al [114] have considered this situation as *wasted evaluations*. Traditionally, CGP has only evaluated CGP offspring with different active nodes indices than its parent. New genetic code encoded in inactive genes can, therefore, be passed to the next generation. The evolution strategy promotes CGP offspring with a better or equal fitness value [221].

A high probability to waste some evaluations during the evolution could be quite high. In section 5.3, the number of active nodes in CGP graphs would vary between 6 and 12 (i.e. 3 to 6 operators). Using the formulae suggested by [114] (i.e $(1 - MutationRate)^{NoOfActiveGenes}$), the probability to waste some evaluations during the partial evolution of iterations would be in range $[0.54, 0.73]$. In section 6.3.1, the probability would spread between $[0.43, 0.81]$. The body of a loop would range between two and eight operators long and the number of active genes between 4 and 16 operators. The mutation rate was increased and a replacement operator added to the template.

In comparison, our online generative hyper-heuristic would control the mutation of active and non-active genes. The co-evolution of both problem solvers and CGP mutation operators has helped in searching in turn "locally" and "globally" the algorithm search space. The algorithm search space may be explored more efficiently; finding some compact and effective solvers.

The number of learning runs for the traveling salesman has remained constant for the traveling salesman problem. For the mimicry problem, the inequality of learning runs between the partial and complete evolution of iterations has now been balanced.

An offline generative hyper-heuristics was the most appropriate method to obtain some nurse-rostering-problem solvers. The number of computer resources required by this problem would produce even shorter learning runs than in section 6.3.4; approximately 2% of the hyper-heuristics budget was used. The quality of those solvers was inferior; they were not demonstrating any scalable properties.

7.3. Validation of a learnt CGP mutation

The two hyper-heuristic reproductive operators most used by autoconstructive CGP were identified in section 7.2.3 (see algorithms 7.3 and 7.4). An evolution strategy is therefore edited to replace a neutral mutation with a *learnt CGP mutation operator* obtained from our experiments in section 7.2. The evolution strategy would remain the same, except the CGP mutation operator (see line 5 of algorithm 7.6).

Algorithm 7.6. The $(\mu + \lambda)$ evolution strategy

1: $CGP_{offspring} \leftarrow$ RandomlyGenerateIdividual($\mu + \lambda$)
2: $CGP_{parent} \leftarrow$ Promote($CGP_{offspring}$)
3: **while** Not solutionFound() or generation $<$ Limit **do**
4: **for** $i \in [1..\lambda]$ **do**
5: $CGP_{offspring}[i] \leftarrow$ **LearntMutation**(CGP_{parent})
6: $CGP_{offspring}[i] \leftarrow$ Evaluate($CGP_{offspring}[i]$)
7: **end for**
8: $CGP_{parent} \leftarrow$ Promote($CGP_{offspring}$)
9: **end while**

We hope to validate the performance of this hyper-heuristic reproductive operator on an unseen problem domain (i.e. the nurse-rostering problem). The hyper-heuristic parameters remain mostly the same (see sections 6.3 and 6.4). The CGP mutation operator defines itself the hyper-heuristic mutation rate (i.e. 0.075 for the partial evolution of loops and 0.0625 for the complete evolution of loops). The problem domain settings remain the same as section 6.2.3 and 6.3.4.

7.3.1 Performance of discovered NRP solvers

The nurse rostering solvers NRP-[E-J] and NRP-[L-O] are examples of generated metaheuristics discovered in our latest experiments (see algorithms [A.51-A.56] and [A.58-A.61] in section 9.2). A minority of these solvers have met the goal sets in the learning objective function (see table 7.3 and section 6.2.3). Some of these metaheuristics may find many times known optima, but outliers can affect the distribution negatively. For example, the solvers NRP-E and NRP-G have found most suitable solutions. However, one outlier with a huge gap has been found (see figure 7.2). Such occurrences could affect the objective learning function negatively, misrepresenting the real performance of a solver. As a result, this may lead to rejecting a suitable solver during the algorithm search and the validation process (i.e. solvers NRP-M and NRP-O in table 7.3).

Solver NRP-N (see algorithm 7.7) has demonstrated the worse performance. This solver has been obtained from a complete evolution of loops. However, best-generated solvers have no loop. The generated part of the metaheuristic is executed once, reducing the problem solution search dramatically.

A detailed statistical analysis was completed for the solvers NRP-[E-J] (see in section 9.2). An example of some results is provided in figures 7.2 and 7.3.

Algorithm 7.7. : **NRP solver N** (NRP-N) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)                          ▷ start generated code
4:     t ← VariableDepthLocalSearch(t)
5:     t ← UnassignedShiftMutation(t)
6:     t ← UnassignedShiftMutation(t)                ▷ end generated code
7:     return Best(p)
8: end function
```

Table 7.3: A comparison of the rosters likely to be obtained during a learning run (i.e by the learning objective function) and those obtained by 100 independent validation runs.

Solver	**Instance**	**Learning objective function**			**Validation**	
		$NoRuns = 3$		**Goal**	$NoRuns = 100$	
		$p.eval = 40$		**met?**	$p.eval = 3,000$	
		μ	σ		μ	σ
NRP-E	**Instance1**	**0.00e+00**	**0.00e+00**	yes	8.61e-04	8.65e-03
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	2.83e-03	9.09e-03
	Ikegami	3.08e-02	2.54e-02	yes	3.58e-02	2.92e-02
NRP-F	**Instance1**	**2.07e+01**	**6.66e+00**	no	1.30e+01	1.06e+01
	BCV-4.13.1	**1.61e+02**	**2.79e+02**	no	2.15e-04	2.16e-03
	Ikegami	2.11e-01	4.08e-02	no	9.15e-02	5.05e-02
NRP-G	**Instance1**	2.20e+01	2.50e-02	no	2.58e-03	1.48e-02
	BCV-4.13.1	**3.20e+02**	**2.77e+02**	no	5.89e+01	1.61e+02
	Ikegami	2.16e-01	6.09e-02	no	4.15e-02	6.24e-02
NRP-H	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	Ikegami	1.33e-01	0.0e+00	yes	5.44e-02	2.05e-02
NRP-I	**Instance1**	**0.00e+00**	**0.00e+00**	yes	3.04e-03	1.53e-02
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	1.54e-03	1.16e-02
	Ikegami	2.11e-01	4.08e-02	no	1.21e-01	6.07e-02
NRP-J	**Instance1**	**0.00e+00**	**0.00e+00**	yes	2.58e-03	1.48e-02
	BCV-4.13.1	**0.00e+00**	**0.00e+00**	yes	1.29e-03	6.01e-03
	Ikegami	1.22e-01	1.60e-02	yes	1.61e-01	5.68e-02
NRP-L	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**6.21e+00**	**10.00e+00**
	BCV-4.13.1	3.18e+02	2.75e+02	yes	4.06e+02	9.09e-03
	Ikegami	2.21e+01	3.04e+02	no	6.21e+00	10.01e+00
NRP-M	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	BCV-4.13.1	7.06e-03	3.01e-02	no	2.17e-02	1.43e-03
	Ikegami	9.91e-01	6.02e-02	no	9.19e-02	4.86e-02
NRP-N	**Instance1**	2.22e+01	2.01e-02	no	**21.86e+00**	**3.81e+00**
	BCV-4.13.1	4.91e+02	4.16e+00	no	5.40e+02	3.09e+02
	Ikegami	1.45e+01	6.28e+00	no	2.40e+01	4.78e+00
NRP-O	**Instance1**	**0.00e+00**	**0.00e+00**	yes	**0.00e+00**	**0.00e+00**
	BCV-4.13.1	**1.01e-02**	**2.02e-03**	no	1.72e-03	6.60e-03
	Ikegami	1.21e-01	4.98e-02	yes	1.04e-01	4.58e-02

Figure 7.2: A statistical comparison of solvers NRP-E, NRP-F and NRP-G for the instance BCV-1.8.4

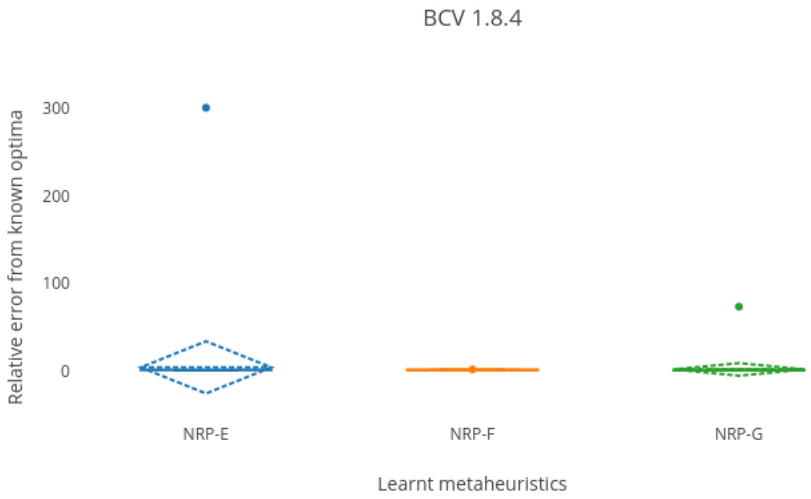
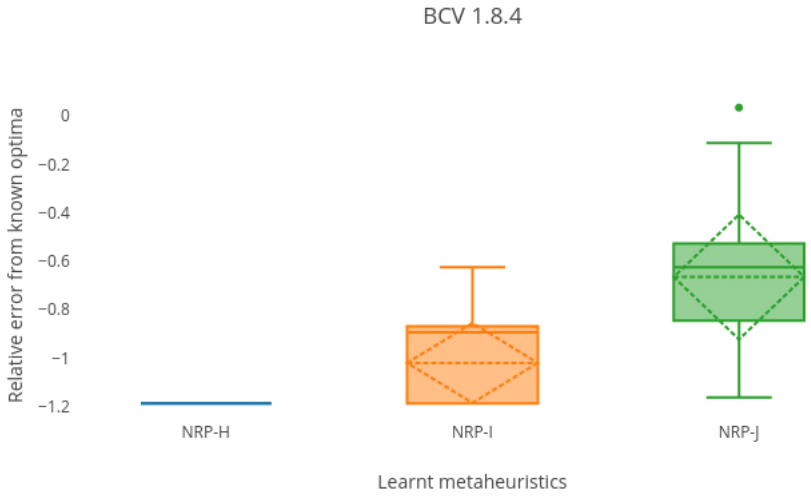


Figure 7.3: A statistical comparison of solvers NRP-H, NRP-I and NRP-J for the instance BCV-1.8.4



7.3.2 Effect of the learnt CGP mutation operators

The learnt CGP mutation operator had a positive impact on the search. The number of algorithm evaluations rose from 2% to 10%, and the number of learning runs increased from 10 to 20 for each type of encoding scheme (i.e. CGP and iterative CGP).

A learning objective function would assess more generated metaheuristics. Changing some "active function" and "active condition" genes would create some new solvers each time the learnt CGP mutation is applied. Altering some active and inactive connection genes contributes to changing the order of the problem-specific operators too. This higher level of control over the algorithm search may have implemented and improved the search for NRP solvers. Still many of these solvers may be unsuitable.

More sections of the algorithm search space are likely to be searched. The offline hyper-heuristic should visit more favourable areas this cannot be guaranteed. The random process of selecting CGP genes for mutation may be partly restricted to some active genes. Also, no restriction or problem domain knowledge has been included.

The algorithm search may have been the least successful when loops were fully evolved. Some metaheuristics have been promoted with no iteration, performing less efficiently than those that repetitively apply some patterns of primitives. Their problem search is much shorter. Such choice suggests the algorithm search may sometimes be underfitting.

7.4. Discussion and conclusion

From our experiments with mimicry and TSP problems, we have identified some CGP mutations operators. Some solvers for the nurse-rostering problem were learnt by one of these refined offline-learning generative hyper-heuristic. The neutral mutation was replaced by our *learnt hyper-heuristic reproductive operator*. We were therefore able to find more suitable iterative nurse-rostering solvers. A CGP mutation operator was validated on a *unseen* problem domain.

Chapter 8. Critical analysis

Contents

8.1 Scalable patterns of primitives 188
8.1.1 The Traveling Salesman Problem 188
8.1.2 The mimicry problem 192
8.1.3 The nurse rostering problem 196
8.2 Automatic design of metaheuristics 200
8.2.1 Templates and directed graphs 201
8.2.2 Effect of the learning objective functions 204
8.2.3 Effectiveness of the learning 207
8.3 Comprehensibility metrics 208
8.3.1 Problem-specific solvers 209
8.3.2 Other forms of GP 212
8.3.3 Effect on human understandability metrics 214
8.3.4 Comparison with other techniques 216
8.3.5 Discussion 221
8.4 Conclusion 222

This chapter critically analyses the results of our experiments to bring some answers to our five objectives introduced in section 1.1. We will focus our attention to the problem and algorithm domain, before our optimisation processes.

8.1. Scalable patterns of primitives

At the start of this work, we made the assumptions it would be beneficial to comprehend how certain combinations of operators can lead to a good or poor performance. Programming languages offer a medium to convey the instructions to complete a specific task, between programmers and computers [140]. The idea of computers communicating algorithms to programmers is not new. Newell et al. [239] stated in their seminal paper *"artificial intelligence must be concerned with how symbol systems must be organised in order to behave intelligently"*.

Our experiments have generated solvers of varying effectiveness and scalability. A critical analysis of these solvers' performance could suggest some combinations of problem-specific operators that could find some suitable solutions.

8.1.1 The Traveling Salesman Problem

8.1.1.1 Effect of geographical features on the generated solvers

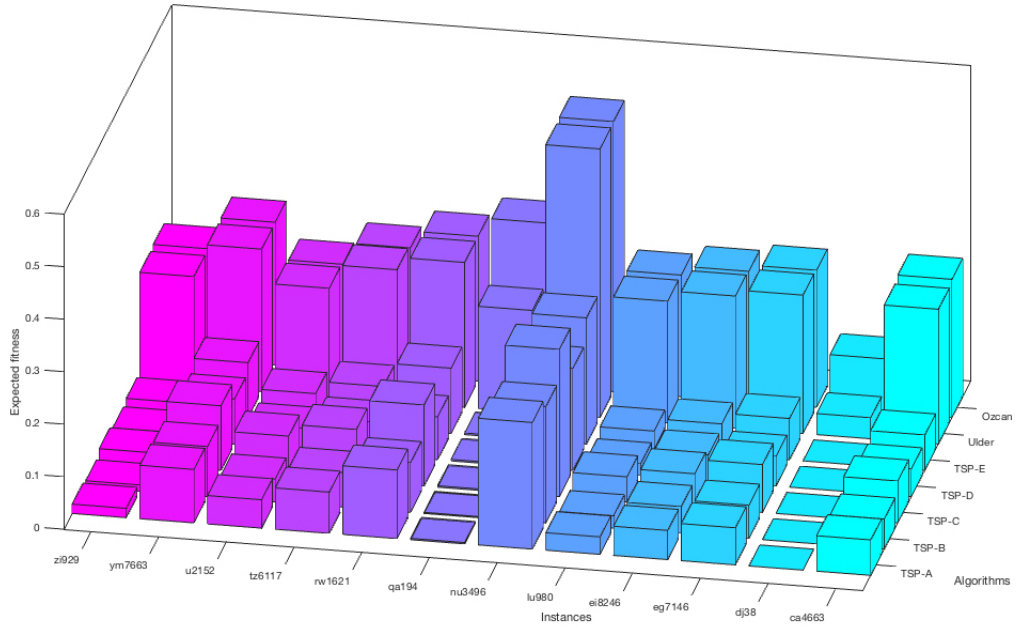
TSP instances vary in term of number cities and geographical features. Landforms and water bodies can restrict the number of routes available between cities, creating some clusters of some towns.

Our instances either represent a drilling networks (i.e. u2152, usa13509 and d18512) or actual geographical data (i.e. the remaining instances). The mapping of some algorithms, instances and some expected fitness values can vary greatly to form an irregular set of columns (see figures 8.1 - 8.3).

These figures suggest our solvers have found better tours for instances that have the least clustered cities. Examples of these instances include *ca4663, d18512, dj38, ei8246, fi10639, ho14473, lu980, mo14185, qa194, sw24978, u2152, usa13509 and zi929*. The expected fitness are often consistently low (see figures 8.1 - 8.3).

Archipelagos and mountain ranges can bring some bottleneck clustering some cities togethe. Greece, Japan, Vietnam and Argentina (i.e gr9882, ja9847, vm22775 and ar9152) have a highest expected fitness (see figures 8.1 - 8.3).

Figure 8.1: A graphical representation of the expected fitness of for the algorithms *TSP-A to TSP-E* and the metaheuristics published by [246, 319]. The instances ranges between 38 to 7663 cities



8.1.1.2 Effect on the number of cities on the generated solvers

A different set of solvers have found better tours than others as the number of cities increases, irrespectively to their unique geographical features.

Figure 8.2: A graphical representation of the expected fitness of for the algorithms *TSP-F to TSP-Q* and the metaheuristics published by [246, 319]. The instances ranges between 38 to 7663 cities

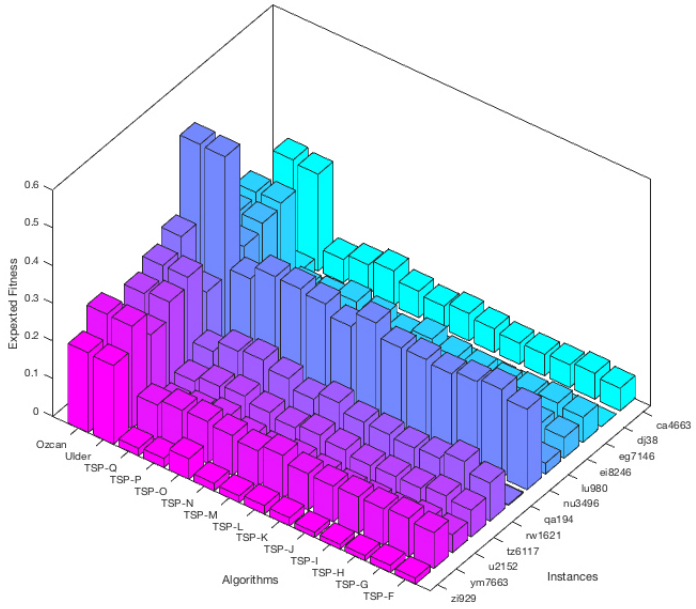


Figure 8.3: A graphical representation of the expected fitness of for the algorithms *TSP-A to TSP-E* and the metaheuristics published by [246, 319]. The instances ranges between 9,152 to 33,708 cities

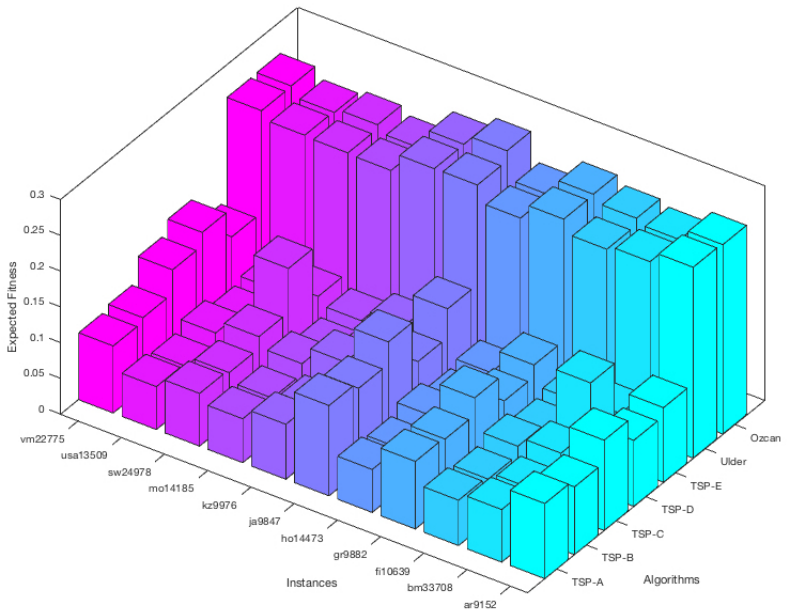
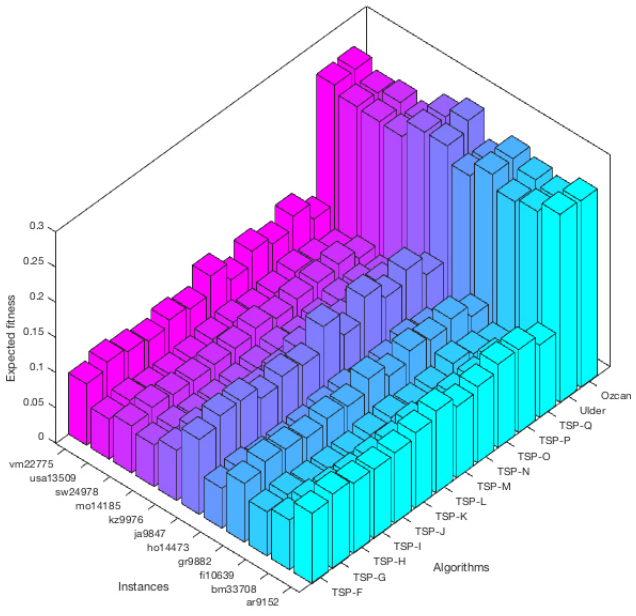


Figure 8.4: A graphical representation of the expected fitness of for the algorithms *TSP-F to TSP-Q* and the metaheuristics published by [246, 319]. The instances ranges between 9,152 to 33,708 cities



- **noCities = 38** : For the exception of solver *TSP-O*, the solvers TSP-[A-Q] have found optima for the instance *dj38*.

- **194** $\leq$ **noCities** $\leq$ **14,473**: Solver TSP-D has found the best tours for these instances. The "TSP-D" tour distribution is likely to be different; the A-measure is often greater than 0.7 (see tables B.22 and B.29). The central tendency (i.e. median and/or arithmetic mean) is generally the lowest (see tables B.18, B.19 and [B.26 - B.37]).

- **noCities** $>$ **14,473**: Solvers *TSP-B, TSP-H* and *TSP-I* have found the best tours of these most challenging instances. Those can be significantly better with a medium effect (see table B.27).

8.1.1.3 Level of disruption

Various mutation operators bring a different level of disruption. First, when a mutation operator swaps randomly two cities (i.e. the *ExchangeMutation* operator), then more suitable tours have been obtained for instances up to 14,000 cities. Secondly, mutation operators that move a position of cities or inverses randomly the order of some cities appear to be more efficient with larger instances (i.e. *InsertionMutation* or *SimpleInversionMutation*).

Larranaga et al. [181] conclusion were confirmed. In contrast, the patterns of primitives were exhaustively tested by human activities with some short TSP instances were used (i.e. $\leq$ 200 cities). Our generative hyper-heuristics have not only found some suitable metaheuristics, but also allow us to comprehend the effect of combining some TSP operators.

8.1.1.4 Metaheuristic design suggestions

Metaheuristics that uses every problem evaluation of a given budget and apply the pattern of primitive *[aMutation, 3_OptLocalSearch, ReplaceLeastFit, SelectElitism]* are more likely to reduce the length of tours to a suitable level. Swapping cities randomly is likely to be more suitable for small instances. More disruptive mutations can be more effective in a large instance.

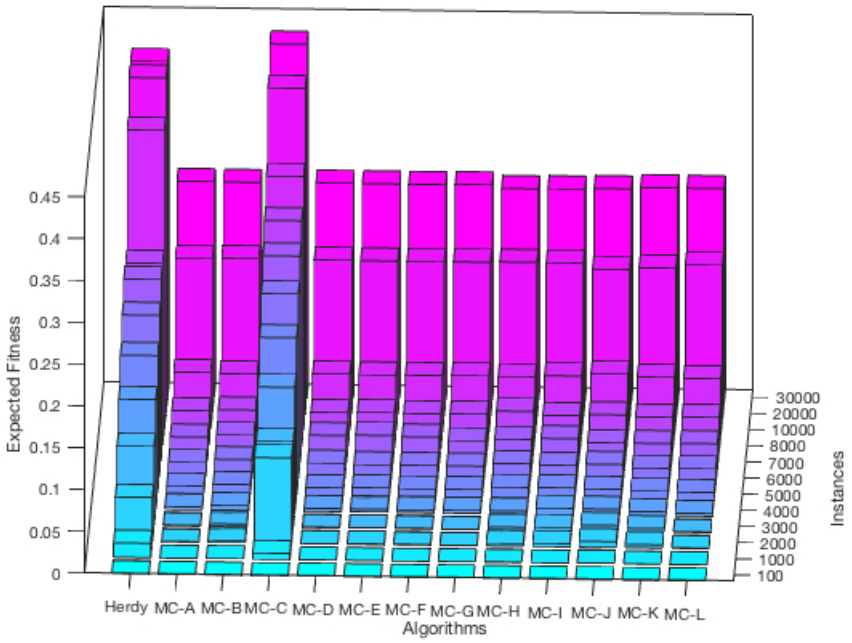
8.1.2 The mimicry problem

8.1.2.1 Fixed number of problem evaluations

Our first validation set samples some instances within the range $100 \leq Length \leq 30,000$. A fixed number of 20,000 problem evaluations was applied during each validation run has used. Some solvers were more scalable than others.

- **Length $\leq$ 1000**: The metaheuristic obtained from the literature (i.e. Herdy [146]) and the generated solvers MC-[A-L] have generally found suitable or optimum imitators (see section 9.2 and figure 8.5).
- **1000 $\leq$ Length $\leq$ 6000**: The best imitators have been found by a reduced set of generated solvers (i.e *MC-A, MD-D, MC-G, MC-E* and *MC-L*). Their distribution is often very similar. The A-measure is very close to 0.5 and the null-hypothesis has been accepted (see section 9.2 and figure 8.5).
- **7000 $\leq$ Length $\leq$ 30000**: The generated solvers *MC-B, MC-H, MC-J, MC-K* have found the best imitators. The expected fitness increases for this set of instances. The fixed number of problem evaluations restrict the possible number of bits that can be corrected, resulting in converging to a high expected fitness (see section 9.2 and figure 8.5).

Figure 8.5: A graphical representation of the expected fitness of each algorithms for the instances ranging between 100 to 30,000 bits.



8.1.2.2 Herdy's ratio

Herdy [146] reported a *(1/1+1)* evolution strategy would required 3,072 generations to correct 250 bits of a 500-bit-long instance. This metaheuristic expected fitness started to rise sharply for instances with more than 1,000 bits; for the generated solvers the imitators quality begun degrading for instances with more than 3,000 bits. At that point the ratio $500\,bits : 3,072\,generations$ stops being respected.

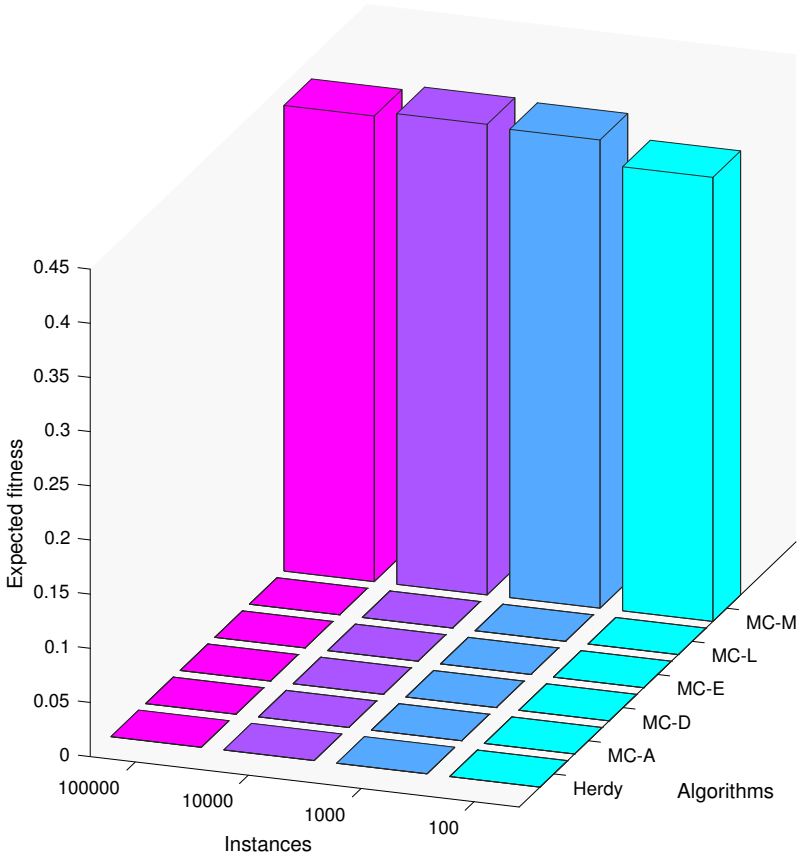
Our implementation of Herdy's metaheuristic has used again some of the mimicry-specific operators. The reported number of generations (i.e 3,072) would therefore apply 6,144 problem evaluations; one problem evaluation for applying *CrossoverUniform* and one for *MutateOneBit* (see table 3.2 and Algorithm A.1 in Appendix A). Therefore, we have expressed again number of problem evaluations required for a certain instance using the expression $Length * [(3072 \times 2)/250]$ (i.e $Length * 24.576$).

8.1.2.3 Variable number of problem evaluations

Our second validation set samples some instances within the range $100 \leq Length \leq 100,000$. Some optimum imitators were found by the solvers *MC-A, MC-D, MC-E* and *MC-L* when the number of problem evaluations was set using Herdy's ratio. Up to 10,000 Herdy's metaheuristics has consistently found some perfect imitators. With larger instances some bits were still not corrected (see tables [B.11 - B.13] and figure 8.6).

The solver *MC-M* failed to find any suitable solutions. This solver was unable to meet the three goals specified in our improved learning objective function. Increasing the number problem evaluations in relation to an instance has brought no improvement (see table 6.20 in section 6.4.2, tables [B.11 - B.13] in Appendix B and figure 8.6).

Figure 8.6: A graphical representation of the expected fitness of each algorithms for the instances ranging between 100 to 100,000 bits.



With both set of validation instances, the most successful metaheuristics have commonly used *mutationOneBitHC* and/or *mutationUniformHC*. These solvers would preserve the number of corrected bits, preventing introducing new errors in an imitator. By repetitively applying these hill-climbers operators the *ReplaceLeastFit* population operator becomes redundant during the search. The population p only needs to be populated again at the end of the search; perhaps one population of mimicry individual may only be required.

8.1.2.4 Metaheuristic design suggestions

Every bit of an imitator can be corrected provided the following conditions are applied.

1. The number of problem evaluations relative to the length of the instance. It should be set to $ProblemEvaluation \leftarrow Length * 24.576$).
2. Only corrected flips are kept.
3. Bits are repetitively and randomly selected and flipped in an imitator.

8.1.3 The nurse rostering problem

8.1.3.1 Two extremes of rosters

The generated solvers have found rosters ranging between two extremes. Some new optima have been found and some other unsuitable rosters too. The mapping between solvers, instances and expected fitness has therefore the greater discrepancy of our three chosen problems; the expected fitness can be approximated between these limits $[-0.8\,..\,2,500]$ (see figures [8.7-8.9]).

Figure 8.7: A graphical representation of the best expected fitness obtained by solvers *NRP-A* - *NRP-J*. The limit approximately varies between $[-0.8..2]$.

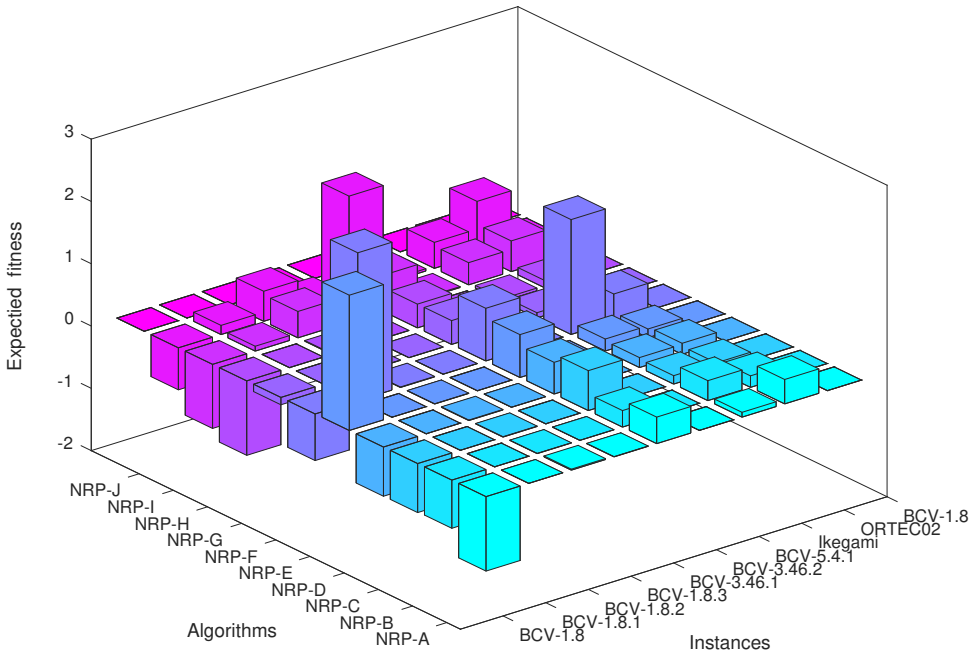


Figure 8.8: A graphical representation of the best expected fitness obtained by solvers *NRP-A* - *NRP-J*. The limit approximately varies between $[0..35]$.

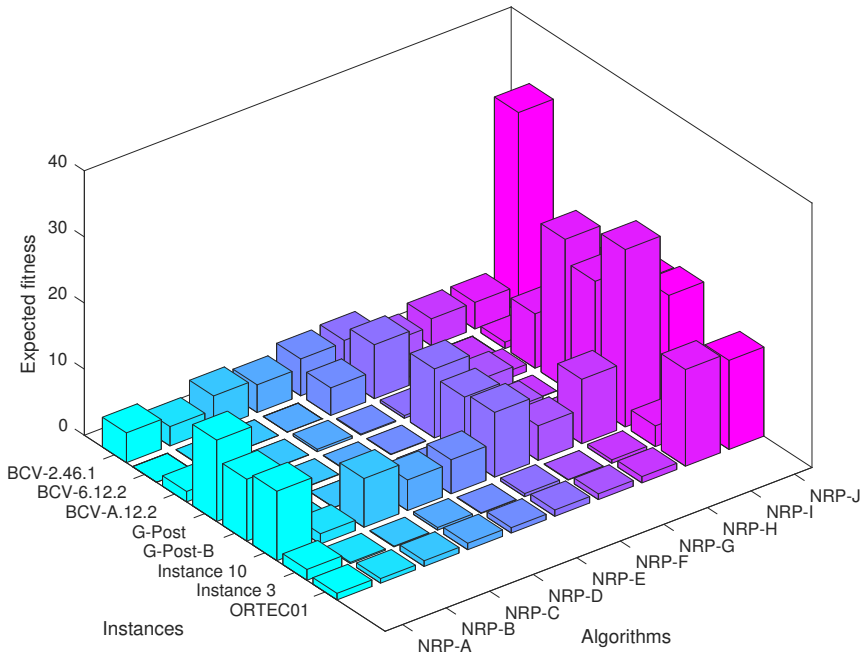
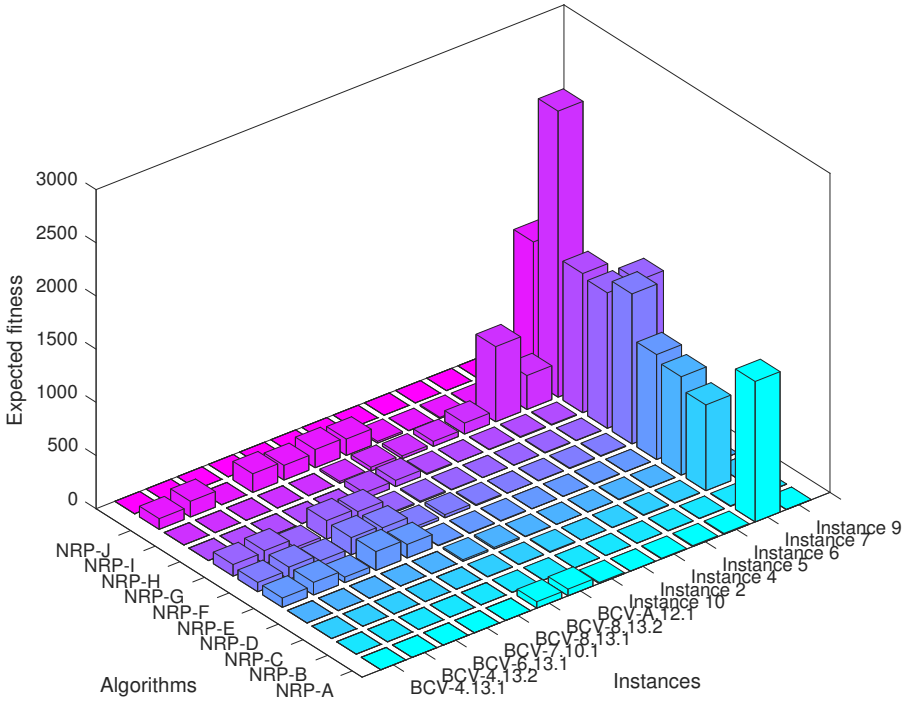


Figure 8.9: A graphical representation of the best expected fitness obtained by solvers *NRP-A* - *NRP-J*. The limit approximately varies between $[0..1, 320]$.



8.1.3.2 Effect of BCV instances on the generated solvers

The BCV instances were initially formulated with the hope of moving forward meta-heuristics research for this problem. Linear programming or selective hyper-heuristics techniques have found a majority of known optima [21, 66, 326].

Many of our generated metaheuristics have successfully found the known optimum, and some new optima have been discovered instance *BCV-1.8.4*. Their medians are therefore the same (see tables [B.39 - B.51]). All these solvers have used a relatively low number of problem evaluations.

8.1.3.3 Effect on more challenging instances on the generated solvers

Integer programming techniques have often solved the remaining benchmarks (i.e. ORTEC01, ORTEC02, G-Post, G-Post-B, Ikegami). Those are considered to challenge more non-deterministic algorithms.

We are pleased that suitable rosters have been found by solvers *NRP-[B-C-D]* and *NRP-H* (see table B.44 in Appendix B). The best-known optima has also been consistently found by solvers *NRP-A, NRP-B, NRP-C, NRP-H* and *NRP-I*.

8.1.3.4 Most scalable generated solvers

Solvers *NRP-B* and *NRP-H* have been the most successful solvers (see tables [B.39 - B.51]). Both measures of centrally tendency solver *NRP-B* have been the lowest for 18 instances out of 30.

Both solvers remove some shifts before applying at least one local search. The patterns of operators satisfy (1) the weekend constraints and (2) an entire work schedule for a nurse. For that reason, the fitness of a roster is reduced. Satisfying these constraints have been particularly useful for some benchmarks with a substantial number of days or nurses (i.e. *instance2, instance3, GPOST-B, Ikegami 3-Shift 1*). All of these algorithms move some shifts to some adjacent days so that the cost of a roster can be lowered.

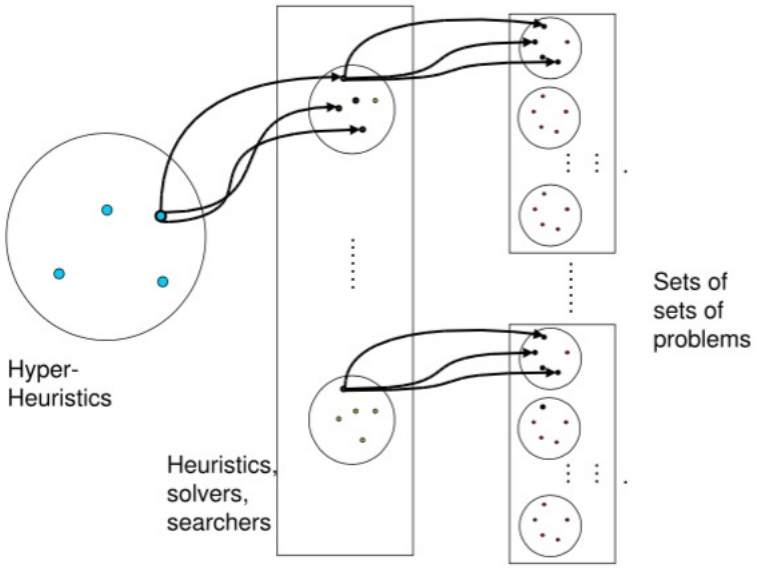
8.1.3.5 Metaheuristic design suggestions

Metaheuristics are likely to find better rosters when some shifts are removed before applying at least one local search and promoting the best roster to a parent solutions.

8.2. Automatic design of metaheuristics

Poli et al [257] represents a hyper-heuristic as a technique that *operates on the [meta]heuristic search space* (see figure 8.10). The framework introduced in chapter 2 has included the hyper-heuristic search as *an algorithm optimisation process*. The solvers are part of the *algorithm domain* and the set of problems in the *problem domain*.

Figure 8.10: An hyper-heuristic searching the multiple areas of the algorithm search space. Each search algorithm has potentially a different set of problems associated to it [257].



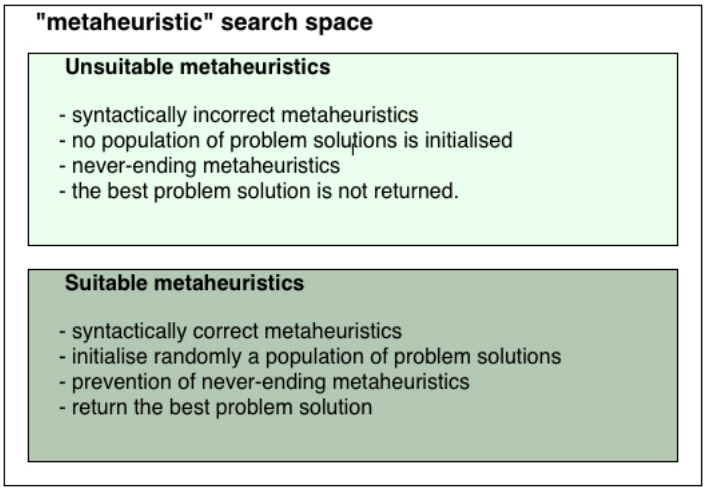
The loose coupling between these three main components has contributed in making use of very little or no knowledge of the other parts; each primary element achieves a single well-defined task. In chapters [5-7] this feature has helped to explore some features that could improve the performance of our graph-based generative hyper-heuristics.

8.2.1 Templates and directed graphs

8.2.1.1 Suitable and unsuitable metaheuristics

Both templates and some directed “acyclic” and “cyclic” graphs encode the generated solvers. The inductive bias has ensured the minimum metaheuristics requirements would be met to prevent the generation of unsuitable solvers. As suggested by Koza [174], the structure of the templates are general enough and problem independent. Known undesirable design aspects are being removed from the fixed part of the algorithm. An extensive section of the algorithm search space can be searched to generate some suitable solvers (see figure 8.11).

Figure 8.11: The metaheuristics design space



8.2.1.2 Syntactically correct metaheuristics

Some grammatical rules have guaranteed the generation of correct nested-loops without specifying a maximum number of nesting level. Examples of solvers with nested loops can be found in solvers *MC-F, MC-K* and *TSP-O* in Appendix 9.2.

The automatic design process has evolved some data flow diagrams. Directed graphs (i.e. acyclic and cyclic) have encoded the evolvable part of a solver. Integer values encode some iterative sequences of some metaheuristics within a string. A feed-forward and feedback mechanism safeguards the connections to valid nodes.

Operators had to be part of a finite set of symbols (i.e. a function and condition set). Each of these syntactic rules introduced in section 4.2 has maintained correct iterative sequences. During the decoding process, each active function gene also represents a syntactically correct line of code.

8.2.1.3 Initialising a population of solutions

We have assumed suitable metaheuristics would start searching the problem fitness landscape by randomly initialising a problem-solution population. Otherwise, the problem fitness landscape cannot be explored. This first step is considered as a necessity [241, 205, 112, 146, 103, 232, 177].

8.2.1.4 Guaranteeing the metaheuristics can terminate

Two termination mechanisms have guaranteed the metaheuristics to terminate. Some templates that only evolve the body of a loop. Koza et al. [174] initially suggested this technique to prevent some iterative algorithms run indefinitely. As a practical necessity, some constraints avoid unending iterations. Some grammatical rules can also make this possible by stating the elements that remain unchanged and the part of the program that is evolved [244, 199, 180, 344, 182, 285].

Some termination criteria have been considered as primitives; some conditions defined externally to a hyper-heuristic similarly as some problem-specific operators. Both sets of primitives were tested before the learning runs so that the metaheuristics would stop.

8.2.1.5 The best problem solution is returned

Two populations of problem solutions were defined in section 3.1.1. Individuals of a population referred as p could survive through the whole search. The templates guarantee this population is updated before its best individual is returned.

8.2.1.6 Adapting the automatic design to a problem domain needs

At the start of our experiments, we made the assumptions a template would remain the same for each problem. We have envisaged the basic "skeleton" of a metaheuristic would be general enough for our three problem domain. As we completed some investigative experiments, it became apparent for the templates to be adapted to the needs of the problem domain. Some problem fitness landscape may be more challenging than other to search for a metaheuristic.

Firstly, the *mimicry problem* fitness landscape was most suited to a minimalistic template; no replacement operators was imposed until the end of the evolution. Secondly, the *nurse rostering problem* has benefited by applying *ReplaceLeastFit* and *SelectEliticism* as the last two operators of the body of any loops. Both templates were amended for this problem. Thirdly, the *traveling salesman problem* has many aspects that can be met from various techniques. Two templates have evolved the body of a loop (see algorithm 5.2 in section 5.3 and algorithm 6.2 in section 6.2.1).

8.2.1.7 Approximating the size of the algorithm fitness landscape

Directed acyclic and cyclic graphs can represent deceptively a large number of possible algorithms. The evolvable part is encoded within a *path of active nodes* or a *sub-graph*. Some pair-wise relationships between elements model the operation flows; both have a start and end. For the exception of a specified maximum graph length, no other restriction was made to the problem operators applied by any solvers. No grammatical or syntactic rule was applied to assume a certain order of operations too.

Directed acyclic graphs restricted the encoding scheme to non-iterative sequences. Counting by-hand the possible number sequences becomes a challenging task rapidly. The number of possible paths rises very quickly in the millions. For example 15 vertices could have approximately 1.81e+29 sequences [2, 268, 216]. In our experiments, a minimum of 100 nodes has been used, so some metaheuristics of substantial sizes could be searched too.

In autoconstructive CGP, the hyper-heuristics reproductive operators are encoded with 10-node-long graphs; this should allow 1.17e+12 possible paths for each type of mutation operators. The size of the algorithm search space remains therefore very substantial.

8.2.1.8 Generative hyper-heuristic design suggestion

The feasibility of a metaheuristic can be guaranteed by a template providing a population of solutions is initialised, and the best solution found during a run is returned. An encoding scheme can enforce some syntactical rules that can be decoded to search the problem fitness landscape. Termination criteria can be implemented in a template or as primitives to ensure a metaheuristic terminates.

8.2.2 Effect of the learning objective functions

A learning objective function associates some solvers and a set of problems (see figure 8.10). This process should reveal the differences present in a problem fitness landscape and an algorithm fitness landscape.

The algorithm search space can be illustrated with the following metaphor; a large mountain range with some high peaks. Metaheuristics with good performances should become the highest points.

8.2.2.1 The no-free-lunch theorem

The state-of-art in evolving EAs [244, 199] was applied in chapter 5. The learning objective function is given in algorithm 5.1 and shows how the no-free-lunch theorem has been used as an inspiration.

Efficient solvers can have a suitable algorithm fitness value computed by a *no-free-lunch* learning objective function. These metaheuristics find some short tours for each instance, lowering the arithmetical mean. The latter can also become skewed to the left. In this case, a generated metaheuristic effectively and solely solve the least challenging learning instance. These metaheuristics may not move away from a local optimum with an increased number of problem evaluations; as demonstrated in chapters [5-6].

We surmised the algorithm search of mimicry solvers was over-fitting. The *no-free-lunch* learning objective function was not sensitive enough to small fluctuations present in the mimicry training set. Additionally, some critical information about NRP solvers were unlikely to be captured. Some roster may become large very quickly and an arithmetic mean could become skewed to the right. Rosters found by our generated metaheuristics have often scored with a more substantial standard deviation and interquartile value (see Appendix B). We surmise the algorithm search was under-fitting NRP solvers.

8.2.2.2 The mean-variance analysis

Another technique has instead calculated a measure of centrality and dispersion, to apply some ideas inspired by the *mean-variance analysis* (see algorithm 6.1). Capturing some clear information about the metaheuristics performance for each learning instance has simulated more accurately the characteristic of scalability. Some expected goals (i.e. *instance goals*) have modelled the "real" performance of the solvers against some realistic aims. In general, the best-generated metaheuristics discussed in section 8.1 have met these conditions. We have observed these metaheuristics would have been rewarded a favourable algorithm fitness value (see figures 6.3 - 6.11 and tables [6.13 - 6.20] in section 6.3).

The logical steps and parametrisation of the second learning objective function have become more general; solvers for several problem domains were suitably assessed. The algorithm fitness value can measure the solver performances based on some set of problem solutions. The hyper-heuristics are now capable of promoting solvers that are likely to be more efficient and scalable. Therefore, the differences existing at the level immediately below have been revealed using a more suitable performance measure [257].

8.2.2.3 Generative hyper-heuristic design suggestion

A learning objective function that incrementally achieves a set of desired results can save a lot of computer resources being used during the algorithm search. A coefficient of variation can capture some clear information that measures the metaheuristics performance with regards to their potential scalability.

8.2.3 Effectiveness of the learning

8.2.3.1 Generalisation

Our supervised learning algorithms have generalised beyond their training sets. Many of our generated metaheuristics have searched the problem fitness landscape efficiently; the discovered solvers have found problem solutions for a wide range of unseen instances (see section 8.1).

At the start of the work, we assumed the elements of our offline learning algorithm would be general enough for each problem domain. Through the completion of this thesis, subtle problem-specific tuning had a positive impact on the algorithm search (see section 8.2). An online learning algorithm has also genetically modified some suitable CGP mutation operators.

8.2.3.2 Effect of the hyper-heuristic reproductive operator

Changes in the hyper-heuristic reproductive operators have aided to explore a broader range of solvers. We believe the added control on the selected genes within a CGP graph has been beneficial. At each hyper-heuristic generation, some active genes have been altered, generating and assessing a new solver.

As a result, some TSP solvers have been discovered again. Some more efficient mimicry solvers have been obtained. A "learnt" CGP mutation was also discovered.

An offline learning algorithm has validated this hyper-heuristic reproductive operator with an "unseen problem". Some NRP solvers have been searched with with more ease. More effective iterative solvers were found; at each generation, a new solver was assessed.

8.2.3.3 Hyper-heuristic design suggestions

An evolution strategy can search more effectively the algorithm search space, providing some active and non-active genes are mutated. Genetically modifying a CGP mutation operator during the algorithm search can aid controlling the selection of genes to be mutated, resulting in exploring the algorithm search space with more precision.

8.3. Comprehensibility metrics

In software engineering, a comprehensible code is a desirable outcome, since it is believed the cost of maintenance can be lowered. To help to identify such code, some specific metrics have been designed to assess the complexity of some programs expressed in a programming language [60, 261, 32, 63].

Some of these metrics have been included in our algorithm domain; they quantify the understandability of some problem-specific solvers. We can therefore enquire into the human understandability of some generated solvers expressed with an imperative pseudo-code. The metaheuristics optimisation processes should change the values of the human understandability metrics introduced in section 2.3.5. Each time some operators or iterations are modified, the vocabulary, length, effort and no of independent paths may become larger or smaller.

For example, some knowledge would be required to maintain and implement some non-deterministic methods. The effort metric of some well-known metaheuristics approximately scores between $[9,000..15,000]$. Luke et al. [205] have expressed these metaheuristics with a set of keywords representing several types of loops, selections, comparisons and assignments. Consequently, the vocabulary and length metrics are quite high, impacting negatively on the effort required to understand these algorithms negatively.

Testing effectively these traditional metaheuristics would require a procedure that considers all the independent paths. Each time a loop or a selection construct is applied a new path is added, increasing the number of independent paths dramatically. These traditional metaheuristics would challenge a programmer to understand its structure and maintain it.

Table 8.1: Software metrics for the traditional metaheuristics as expressed by Luke et al.[205]

Algorithm	No independent paths	Vocabulary	Length	Effort
GA	5	30	72	15,191.73
ES	8	28	65	12,499.12
ILS	5	23	56	9,330.60

8.3.1 Problem-specific solvers

Knowledge about a problem domain and an algorithm representation was required to establish an appropriate communication between programmers-to-computers and computers-to-programmers. The difficulty to "make sense of " these algorithms was then eased dramatically by adopting a set of symbols that a group of programmers and CGP can use (see chapters [3-7]). Otherwise, we would not have been able to critically analyse given pattern of primitives, their performance and the benefits brought to search a problem fitness landscape [169, 38, 60].

Once coded with an imperative pseudo-code, the understandability metrics were computed for each solver of appendix A. The solvers' performance discussed in section 8.1, results reported in chapters [5-7] and appendix B have also been summarised (see tables [8.2-8.4]).

Table 8.2: Software metrics applied to the generated mimicry solvers and Herdy's evolution strategy [146]

Algorithm	No independent paths	Vocabulary	Length	Effort	Class of instances
Herdy [146]	2	24	43	3,450.18	Length $\leq$ 1,000
MC-A	2	25	47	3,928.70	NoOfBits $\leq$ 100,000
MC-B	2	28	65	9,561.83	NoOfBits $\leq$ 10,000
MC-C	2	27	58	7,266.89	Length $\leq$ 1,000
MC-D	2	17	43	3,623.94	NoOfBits $\leq$ 100,000
MC-E	3	27	55	5,135.28	NoOfBits $\leq$ 100,000
MC-F	3	27	55	5,135.28	NoOfBits $\leq$ 10,000
MC-G	2	25	43	3,145.05	NoOfBits $\leq$ 10,000
MC-H	2	25	43	3,145.05	NoOfBits $\leq$ 10,000
MC-I	2	25	51	4,440.69	NoOfBits $\leq$ 10,000
MC-J	2	24	44	3,698.54	NoOfBits $\leq$ 10,000
MC-K	3	28	61	6,798.04	NoOfBits $\leq$ 10,000
MC-L	2	21	39	2,644.45	NoOfBits $\leq$ 10,000
MC-M	2	21	39	2,644.45	Not effective
MC-N	2	26	50	5,327.17	Not effective
MC-O	2	24	46	4,429.07	Not effective

Table 8.4: Software metrics applied to the generated NRP solvers

Algorithm	No independent paths	Vocabulary	Length	Effort	Class of instances
NRP-A	2	21	58	6,106.81	Most BCV instances
NRP-B	2	22	58	7,032.45	Most instances
NRP-C	2	24	63	8,449.89	Most BCV instances
NRP-D	2	22	57	7,897.97	Most BCV instances
NRP-E	2	27	60	7,274.98	Some BCV instances
NRP-F	2	26	59	6,877.68	Some BCV instances
NRP-G	2	27	64	8,277.31	Some BCV instances
NRP-H	2	21	50	5,270.78	Most nstances
NRP-I	2	22	47	5,164.98	BCV instances
NRP-J	2	24	58	6,204.98	Some BCV instances
NRP-K	2	24	56	8,106.21	not effective
NRP-L	2	23	57	7,956.30	not effective
NRP-M	2	25	59	10,215.82	not effective
NRP-N	1	15	29	1,586.20	not effective
NRP-O	2	21	38	3,838.89	not effective

Table 8.3: Software metrics applied to the generated TSP solvers, some metaheuristic written by human-activity, and also some solvers generated with tree-base GP

Algorithm	No independent paths	Vocabulary	Length	Effort	Class of instances
Ozcan [246]	2	21	41	3,781.79	noCities $<$ 38
Ulder [319]	2	25	43	4,351.87	noCities $<$ 38
Ntombela [241]	3	29	60	4,372.18	noCities $\leq$ 247
Loyola-1 [203]	5	19	65	5,245.34	noCities $\leq$ 1,400
Loyola-2 [203]	6	16	51	6,156.00	noCities $\leq$ 1,400
TSP-A	2	26	51	4,794.45	noCities $<$ 38
TSP-B	2	25	49	4,266.54	noCities $>$ 14,473
TSP-C	2	29	57	6,215.86	noCities $<$ 38
TSP-D	2	25	57	6,823.37	noCities $<$ 14,473
TSP-E	3	23	73	12,831.41	noCities $<$ 38
TSP-F	2	25	61	6,161.24	noCities $<$ 38
TSP-G	2	26	66	7,941.86	noCities $<$ 38
TSP-H	2	25	49	3,925.22	noCities $>$ 14,473
TSP-I	2	24	52	5,165.72	noCities $>$ 14,473
TSP-J	2	24	76	10,574.11	noCities $<$ 38
TSP-K	2	25	48	3,677.93	noCities $<$ 38
TSP-L	2	27	66	9,069.47	noCities $<$ 38
TSP-M	2	26	68	8,693.93	noCities $<$ 38
TSP-N	2	27	72	11,214.60	noCities $<$ 38
TSP-O	3	31	71	8,279.61	not effective
TSP-P	2	26	60	8,523.46	noCities $<$ 38
TSP-Q	2	25	60	7,430.17	noCities $<$ 38
TSP-R	2	25	49	4,266.54	not effective
TSP-S	2	25	57	6,215.86	not effective
TSP-T	2	26	61	8,181.64	not effective
TSP-U	3	28	76	21,564.78	not effective
TSP-V	2	22	64	9,479.48	not effective
TSP-W	2	31	80	22,219.57	not effective
TSP-X	2	25	60	6,934.83	noCities $\leq$ 29
TSP-Y	2	25	78	12,878.96	not effective
TSP-Z	2	23	83	21,868.19	not effective

In comparison to the traditional metaheuristics listed table 8.1, a programmer with some good problem domain knowledge could understand with more ease the most effective solvers. Those have often scored a low value for the effort metric (see the algorithms highlighted in green in tables [8.3-8.4]. Their vocabulary and length are often reasonably small. The total number of operators and operands should, therefore, be quite low. In comparison, the metrics are often close to some problem specific metaheuristics reported in the literature (i.e. Herdy [146], Ozcan[246] and Ulder[319]).

8.3.2 Other forms of GP

At the start of this work, only a minority of the literature dedicated to generative hyper-heuristics have been publishing examples of some discovered algorithms; the main focus remains the problem solutions obtained from these techniques. More recent publication has continued with this trend [287, 254, 325, 55, 5, 213].

Many of these have applied a tree-based GP; we anticipate those may be very challenging to comprehend as the program may grow large during the learning phase (see figure 8.12). Unlike [241, 203], no maximum tree depth was specified. Nonetheless, some TSP solvers generated by tree-based GP have more independent paths. Their vocabulary and length can vary and their effort metrics achieve a similar score than the most effective TSP solvers. These results are either driven by using a very large template or the evolution has repetitively applied the same combinations of operands and operators.

Figure 8.12: An example of published algorithm generated by a generative tree-based hyper-heuristics.

```
(- (+ (- (+ (* NBH (- W W)) (- (- (+ (* NBH (*
(- W W) (- W W))) (- (- (* (+ BH BWL) (- W
W)) NBH) (* BH BH))) (- (+ (% H OBH) (+ W
NBH)) (- (* W W) (* BH BH)))) (+ (* BH BH)
(- (+ BH BWL) OBH)))) (- (+ (* BH BH) (- (+
BH BWL) OBH)) (- W (* BH BH)))) (- (* (+ BH
BWL) (- W W)) (+ BH BWL))) (- (+ (+ W BH)
(+ W NBH)) (- (* W W) (* W W))))
```

Binary code, parse trees, and registers have also been used. These chosen EAs have solved a variety of problems that differ from our choices; function optimisation and the Royal-Road problem were examples of use.

Table 8.5: Summary of metaheuristics understandability metrics evolved with a non-graph-based form of genetic programming

Algorithm	No independent paths	Vocabulary	Length	Effort
Oltean et al [245]	1	10	16	141.74
Oltean et al [245]	1	11	24	415.13
Oltean et al [244]	2	20	48	1936.22
Diosan et al [91]	2	19	48	1699.17
Lourenco et al. [199]	2	15	14	187.53
Lourenco et al [202]	2	14	19	651.06
Lourenco et al [202]	2	14	19	651.06
Lourenco et al [202]	2	15	15	234.41
Lourenco et al [202]	2	14	19	651.06
Martin et al. [213]	2	25	67	5744.09

These solvers often score much lower values, than the ones obtained by human activity (see table 8.5). A smaller number of functions, operators and compact grammatical rules may have limited the metaheuristics expressibility; the vocabulary is therefore quite low. Sometimes indices symbolise some problem solutions [91]; this increases the effort metric but can move away from a ”human” programmer expectations.

Martin et al. Martin et al. [213] have matched the score achieved by our learnt metaheuristics. An extensive set of constants and genetic operators was used, but no replacement operator appears to be in the function set. Similarly to the technique used by [203], hybrid metaheuristics employs some mathematical operators alongside some genetic operators. This vocabulary may be unfamiliar to the “human-designers of metaheuristics”; these researchers are more likely to use some genetic operators.

8.3.3 Effect on human understandability metrics

8.3.3.1 Vocabulary

The variables, constants, operators, functions, reserved keywords offer a complete set of symbols that comprises most elements of an imperative programming language. Many of the lines remain unchanged as they are written in a template. For example, the variables p and t, the assignment operator $\leftarrow$, the population operators $SelectElitism()$ and $ReplaceLeastFit()$). These distinct symbols remain constant during the meta-heuristic search; those are now referred as $NoOp_{temp}$ and $NoOperand_{temp}$ (see expression 8.1).

Some others lines were encoded in CGP graphs (iterative and non-iterative). In chapters [5-7] these lines were considered as active function and condition genes. The part evolved is therefore modelled in expression 8.2 in a similar way; $NoOp_{graph}$ and $NoOperand_{graph}$ are likely to change values during the search. Therefore we can write again the variable $NoOp$ and $NoOperand$ (see expressions 8.4 and 8.5).

$$Vocabulary_{temp} \leftarrow NoOp_{temp} + NoOperand_{temp} \tag{8.1}$$

$$Vocabulary_{graph} \leftarrow NoOp_{graph} + NoOperand_{graph} \tag{8.2}$$

$$Vocabulary \leftarrow Vocabulary_{temp} + Vocabulary_{graph} \tag{8.3}$$

$$NoOp \leftarrow NoOp_{temp} + NoOp_{graph} \tag{8.4}$$

$$NoOperand \leftarrow NoOperand_{temp} + NoOperand_{graph} \tag{8.5}$$

$$totOp \leftarrow totOp_{temp} + totOp_{graph} \tag{8.6}$$

$$totOperand \leftarrow totOperand_{temp} + totOperand_{graph} \tag{8.7}$$

8.3.3.2 Length

The Length of a program varies each time the evolve part of metaheuristics varies. The total number of operands and operators used are likely to change. There it would be beneficial to adapt the length metric with one for the template and one for the graphs (see expressions [8.8-8.10]).

$$Length_{temp} \leftarrow totOp_{temp} + totOperand_{temp} \tag{8.8}$$

$$Length_{graph} \leftarrow totOp_{graph} + totOperand_{graph} \tag{8.9}$$

$$Length \leftarrow Length_{temp} + Length_{graph} \tag{8.10}$$

8.3.3.3 Effort

The effort metric is mostly affected by the changes brought by the evolved part of the metaheuristic. This expression has been expressed to illustrate how variables modelling the templates and graphs affect the metric (see expression 8.11). Any variations in the values of $NoOp_{graph}$, $NoOperand_{graph}$, $totOp_{graph}$ and $totOperand_{graph}$ would affect the effort metric too.

$$Effort \leftarrow \frac{(Length \times NoOp \times totOperand)log_2(Vocabulary)}{2(NoOperand)} \tag{8.11}$$

8.3.3.4 No of independent path

The templates impose a certain number of nodes and edges to the metaheuristics. Each line of code is represented by a node, resulting in converting the template and evolving part of the metaheuristics into a graph. Those are now referred as $NoEdges_{temp}$, $NoEdges_{graph}$, $NoNodes_{temp}$ and $NoNodes_{graph}$.

$$NoEdges \leftarrow NoEdges_{temp} + NoEdges_{graph}) \tag{8.12}$$

$$NoNodes \leftarrow NoNodes_{temp} + NoNodes_{graph}) \tag{8.13}$$

$$NoIndependentPath \leftarrow NoEdges - NoNodes + 2 \tag{8.14}$$

8.3.4 Comparison with other techniques

8.3.4.1 Graph-based hyper-heuristics

The understandability metrics can be affected by when a CGP mutation is applied, especially when active genes are mutated. When a hyper-heuristic reproductive operator only alters non-coding genes; the encoded metaheuristics remain the same. The evolution of hyper-heuristics reproductive operators should overcome this undesirable effect. The experiments conducted in chapter 7 have guaranteed some coding and non-coding genes were mutated. As result, a wider range of understandability metrics would have been explored.

Lemma 1

The vocabulary, length and effort metric is only susceptible to changes made to active coding genes.

The termination criteria introduced in section 3.1.2 rely on various variables; some of them may have several conditions too. The values of $NoOperand_{graph}$ and $NoOp_{graph}$ should vary the most when some termination criterion of an active loop header is mutated, or a loop is introduced. As a consequence, some operands, comparisons, boolean and logical operators expressing a termination criterion may be added or removed by iterative CGP; increasing or decreasing these values. This only occurs with when iterations are fully evolved with iterative Cartesian Genetic programming.

Lemma 2

The vocabulary are likely to vary the most when iterations are fully evolved.

Depending on the number of occurrences of an operator in a solver, the value of $NoOp_{graph}$ may vary slightly within the integer range $[-1..1]$. If a new problem-specific operator is introduced, then the lexicon would become larger by one unit. Conversely, when a problem-specific operator is applied once, then $NoOp_{graph}$ would decrease by one unit if it is removed. Otherwise, the number of operands remains unchanged.

Lemma 3

Activating and deactivating nodes can affect the most the length of a metaheuristic.

Independent paths arise when *"goto"* statements are applied in metaheuristics; those are represented by one node with two edges. The solvers rely on such statements to move out of a loop when a termination criterion is met.

When the partial evolution of iterations evolves the body of a loop, a template has the same number of edges and nodes. The equivalence $NoEdges_{temp} = NoNodes_{temp}$ hold setting the number of independent paths to 2 (see equation 8.14). The first path executes the evolved body of a loop and the second path never enter the loop as the termination criterion is met. Activating or deactivating a CGP node would increase or decrease the value of $NoNode_{graph}$ and $NoEdges_{graph}$ by one unit. One node and one edge are added or removed. The equivalence $NoEdges_{graph} = NoNodes_{graph}$ also hold maintaining the number of independent paths to 2 (see equation 8.14).

The templates used to fully evolve iterations has only one independent path. Its number of edges is smaller than the number of nodes (i.e. lines). Each time a loop is added or removed, then the value of $NoEdges_{graph}$ varies by two units and $NoNodes_{graph}$ by one. The number of independent paths increases or decreases by one unit. Both equivalences $NoEdges_{graph} = NoNodes_{graph}$ and $NoEdges_{temp} = NoNodes_{temp}$ do not hold any more.

Lemma 4

Adding or removing an iteration can affect the number of independent paths.

8.3.4.2 Exhaustive search

The body of some loops was enumerated by an exhaustive search; every possible combination of a fixed number of problem-specific operators was exhaustively tested and assessed. Some templates were used again to leave one part of the algorithm unchanged; those were the same as used with a CGP hyper-heuristics. Therefore, the equations [8.1 - 8.14] can suitably model the understandability metrics for this enumeration process.

The vocabulary metric changes during the enumeration process within a range of values. It is assumed all the operands have been introduced by the templates (i.e expression 8.2 becomes $Vocabulary_{graph} \leftarrow NoOp_{graph}$, $NoOperande_{graph} = 0$).

The enumerating process can only change the variable $NoOp_{graph}$. A minimum vocabulary occurs when all the enumerated operators are the same; then $NoOp_{graph} = 1$. When a combination use all distinct operators, the biggest vocabulary metric is reached. In this case, $NoOp_{graph} = NoEnumeratedOp$.

Lemma 5

The enumeration of a fixed number of operator restricts the metaheuristic vocabulary in the range expressed by the inequality

$(Vocabulary_{temp} + 1) \leq Vocabulary \leq (Vocabulary_{temp} + NoEnumeratedOp)$

The body of a loop is fixed by the number of enumerated operators. The $Length_{graph}$ metric should be proportional to the number of enumerated operators. As the variable $NoEnumeratedOp$ becomes larger more operands and operators are applied, affecting the overall length.

The number of operators and operands can vary for each opcode in a given function set. The variables $TotOp_{graph}$ should, therefore, vary between the minimum and maximum numbers of operators in a function set. The same assumption can be made with the number of operands (i.e. $TotOperand_{graph}$). As a result, the minimum value of the $Length_{graph}$ would represent a metaheuristic using a combination of a problem-specific operator with the least number of operators and operands. The maximum value would combine the opcode with the greater length.

Lemma 6
The number of enumerated operators would affect the most the effort to understand a metaheuristics. A decreased length and the vocabulary would promote comprehension, but an increased values in these metrics would raise the barrier of understanding.

8.3.4.3 Selective hyper-heuristics

Metaheuristic produced by selective hyper-heuristics may require a more straightforward representation of the understandability metric. This technique concatenates some problem-specific operators in each generation; the outcome algorithm is an enormous list of problem-specific operators. We, therefore, apply the original definition of the understandability metrics; no template prevents having some constant and variable values during the algorithm optimisation (see expressions [2.24-2.27] given in section 2.3.5).

The majority of the operators included in the function set should be randomly selected during a run. For this reason, it is presumed the value of $NoOp$ increases as asymptote. Until all the operators are selected, the value of $NoOp$ increases by one unit each time a new distinct operator is selected. Then this variable remains constant. We, therefore, believe the equalities [8.15 - 8.16] hold.

$$NoOp = SizeOfFunctionSet \tag{8.15}$$

$$\lim_{iteration \to maxIteration} NoOp = SizeOfFunctionSet \tag{8.16}$$

The number of operands is often modelled similarly as some memory addresses. Vectors store all problem solutions with a unique index, represented as variables. The number of operands becomes $NoOperand = NoProblemSolutions$. It is assumed, all these operands are applied from the first iteration, resulting in remaining unchanged as the number of selected operators become larger.

At the end of a run, the vocabulary metric depends on the number of problem solutions, and the function set size. The vocabulary metric is likely to remain the same if the function set size and the number of problem solutions both remain unchanged.

Lemma 7

The vocabulary of a concatenated list of selected problem operators can be expressed as the sum of the function set size and the number of problem solutions used. The maximum number of operators tends to the size of the function set. The number of operands tends to the number of solutions.

The selective hyper-heuristic would select an operator and two operands [307]. As the selective hyper-heuristic progresses the length metric increases linearly; $totOp_{it} \leftarrow totOp_{it-1} + 2$ and $totOperand_{it} \leftarrow totOperand_{it-1} + 3$ grows larger each type an operator is randomly selected and applied; increasing at the same time the effort metric.

The effort values of very short metaheuristics runs in the thousand very quickly (see tables[8.3-8.4]). We would assume these concatenated list of operators obtained from the selective hyper-heuristics would grow very quickly towards ∞. It should reflect adequately the enormous barrier brought to comprehension. An automated analytical tool would need to be written to analyse the frequency of the operators instead. The focus has shifted toward toward computer understandability instead of human understandability.

Lemma 8
The length and the effort metric increases continually during the execution of the selective hyper-heuristics. Their values are expected to move toward ∞.

8.3.5 Discussion

A computer can search for algorithms, but many may challenge human understanding. The space of automatically generated algorithms is much larger than the space of humanly-designed algorithms and is likely to contain many unfamiliar algorithms [270]. For example, algorithms encoded in some data structures appear to be very different than the one a human would design.

Our set of symbols has been simplified to the context of some specific problem domains, metaheuristics and CGP. With the help of some templates, directed acyclic graphs and directed graphs have produced some comprehensible solvers; the understandability metrics score very closely those computed by some "humanly-written" metaheuristics.

The idea of computers communicating algorithms to programmers is not new. Newell et al. [239] stated in their seminal paper *"artificial intelligence must be concerned with how symbol systems must be organised to behave intelligently"*. Lowering the barriers of understandability is a step forward in this direction.

Not all the discovered patterns of primitives have found some suitable solutions. Some ineffective algorithms have also scored some low understandability metrics. Therefore inspecting automatically-designed algorithms become a necessary step to assist in the validation process.

8.4. Conclusion

This chapter has critically analysed the effect of our experiments on the common and essential elements of the optimisation of metaheuristics; (1) *the problem domain*, (2) *the metaheuristic domain* and (3) *the metaheuristic optimisation process*. Some comparisons with the state-of-the-art have positioned our techniques favourably. In general, near-optimum found by our discovered solvers were nearer the optimum solutions than a selective hyper-heuristic or metaheuristics written by human activities.

Our CGP hyper-heuristics have found some longer algorithms with fewer metaheuristic evaluations. An exhaustive search would have become infeasible very quickly. Also, the generated metaheuristics were human-understandable; some known and unknown patterns of problem operators were identified. Next chapter will conclude this thesis. Some recommendations and future work are discussed.

Chapter 9. Conclusion

Contents

9.1 Recommendations 225

9.2 Future work 226

Designing effective metaheuristics can be difficult and time-consuming. Non-deterministic problem-specific operators may disrupt excessively or insufficiently some problem solutions. Therefore, searching a problem fitness landscape can become ineffective for some instances and finding sufficiently good solutions of computational hard problems across a set of instances may not be guaranteed.

Some combinations of problem-specific operators can successfully produce a desired outcome when the weaknesses of certain operators are balanced with the strengths of others. Indeed, studying metaheuristics and their performance would result in understanding how certain patterns of problem-specific operators may be more effective than others. The hyper-heuristics community (i.e selective and generative) largely focus on the problem solutions, instead of studying the metaheuristics obtained from their algorithm optimisation techniques. Without controlling the size of a metaheuristic, some algorithms can become unfamiliar and unreasonably challenge the human intellect.

We argue selective hyper-heuristics may not be suited for this purpose; selected operators are concatenated while search a problem fitness landscape. Tree-based GP can also represent large algorithms, without using a method to control bloat. Extracting some generated metaheuristics and expressing them in a language that is more comprehensible to humans may become a challenging task. This could prevent a more effective way for programmers and computer to communicate with each other. We believe it would be desirable if not only the programmers could communicate to the computer, but the reverse occurs too.

We have shown some domain knowledge and a graph-based GP can discover comprehensible and effective metaheuristics. We argue our generated solvers are expressed similarly than the ones that have been designed by human activities. A lists some generated metaheuristics were decoded and expressed again with a speudocode very close to an imperative programming language.Some well-known software metrics quantify the vocabulary, length and effort to a comparable amount of those written by human activity. These results reflect appropriately these stochastic methods require some specialist knowledge to be designed, implemented and validated.

Generative hyper-heuristics to date has mainly concentrated on evolving the body of a loop or have applied some hard limits to control the halting problem. We argue the techniques which allow to take advantage of the characteristics of directed graphs will be essential to move forward the generation of deterministic and non-deterministic algorithms, without imposing hard limits.

We have shown some improvements made to CGP can evolve efficaciously some metaheuristics. We have discussed how a measure of centrality and dispersion is beneficial to evaluate a metaheuristic. A desired state of scalability is more suitably modelled; the means for an end can be measured more effectively with a set of goals. We have also demonstrated an online hyper-heuristic that genetically improves its reproductive operator during the metaheuristic search can bring some advantages.

The failure of applying this technique to a scheduling problem with a complex set of expected constraints is disappointing, but expected. It was partially anticipated this online hyper-heuristic is general enough for our three problem domains. If more time resources would be available then the metaheuristic search needs would be met more effectively. This problem is a general one. Nonetheless, we have validated the performance of a genetically-improved hyper-heuristic reproductive operator to evolve some solvers for this challenging scheduling problem. This technique has brought some concepts of cross-domain selective hyper-heuristics, genetic improvements, and autoconstructivism together.

9.1. Recommendations

The recommendations made by [78, 222, 237, 302, 104] remain sound. Some practical suggestions can be made.

1. The problem fitness values should indicate the distance away from the instance known optima. This meta-information compares automatically against the state-of-the-art in a meaningful and a general manner across every instance and problem domains.

2. The interpretation of any algorithms should be kept separated from the algorithm optimisation process and the problem domain. A higher cohesion can enhance swapping some elements more efficiently; their effect on the algorithm and problem domain can be studied more easily. Code re-use, increased maintainability and fewer operations should reduce the coding time.

3. Diagnosing issues with a learning algorithm requires a good domain knowledge.

   (a) Testing individually each problem operators should help to identify their effect on problem solutions. These investigative experiments can help recognising the level of disruption brought by an operator. Generated solvers can be analysed with an intensified problem domain knowledge.

   (b) Templates can be written with more problem domain knowledge.

   (c) Recognising when a similar problem fitness value for each learning instance may suggest the metaheuristic search lack of sensitivity to fluctuations between learning instances.

4. Hyper-heuristics reproductive operators should alter some coding and non-coding genes. This method optimises the use of resources.

5. Recording the evolved metaheuristics can help establish whether the generative hyper-heuristic is optimising suitably the solvers.

   (a) Studying the history of generated metaheuristics can validate whether the assumptions modelled in a learning objective function are suitable.

(b) Mapping the evolved hyper-heuristic reproductive operators during an online hyper-heuristic search helps to determine some possible improvements in an offline learning algorithm across different problem domains.

9.2. Future work

Many interesting questions arise from this thesis. Further investigations could explore the benefits and clarify in which circumstances the techniques introduced to CGP can be the most appropriate. Examples include:

- Digital circuits
- Optimisation of assembly code
- Numerical analysis
- Protein-structure prediction
- Evolution of branch-and-bound algorithm for solving problems with a high-level of constraints.
- Vehicle routing problem
- Other real-world problems

However, the concept of evolving some hyper-heuristics reproductive operators that are general enough to search algorithms across many different problem and algorithm domains effectively is an obvious step. It would also be interesting to extend iterative CGP with the ability to evolve algorithms with their variables as well as their selections.

For a very long time researchers have pursued the goal that machines could learn to solve a problem for which they were not given precise methods. At the start of this work, hyper-heuristics was branded as an emerging methodology that would automate the learning of heuristics. During the completion of this thesis, interest in selective hyper-heuristic appear to have shifted towards a generative form of hyper-heuristics. A consequence is a generalisation that a "heuristic" can apply deterministic and non-deterministic methods. The literature has recently adopted the terminology *algorithm synthesis* or *exploration of the automated design search space*.

The concept of embedding fully human understandability within a generative framework should also be considered. Otherwise, the generated (meta-)heuristics are likely to remain in a black box. This could bring together the concept of a "learning machine" and "software engineering" a bit closer. Then perhaps the complete generation of a program may become a step closer.

Very little progress has been made to develop some theoretical understanding of many hyper-heuristics approaches. Some of them appear to focus on one specific domain and some engineering aspects. Many research communities should share some common benchmarks so that their techniques could be compared more efficiently against other machine learning techniques. Then perhaps we would be able to deepen the theoretical knowledge and increase collaboration.

Appendix A: Algorithms

The three subsequent sections provide all the algorithms mentioned in this thesis. Three forms of designs methods were used to generate these algorithms:

1. A CGP hyper-heuristic (see chapters [5-7])
2. A tree-based GP hyper-heuristics (as documented in the literature)
3. Some humanly-designed algorithms (as reported in the literature)

In addition to the problem-specific operators introduced in chapter 3, the reserved keywords **function**, **while**, **do**, **end** and **return** are also used to express our solvers. Finally, the comparison operators $\leq$ and $>$, the logical operator **or** and assignment operator $\leftarrow$ have been included in the language or vocabulary.

The Mimicry Problem

This section provides the mimicry solvers obtained either from the literature or our experiments.

Algorithm A.1. : **Mimicry solver Herdy** (Herdy(1991) [146])

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
4:         t ← SelectElitism(p)
5:         t ← CrossoverUniform(t)
6:         t ← MutateOneBit(t)
7:         p ← ReplaceLeastFit(p,t)
8:     end while
9:     return Best(p)
10: end function
```

Algorithm A.2. : **Mimicry solver A** (MC-A) was discovered from the experiments described in section 6.3

```
 1: function FINDSOLUTION(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
 5:         t ← MutateOneBitHC(t)                    ▷ start generated code
 6:         t ← MutateUniformHC(t)
 7:         t ← MutateUniformHC(t)
 8:         t ← MutateUniformHC(t)                   ▷ end generated code
 9:     end while
10:     p ← ReplaceLeastFit(p,t)
11:     return Best(p)
12: end function
```

Algorithm A.3. **Mimicry solver B** (MC-B) was discovered from the experiments described in section 6.3

```
 1: function FINDSOLUTION(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
 5:         t ← MutateOneBitHC(t)                    ▷ start generated code
 6:         t ← MutateOneBitHC(t)
 7:         p ← ReplaceLeastFit(p, t)
 8:         t ← SelectElitism(p)
 9:         t ← CrossoverOnePoint(p, t)
10:         t ← CrossoverTwoPoints(p, t)
11:         t ← MutateUniformHC(t)
12:         t ← MutateUniformSubSequenceHC(t)        ▷ end generated code
13:     end while
14:     p ← ReplaceLeastFit(p,t)
15:     return Best(p)
16: end function
```

Algorithm A.4. **Mimicry solver C** (MC-C) was discovered from the experiments described in section 6.3

```
 1: function FINDSOLUTION(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
 5:         t ← CrossoverUniform(p, t)               ▷ start generated code
 6:         t ← MutateUniformHC(t)
 7:         p ← ReplaceLeastFit(p, t)
 8:         t ← SelectElitism(p)
 9:         t ← MutateOneBitHC(t)
10:         t ← CrossoverOnePoint(p, t)              ▷ end generated code
11:     end while
12:     p ← ReplaceLeastFit(p,t)
13:     return Best(p)
14: end function
```

Algorithm A.5. Mimicry Solver D (MC-D) was discovered from the experiments described in section 6.5

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                    ▷ start generated code
5:         t ← MutationOneBitHC(t)
6:         t ← MutationOneBitHC(t)
7:         t ← MutationOneBitHC(t)
8:     end while                                        ▷ end generated code
9:     p ← ReplaceLeastFit(p,t)
10:    return Best(p)
11: end function
```

Algorithm A.6. Mimicry Solver E (MC-E) was discovered from the experiments described in section 6.5

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do   ▷ start generated code
5:         t ← MutationOneBit(t)
6:         while EvalCount ≤ MaxEvals or IsBetter(1) do
7:             t ← MutationOneBitHC(t)
8:         end while
9:     end while                                        ▷ end generated code
10:    p ← ReplaceLeastFit(p,t)
11:    return Best(p)
12: end function
```

Algorithm A.7. Mimicry Solver F (MC-F) was discovered from the experiments described in section 6.5

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do   ▷ start generated code
5:         t ← CrossoverOnePoint(p, t)
6:         while EvalCount ≤ MaxEvals or IsBetter(1) do
7:             t ← MutationOneBitHC(t)
8:         end while
9:     end while                                        ▷ end generated code
10:    p ← ReplaceLeastFit(p,t)
11:    return Best(p)
12: end function
```

Algorithm A.8. Mimicy Solver G (MC-G) was discovered from the experiments described in section 7.2.1

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← MutationOneBitHC(t)                    ▷ start generated code
6:         t ← MutateUniformHC(t)                     ▷ end generated code
7:     end while
8:     p ← ReplaceLeastFit(p,t)
9:     return Best(p)
10: end function
```

Algorithm A.9. Mimicry Solver H (MC-H) was discovered from the experiments described in section 7.2.1

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← MutationOneBitHC(t)                    ▷ start generated code
6:         t ← MutateUniformSubSequenceHC(t)          ▷ end generated code
7:     end while
8:     p ← ReplaceLeastFit(p,t)
9:     return Best(p)
10: end function
```

Algorithm A.10. Mimicry Solver I (MC-I) was discovered from the experiments described in section 7.2.1

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← MutationOneBitHC(t)                    ▷ start generated code
6:         t ← MutationOneBitHC(t)
7:         t ← MutateUniformSubSequenceHC(t)
8:         t ← MutateUniformSubSequenceHC(t)          ▷ end generated code
9:     end while
10:    p ← ReplaceLeastFit(p,t)
11:    return Best(p)
12: end function
```

Algorithm A.11. Mimicry Solver J (MC-J) was discovered from the experiments described in section 7.2.1

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do        ▷ start generated code
5:         t ← MutateUniformSubSequenceHC(t)
6:         p ← ReplaceLeastFit(p, t)
7:     end while                                             ▷ end generated code
8:     p ← ReplaceLeastFit(p,t)
9:     return Best(p)
10: end function
```

Algorithm A.12. Mimicry Solver K (MC-K) was discovered from the experiments described in section 7.2.1

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do        ▷ end generated code
5:         t ← MutationOneBitHC(t)
6:         Limit ← initialiseNewLimit(MaxEvals, EvalCount)
7:         while EvalCount ≤ Limit or p.fitness > goal do
8:             t ← MutationOneBitHC(t)
9:         end while
10:        t ← MutateUniformSubSequenceHC(t)
11:    end while                                             ▷ end generated code
12:    p ← ReplaceLeastFit(p,t)
13:    return Best(p)
14: end function
```

Algorithm A.13. Mimicry Solver L (MC-L) was discovered from the experiments described in section 7.2.1

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                         ▷ end generated code
5:         t ← MutationOneBitHC(t)
6:         t ← MutationOneBitHC(t)
7:     end while                                             ▷ end generated code
8:     p ← ReplaceLeastFit(p,t)
9:     return Best(p)
10: end function
```

Algorithm A.14. Mimicry solver M (MC-M) was discovered from the experiments described in section 6.3. This metaheuristic is ineffective; no further analysis is completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← MutateUniformSubSequenceHC(t)          ▷ start generated code
6:         t ← CrossoverTwoPoints(p, t)
7:         t ← MutateUniformVariableRate(t)
8:         t ← MutateUniformSubSequenceHC(t)
9:         t ← MutateUniformSubSequenceHC(t)          ▷ end generated code
10:    end while
11:    p ← ReplaceLeastFit(p,t)
12:    return Best(p)
13: end function
```

Algorithm A.15. Mimicry solver N (MC-N) was discovered from the experiments described in section 6.3. This metaheuristic is ineffective; no further analysis is completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← MutateUniformSubSequenceHC(t)          ▷ start generated code
6:         t ← CrossoverTwoPoints(p, t)
7:         t ← MutateUniformVariableRate(t)
8:         t ← MutateUniformSubSequenceHC(t)
9:         t ← MutateUniformSubSequenceHC(t)          ▷ end generated code
10:    end while
11:    p ← ReplaceLeastFit(p,t)
12:    return Best(p)
13: end function
```

Algorithm A.16. Mimicry solver O (MC-O) was discovered from the experiments described in section 6.5. This metaheuristic is ineffective; no further analysis is completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     Limit ← RandomlyGetNoOfEval()              ▷ start generated code
5:     while EvalCount ≤ Limit or p.fitness > goal do
6:         t ← MutateOneBit(p, t)
7:         t ← MutateUniformVariableRate(t)
8:         t ← MutateUniformSubSequenceHC(t)
9:     end while                                  ▷ end generated code
10:    p ← ReplaceLeastFit(p,t)
11:    return Best(p)
12: end function
```

The Traveling Salesman Problem

This section provides some TSP solvers obtained from different sources. Algorithms A.17 and A.18 were obtained from the literature; those have been written by human activities. Algorithms A.45 and A.46 were automatically designed by a tree-based GP technique. Those incrementally construct tours instead of using a metaheuristic to search the TSP fitness landscape. Solver TSP[A-Q] were obtained by our experiments.

Algorithm A.17. : **TSP Solver Ulder (Ulder(1991)** [319]

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     while p.fitness > 0 or IsBetter(p,5) do
4:         t ← SelectionElitism(p)
5:         t ← VotingRecombinationCrossover(t)
6:         t ← 2-OptLocalSearch(t)
7:         p ← ReplaceLeastFit(t,p)
8:     end while
9:     return Best(p)
10: end function
```

Algorithm A.18. : **TSP Solver Ozcan (Ozcan(2004)** [247]

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     while EvalCount ≤ MaxEvals do
4:         t ← SelectionElitism(p)
5:         t ← InsertionMutation(t)
6:         t ← Order-BasedCrossover(t)
7:         t ← SimpleInversionMutation(t)
8:         p ← ReplaceLeastFit(t, p)
9:     end while
10:    return Best(p)
11: end function
```

Algorithm A.19. : **TSP Solver A** (TSP-A) was discovered from the experiments described in section 5.3

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectionElitism(p)
6:         t ← InsertionMutation(t)                    ▷ start generated code
7:         t ← Order − BasedCrossover(t)
8:         t ← 3 − OptLocalSearch(t)
9:         p ← ReplaceLeastFit(t, p)                   ▷ end generated code
10:    end while
11:    return Best(p)
12: end function
```

Algorithm A.20. : **TSP Solver B** (TSP-B) was discovered from the experiments described in section 5.3

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectElitism(p)
6:         t ← ExchangeMutation(t)                     ▷ start generated code
7:         t ← 3 − OptLocalSearch(t)
8:         p ← ReplaceLeastFit(t, p)                   ▷ end generated code
9:     end while
10:    return Best(p)
11: end function
```

Algorithm A.21. : **TSP Solver C** (TSP-C) was discovered from the experiments described in section 5.3

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectElitism(p)
6:         t ← ExchangeMutation(t)                     ▷ start generated code
7:         t ← Best − 2 − OptLocalSearch(t)
8:         t ← ExchangeMutation(t)
9:         t ← 3 − OptLocalSearch(t)
10:        p ← ReplaceLeastFit(t, p)                   ▷ end generated code
11:    end while
12:    return Best(p)
13: end function
```

The Nurse Rostering Problem

This section provides all the solvers obtained by our experiments.

Algorithm A.22. TSP Solver D (TSP-D) was discovered from the experiments described in section 5.4

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEval do                    ▷ start generated code
5:         t ← 3 − OptLocalSearch(t)
6:         p ← Restart(p)
7:         p ← ReplaceLeastFit(t, p)
8:         t ← SelectElitism(p)
9:         t ← ExchangeMutation(t)                     ▷ end generated code
10:    end while
11:    p ← ReplaceLeastFit(t, p)
12:    return Best(p)
13: end function
```

Algorithm A.23. TSP Solver E (TSP-E) was discovered from the experiments described in section 5.4

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEval/2 do                  ▷ start generated code
5:         t ← 3 − OptLocalSearch(t)
6:         while EvalCount ≤ MaxEval/2 do
7:             t ← Best2 − OptionLocalSearch(t)
8:             t ← ExchangeMutation(t)
9:             t ← 3 − OptionLocalSearch(t)
10:            p ← replaceLeastFit(t, p)
11:            t ← SelectElitism()
12:        end while
13:        t ← SimpleInversionMutation(t)
14:    end while
15:    t ← 3 − OptionLocalSearch(t)
16:    t ← OrderBaseCrossover(t)                       ▷ end generated code
17:    p ← ReplaceLeastFit(t, p)
18:    return Best(p)
19: end function
```

Algorithm A.47. : **NRP solver A** (NRP-A) was discovered from the experiments described in section 6.3

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← GreedyVariableDepthLocalSearch(t)
5:     while EvalCount ≤ MaxEvals do
6:         t ← SmallGreedyRuinRecreate(t)              ▷ start generated code
7:         t ← SimpleGreedyRuinRecreate(t)
8:         p ← ReplaceLeastFit(t, p)
9:         t ← SelectElitism(t)
10:        t ← GreedyVariableDepthLocalSearch(t)       ▷ end generated code
11:        p ← ReplaceLeastFit(t, p)
12:        t ← SelectElitism(p)
13:    end while
14:    return Best(p)
15: end function
```

Algorithm A.24. : **TSP Solver F** (TSP-F) was discovered from the experiments described in section 6.3

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← 3-Opt-LocalSearch(t)
5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
6:         t ← 3 − OptLocalSearch(t)                    ▷ start generated code
7:         p ← ReplaceLeastFit(t)
8:         t ← SelectElitism(p)
9:         t ← ExchangeMutation(t)
10:        t ← 3 − OptLocalSearch(t)                    ▷ end generated code
11:        p ← ReplaceLeastFit(t, p)
12:        t ← SelectElitism(p)
13:    end while
14:    return Best(p)
15: end function
```

Algorithm A.25. : **TSP Solver G** (TSP-G) -was discovered from the experiments described in section 6.3

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← 3-Opt-LocalSearch(t)
5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
6:         t ← Best2 − OptLocalSearch(t)                ▷ start generated code
7:         t ← Best2 − OptLocalSearch(t)
8:         t ← InsertMutationt(t)
9:         t ← 3 − OptLocalSearch(t)
10:        p ← ReplaceLeastFit(t)
11:        t ← SelectElitism(p)                         ▷ end generated code
12:        p ← ReplaceLeastFit(t,p)
13:        t ← SelectElitism(p)
14:    end while
15:    return Best(p)
16: end function
```

Algorithm A.26. : **TSP Solver H** (TSP-H) - was discovered from the experiments described in section 6.3

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← 3-Opt-LocalSearch(t)
5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
6:         t ← SimpleInversionMutation(t)               ▷ start generated code
7:         t ← 3 − OptLocalSearch(t)                    ▷ end generated code
8:         p ← ReplaceLeastFit(t,p)
9:         t ← SelectElitism(P)
10:    end while
11:    return Best(p)
12: end function
```

Algorithm A.27. TSP Solver I (TSP-I) was discovered from the experiments described in section 6.5

```
 1: function FindSolution(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do        ▷ start generated code
 5:         p ← ReplaceLeastFit(t, p)
 6:         t ← SelectElitism(p)
 7:         t ← InsertionMutation(t)
 8:         t ← 3 − OptLocalSearch(t)
 9:     end while                                             ▷ end generated code
10:     p ← replaceLeastFit(t, p)
11:     return Best(p)
12: end function
```

Algorithm A.28. TSP Solver J (TSP-J) was discovered from the experiments described in section 6.5

```
 1: function FindSolution(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do        ▷ start generated code
 5:         t ← InsertionMutation(t)
 6:         while EvalCount ≤ MaxEval Or IsBetter(1) do
 7:             p ← ReplaceLeastFit(t, p)
 8:             t ← SelectElitism(p)
 9:             t ← 3 − OptLocalSearch(t)
10:         end while
11:         t ← 3 − OptLocalSearch(t)
12:         p ← replaceLeastFit(t, p)
13:         t ← SelectElitism(p)                              ▷ end generated code
14:     end while
15:     p ← ReplaceLeastFit(t, p)
16:     return Best(p)
17: end function
```

Algorithm A.29. TSP Solver K (TSP-K) was discovered from the experiments described in section 6.5

```
 1: function FindSolution(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     Limit ← RandomlyGetNoOfEval()                         ▷ start generated code
 5:     while EvalCount ≤ Limit do
 6:         t ← 3 − OptLocalSearch(t)
 7:         p ← ReplaceLeastFit(t, p)
 8:         t ← SelectElitism(t)
 9:         t ← ExchangeMutation(t)                           ▷ end generated code
10:     end while
11:     p ← ReplaceLeastFit(t, p)
12:     return Best(p)
13: end function
```

Algorithm A.30. : **TSP Solver L** (TSP-L) was discovered from the experiments described in section 7.2

```
 1: function FindSolution(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     t ← 3-Opt-LocalSearch(p)
 5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
 6:         t ← ExchangeMutation(t)                    ▷ start generated code
 7:         t ← 2 − OptLocalSearch(t)
 8:         t ← 3 − OptLocalSearch(t)
 9:         t ← OrderBasedCrossover(t)
10:         p ← ReplaceLeastFit(t, p)
11:         t ← SelectElitism(p)                       ▷ endgenerated code
12:         p ← ReplaceLeastFit(t,p)
13:         t ← SelectElitism(p)
14:     end while
15:     return Best(p)
16: end function
```

Algorithm A.31. : **TSP Solver M** (TSP-M) was discovered from the experiments described in section 7.2

```
 1: function FindSolution(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     t ← 3-Opt-LocalSearch(p)
 5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
 6:         p ← ReplaceLeastFit(t, p)                  ▷ start generated code
 7:         t ← SelectElitism(p)
 8:         t ← 2 − OptLocalSearch(t)
 9:         t ← ExchangeMutation(t)
10:         t ← 3 − OptLocalSearch(t)
11:         t ← 3 − OptLocalSearch(t)                  ▷ end generated code
12:         p ← ReplaceLeastFit(t,p)
13:         t ← SelectElitism(p)
14:     end while
15:     return Best(p)
16: end function
```

Algorithm A.32. : **TSP Solver N** (TSP-N) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← 3-Opt-LocalSearch(p)
5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
6:         t ← Best2 − OptLocalSearch()                ▷ start generated code
7:         t ← ExchangeMutation(t)
8:         t ← 3 − OptLocalSearch(t)
9:         p ← ReplaceLeastFit(t, p)
10:        t ← SelectElitism(p)
11:        t ← OrderBasedCrossover(t)
12:        t ← PartiallyMapCrossover(t)                ▷ end generated code
13:        p ← ReplaceLeastFit(t,p)
14:        t ← SelectElitism(p)
15:    end while
16:    return Best(p)
17: end function
```

Algorithm A.33. : **TSP Solver O** (TSP-O) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     Limit1 ← RandomlyGetNoOfEval()                  ▷ start generated code
5:     while EvalCount ≤ Limit1 do
6:         t ← 3 − OptLocalSearch(t)
7:         Limit2 ← RandomlyGetNoOfEval(t)
8:         while EvalCount ≤ Limit2 or IsBetter(noEval) do
9:             t ← 3 − OptLocalSearch(t)
10:            p ← ReplaceLeastFit(t, p)
11:            t ← SelectElitism(t)
12:            t ← SimpleInversionMutation(t)
13:            t ← Best2 − OptLocalSearch(t)
14:        end while
15:    end while                                       ▷ end generated code
16:    p ← ReplaceLeastFit(t, p)
17:    return Best(p)
18: end function
```

Algorithm A.34. : **TSP Solver P** (TSP-P) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do        ▷ start generated code
5:         t ← Best2 − OptLocalSearch(t)
6:         p ← ReplaceLeastFit(t, p)
7:         t ← SelectElitism(p)
8:         t ← InsertMutation(t)
9:         t ← 3 − OptLocalSearch(t)
10:        t ← SimpleInversionMutation(t)
11:        t ← 3 − OptLocalSearch(t)                         ▷ end generated code
12:    end while
13:    p ← ReplaceLeastFit(t, p)
14:    return Best(p)
15: end function
```

Algorithm A.35. : **TSP Solver Q** (TSP-Q) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEval or p.fitness > 0 do         ▷ start generated code
5:         t ← 3_OptLocalSearch(t)
6:         p ← ReplaceLeastFit(t, p)
7:         t ← SelectElitism(p)
8:         t ← OrderBasedCrossover(t)
9:         t ← ExchangeMutation(t)
10:        t ← 3_OptLocalSearch(t)                           ▷ end generated code
11:    end while
12:    p ← ReplaceLeastFit(t, p)
13:    return Best(p)
14: end function
```

Algorithm A.36. : **TSP Solver R (TSP-R)** was discovered from the experiments described in section 5.3. This metaheuristic is ineffective; no further analysis is completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectElitism(p)
6:         t ← 3 − OptLocalSearch(t)                         ▷ start generated code
7:         t ← InsertionMutation(t)
8:         p ← ReplaceLeastFit(t, p)                         ▷ end generated code
9:     end while
10:    return Best(p)
11: end function
```

Algorithm A.37. : **TSP Solver S (TSP-S)** was discovered from the experiments described in section 5.3. This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectElitism(p)
6:         t ← OrderBasedCrossover(t)                ▷ start generated code
7:         t ← InsertionMutation(t)
8:         t ← InsertionMutation(t)
9:         p ← ReplaceLeastFit(t, p)
10:        t ← SelectElitism(p)                      ▷ end generated code
11:    end while
12:    return Best(p)
13: end function
```

Algorithm A.38. : **TSP Solver T (TSP-T)** was discovered from the experiments described in section 5.3.This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     p ← 3-Opt-LocalSearch(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
5:         t ← SelectElitism(p)
6:         t ← 3 − OptLocalSearch(t)                 ▷ start generated code
7:         t ← ScrambleSubtourMutation(t)
8:         t ← 2_OptLocalSearch(t)
9:         t ← Best2 − OptLocalSearch(t)
10:        t ← OrderBasedCrossover(t)
11:        p ← ReplaceLeastFit(t, p)                 ▷ end generated code
12:    end while
13:    return Best(p)
14: end function
```

Algorithm A.39. : **TSP Solver U (TSP-U)** was discovered from the experiments described in section 5.4.This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                    ▷ start generated code
5:         t ← Best2 − OptLocalSearch(t)
6:         while EvalCount > MaxEvals/2 and EvalCount ≤ MaxEvals do
7:             t ← 3 − OptLocalSearch(t)
8:             p ← ReplaceLeastFit(t, p)
9:             t ← SelectElitism(p)
10:        end while
11:        t ← SubtourExchangeCrossover(t)
12:    end while                                        ▷ end generated code
13:    p ← ReplaceLeastFit(t,p)
14:    return Best(p)
15: end function
```

Algorithm A.40. : **TSP Solver V (TSP-V)** was discovered from the experiments described in section 5.4. This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                    ▷ start generated code
5:         t ← SubtourExchangeCrossover(t)
6:         t ← 3 − OptLocalSearch(t)
7:         t ← 3 − OptLocalSearch(t)
8:     end while
9:     p ← ReplaceLeastFit(t, p)
10:    t ← SelectElitism(p)                             ▷ end generated code
11:    p ← ReplaceLeastFit(t,p)
12:    return Best(p)
13: end function
```

Algorithm A.41. : **TSP Solver W (TSP-W)** was discovered from the experiments described in section 5.4. This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEval/2 do                ▷ start generated code
5:         t ← SubtourExchangeCrossover(t)
6:         t ← SimpleInversionMutation(t)
7:         t ← Best2 − OptLocalSearch(t)
8:         t ← SubtourExchangeCrossover(t)
9:         t ← OrderBasedCrossover(t)
10:    end while
11:    t ← OrderBasedCrossover(t)
12:    t ← 3 − OptLocalSearch(t)
13:    t ← SimpleInversionMutation(t)
14:    t ← InsertionMutation(t)                      ▷ end generated code
15:    p ← ReplaceLeastFit(t,p)
16:    return Best(p)
17: end function
```

Algorithm A.42. : **TSP Solver X** (TSP-X) was discovered from the experiments described in section 6.3. This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← 3-Opt-LocalSearch(p)
5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
6:         t ← 3 − OptLocalSearch(t)                 ▷ start generated code
7:         t ← SubtourExchangeCrossover(t)
8:         t ← SimpleInversionMutation(t)
9:         t ← ExchangeMutation(t)
10:        t ← 3 − OptLocalSearch(t)                 ▷ end generated code
11:        p ← ReplaceLeastFit(t,p)
12:        t ← SelectElitism(p)
13:    end while
14:    return Best(p)
15: end function
```

Algorithm A.43. : **TSP Solver Y** (TSP-Y) was discovered from the experiments described in section 6.3. This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← 3-Opt-LocalSearch(p)
5:     while EvalCount ≤ MaxEvals or p.fitness > 0 do
6:         t ← 3 − OptLocalSearch(t)                    ▷ start generated code
7:         t ← Best2 − OptLocalSearch(t)
8:         p ← ReplaceLeastFit(t, p)
9:         t ← SelectElitism(p)
10:        t ← 3 − OptLocalSearch(t)
11:        t ← 3 − OptLocalSearch(t)
12:        t ← InsertionMutation(t)
13:        t ← InsertionMutation(t)                     ▷ end generated code
14:        p ← ReplaceLeastFit(t,p)
15:        t ← SelectElitism(p)
16:    end while
17:    return Best(p)
18: end function
```

Algorithm A.44. : **TSP Solver Z** (TSP-Z) was discovered from the experiments described in section 6.5. This metaheuristic was ineffective; no further analysis will be completed.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                    ▷ start generated code
5:         t ← SubtourExchangeCrossover(t)
6:         t ← 3 − OptLocalSearch(t)
7:         t ← OrderBasedCrossover(t)
8:         p ← ReplaceLeastFit(t, p)
9:         t ← SelectElitism(p)
10:        t ← SubtourExchangeCrossover(t)
11:        t ← SimpleInversionMutation(t)
12:        t ← SimpleInversionMutation(t)
13:        t ← 3 − OptLocalSearch(t)
14:        t ← SubtourExchangeCrossover(t)
15:        t ← OrderBasedCrossover(t)
16:        p ← ReplaceLeastFit(t, p)
17:        t ← InsertionMutation(t)
18:    end while                                        ▷ end generated code
19:    p ← ReplaceLeastFit(t,p)
20:    return Best(p)
21: end function
```

Algorithm A.45. : **TSP solver Loloya-1**. It is a tour construction algorithm as suggested by Loyola et al [203]

```
1: i ← 0
2: while i < c do                                   ▷ start generated code
3:     if nearest − insertion() = 1 then
4:         i ← i + 1;
5:         2 − Opt();
6:     end if
7:     best − neighbor();
8:     i ← i + 1;
9:     2 − Opt();
10:    if nearest − insertion() = 1 then
11:        i ← i + 1;
12:        2 − Opt();
13:    end if
14:    worst − neighbor();
15:    i ← i + 1;
16:    2 − Opt();
17: end while                                       ▷ end generated code
```

Algorithm A.46. : **TSP solver Loloya-2** It is a tour construction algorithm as suggested by Loyola et al [203]

```
1: i ← 0
2: nearest − insertion();                           ▷ start generated code
3: best − neighbor();
4: near − center();
5: i ← 3;
6: if far − center() = 1 then
7:     i ← i + 1;
8:     if worst − neighbor() = 1 then
9:         i ← i + 1;
10:        if worst − neighbor() = 1 then
11:            i ← i + 1;
12:            while i < c do
13:                nearest − insertion();
14:                i ← i + 1;
15:                2 − Opt();
16:            end while
17:        end if
18:    end if
19: end if                                          ▷ end generated code
```

Algorithm A.48. : **NRP solver B** (NRP-B) was discovered from the experiments described in section 6.3

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← GreedyVariableDepthLocalSearch(t)
5:     while EvalCount ≤ MaxEvals do
6:         t ← UnassignedShiftMutation(t)                ▷ start generated code
7:         t ← NewSwapLocalSearch(t)
8:         t ← NewSwapLocalSearch(t)
9:         t ← VariableDepthLocalSearch(t)               ▷ end generated code
10:        p ← ReplaceLeastFit(t, p)
11:        t ← SelectElitism(p)
12:    end while
13:    return Best(p)
14: end function
```

Algorithm A.49. : **NRP solver C** (NRP-C) was discovered from the experiments described in section 6.3

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← GreedyVariableDepthLocalSearch(t)
5:     while EvalCount ≤ MaxEvals do
6:         t ← VariableDepthLocalSearch(t)               ▷ start generated code
7:         t ← LargeGreedyRuinRecreate(t)
8:         t ← GreedyVariableDepthLocalSearch(t)
9:         t ← SimpleGreedyRuinRecreate(t)
10:        t ← VariableDepthLocalSearch(t)
11:        t ← NewSwapLocalSearch(t)                     ▷ end generated code
12:        p ← ReplaceLeastFit(t, p)
13:        t ← SelectElitism(p)
14:    end while
15:    return Best(p)
16: end function
```

Algorithm A.50. : **NRP solver D** (NRP-D) was discovered from the experiments described in section 6.5

```
function FindSolution(ProblemParam,μ, λ)
    p ← InitPopulation(ProblemParam, μ, λ)
    t ← SelectElitism(p)
    while EvalCount ≤ MaxEvals do                          ▷ start generated code
        t ← VariableDepthLocalSearch(t)
        p ← ReplaceLeastFit(t, p)
        t ← SelectElitism(p)
        t ← SimpleGreedyRuinRecreate(t)
        t ← SmallGreedyRuinRecreate(t)
        p ← ReplaceLeastFit(t, p)
        t ← SelectElitism(p)
    end while                                              ▷ end generated code
    return Best(p)
end function
```

Algorithm A.51. : **NRP solver E** (NRP-E) was discovered from the experiments described in section 7.2

```
 1: function FindSolution(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     t ← GreedyVariableDepthLocalSearch(t)
 5:     while EvalCount ≤ MaxEvals do
 6:         p ← ReplaceLeastFit(t, p)                       ▷ start generated code
 7:         t ← SelectElitism(p)
 8:         t ← NewSwapLocalSearch(t)
 9:         t ← LargeGreedyRuinRecreate(t)
10:         t ← VariableDepthLocalSearch(t)                 ▷ end generated code
11:         p ← ReplaceLeastFit(t, p)
12:         t ← SelectElitism(t)
13:     end while
14:     return Best(p)
15: end function
```

Algorithm A.52. : **NRP solver F** (NRP-F) was discovered from the experiments described in section 7.2

```
 1: function FindSolution(ProblemParam,μ, λ)
 2:     p ← InitPopulation(ProblemParam, μ, λ)
 3:     t ← SelectElitism(p)
 4:     t ← GreedyVariableDepthLocalSearch(t)
 5:     while EvalCount ≤ MaxEvals do
 6:         t ← VariableDepthLocalSearch(t)                 ▷ start generated code
 7:         p ← ReplaceLeastFit(t, p)
 8:         t ← SelectElitism(p)
 9:         t ← SmallGreedyRuinRecreate(t)                  ▷ end generated code
10:         p ← ReplaceLeastFit(t, p)
11:         t ← SelectElitism(t)
12:     end while
13:     return Best(p)
14: end function
```

Algorithm A.53. : **NRP solver G** (NRP-G) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← GreedyVariableDepthLocalSearch(t)
5:     while EvalCount ≤ MaxEvals do
6:         t ← LargeGreedyRuinRecreate(t)                ▷ start generated code
7:         p ← ReplaceLeastFit(t, p)
8:         t ← SelectElitism(p)
9:         t ← GreedyVariableDepthLocalSearch(t)
10:        t ← NewSwapLocalSearch(t)
11:        t ← SimpleGreedyRuinRecreate(t)               ▷ end generated code
12:        p ← ReplaceLeastFit(t, p)
13:        t ← SelectElitism(t)
14:    end while
15:    return Best(p)
16: end function
```

Algorithm A.54. : **NRP solver H** (NRP-H) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                     ▷ start generated code
5:         p ← ReplaceLeastFit(t, p)
6:         t ← SelectElitism(p)
7:         t ← SmallGreedyRuinRecreate(t)
8:         t ← GreedyVariableDepthLocalSearch(t)
9:         p ← ReplaceLeastFit(t, p)
10:        t ← SelectElitism(p)
11:    end while                                         ▷ end generated code
12:    return Best(p)
13: end function
```

Algorithm A.55. : **NRP solver I** (NRP-I) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                     ▷ start generated code
5:         t ← GreedyVariableDepthLocalSearch(t)
6:         t ← HorizontalSwapLocalSearch(t)
7:         t ← NewSwapLocalSearch(t)
8:         p ← ReplaceLeastFit(t, p)
9:         t ← SelectElitism(t)
10:    end while                                         ▷ end generated code
11:    return Best(p)
12: end function
```

Algorithm A.56. : **NRP solver J** (NRP-J) was discovered from the experiments described in section 7.2

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals or p.fitness > 0 or Walk() do ▷ start gen. code
5:         p ← ReplaceLeastFit(t, p)
6:         t ← SelectElitism(t)
7:         t ← NewSwapLocalSearch(t)
8:         t ← GreedyVariableDepthLocalSearch(t)
9:         p ← ReplaceLeastFit(t, p)
10:        t ← SelectElitism(t)
11:    end while                                          ▷ end generated code
12:    return Best(p)
13: end function
```

Algorithm A.57. : **NRP solver K** (NRP-K) was discovered from the experiments described in section 7.2.

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← GreedyVariableDepthLocalSearch(t)
5:     while EvalCount ≤ MaxEvals do
6:         t ← VariableDepthLocalSearch(t)                ▷ start generated code
7:         t ← NewSwapLocalSearch(t)
8:         p ← RestartPopulation()(p)
9:         t ← MultiEventCrossover(t)
10:        t ← SmallGreedyRuinRecreate(t)                 ▷ end generated code
11:        p ← ReplaceLeastFit(t, p)
12:        t ← SelectElitism(t)
13:    end while
14:    return Best(p)
15: end function
```

Algorithm A.58. : **NRP solver L** (NRP-L) was discovered from the experiments described in section 7.2.

```
1: function FindSolution(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← GreedyVariableDepthLocalSearch(t)
5:     while EvalCount ≤ MaxEvals do
6:         t ← HorizontalSwapLocalSearch(t)               ▷ start generated code
7:         t ← HorizontalSwapLocalSearch(t)
8:         t ← UnassignedShiftMutation(t)
9:         p ← ReplaceLeastFit(t, p)
10:        t ← SelectElitism(p)                           ▷ end generated code
11:        p ← ReplaceLeastFit(t, p)
12:        t ← SelectElitism(t)
13:    end while
14:    return Best(p)
15: end function
```

Algorithm A.59. : **NRP solver M** (NRP-M) was discovered from the experiments described in section 7.2.

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     t ← GreedyVariableDepthLocalSearch(t)
5:     while EvalCount ≤ MaxEvals do
6:         t ← VariableDepthLocalSearch                    ▷ start generated code
7:         t ← NewSwapLocalSearch(t)
8:         t ← UnassignedShiftMutation(t)
9:         p ← VariableDepthLocalSearch
10:        t ← SmallGreedyRuinRecreate                     ▷ end generated code
11:        p ← ReplaceLeastFit(t, p)
12:        t ← SelectElitism(t)
13:    end while
14:    return Best(p)
15: end function
```

Algorithm A.60. : **NRP solver N** (NRP-N) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)                                ▷ start generated code
4:     t ← VariableDepthLocalSearch(t)
5:     t ← UnassignedShiftMutation(t)
6:     t ← UnassignedShiftMutation(t)                      ▷ end generated code
7:     return Best(p)
8: end function
```

Algorithm A.61. : **NRP solver O** (NRP-O) was discovered from the experiments described in section 7.2

```
1: function FINDSOLUTION(ProblemParam,μ, λ)
2:     p ← InitPopulation(ProblemParam, μ, λ)
3:     t ← SelectElitism(p)
4:     while EvalCount ≤ MaxEvals do                       ▷ start generated code
5:         t ← GreedyVariableDepthLocalSearch(t)
6:         t ← HorizontalSwapLocalSearch(t)
7:         t ← NewSwapLocalSearch(t)
8:         p ← ReplaceLeastFit(t, p)
9:         t ← SelectElitism(t)
10:    end while                                           ▷ end generated code
11:    return Best(p)
12: end function
```

Appendix B: Statistical results

This appendix provides a complete set of results obtained from our algorithm optimisation processes and the literature. We compare the performance of the majority of the algorithms given in the previous appendix.

An extensive collection of unknown instances has validated the performance of these problem-specific metaheuristics. No learning instance is reported in this appendix. We have completed 100 independent runs for each algorithm for some solutions unknown benchmark.

The arithmetical mean suggests the expected solutions that can be obtained by a problem-specific metaheuristic for a benchmark. This simple measure of central tendency is commonly used in many scientific fields, including selective and generative hyper-heuristic. Nonetheless, some misleading results can be reported with skewed data sets and outliers.

The distributions of the problem solutions exhibit such properties. Therefore, a median estimates more accurately the middle value within a data set; extreme values or outliers do not affect this measure of central tendency. Therefore we believe the arithmetical mean and the median should indicate whether problem-specific metaheuristics are likely to lower the problem solutions over time.

A standard deviation (std) measures the range of solutions that could be found by a problem-specific metaheuristic. A large value may indicate a high uncertainty to obtain suitable near-optima. The interquartile range (IQR) quantifies the statistical dispersion of the midspread, instead of the whole range. Alongside the standard deviation, both measures of dispersion describe the likelihood to obtain an outlier with an algorithm for specific benchmarks.

The best results for each instance have been reported **in green for by the mean and standard deviation** and **in orange for the median and interquartile range**. This differentiation helps us appreciating more effectively a distribution skewness and the variability of the solutions that can be found.

We have compared the solutions found by some discovered solvers using the Mann-Whitney U test. This non-parametric test indicates whether two distributions of problem solutions have the same medians. The Mann-Whitney U test can, therefore, inform us whether two metaheuristics have the same performance and operate on the same set of problem solutions.

We assume that each problem solution obtained for a benchmark is considered to be independent of each other. The problem fitness value is ordinal, and one can at least say, of any two observations, which is the greater. The *null hypothesis H0* suggests two sets of problem solutions are equal. The *alternative hypothesis H1* suggests otherwise. All our non-parametric tests have completed over 100 independent runs with a p-value set to 0.01.

The "A-measure" effect size was also calculated, to assess the effect on the medians. This additional information suggests no effect exists (0.5), a small effect (0.56) and a common effect (0.64). Finally, a big effect is indicated by an "A" measure greater or equal to 0.71] [304].

A symbol $=$ indicates no significant difference exists between Alg A and Alg B (i.e. the results of). A symbol $+$ denotes that Alg A is significantly better than Alg B and finally a symbol $-$ that Alg A is significantly worse than Alg B.

We should appreciate more effectively which algorithms would perform well on a benchmark or a class of them.

The Mimicry Problem

Tables [B.1-B.10] statistically compares the distribution of imitators found by some of the mimicry solvers provided in Appendix A. 100 independent runs with 20,000 evaluations were completed. The mutation rate was set to 0.001 and the adaptive mutation rate to 0.05. A formal description of this problem domain was provided in section 3.2.

Table B.1: Statistical comparison for some mimicry solvers generated in chapters [5-7]

Instance		100	1000	2000	3000	4000	5000
MC-A	**mean**	0.000e+00	0.000e+00	6.433e-04	6.433e-04	3.387e-03	9.132e-03
	std	(0.0e+00)	(0.0e+00)	(4.8e-04)	(4.8e-04)	(8.7e-04)	(1.4e-03)
	median	0.000e+00	0.000e+00	6.667e-04	6.667e-04	3.375e-03	9.200e-03
	IQR	(0.0e+00)	(0.0e+00)	(6.7e-04)	(6.7e-04)	(1.3e-03)	(1.8e-03)
MC-B	**mean**	0.000e+00	1.500e-04	2.933e-03	2.933e-03	5.542e-03	1.148e-02
	std	(0.0e+00)	(4.1e-04)	(2.8e-03)	(2.8e-03)	(5.1e-03)	(9.6e-03)
	median	0.000e+00	0.000e+00	2.167e-03	2.167e-03	4.250e-03	9.200e-03
	IQR	(0.0e+00)	(0.0e+00)	(3.0e-03)	(3.0e-03)	(6.0e-03)	(1.2e-02)
MC-C	**mean**	0.000e+00	7.310e-03	1.020e-01	1.020e-01	1.476e-01	1.876e-01
	std	(0.0e+00)	(2.8e-03)	(5.3e-03)	(5.3e-03)	(5.1e-03)	(5.2e-03)
	median	0.000e+00	7.000e-03	1.020e-01	1.020e-01	1.471e-01	1.871e-01
	IQR	(0.0e+00)	(3.0e-03)	(7.2e-03)	(7.2e-03)	(5.6e-03)	(6.6e-03)
MC-D	**mean**	0.000e+00	4.000e-05	5.767e-04	5.767e-04	3.505e-03	8.944e-03
	std	(0.0e+00)	(1.4e-04)	(4.4e-04)	(4.4e-04)	(9.2e-04)	(1.4e-03)
	median	0.000e+00	0.000e+00	6.667e-04	6.667e-04	3.250e-03	9.000e-03
	IQR	(0.0e+00)	(0.0e+00)	(6.7e-04)	(6.7e-04)	(1.4e-03)	(1.6e-03)
MC-E	**mean**	0.000e+00	0.000e+00	6.333e-04	6.333e-04	3.285e-03	9.573e-03
	std	(0.0e+00)	(0.0e+00)	(4.5e-04)	(4.5e-04)	(8.1e-04)	(2.6e-03)
	median	0.000e+00	0.000e+00	6.667e-04	6.667e-04	3.250e-03	9.000e-03
	IQR	(0.0e+00)	(0.0e+00)	(6.7e-04)	(6.7e-04)	(1.0e-03)	(2.0e-03)
MC-F	**mean**	0.000e+00	0.000e+00	6.833e-04	6.833e-04	3.460e-03	9.110e-03
	std	(0.0e+00)	(0.0e+00)	(4.4e-04)	(4.4e-04)	(9.7e-04)	(1.4e-03)
	median	0.000e+00	0.000e+00	6.667e-04	6.667e-04	3.250e-03	9.000e-03
	IQR	(0.0e+00)	(0.0e+00)	(6.7e-04)	(6.7e-04)	(1.3e-03)	(2.0e-03)
MC-G	**mean**	0.000e+00	0.000e+00	5.667e-04	5.667e-04	3.327e-03	9.028e-03
	std	(0.0e+00)	(0.0e+00)	(4.2e-04)	(4.2e-04)	(8.6e-04)	(1.4e-03)
	median	0.000e+00	0.000e+00	3.333e-04	3.333e-04	3.250e-03	8.600e-03
	IQR	(0.0e+00)	(0.0e+00)	(3.3e-04)	(3.3e-04)	(1.1e-03)	(1.8e-03)
MC-H	**mean**	0.000e+00	3.700e-04	3.467e-03	3.467e-03	3.460e-03	1.235e-02
	std	(0.0e+00)	(6.8e-04)	(3.7e-03)	(3.7e-03)	(9.7e-04)	(9.1e-03)
	median	0.000e+00	0.000e+00	2.333e-03	2.333e-03	3.250e-03	1.100e-02
	IQR	(0.0e+00)	(1.0e-03)	(4.0e-03)	(4.0e-03)	(1.3e-03)	(1.0e-02)
MC-I	**mean**	0.000e+00	2.900e-04	3.883e-03	3.883e-03	7.475e-03	1.237e-02
	std	(0.0e+00)	(5.2e-04)	(3.7e-03)	(3.7e-03)	(7.1e-03)	(8.8e-03)
	median	0.000e+00	0.000e+00	2.667e-03	2.667e-03	5.250e-03	1.080e-02
	IQR	(0.0e+00)	(1.0e-03)	(5.2e-03)	(5.2e-03)	(8.1e-03)	(1.2e-02)
MC-J	**mean**	0.000e+00	1.230e-03	6.945e-03	6.945e-03	6.945e-03	1.161e-02
	std	(0.0e+00)	(1.2e-03)	(6.4e-03)	(6.4e-03)	(6.4e-03)	(8.7e-03)
	median	0.000e+00	1.000e-03	4.000e-03	4.000e-03	4.000e-03	9.500e-03
	IQR	(0.0e+00)	(2.0e-03)	(8.5e-03)	(8.5e-03)	(8.5e-03)	(1.2e-02)
MC-K	**mean**	0.000e+00	1.200e-04	3.857e-03	3.857e-03	4.625e-03	9.666e-03
	std	(0.0e+00)	(3.3e-04)	(3.8e-03)	(3.8e-03)	(2.9e-03)	(5.5e-03)
	median	0.000e+00	0.000e+00	2.500e-03	2.500e-03	4.125e-03	8.700e-03
	IQR	(0.0e+00)	(0.0e+00)	(4.3e-03)	(4.3e-03)	(3.6e-03)	(4.9e-03)
MC-L	**mean**	0.000e+00	0.000e+00	6.167e-04	6.167e-04	3.353e-03	9.188e-03
	std	(0.0e+00)	(0.0e+00)	(4.0e-04)	(4.0e-04)	(9.0e-04)	(1.2e-03)
	median	0.000e+00	0.000e+00	6.667e-04	6.667e-04	3.500e-03	9.200e-03
	IQR	(0.0e+00)	(0.0e+00)	(3.3e-04)	(3.3e-04)	(1.3e-03)	(1.6e-03)
Herdy	**mean**	6.000e-04	1.729e-02	5.275e-02	9.417e-02	1.311e-01	1.643e-01
	std	(7.2e-04)	(2.8e-03)	(4.1e-03)	(3.9e-03)	(4.3e-03)	(4.2e-03)
	median	0.000e+00	1.750e-02	5.300e-02	9.413e-02	1.310e-01	1.634e-01
	IQR	(1.0e-03)	(4.3e-03)	(6.2e-03)	(4.6e-03)	(5.8e-03)	(6.8e-03)

Table B.2: Statistical comparison for some mimicry solvers generated in chapters [5-7]

Instance		6000	7000	8000	10000	20000	30000
MC-A	**mean**	1.762e-02	2.870e-02	4.091e-02	6.751e-02	1.838e-01	2.569e-01
	std	(1.7e-03)	(2.0e-03)	(2.1e-03)	(2.5e-03)	(2.5e-03)	(2.1e-03)
	median	1.742e-02	2.914e-02	4.106e-02	6.750e-02	1.838e-01	2.568e-01
	IQR	(2.2e-03)	(2.5e-03)	(2.4e-03)	(2.8e-03)	(3.8e-03)	(2.7e-03)
MC-B	**mean**	2.189e-02	3.162e-02	4.256e-02	6.618e-02	1.848e-01	2.565e-01
	std	(1.3e-02)	(1.6e-02)	(1.9e-02)	(2.2e-02)	(2.5e-02)	(2.6e-02)
	median	1.875e-02	2.829e-02	4.244e-02	6.405e-02	1.804e-01	2.587e-01
	IQR	(1.4e-02)	(2.1e-02)	(2.6e-02)	(3.2e-02)	(3.3e-02)	(3.5e-02)
MC-C	**mean**	2.206e-01	2.466e-01	2.695e-01	3.033e-01	3.891e-01	4.223e-01
	std	(4.8e-03)	(4.4e-03)	(4.4e-03)	(4.2e-03)	(3.0e-03)	(2.3e-03)
	median	2.206e-01	2.459e-01	2.697e-01	3.035e-01	3.891e-01	4.223e-01
	IQR	(5.9e-03)	(7.0e-03)	(6.3e-03)	(5.4e-03)	(4.3e-03)	(3.1e-03)
MC-D	**mean**	1.916e-02	3.286e-02	4.090e-02	6.742e-02	1.843e-01	2.569e-01
	std	(5.2e-03)	(6.2e-03)	(2.1e-03)	(2.2e-03)	(2.2e-03)	(2.3e-03)
	median	1.800e-02	2.979e-02	4.113e-02	6.735e-02	1.842e-01	2.569e-01
	IQR	(2.3e-03)	(1.2e-02)	(2.6e-03)	(2.6e-03)	(2.9e-03)	(3.1e-03)
MC-E	**mean**	1.787e-02	3.358e-02	4.137e-02	6.723e-02	1.841e-01	2.569e-01
	std	(1.7e-03)	(6.6e-03)	(2.1e-03)	(2.4e-03)	(2.4e-03)	(2.3e-03)
	median	1.792e-02	3.007e-02	4.113e-02	6.755e-02	1.840e-01	2.566e-01
	IQR	(2.6e-03)	(1.2e-02)	(3.4e-03)	(3.5e-03)	(2.9e-03)	(3.5e-03)
MC-F	**mean**	1.775e-02	3.227e-02	4.077e-02	6.741e-02	1.835e-01	2.567e-01
	std	(1.4e-03)	(6.2e-03)	(2.0e-03)	(2.3e-03)	(2.6e-03)	(2.3e-03)
	median	1.783e-02	2.943e-02	4.063e-02	6.740e-02	1.834e-01	2.567e-01
	IQR	(1.5e-03)	(1.2e-02)	(2.6e-03)	(3.1e-03)	(3.3e-03)	(3.0e-03)
MC-G	**mean**	1.762e-02	2.867e-02	4.108e-02	6.746e-02	1.840e-01	2.571e-01
	std	(1.7e-03)	(1.8e-03)	(2.1e-03)	(2.4e-03)	(2.2e-03)	(2.2e-03)
	median	1.783e-02	2.843e-02	4.113e-02	6.710e-02	1.840e-01	2.575e-01
	IQR	(2.5e-03)	(2.6e-03)	(2.9e-03)	(3.2e-03)	(2.5e-03)	(2.9e-03)
MC-H	**mean**	2.128e-02	3.113e-02	4.514e-02	6.726e-02	1.846e-01	2.532e-01
	std	(1.3e-02)	(1.7e-02)	(1.9e-02)	(2.0e-02)	(2.8e-02)	(2.2e-02)
	median	1.975e-02	2.879e-02	4.456e-02	6.690e-02	1.813e-01	2.522e-01
	IQR	(1.9e-02)	(2.2e-02)	(2.7e-02)	(2.1e-02)	(3.4e-02)	(2.9e-02)
MC-I	**mean**	1.999e-02	3.253e-02	4.088e-02	6.878e-02	1.846e-01	2.537e-01
	std	(1.3e-02)	(1.8e-02)	(1.8e-02)	(2.3e-02)	(2.8e-02)	(2.4e-02)
	median	1.750e-02	3.157e-02	3.919e-02	7.160e-02	1.813e-01	2.532e-01
	IQR	(1.6e-02)	(2.5e-02)	(2.6e-02)	(3.6e-02)	(3.4e-02)	(3.5e-02)
MC-J	**mean**	2.180e-02	2.815e-02	4.240e-02	7.163e-02	1.777e-01	2.546e-01
	std	(1.3e-02)	(1.4e-02)	(1.9e-02)	(2.5e-02)	(2.6e-02)	(2.3e-02)
	median	1.975e-02	2.729e-02	3.975e-02	6.935e-02	1.767e-01	2.531e-01
	IQR	(1.6e-02)	(1.9e-02)	(2.7e-02)	(3.5e-02)	(3.5e-02)	(2.6e-02)
MC-K	**mean**	1.808e-02	2.821e-02	4.031e-02	6.870e-02	1.799e-01	2.564e-01
	std	(6.7e-03)	(8.8e-03)	(8.2e-03)	(1.4e-02)	(1.5e-02)	(1.2e-02)
	median	1.817e-02	2.736e-02	3.975e-02	6.895e-02	1.813e-01	2.569e-01
	IQR	(7.8e-03)	(7.9e-03)	(7.1e-03)	(1.0e-02)	(1.3e-02)	(1.2e-02)
MC-L	**mean**	1.780e-02	2.873e-02	4.090e-02	6.766e-02	1.839e-01	2.562e-01
	std	(1.7e-03)	(2.0e-03)	(2.1e-03)	(2.5e-03)	(2.2e-03)	(1.8e-03)
	median	1.767e-02	2.857e-02	4.094e-02	6.765e-02	1.838e-01	2.564e-01
	IQR	(2.2e-03)	(3.0e-03)	(2.5e-03)	(3.7e-03)	(3.3e-03)	(2.7e-03)
Herdy	**mean**	1.930e-01	2.165e-01	2.556e-01	3.569e-01	3.994e-01	3.993e-01
	std	(3.8e-03)	(4.1e-03)	(3.7e-03)	(3.0e-03)	(2.1e-03)	2.0e-03
	median	1.929e-01	2.162e-01	2.554e-01	3.568e-01	3.995e-01	3.994e-01
	IQR	(4.3e-03)	(5.3e-03)	(4.7e-03)	(4.4e-03)	(2.9e-03)	2.9e-01

Table B.3: Statistical comparison of imitators obtained by Herdy [146] and the generated solvers MC-[A-L]

	Herdy vs											
Instance	**MC-A**	**MC-B**	**MC-C**	**MC-D**	**MC-E**	**MC-F**	**MC-G**	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**
100	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)
1000	0.73 (−)	0.67 (−)	0.99 (+)	0.71 (−)	0.73 (−)	0.73 (−)	0.73 (−)	0.59 (−)	0.61 (−)	0.65 (+)	0.68 (−)	0.73 (−)
2000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	0.99 (−)	0.99 (−)	0.91 (−)	0.99 (−)	1 (−)
3000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
4000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
5000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
6000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
7000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
8000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
10000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
20000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
30000	1 (−)	1 (−)	1 (+)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)

Table B.4: Statistical comparison of imitators obtained by generated solver MC-A and the generated solvers MC-[B-L]

	MC-A vs										
Instance	**MC-B**	**MC-C**	**MC-D**	**MC-E**	**MC-F**	**MC-G**	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**
100	0.50 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)
1000	0.56 (−)	0.99 (+)	0.54 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.65 (−)	0.63 (−)	0.83 (+)	0.56 (−)	0.5 (=)
2000	0.82 (+)	1 (+)	0.54 (=)	0.51 (=)	0.54 (=)	0.54 (=)	0.8 (+)	0.84 (+)	0.88 (+)	0.84 (+)	0.5 (=)
3000	0.82 (+)	1 (+)	0.54 (=)	0.51 (=)	0.54 (=)	0.54 (=)	0.8 (+)	0.84 (+)	0.88 (+)	0.84 (+)	0.5 (=)
4000	0.56 (+)	1 (+)	0.53 (=)	0.54 (=)	0.51 (=)	0.53 (=)	0.51 (=)	0.66 (+)	0.61 (+)	0.61 (+)	0.51 (=)
5000	0.5 (−)	1 (+)	0.54 (=)	0.51 (=)	0.51 (=)	0.53 (=)	0.59 (+)	0.59 (+)	0.53 (+)	0.56 (−)	0.51 (=)
6000	0.57 (+)	1 (+)	0.59 (=)	0.55 (=)	0.53 (=)	0.51 (=)	0.56 (+)	0.5 (=)	0.58 (+)	0.54 (+)	0.53 (=)
7000	0.51 (=)	1 (+)	0.65 (+)	0.67 (+)	0.6 (+)	0.53 (=)	0.5 (=)	0.56 (+)	0.54 (−)	0.6 (−)	0.51 (=)
8000	0.53 (+)	1 (+)	0.51 (=)	0.55 (=)	0.53 (=)	0.52 (=)	0.55 (+)	0.54 (−)	0.53 (−)	0.58 (−)	0.51 (=)
10000	0.54 (=)	1 (+)	0.51 (=)	0.53 (=)	0.51 (=)	0.52 (=)	0.52 (−)	0.54 (+)	0.54 (+)	0.55 (+)	0.51 (=)
20000	0.55 (=)	1 (+)	0.55 (=)	0.53 (=)	0.53 (=)	0.53 (=)	0.53 (−)	0.53 (−)	0.6 (−)	0.61 (−)	0.51 (=)
30000	0.52 (+)	1 (+)	0.51 (=)	0.50 (=)	0.51 (=)	0.54 (=)	0.58 (−)	0.55 (=)	0.58 (−)	0.51 (+)	0.6 (=)

Table B.5: Statistical comparison of imitators obtained by generated solver MC-B and the generated solvers MC-[C-L]

	MC-B vs									
Instance	**MC-C**	**MC-D**	**MC-E**	**MC-F**	**MC-G**	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**
100	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)
1000	0.99 (+)	0.53 (=)	0.56 (−)	0.56 (−)	0.56 (−)	0.58 (−)	0.56 (=)	0.78 (+)	0.51 (=)	0.56 (−)
2000	1 (+)	0.83 (−)	0.82 (−)	0.81 (−)	0.84 (−)	0.52 (=)	0.56 (=)	0.69 (+)	0.55 (=)	0.82 (−)
3000	1 (+)	0.83 (−)	0.82 (−)	0.81 (−)	0.84 (−)	0.52 (=)	0.56 (=)	0.69 (+)	0.55 (=)	0.82 (−)
4000	1 (+)	0.56 (−)	0.57 (−)	0.56 (−)	0.57 (−)	0.56 (−)	0.58 (=)	0.55 (=)	0.5 (=)	0.57 (−)
5000	1 (+)	0.51 (=)	0.51 (=)	0.50 (=)	0.51 (=)	0.55 (=)	0.55 (=)	0.52 (=)	0.51 (=)	0.50 (=)
6000	1 (+)	0.53 (−)	0.56 (−)	0.56 (−)	0.57 (−)	0.52 (=)	0.55 (=)	0.5 (=)	0.57 (−)	0.56 (−)
7000	1 (+)	0.58 (+)	0.59 (+)	0.56 (+)	0.5 (+)	0.52 (=)	0.51 (=)	0.56 (=)	0.55 (=)	0.50 (=)
8000	1 (+)	0.53 (=)	0.53 (=)	0.54 (=)	0.53 (−)	0.54 (=)	0.52 (=)	0.5 (=)	0.53 (=)	0.53 (=)
10000	1 (+)	0.54 (+)	0.53 (+)	0.54 (+)	0.54 (+)	0.52 (=)	0.52 (=)	0.57 (=)	0.54 (+)	0.54 (+)
20000	1 (+)	0.56 (+)	0.56 (+)	0.55 (+)	0.56 (+)	0.5 (=)	0.5 (=)	0.57 (=)	0.52 (+)	0.56 (+)
30000	1 (+)	0.52 (=)	0.51 (=)	0.52 (=)	0.51 (=)	0.54 (=)	0.53 (=)	0.52 (=)	0.51 (=)	0.53 (=)

Table B.6: Statistical comparison of imitators obtained by generated solver MC-C and the generated solvers MC-[D-L]

	MC-C vs								
Instance	**MC-D**	**MC-E**	**MC-F**	**MC-G**	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**
100	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)
1000	0.99 (−)	0.99 (−)	0.99 (−)	0.99 (−)	0.99 (−)	0.99 (−)	0.98 (−)	0.99 (−)	0.99 (−)
2000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
3000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
4000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
5000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
6000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
7000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
8000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
10000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
20000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
30000	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)

Table B.7: Statistical comparison of imitators obtained by generated solver MC-D and the generated solvers MC-[E-L]

	MC-D vs							
Instance	**MC-E**	**MC-F**	**MC-G**	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**
100	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)
1000	0.54 (−)	0.54 (−)	0.54 (−)	0.62 (−)	0.6 (−)	0.82 (+)	0.52 (=)	0.54 (−)
2000	0.54 (=)	0.58 (=)	0.51 (=)	0.82 (+)	0.85 (+)	0.89 (+)	0.85 (+)	0.54 (=)
3000	0.54 (=)	0.58 (=)	0.51 (=)	0.82 (+)	0.85 (+)	0.89 (+)	0.85 (+)	0.54 (=)
4000	0.56 (=)	0.52 (=)	0.55 (=)	0.52 (=)	0.65 (+)	0.6 (+)	0.59 (+)	0.54 (=)
5000	0.54 (=)	0.52 (=)	0.5 (=)	0.6 (+)	0.61 (+)	0.53 (+)	0.53 (−)	0.55 (=)
6000	0.55 (=)	0.57 (=)	0.58 (=)	0.53 (+)	0.53 (−)	0.54 (+)	0.51 (+)	0.57 (=)
7000	0.53 (=)	0.54 (=)	0.67 (−)	0.58 (−)	0.53 (+)	0.63 (−)	0.71 (−)	0.66 (−)
8000	0.56 (=)	0.52 (=)	0.53 (=)	0.55 (+)	0.54 (−)	0.53 (−)	0.58 (−)	0.5 (=)
10000	0.52 (=)	0.51 (=)	0.5 (=)	0.51 (−)	0.55 (+)	0.54 (+)	0.55 (+)	0.52 (=)
20000	0.53 (=)	0.59 (=)	0.52 (=)	0.53 (−)	0.53 (−)	0.61 (−)	0.64 (−)	0.55 (=)
30000	0.51 (=)	0.52 (=)	0.53 (=)	0.58 (−)	0.55 (−)	0.58 (−)	0.51 (−)	0.50 (=)

Table B.8: Statistical comparison of imitators obtained by generated solvers MC-[E-F] and the generated solvers MC-[G-L]

	MC-E vs						MC-F vs					
Instance	**MC-F**	**MC-G**	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**	**MC-G**	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**
100	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)
1000	0.5 (=)	0.5 (=)	0.65 (−)	0.63 (−)	0.83 (+)	0.56 (−)	0.5 (=)	0.5 (=)	0.65 (−)	0.63 (−)	0.83 (+)	0.56 (−)	0.5 (=)
2000	0.54 (=)	0.55 (=)	0.8 (+)	0.84 (+)	0.89 (+)	0.84 (+)	0.51 (=)	0.58 (=)	0.79 (+)	0.83 (+)	0.88 (+)	0.83 (+)	0.55 (=)
3000	0.54 (=)	0.55 (=)	0.8 (+)	0.84 (+)	0.89 (+)	0.84 (+)	0.51 (=)	0.58 (=)	0.79 (+)	0.83 (+)	0.88 (+)	0.83 (+)	0.55 (=)
4000	0.54 (=)	0.51 (=)	0.54 (=)	0.66 (+)	0.62 (+)	0.62 (+)	0.53 (=)	0.53 (=)	0.5 (=)	0.65 (+)	0.61 (+)	0.6 (+)	0.52 (=)
5000	0.52 (=)	0.54 (=)	0.57 (+)	0.58 (+)	0.51 (+)	0.57 (−)	0.5 (=)	0.53 (=)	0.59 (+)	0.6 (+)	0.52 (+)	0.55 (−)	0.52 (=)
6000	0.53 (=)	0.54 (=)	0.55 (+)	0.51 (−)	0.57 (+)	0.53 (+)	0.52 (=)	0.51 (=)	0.56 (+)	0.5 (−)	0.57 (+)	0.53 (+)	0.5 (=)
7000	0.57 (=)	0.7 (−)	0.6 (−)	0.55 (+)	0.65 (−)	0.72 (−)	0.69 (−)	0.62 (−)	0.57 (−)	0.52 (+)	0.62 (−)	0.68 (−)	0.62 (−)
8000	0.58 (=)	0.53 (=)	0.54 (+)	0.55 (−)	0.54 (−)	0.61 (−)	0.56 (=)	0.55 (=)	0.55 (+)	0.54 (−)	0.53 (−)	0.57 (−)	0.53 (=)
10000	0.52 (=)	0.52 (=)	0.51 (−)	0.55 (+)	0.54 (+)	0.56 (+)	0.54 (=)	0.51 (=)	0.51 (−)	0.55 (+)	0.54 (+)	0.55 (+)	0.52 (=)
20000	0.56 (=)	0.5 (=)	0.53 (−)	0.53 (−)	0.61 (−)	0.63 (−)	0.52 (=)	0.57 (=)	0.52 (−)	0.52 (−)	0.6 (−)	0.6 (−)	0.54 (=)
30000	0.52 (=)	0.54 (=)	0.58 (−)	0.55 (−)	0.58 (−)	0.51 (+)	0.58 (−)	0.55 (=)	0.58 (−)	0.55 (−)	0.57 (−)	0.5 (+)	0.58 (=)

Table B.9: Statistical comparison of imitators obtained by generated solvers MC-[G-I] and the generated solvers MC-[J-L]

	MC-G vs					**MC-H vs**				**MC-I vs**		
Instance	**MC-H**	**MC-I**	**MC-J**	**MC-K**	**MC-L**	**MC-I**	**MC-J**	**MC-K**	**MC-L**	**MC-J**	**MC-K**	**MC-L**
100	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)
1000	0.65 (−)	0.63 (−)	0.83 (+)	0.56 (−)	0.5 (=)	0.52 (=)	0.72 (+)	0.59 (−)	0.65 (−)	0.74 (+)	0.57 (−)	0.63 (−)
2000	0.82 (+)	0.86 (+)	0.89 (+)	0.85 (+)	0.55 (=)	0.54 (=)	0.67 (+)	0.53 (=)	0.8 (−)	0.63 (+)	0.51 (=)	0.84 (−)
3000	0.82 (+)	0.86 (+)	0.89 (+)	0.85 (+)	0.55 (=)	0.54 (=)	0.67 (+)	0.53 (=)	0.8 (−)	0.63 (+)	0.51 (=)	0.84 (−)
4000	0.53 (=)	0.66 (+)	0.62 (+)	0.62 (+)	0.51 (=)	0.65 (+)	0.61 (+)	0.6 (+)	0.52 (=)	0.52 (=)	0.58 (−)	0.66 (−)
5000	0.6 (+)	0.6 (+)	0.53 (+)	0.54 (+)	0.55 (=)	0.5 (=)	0.53 (=)	0.58 (−)	0.59 (−)	0.53 (=)	0.58 (−)	0.59 (−)
6000	0.56 (+)	0.5 (−)	0.58 (+)	0.54 (+)	0.52 (=)	0.53 (=)	0.51 (=)	0.55 (−)	0.55 (−)	0.55 (=)	0.5 (=)	0.5 (+)
7000	0.5 (+)	0.56 (+)	0.53 (−)	0.6 (−)	0.51 (=)	0.53 (=)	0.54 (=)	0.53 (−)	0.5 (−)	0.57 (=)	0.57 (−)	0.56 (−)
8000	0.54 (+)	0.54 (−)	0.53 (−)	0.59 (−)	0.53 (=)	0.56 (=)	0.54 (=)	0.57 (−)	0.55 (−)	0.52 (=)	0.52 (+)	0.54 (+)
10000	0.51 (−)	0.55 (+)	0.54 (+)	0.55 (+)	0.52 (=)	0.52 (=)	0.55 (=)	0.53 (=)	0.52 (+)	0.54 (=)	0.51 (−)	0.54 (−)
20000	0.53 (−)	0.53 (−)	0.61 (−)	0.63 (−)	0.53 (=)	0.5 (=)	0.57 (=)	0.53 (−)	0.53 (+)	0.57 (=)	0.53 (−)	0.53 (+)
30000	0.59 (−)	0.55 (−)	0.58 (−)	0.52 (−)	0.63 (−)	0.51 (=)	0.53 (=)	0.56 (+)	0.57 (+)	0.52 (=)	0.54 (+)	0.54 (+)

Table B.10: Statistical comparison of imitators obtained by generated solvers MC-[J-K] and the generated solver MC-L]

Instance	MC-J vs MC-K		MC-J vs MC-L		MC-K vs MC-L	
	0.5	(=)	0.5	(=)	0.5	(=)
1000	0.79	(−)	0.83	(−)	0.56	(−)
2000	0.64	(−)	0.89	(−)	0.84	(−)
3000	0.64	(−)	0.89	(−)	0.84	(−)
4000	0.55	(+)	0.61	(−)	0.62	(−)
5000	0.54	(−)	0.52	(−)	0.56	(+)
6000	0.56	(−)	0.57	(−)	0.53	(−)
7000	0.51	(=)	0.54	(+)	0.6	(+)
8000	0.5	(−)	0.53	(+)	0.58	(+)
10000	0.53	(−)	0.53	(−)	0.54	(−)
20000	0.54	(+)	0.6	(+)	0.62	(+)
30000	0.54	(+)	0.57	(+)	0.53	(−)

The statistical comparison of the imitators found by some mimicry solvers is provided in table B.11. One of them was designed by humanly activities (i.e Herdy) and the remaining were generated with a CGP hyper-heuristic. The number of problem evaluations was computed using the expression $(3,072 * 2)/250$ (see section 8.1.2). 100 independent runs were completed.

Table B.11: Statistical comparison of some imitators obtained by some mimicry solvers.

		100	**1000**	**10000**	**100000**
Herdy	**mean**	0.000e+00	1.300e-04	1.370e-04	1.371e-04
	std	(0.0e+00)	(3.4e-04)	(1.2e-04)	(3.3e-05)
	median	0.000e+00	0.000e+00	1.000e-04	1.300e-04
	IQR	(0.0e+00)	(0.0e+00)	(1.5e-04)	(4.0e-05)
MC-A	**mean**	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	std	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
MC-D	**mean**	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	std	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
MC-E	**mean**	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	std	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
MC-L	**mean**	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	std	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
MC-M	**mean**	4.099e-01	4.322e-01	4.344e-01	4.298e-01
	std	(6.9e-02)	(3.6e-02)	(3.4e-02)	(4.0e-02)
	median	4.100e-01	4.345e-01	4.348e-01	4.362e-01
	IQR	(1.0e-01)	(5.0e-02)	(4.7e-02)	(5.3e-02)

Table B.12: Statistical comparison of imitators obtained by Herdy [146] and the generated solver MC-A, and the generated solvers MC-A, MC-E, MC-L, MC-M

	Herdy vs					MC-A vs			
Instances	**MC-A**	**MC-D**	**MC-E**	**MC-L**	**MC-M**	**MC-D**	**MC-E**	**MC-L**	**MC-M**
100	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	1 (+)	0.5 (=)	0.5 (=)	0.5 (=)	1 (+)
1000	0.56 (−)	0.56 (−)	0.56 (−)	0.56 (−)	1 (+)	0.5 (=)	0.5 (=)	0.5 (=)	1 (+)
10000	0.88 (−)	0.88 (−)	0.88 (−)	0.88 (−)	1 (+)	0.5 (=)	0.5 (=)	0.5 (=)	1 (+)
100000	1 (−)	1 (−)	1 (−)	1 (−)	1 (+)	0.5 (=)	0.5 (=)	0.5 (=)	1 (+)

Table B.13: Statistical comparison of imitators obtained the generated solver MC-D, and the generated solvers MC-E, MC-L, MC-M

	MC-D vs			MC-E vs		MC-L vs
Instances	**MC-E**	**MC-L**	**MC-M**	**MC-L**	**MC-M**	**MC-M**
100	0.5 (=)	0.5 (=)	1 (+)	0.5 (=)	1 (+)	1 (+)
1000	0.5 (=)	0.5 (=)	1 (+)	0.5 (=)	1 (+)	1 (+)
10000	0.5 (=)	0.5 (=)	1 (+)	0.5 (=)	1 (+)	1 (+)
100000	0.5 (=)	0.5 (=)	1 (+)	0.5 (=)	1 (+)	1 (+)

The Traveling Salesman Problem

This section provides the results of a detailed statistical analysis completed for some of the TSP solvers obtained by our experiments discussed in chapters [5-7]. We have completed 100 independent runs completed with 6,000 problem evaluations. The *National traveling salesman problems* [1] provide the details of the instances.

Table B.14: Statistical comparison of tours obtained by solvers TSP[A-J] for the instances u2152,usa13509, d18512, dj38, q194 and zi929

		u2152	usa13509	d18512	dj38	qa194	zi929
TSP-A	**mean**	5.405e-02	6.004e-02	4.250e-02	0.000e+00	2.563e-03	1.776e-02
	std	(9.4e-03)	(8.1e-03)	(1.7e-03)	(0.0e+00)	(1.8e-03)	(4.6e-03)
	median	5.293e-02	5.791e-02	4.243e-02	0.000e+00	2.673e-03	1.765e-02
	IQR	(1.0e-02)	(8.2e-03)	(2.1e-03)	(0.0e+00)	(3.1e-03)	(5.4e-03)
TSP-B	**mean**	4.333e-02	5.491e-02	4.006e-02	0.000e+00	2.092e-03	1.218e-02
	std	(7.1e-03)	(7.5e-03)	(1.8e-03)	((0.0e+00)	(1.3e-03)	(3.4e-03)
	median	4.298e-02	5.341e-02	3.976e-02	0.000e+00	2.673e-03	1.205e-02
	IQR	(8.4e-03)	(6.4e-03)	(2.2e-03)	(0.0e+00)	(2.7e-03)	(5.1e-03)
TSP-C	**mean**	7.511e-02	6.768e-02	4.629e-02	0.000e+00	2.722e-03	2.335e-02
	std	(1.2e-02)	(1.1e-02)	(2.0e-03)	(0.0e+00)	(1.7e-03)	(4.8e-03)
	median	7.295e-02	6.457e-02	4.597e-02	0.000e+00	3.368e-03	2.281e-02
	IQR	(1.6e-02)	(9.3e-03)	(2.9e-03)	(0.0e+00)	(3.1e-03)	(6.6e-03)
TSP-D	**mean**	4.055e-02	5.442e-02	4.077e-02	0.000e+00	2.821e-03	1.319e-02
	std	(1.6e-02)	(1.8e-02)	(5.1e-03)	(0.0e+00)	(8.0e-03)	(1.6e-02)
	median	3.958e-02	5.143e-02	4.010e-02	0.000e+00	2.673e-03	1.145e-02
	IQR	(7.3e-03)	(3.9e-03)	(2.4e-03)	(0.0e+00)	(2.7e-03)	(4.4e-03)
TSP-E	**mean**	5.603e-02	6.075e-02	4.349e-02	0.000e+00	3.188e-03	1.906e-02
	std	(9.0e-03)	(7.5e-03)	(1.8e-03)	(0.0e+00)	(2.7e-03)	(5.0e-03)
	median	5.549e-02	5.847e-02	4.337e-02	0.000e+00	2.887e-03	1.841e-02
	IQR	(1.1e-02)	(8.3e-03)	(2.4e-03)	(0.0e+00)	(1.2e-03)	(5.2e-03)
TSP-F	**mean**	4.759e-02	5.675e-02	4.217e-02	0.000e+00	2.253e-03	1.625e-02
	std	(7.3e-03)	(5.6e-03)	(1.9e-03)	(0.0e+00)	(1.5e-03)	(4.2e-03)
	median	4.753e-02	5.544e-02	4.198e-02	0.000e+00	2.673e-03	1.593e-02
	IQR	(9.2e-03)	(5.8e-03)	(2.9e-03)	(0.0e+00)	(2.5e-03)	(5.3e-03)
TSP-G	**mean**	5.213e-02	6.157e-02	4.231e-02	0.000e+00	4.651e-03	1.920e-02
	std	(8.0e-03)	(1.0e-02)	(2.1e-03)	(0.0e+00)	(5.0e-03)	(4.6e-03)
	median	5.171e-02	5.900e-02	4.199e-02	0.000e+00	3.208e-03	1.906e-02
	IQR	(1.1e-02)	(8.4e-03)	(2.5e-03)	(0.0e+00)	(2.3e-03)	(6.2e-03)
TSP-H	**mean**	4.127e-02	5.553e-02	4.019e-02	0.000e+00	2.884e-03	1.415e-02
	std	(6.7e-03)	(7.6e-03)	(2.1e-03)	(0.0e+00)	(2.6e-03)	(4.0e-03)
	median	4.090e-02	5.397e-02	3.968e-02	0.000e+00	2.673e-03	1.383e-02
	IQR	(9.2e-03)	(6.2e-03)	(2.5e-03)	(0.0e+00)	(8.6e-04)	(6.0e-03)
TSP-I	**mean**	4.259e-02	5.478e-02	4.029e-02	0.000e+00	2.889e-03	1.286e-02
	std	(6.3e-03)	(6.3e-03)	(1.8e-03)	(0.0e+00)	(2.6e-03)	(3.4e-03)
	median	4.173e-02	5.296e-02	4.002e-02	0.000e+00	2.673e-03	1.262e-02
	IQR	(7.6e-03)	(5.7e-03)	(2.6e-03)	(0.0e+00)	(0.0e+00)	(4.3e-03)
TSP-J	**mean**	4.234e-02	5.719e-02	4.061e-02	0.000e+00	2.316e-03	1.377e-02
	std	(7.2e-03)	(9.1e-03)	(1.7e-03)	(0.0e+00)	(1.5e-03)	(4.4e-03)
	median	4.155e-02	5.403e-02	4.039e-02	0.000e+00	2.673e-03	1.327e-02
	IQR	(1.0e-02)	(7.5e-03)	(2.4e-03)	(0.0e+00)	(7.5e-04)	(5.8e-03)

Table B.15: Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for the instances u2152,usa13509, d18512, dj38, q194 and zi929

		u2152	usa13509	d18512	dj38	qa194	zi929
TSP-K	**mean**	5.070e-02	6.226e-02	4.341e-02	0.000e+00	5.185e-03	2.157e-02
	std	(1.1e-02)	(1.0e-02)	(2.6e-03)	(0.0e+00)	(6.3e-03)	(9.5e-03)
	median	4.845e-02	5.880e-02	4.335e-02	0.000e+00	3.101e-03	1.882e-02
	IQR	(1.4e-02)	(1.2e-02)	(4.0e-03)	(0.0e+00)	(2.4e-03)	(9.7e-03)
TSP-L	**mean**	5.828e-02	6.520e-02	4.471e-02	0.000e+00	4.491e-03	2.284e-02
	std	(8.4e-03)	(1.0e-02)	(1.9e-03)	(0.0e+00)	(4.6e-03)	(5.3e-03)
	median	5.875e-02	6.285e-02	4.473e-02	0.000e+00	3.529e-03	2.238e-02
	IQR	(8.6e-03)	(1.3e-02)	(2.5e-03)	(0.0e+00)	(2.2e-03)	(7.7e-03)
TSP-M	**mean**	4.962e-02	5.985e-02	4.259e-02	0.000e+00	2.913e-03	1.780e-02
	std	(6.5e-03)	(9.0e-03)	(1.9e-03)	(0.0e+00)	(2.7e-03)	(4.0e-03)
	median	4.960e-02	5.695e-02	4.230e-02	0.000e+00	2.673e-03	1.703e-02
	IQR	(6.7e-03)	(7.4e-03)	(2.5e-03)	(0.0e+00)	(1.9e-03)	(5.4e-03)
TSP-N	**mean**	5.659e-02	6.360e-02	4.377e-02	0.000e+00	3.569e-03	1.999e-02
	std	(9.4e-03)	(1.0e-02)	(1.9e-03)	(0.0e+00)	(3.9e-03)	(4.9e-03)
	median	5.562e-02	6.100e-02	4.334e-02	0.000e+00	3.101e-03	1.975e-02
	IQR	(1.1e-02)	(9.4e-03)	(2.7e-03)	(0.0e+00)	(1.1e-03)	(5.8e-03)
TSP-O	**mean**	7.632e-02	7.206e-02	4.835e-02	5.858e-03	5.089e-02	5.424e-02
	std	(1.6e-02)	(1.2e-02)	(3.3e-03)	(1.4e-01)	(2.1e-02)	(1.6e-02)
	median	7.548e-02	6.918e-02	4.754e-02	0.000e+00	4.790e-02	5.320e-02
	IQR	(2.3e-02)	(1.2e-02)	(3.8e-03)	(0.0e+00))	(3.1e-02)	(2.0e-02)
TSP-P	**mean**	6.404e-02	6.667e-02	4.523e-02	0.000e+00	4.288e-03	2.424e-02
	std	(8.8e-03)	(9.0e-03)	(2.0e-03)	(0.0e+00)	(5.0e-03)	(5.3e-03)
	median	6.382e-02	6.394e-02	4.501e-02	0.000e+00	3.529e-03	2.374e-02
	IQR	(1.1e-02)	(1.1e-02)	(2.4e-03)	(0.0e+00)	(3.0e-03)	(8.1e-03)
TSP-Q	**mean**	5.277e-02	5.844e-02	4.307e-02	0.000e+00	2.599e-03	2.018e-02
	std	(7.3e-03)	(6.8e-03)	(2.1e-03)	(0.0e+00)	(2.3e-03)	(5.0e-03)
	median	5.299e-02	5.669e-02	4.294e-02	0.000e+00	2.673e-03	1.977e-02
	IQR	(1.1e-02)	(7.3e-03)	(2.9e-03)	(0.0e+00)	(3.1e-03)	(6.0e-03)
Ulder	**mean**	2.066e-01	2.430e-01	2.329e-01	4.009e-02	1.945e-01	2.089e-01
	std	(1.6e-02)	(5.0e-03)	(1.3e-02)	(4.8e-02)	(5.4e-02)	(3.1e-02)
	median	2.053e-01	2.447e-01	2.294e-01	1.788e-02	1.884e-01	2.040e-01
	IQR	(2.0e-02)	(5.5e-03)	(1.7e-02)	(7.2e-02)	(7.9e-02)	(3.9e-02)
Ozcan	**mean**	2.024e-01	2.411e-01	2.319e-01	1.023e-01	3.111e-01	2.120e-01
	std	(1.0e-02)	(9.0e-03)	(1.3e-02)	(8.3e-02)	(3.1e-02)	(2.7e-02)
	median	2.043e-01	2.447e-01	2.294e-01	5.228e-02	3.124e-01	2.079e-01
	IQR	(1.7e-02)	(5.5e-03)	(1.3e-02)	(1.4e-01)	(3.1e-02)	(3.6e-02)

Table B.16: Statistical comparison of tours obtained by solvers TSP[A-J] for the instances lu980,rw1621,nu3496, ca4663, tz6117, eg7146.

		lu980	rw1621	nu3496	ca4663	tz6117	eg7146
TSP-A	**mean**	3.332e-02	1.307e-01	9.042e-02	6.753e-02	7.858e-02	7.065e-02
	std	(6.9e-03)	(3.1e-02)	(1.7e-02)	(1.1e-02)	(1.0e-02)	(1.8e-02)
	median	3.311e-02	1.243e-01	8.762e-02	6.530e-02	7.698e-02	6.539e-02
	IQR	(9.3e-03)	(3.6e-02)	(2.1e-02)	(1.3e-02)	(1.1e-02)	(1.9e-02)
TSP-B	**mean**	2.356e-02	9.460e-02	7.625e-02	5.736e-02	6.598e-02	6.356e-02
	std	(6.1e-03)	(1.8e-02)	(1.8e-02)	(1.2e-02)	(9.5e-03)	(1.7e-02)
	median	2.324e-02	9.157e-02	7.270e-02	5.517e-02	6.387e-02	5.908e-02
	IQR	(7.9e-03)	(2.8e-02)	(2.0e-02)	(9.3e-03)	(8.2e-03)	(1.7e-02)
TSP-C	**mean**	4.762e-02	1.545e-01	9.970e-02	7.973e-02	9.784e-02	9.094e-02
	std	(1.1e-02)	(2.8e-02)	(2.0e-02)	(1.3e-02)	(1.7e-02)	(2.8e-02)
	median	4.550e-02	1.505e-01	9.432e-02	7.536e-02	9.488e-02	8.183e-02
	IQR	(1.5e-02)	(3.8e-02)	(2.5e-02)	(1.5e-02)	(1.4e-02)	(3.6e-02)
TSP-D	**mean**	2.232e-02	8.174e-02	7.014e-02	5.276e-02	6.201e-02	5.662e-02
	std	(1.3e-02)	(1.8e-02)	(9.7e-03)	(6.3e-03)	(1.0e-02)	(1.3e-02)
	median	2.072e-02	7.750e-02	6.855e-02	5.282e-02	6.082e-02	5.542e-02
	IQR	(7.0e-03)	(1.5e-02)	(1.1e-02)	(6.7e-03)	(8.2e-03)	(1.0e-02)
TSP-E	**mean**	3.475e-02	1.245e-01	9.275e-02	6.739e-02	7.967e-02	7.866e-02
	std	(7.3e-03)	(2.9e-02)	(2.3e-02)	(8.3e-03)	(1.6e-02)	(2.3e-02)
	median	3.369e-02	1.192e-01	8.738e-02	6.686e-02	7.651e-02	7.081e-02
	IQR	(1.1e-02)	(3.4e-02)	(2.3e-02)	(1.1e-02)	(1.4e-02)	(1.9e-02)
TSP-F	**mean**	2.846e-02	1.015e-01	8.169e-02	5.999e-02	7.086e-02	6.663e-02
	std	(5.9e-03)	(1.6e-02)	(1.5e-02)	(7.0e-03)	(7.6e-03)	(9.1e-03)
	median	2.809e-02	1.001e-01	7.790e-02	5.996e-02	6.921e-02	6.489e-02
	IQR	(8.4e-03)	(1.9e-02)	(1.8e-02)	(9.0e-03)	(9.4e-03)	(1.2e-02)
TSP-G	**mean**	3.275e-02	1.269e-01	8.970e-02	6.858e-02	7.756e-02	7.294e-02
	std	(5.9e-03)	(2.6e-02)	(1.8e-02)	(1.4e-02)	(1.3e-02)	(1.5e-02)
	median	3.245e-02	1.231e-01	8.607e-02	6.595e-02	7.530e-02	6.923e-02
	IQR	(7.8e-03)	(3.7e-02)	(2.6e-02)	(1.4e-02)	(1.2e-02)	(1.7e-02)
TSP-H	**mean**	2.410e-02	1.051e-01	8.327e-02	6.027e-02	6.722e-02	6.725e-02
	std	(5.2e-03)	(2.1e-02)	(1.7e-02)	(1.2e-02)	(7.2e-03)	(1.7e-02)
	median	2.332e-02	1.000e-01	8.065e-02	5.702e-02	6.641e-02	6.251e-02
	IQR	(7.1e-03)	(2.1e-02)	(2.1e-02)	(1.3e-02)	(9.0e-03)	(2.1e-02)
TSP-I	**mean**	2.453e-02	1.011e-01	7.868e-02	5.688e-02	6.715e-02	6.185e-02
	std	(5.4e-03)	(2.2e-02)	(1.6e-02)	(7.1e-03)	(8.2e-03)	(1.5e-02)
	median	2.425e-02	9.896e-02	7.501e-02	5.570e-02	6.567e-02	5.853e-02
	IQR	(6.6e-03)	(3.1e-02)	(1.4e-02)	(8.4e-03)	(9.3e-03)	(1.6e-02)
TSP-J	**mean**	2.443e-02	1.061e-01	7.840e-02	5.579e-02	6.560e-02	6.494e-02
	std	(5.0e-03)	(2.2e-02)	(1.5e-02)	(9.1e-03)	(7.1e-03)	(1.6e-02)
	median	2.429e-02	9.994e-02	7.544e-02	5.495e-02	6.425e-02	6.054e-02
	IQR	(5.8e-03)	(2.8e-02)	(1.6e-02)	(1.1e-02)	(8.0e-03)	(1.8e-02)

Table B.17: Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for the instances lu980,rw1621,nu3496, ca4663, tz6117, eg7146.

		lu980	rw1621	nu3496	ca4663	tz6117	eg7146
TSP-K	**mean**	3.663e-02	1.075e-01	8.784e-02	6.291e-02	7.589e-02	7.091e-02
	std	(1.6e-02)	(2.1e-02)	(2.0e-02)	(1.0e-02)	(1.3e-02)	(1.3e-02)
	median	3.338e-02	1.051e-01	8.202e-02	6.160e-02	7.252e-02	6.838e-02
	IQR	(1.9e-02)	(3.1e-02)	(2.4e-02)	(1.2e-02)	(1.8e-02)	(1.7e-02)
TSP-L	**mean**	3.720e-02	1.287e-01	9.919e-02	7.247e-02	8.405e-02	8.800e-02
	std	(6.8e-03)	(2.3e-02)	(2.5e-02)	(1.7e-02)	(1.5e-02)	(2.7e-02)
	median	3.655e-02	1.236e-01	9.159e-02	6.802e-02	7.983e-02	7.922e-02
	IQR	(7.7e-03)	(2.9e-02)	(3.0e-02)	(1.4e-02)	(1.3e-02)	(2.8e-02)
TSP-M	**mean**	3.079e-02	1.069e-01	8.208e-02	6.100e-02	7.257e-02	6.726e-02
	std	(5.5e-03)	(1.7e-02)	(1.4e-02)	(5.4e-03)	(6.9e-03)	(1.0e-02)
	median	3.060e-02	1.072e-01	7.955e-02	6.027e-02	7.209e-02	6.504e-02
	IQR	(6.1e-03)	(2.6e-02)	(1.7e-02)	(5.9e-03)	(9.7e-03)	(1.4e-02)
TSP-N	**mean**	3.565e-02	1.305e-01	9.186e-02	6.979e-02	8.212e-02	8.381e-02
	std	(7.2e-03)	(3.2e-02)	(2.0e-02)	(1.1e-02)	(1.6e-02)	(3.0e-02)
	median	3.461e-02	1.245e-01	8.909e-02	6.779e-02	7.908e-02	7.485e-02
	IQR	(9.5e-03)	(4.0e-02)	(2.3e-02)	(1.3e-02)	(1.2e-02)	(2.7e-02)
TSP-O	**mean**	7.095e-02	1.485e-01	1.181e-01	9.267e-02	9.504e-02	9.411e-02
	std	(1.6e-02)	(2.2e-02)	(2.1e-02)	(1.8e-02)	(9.9e-03)	(9.3e-02)
	median	6.936e-02	1.483e-01	1.162e-01	8.760e-02	9.396e-02	9.361e-02
	IQR	(2.3e-02)	(2.7e-02)	(3.2e-02)	(1.5e-02)	(1.3e-02)	(1.7e-02)
TSP-P	**mean**	4.350e-02	1.412e-01	9.740e-02	7.767e-02	8.962e-02	7.476e-02
	std	(7.6e-03)	(2.1e-02)	(1.5e-02)	(1.0e-02)	(8.4e-03)	(1.4e-02)
	median	4.237e-02	1.390e-01	9.629e-02	7.630e-02	8.930e-02	7.129e-02
	IQR	(1.0e-02)	(2.3e-02)	(1.9e-02)	(1.5e-02)	(1.1e-02)	(1.5e-02)
TSP-Q	**mean**	3.391e-02	1.098e-01	8.626e-02	6.113e-02	7.247e-02	6.890e-02
	std	(6.2e-03)	(1.7e-02)	(1.5e-02)	(6.2e-03)	(6.6e-03)	(1.0e-02)
	median	3.435e-02	1.104e-01	8.564e-02	6.044e-02	7.218e-02	6.761e-02
	IQR	(9.2e-03)	(2.3e-02)	(1.5e-02)	(7.9e-03)	(8.8e-03)	(1.3e-02)
Ulder	**mean**	2.320e-01	2.759e-01	2.514e-01	2.567e-01	2.520e-01	2.636e-01
	std	(1.9e-02)	(1.7e-02)	(1.2e-02)	(3.8e-02)	(6.9e-03)	(7.8e-03)
	median	2.331e-01	2.718e-01	2.510e-01	2.714e-01	2.520e-01	2.610e-01
	IQR	(2.1e-02)	(2.8e-02)	(1.4e-02)	(2.1e-02)	(1.4e-02)	(1.2e-02)
Ozcan	**mean**	2.317e-01	2.757e-01	2.534e-01	2.637e-01	2.435e-01	2.609e-01
	std	(1.4e-02)	(1.7e-02)	(1.3e-02)	(7.9e-03)	(3.3e-16)	(6.4e-03)
	median	2.301e-01	2.708e-01	2.535e-01	2.602e-01	2.435e-01	2.609e-01
	IQR	(2.1e-02)	(2.8e-02)	(1.9e-02)	(1.0e-02)	(0.0e+00)	(7.1e-03)

Table B.18: Statistical comparison of tours obtained by solvers TSP[A-J] for the instances ym7663,ei8246,ar9152, ja9847, gr9882, and kz9976.

		ym7663	ei8246	ar9152	ja9847	gr9882	kz9976
TSP-A	**mean**	1.018e-01	5.558e-02	1.052e-01	1.263e-01	9.454e-02	7.686e-02
	std	(1.6e-02)	(5.8e-03)	(1.5e-02)	(2.7e-02)	(1.5e-02)	(8.5e-03)
	median	9.891e-02	5.483e-02	1.030e-01	1.247e-01	9.224e-02	7.509e-02
	IQR	(2.0e-02)	(5.9e-03)	(2.0e-02)	(4.1e-02)	(2.2e-02)	(1.1e-02)
TSP-B	**mean**	9.535e-02	5.095e-02	9.474e-02	1.168e-01	9.206e-02	6.705e-02
	std	(2.3e-02)	(5.0e-03)	(1.8e-02)	(4.2e-02)	(1.9e-02)	(9.1e-03)
	median	8.891e-02	5.044e-02	9.094e-02	1.037e-01	8.477e-02	6.491e-02
	IQR	(1.8e-02)	(6.8e-03)	(1.6e-02)	(6.0e-02)	(2.8e-02)	(7.7e-03)
TSP-C	**mean**	1.230e-01	6.351e-02	1.263e-01	1.479e-01	1.160e-01	9.648e-02
	std	(2.9e-02)	(6.3e-03)	(2.3e-02)	(3.9e-02)	(2.4e-02)	(1.5e-02)
	median	1.139e-01	6.274e-02	1.207e-01	1.375e-01	1.160e-01	9.473e-02
	IQR	(4.2e-02)	(7.8e-03)	(2.6e-02)	(6.1e-02)	(4.0e-02)	(2.2e-02)
TSP-D	**mean**	8.590e-02	4.930e-02	9.239e-02	8.663e-02	7.806e-02	6.193e-02
	std	(1.5e-02)	(5.1e-03)	(1.8e-02)	(1.6e-02)	(1.9e-02)	(4.6e-03)
	median	8.304e-02	4.843e-02	9.031e-02	8.285e-02	7.522e-02	6.144e-02
	IQR	(1.1e-02)	(4.7e-03)	(1.3e-02)	(2.3e-02)	(1.1e-02)	(5.9e-03)
TSP-E	**mean**	1.050e-01	5.838e-02	1.044e-01	1.268e-01	9.451e-02	7.999e-02
	std	(2.0e-02)	(8.6e-03)	(1.9e-02)	(4.3e-02)	(1.7e-02)	(1.2e-02)
	median	9.831e-02	5.627e-02	1.002e-01	1.173e-01	8.892e-02	7.635e-02
	IQR	(1.9e-02)	(8.5e-03)	(1.7e-02)	(5.0e-02)	(2.0e-02)	(1.3e-02)
TSP-F	**mean**	9.473e-02	5.470e-02	9.980e-02	1.064e-01	8.317e-02	7.135e-02
	std	(1.0e-02)	(5.8e-03)	(1.1e-02)	(2.5e-02)	(7.6e-03)	(8.4e-03)
	median	9.348e-02	5.330e-02	9.731e-02	1.013e-01	8.334e-02	6.900e-02
	IQR	(1.2e-02)	(5.5e-03)	(1.3e-02)	(3.4e-02)	(1.0e-02)	(9.4e-03)
TSP-G	**mean**	1.017e-01	5.770e-02	1.059e-01	1.185e-01	9.271e-02	7.855e-02
	std	(1.5e-02)	(7.9e-03)	(1.4e-02)	(2.9e-02)	(1.4e-02)	(1.3e-02)
	median	9.938e-02	5.575e-02	1.042e-01	1.152e-01	9.036e-02	7.582e-02
	IQR	(1.4e-02)	(8.5e-03)	(2.2e-02)	(3.6e-02)	(2.0e-02)	(1.5e-02)
TSP-H	**mean**	9.628e-02	5.189e-02	9.955e-02	1.185e-01	8.896e-02	6.957e-02
	std	(1.4e-02)	(5.0e-03)	(1.4e-02)	(3.2e-02)	(1.3e-02)	(1.0e-02)
	median	9.368e-02	5.148e-02	9.788e-02	1.132e-01	8.653e-02	6.781e-02
	IQR	(1.7e-02)	(6.1e-03)	(2.1e-02)	(4.7e-02)	(1.7e-02)	(9.4e-03)
TSP-I	**mean**	9.663e-02	5.031e-02	9.785e-02	1.004e-01	8.337e-02	6.802e-02
	std	(1.6e-02)	(4.1e-03)	(1.4e-02)	(2.7e-02)	(1.1e-02)	(9.2e-03)
	median	9.246e-02	4.992e-02	9.585e-02	9.445e-02	8.098e-02	6.618e-02
	IQR	(2.1e-02)	(5.2e-03)	(2.1e-02)	(3.2e-02)	(1.4e-02)	(8.9e-03)
TSP-J	**mean**	9.440e-02	5.200e-02	1.000e-01	1.100e-01	8.930e-02	6.997e-02
	std	(1.5e-02)	(5.1e-03)	(1.4e-02)	(2.5e-02)	(1.2e-02)	(8.9e-03)
	median	9.124e-02	5.132e-02	9.971e-02	1.080e-01	8.833e-02	6.908e-02
	IQR	(1.5e-02)	(5.9e-03)	(1.7e-02)	(3.8e-02)	(1.8e-02)	(1.1e-02)

Table B.19: Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for the instances ym7663,ei8246,ar9152, ja9847, gr9882, and kz9976.

		ym7663	ei8246	ar9152	ja9847	gr9882	kz9976
TSP-K	**mean**	9.652e-02	5.861e-02	1.068e-01	1.042e-01	8.540e-02	7.503e-02
	std	(1.0e-02)	(7.6e-03)	(1.3e-02)	(2.3e-02)	(8.4e-03)	(1.2e-02)
	median	9.568e-02	5.696e-02	1.048e-01	9.913e-02	8.489e-02	7.253e-02
	IQR	(1.4e-02)	(8.4e-03)	(1.8e-02)	(3.2e-02)	(1.2e-02)	(1.5e-02)
TSP-L	**mean**	1.139e-01	6.056e-02	1.163e-01	1.379e-01	1.028e-01	8.585e-02
	std	(2.5e-02)	(8.2e-03)	(1.9e-02)	(4.7e-02)	(2.1e-02)	(1.6e-02)
	median	1.046e-01	5.840e-02	1.114e-01	1.266e-01	9.713e-02	8.160e-02
	IQR	(2.6e-02)	(7.5e-03)	(2.3e-02)	(7.6e-02)	(2.8e-02)	(1.9e-02)
TSP-M	**mean**	9.554e-02	5.528e-02	1.012e-01	1.044e-01	8.539e-02	7.193e-02
	std	(9.8e-03)	(4.6e-03)	(1.2e-02)	(2.2e-02)	(7.8e-03)	(7.7e-03)
	median	9.434e-02	5.492e-02	1.003e-01	1.046e-01	8.506e-02	7.045e-02
	IQR	(1.1e-02)	(4.6e-03)	(1.6e-02)	(3.3e-02)	(1.2e-02)	(9.0e-03)
TSP-N	**mean**	1.050e-01	5.855e-02	1.081e-01	1.370e-01	9.990e-02	8.156e-02
	std	(1.9e-02)	(7.0e-03)	(1.5e-02)	(4.1e-02)	(2.1e-02)	(1.2e-02)
	median	9.961e-02	5.755e-02	1.066e-01	1.259e-01	9.535e-02	7.978e-02
	IQR	(1.8e-02)	(7.3e-03)	(2.0e-02)	(6.1e-02)	(3.0e-02)	(1.5e-02)
TSP-O	**mean**	1.115e-01	7.036e-02	1.184e-01	1.147e-01	9.369e-02	9.069e-02
	std	(1.1e-02)	(9.7e-03)	(1.2e-02)	(2.0e-02)	(8.2e-03)	(1.4e-02)
	median	1.101e-01	6.816e-02	1.177e-01	1.123e-01	9.252e-02	8.697e-02
	IQR	(1.5e-02)	(1.4e-02)	(1.6e-02)	(2.8e-02)	(1.1e-02)	(1.7e-02)
TSP-P	**mean**	1.098e-01	6.367e-02	1.179e-01	1.323e-01	1.023e-01	9.144e-02
	std	(1.3e-02)	(7.4e-03)	(1.5e-02)	(2.6e-02)	(1.5e-02)	(1.4e-02)
	median	1.078e-01	6.275e-02	1.164e-01	1.285e-01	1.017e-01	8.894e-02
	IQR	(1.7e-02)	(8.4e-03)	(2.3e-02)	(4.1e-02)	(2.1e-02)	(1.9e-02)
TSP-Q	**mean**	9.325e-02	5.710e-02	1.027e-01	1.044e-01	8.439e-02	7.184e-02
	std	(9.4e-03)	(7.4e-03)	(1.3e-02)	(2.4e-02)	(6.9e-03)	(6.7e-03)
	median	9.236e-02	5.530e-02	1.024e-01	9.959e-02	8.400e-02	7.153e-02
	IQR	(1.2e-02)	(7.6e-03)	(1.8e-02)	(3.8e-02)	(9.8e-03)	(7.8e-03)
Ulder	**mean**	2.712e-01	2.519e-01	2.670e-01	2.661e-01	2.648e-01	2.658e-01
	std	(8.7e-03)	(5.2e-03)	(5.6e-03)	(8.5e-03)	(8.0e-03)	(9.9e-03)
	median	2.713e-01	2.515e-01	2.674e-01	2.658e-01	2.633e-01	2.615e-01
	IQR	(1.2e-02)	(5.0e-03)	(5.2e-03)	(6.9e-03)	(1.4e-02)	(7.3e-03)
Ozcan	**mean**	2.723e-01	2.488e-01	2.649e-01	2.803e-01	2.659e-01	2.623e-01
	std	(1.1e-02)	(9.0e-03)	(1.1e-02)	(1.5e-02)	(1.1e-02)	(6.3e-03)
	median	2.720e-01	2.439e-01	2.669e-01	2.789e-01	2.637e-01	2.612e-01
	IQR	(1.5e-02)	(1.2e-02)	(1.5e-02)	(2.3e-02)	(1.5e-02)	(5.4e-03)

Table B.20: Statistical comparison of tours obtained by solvers TSP[A-J] for some instances with a greater number 10,000 cities.

		fi10639	ho14473	mo14185	it16862	vm22775	sw24978	bm33708
TSP-A	**mean**	6.342e-02	1.508e-01	6.144e-02	9.769e+00	9.373e-02	7.365e-02	7.352e-02
	std	(5.4e-03)	(1.8e-02)	(5.5e-03)	(1.1e-01)	(1.6e-02)	(7.4e-03)	(5.3e-03)
	median	6.302e-02	1.482e-01	6.046e-02	9.757e+00	9.392e-02	7.201e-02	7.289e-02
	IQR	(7.4e-03)	(2.2e-02)	(5.4e-03)	(1.6e-01)	(2.0e-02)	(1.0e-02)	(7.8e-03)
TSP-B	**mean**	5.926e-02	1.452e-01	5.666e-02	9.714e+00	9.951e-02	6.906e-02	6.915e-02
	std	(6.2e-03)	(2.4e-02)	(4.9e-03)	(1.5e-01)	(2.6e-02)	(9.0e-03)	(7.4e-03)
	median	5.858e-02	1.393e-01	5.533e-02	9.667e+00	9.759e-02	6.667e-02	6.776e-02
	IQR	(7.9e-03)	(2.8e-02)	(6.2e-03)	(1.6e-01)	(4.4e-02)	(8.9e-03)	(5.9e-03)
TSP-C	**mean**	7.144e-02	1.806e-01	7.098e-02	9.875e+00	1.328e-01	8.565e-02	8.492e-02
	std	(7.9e-03)	(2.9e-02)	(8.3e-03)	(1.6e-01)	(3.8e-02)	(1.2e-02)	(1.3e-02)
	median	7.057e-02	1.748e-01	6.900e-02	9.854e+00	1.312e-01	8.318e-02	8.019e-02
	IQR	(7.3e-03)	(3.8e-02)	(8.2e-03)	(2.2e-01)	(6.2e-02)	(1.9e-02)	(1.4e-02)
TSP-D	**mean**	5.926e-02	1.241e-01	5.633e-02	9.699e+00	1.513e-01	1.474e-01	1.495e-01
	std	(1.4e-02)	(1.0e-02)	(1.1e-02)	(8.1e-02)	(5.4e-02)	(5.6e-02)	(5.3e-02)
	median	5.774e-02	1.218e-01	5.448e-02	9.695e+00	1.364e-01	1.383e-01	1.493e-01
	IQR	(6.6e-03)	(1.2e-02)	(4.0e-03)	(1.4e-01)	(7.8e-02)	(1.1e-01)	(9.9e-02)
TSP-E	**mean**	6.664e-02	1.476e-01	6.372e-02	9.790e+00	1.115e-01	7.455e-02	7.308e-02
	std	(6.2e-03)	(2.3e-02)	(5.6e-03)	(1.6e-01)	(2.7e-02)	(8.3e-03)	(6.5e-03)
	median	6.583e-02	1.426e-01	6.256e-02	9.744e+00	1.058e-01	7.206e-02	7.129e-02
	IQR	(7.3e-03)	(3.2e-02)	(6.1e-03)	(1.8e-01)	(3.6e-02)	(1.2e-02)	(7.5e-03)
TSP-F	**mean**	6.262e-02	1.317e-01	5.922e-02	9.712e+00	8.696e-02	6.653e-02	7.056e-02
	std	(5.0e-03)	(1.1e-02)	(3.4e-03)	(9.1e-02)	(1.4e-02)	(3.8e-03)	(4.0e-03)
	median	6.236e-02	1.311e-01	5.854e-02	9.697e+00	9.197e-02	6.559e-02	7.016e-02
	IQR	(5.1e-03)	(1.8e-02)	(5.4e-03)	(1.6e-01)	(2.3e-02)	(3.8e-03)	(5.2e-03)
TSP-G	**mean**	6.560e-02	1.451e-01	6.142e-02	9.779e+00	9.651e-02	7.155e-02	7.496e-02
	std	(6.8e-03)	(1.6e-02)	(5.4e-03)	(1.1e-01)	(1.7e-02)	(5.8e-03)	(6.0e-03)
	median	6.461e-02	1.467e-01	6.009e-02	9.786e+00	9.715e-02	7.102e-02	7.490e-02
	IQR	(6.6e-03)	(2.1e-02)	(6.7e-03)	(1.5e-01)	(2.4e-02)	(8.3e-03)	(8.1e-03)
TSP-H	**mean**	6.051e-02	1.416e-01	5.724e-02	9.724e+00	8.977e-02	6.789e-02	7.078e-02
	std	(5.8e-03)	(1.6e-02)	(4.5e-03)	(1.1e-01)	(1.6e-02)	(6.1e-03)	(5.0e-03)
	median	6.000e-02	1.416e-01	5.658e-02	9.728e+00	8.965e-02	6.698e-02	7.013e-02
	IQR	(6.3e-03)	(2.3e-02)	(5.3e-03)	(1.5e-01)	(2.4e-02)	(7.0e-03)	(7.9e-03)
TSP-I	**mean**	5.935e-02	1.358e-01	5.592e-02	9.683e+00	8.337e-02	6.658e-02	6.954e-02
	std	(5.3e-03)	(1.2e-02)	(4.2e-03)	(9.9e-02)	(1.4e-02)	(5.7e-03)	(5.6e-03)
	median	5.874e-02	1.362e-01	5.504e-02	9.651e+00	8.362e-02	6.557e-02	6.877e-02
	IQR	(6.7e-03)	(1.7e-02)	(5.8e-03)	(1.4e-01)	(2.2e-02)	(7.5e-03)	(6.4e-03)
TSP-J	**mean**	5.941e-02	1.431e-01	5.792e-02	9.721e+00	9.192e-02	6.806e-02	7.128e-02
	std	(5.2e-03)	(1.4e-02)	(4.9e-03)	(9.3e-02)	(1.7e-02)	(5.6e-03)	(6.5e-03)
	median	5.996e-02	1.421e-01	5.652e-02	9.739e+00	9.194e-02	6.688e-02	7.037e-02
	IQR	(7.4e-03)	(1.8e-02)	(6.0e-03)	(1.3e-01)	(2.2e-02)	(7.6e-03)	(6.7e-03)

Table B.21: Statistical comparison of tours obtained by generated solvers TSP[K-Q], Ulder [319] and Ozcan [247] for some instances with a greater number 10,000 cities.

		fi10639	**ho14473**	**mo14185**	**it16862**	**vm22775**	**sw24978**	**bm33708**
TSP-K	**mean**	6.431e-02	1.333e-01	6.108e-02	9.745e+00	8.597e-02	6.939e-02	7.175e-02
	std	(7.4e-03)	(1.1e-02)	(5.5e-03)	(8.2e-02)	(1.3e-02)	(5.7e-03)	(4.6e-03)
	median	6.325e-02	1.333e-01	6.030e-02	9.749e+00	8.933e-02	6.819e-02	7.140e-02
	IQR	(8.0e-03)	(1.8e-02)	(6.1e-03)	(1.2e-01)	(2.4e-02)	(7.4e-03)	(6.6e-03)
TSP-L	**mean**	6.776e-02	1.575e-01	6.675e-02	9.810e+00	1.113e-01	7.657e-02	7.789e-02
	std	(6.7e-03)	(2.2e-02)	(7.3e-03)	(1.4e-01)	(2.6e-02)	(9.8e-03)	(8.0e-03)
	median	6.715e-02	1.550e-01	6.486e-02	9.796e+00	1.107e-01	7.510e-02	7.769e-02
	IQR	(8.2e-03)	(3.0e-02)	(9.4e-03)	(1.9e-01)	(4.1e-02)	(1.2e-02)	(1.0e-02)
TSP-M	**mean**	6.326e-02	1.343e-01	6.035e-02	9.730e+00	8.460e-02	6.895e-02	7.118e-02
	std	(5.3e-03)	(1.0e-02)	(4.1e-03)	(9.0e-02)	(1.3e-02)	(5.1e-03)	(4.6e-03)
	median	6.344e-02	1.333e-01	5.965e-02	9.724e+00	9.001e-02	6.831e-02	7.081e-02
	IQR	(7.0e-03)	(1.4e-02)	(5.7e-03)	(1.5e-01)	(2.4e-02)	(6.7e-03)	(6.1e-03)
TSP-N	**mean**	6.562e-02	1.532e-01	6.395e-02	9.809e+00	1.039e-01	7.507e-02	7.624e-02
	std	(4.6e-03)	(1.8e-02)	(5.6e-03)	(1.4e-01)	(2.6e-02)	(9.2e-03)	(9.1e-03)
	median	6.543e-02	1.507e-01	6.279e-02	9.784e+00	1.005e-01	7.271e-02	7.317e-02
	IQR	(5.7e-03)	(2.2e-02)	(6.3e-03)	(1.5e-01)	(4.1e-02)	(1.3e-02)	(9.2e-03)
TSP-O	**mean**	7.993e-02	1.358e-01	7.103e-02	9.849e+00	9.254e-02	7.688e-02	7.752e-02
	std	(7.7e-03)	(1.2e-02)	(4.7e-03)	(9.1e-02)	(1.5e-02)	(6.0e-03)	(5.4e-03)
	median	7.827e-02	1.352e-01	7.090e-02	9.847e+00	9.415e-02	7.648e-02	7.720e-02
	IQR	(1.1e-02)	(1.7e-02)	(5.4e-03)	(1.1e-01)	(2.7e-02)	(8.6e-03)	(7.6e-03)
TSP-P	**mean**	7.250e-02	1.582e-01	7.004e-02	9.830e+00	1.106e-01	8.199e-02	8.221e-02
	std	(6.4e-03)	(1.4e-02)	(6.3e-03)	(1.2e-01)	(1.8e-02)	(8.3e-03)	(7.1e-03)
	median	7.198e-02	1.567e-01	6.837e-02	9.819e+00	1.079e-01	8.032e-02	8.138e-02
	IQR	(6.9e-03)	(1.8e-02)	(8.9e-03)	(1.8e-01)	(2.8e-02)	(1.2e-02)	(8.5e-03)
TSP-Q	**mean**	6.365e-02	1.330e-01	6.078e-02	9.725e+00	8.514e-02	6.857e-02	7.204e-02
	std	(5.7e-03)	(1.2e-02)	(4.6e-03)	(7.7e-02)	(1.3e-02)	(5.0e-03)	(4.7e-03)
	median	6.350e-02	1.317e-01	5.988e-02	9.728e+00	8.866e-02	6.761e-02	7.175e-02
	IQR	(7.5e-03)	(1.9e-02)	(5.8e-03)	(1.2e-01)	(2.2e-02)	(5.7e-03)	(7.1e-03)
Ulder	**mean**	2.458e-01	2.606e-01	2.401e-01	1.174e+01	2.542e-01	2.415e-01	2.516e-01
	std	(3.3e-02)	(6.2e-03)	(9.0e-03)	(1.6e-01)	(1.4e-02)	(3.9e-03)	(2.9e-03)
	median	2.558e-01	2.622e-01	2.374e-01	1.165e+01	2.510e-01	2.403e-01	2.521e-01
	IQR	(1.4e-02)	(1.4e-02)	(9.3e-03)	(2.5e-01)	(2.6e-02)	(7.9e-03)	(4.2e-03)
Ozcan	**mean**	2.555e-01	2.608e-01	2.373e-01	1.174e+01	2.551e-01	2.472e-01	2.486e-01
	std	(6.8e-03)	(5.9e-03)	(7.0e-03)	(1.6e-01)	(1.3e-02)	(5.4e-03)	(4.4e-03)
	median	2.558e-01	2.622e-01	2.375e-01	1.165e+01	2.528e-01	2.455e-01	2.488e-01
	IQR	(9.7e-03)	(1.4e-02)	(8.7e-03)	(2.5e-01)	(2.5e-02)	(6.7e-03)	(6.5e-03)

Table B.22: Statistical comparison of tours obtained by Ulder [319] and Ozcan [247] and the generated solvers TSP-[A-E]

	Ulder vs					**Ozcan vs**				
	TSP-A	**TSP-B**	**TSP-C**	**TSP-D**	**TSP-E**	**TSP-A**	**TSP-B**	**TSP-C**	**TSP-D**	**TSP-E**
u2152	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
usa13509	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
d18512	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
dj38	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
qa194	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
zi929	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
lu980	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
rw1621	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
nu3496	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ca4663	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
tz6117	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
eg7146	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ym7663	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ei8246	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ar9152	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ja9847	1 (−)	1 (−)	0.99 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
gr9882	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
kz9976	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
fi10639	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ho14473	1 (−)	1 (−)	0.98 (−)	1 (−)	1 (−)	1 (−)	1 (−)	0.99 (−)	1 (−)	1 (−)
mo14185	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
it16862	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
vm22775	1 (−)	1 (−)	1 (−)	0.95 (−)	0.95 (−)	1 (−)	1 (−)	1 (−)	0.96 (−)	0.96 (−)
sw24978	1 (−)	1 (−)	1 (−)	0.97 (−)	0.97 (−)	1 (−)	1 (−)	1 (−)	0.99 (−)	0.99 (−)
bm33708	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	0.99 (−)	0.99 (−)

Table B.23: Statistical comparison of tours obtained by Ulder [319] and the generated solvers TSP-[F-Q]

	Ulder vs											
	TSP-F	**TSP-G**	**TSP-H**	**TSP-I**	**TSP-J**	**TSP-K**	**TSP-L**	**TSP-M**	**TSP-N**	**TSP-O**	**TSP-P**	**TSP-Q**
u2152	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
usa13509	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
d18512	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
dj38	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	0.94 (−)	0.81 (−)	0.94 (−)	0.94 (−)
qa194	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
zi929	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
lu980	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
rw1621	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
nu3496	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ca4663	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
tz6117	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
eg7146	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ym7663	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ei8246	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ar9152	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ja9847	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
gr9882	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
kz9976	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
fi10639	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
ho14473	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
mo14185	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
it16862	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
vm22775	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
sw24978	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)
bm33708	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)

Table B.24: Statistical comparison of tours obtained by Ozcan [247] and the generated solvers TSP-[F-Q]

	Ozcan vs TSP-F		Ozcan vs TSP-G		Ozcan vs TSP-H		Ozcan vs TSP-I		Ozcan vs TSP-J		Ozcan vs TSP-K		Ozcan vs TSP-L		Ozcan vs TSP-M		Ozcan vs TSP-N		Ozcan vs TSP-O		Ozcan vs TSP-P		Ozcan vs TSP-Q	
u2152	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
usa13509	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
d18512	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
dj38	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	0.95	(−)	1	(−)	1	(−)
qa194	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
zi929	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
lu980	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
rw1621	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
nu3496	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
ca4663	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
tz6117	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
eg7146	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
ym7663	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
ei8246	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
ar9152	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
ja9847	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
gr9882	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
kz9976	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
fi10639	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
ho14473	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
mo14185	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
it16862	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
vm22775	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
sw24978	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)
bm33708	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)	1	(−)

Table B.25: Statistical comparison of tours obtained by the generated solvers TSP-[A-E]

	TSP-A vs TSP-B		TSP-A vs TSP-C		TSP-A vs TSP-D		TSP-A vs TSP-E		TSP-B vs TSP-C		TSP-B vs TSP-D		TSP-B vs TSP-E		TSP-C vs TSP-D		TSP-C vs TSP-E		TSP-D vs TSP-E	
u2152	0.84	(−)	0.93	(+)	0.93	(−)	0.57	(=)	0.99	(+)	0.68	(−)	0.88	(+)	0.99	(−)	0.91	(−)	0.95	(+)
usa13509	0.77	(−)	0.79	(+)	0.85	(−)	0.54	(=)	0.92	(+)	0.61	(−)	0.79	(+)	0.95	(−)	0.76	(−)	0.87	(+)
d18512	0.85	(−)	0.94	(+)	0.81	(−)	0.66	(+)	0.99	(+)	0.55	(=)	0.92	(+)	0.98	(−)	0.86	(−)	0.88	(+)
dj38	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)
qa194	0.6	(−)	0.56	(+)	0.6	(−)	0.57	(=)	0.69	(+)	0.5	(=)	0.68	(+)	0.69	(−)	0.51	(=)	0.68	(+)
zi929	0.84	(−)	0.81	(+)	0.86	(−)	0.58	(=)	0.98	(+)	0.55	(=)	0.89	(+)	0.97	(−)	0.75	(−)	0.9	(+)
lu980	0.85	(−)	0.87	(+)	0.93	(−)	0.55	(=)	0.98	(+)	0.61	(−)	0.88	(+)	0.99	(−)	0.84	(−)	0.95	(+)
rw1621	0.86	(−)	0.74	(+)	0.95	(−)	0.56	(=)	0.97	(+)	0.74	(−)	0.83	(+)	0.99	(−)	0.8	(−)	0.95	(+)
nu3496	0.75	(−)	0.64	(+)	0.87	(−)	0.52	(=)	0.84	(+)	0.59	(−)	0.76	(+)	0.94	(−)	0.62	(−)	0.88	(+)
ca4663	0.8	(−)	0.8	(+)	0.92	(−)	0.53	(=)	0.95	(+)	0.64	(−)	0.83	(+)	0.99	(−)	0.81	(−)	0.94	(+)
tz6117	0.87	(−)	0.88	(+)	0.95	(−)	0.52	(=)	0.97	(+)	0.66	(−)	0.85	(+)	0.99	(−)	0.86	(−)	0.94	(+)
eg7146	0.66	(−)	0.75	(+)	0.8	(−)	0.63	(+)	0.85	(+)	0.64	(−)	0.77	(+)	0.95	(−)	0.66	(−)	0.9	(+)
ym7663	0.67	(−)	0.73	(+)	0.84	(−)	0.52	(=)	0.83	(+)	0.63	(−)	0.7	(+)	0.95	(−)	0.71	(−)	0.87	(+)
ei8246	0.74	(−)	0.84	(+)	0.85	(−)	0.6	(=)	0.95	(+)	0.62	(−)	0.8	(+)	0.97	(−)	0.74	(−)	0.89	(+)
ar9152	0.74	(−)	0.79	(+)	0.78	(−)	0.54	(=)	0.91	(+)	0.53	(=)	0.71	(+)	0.95	(−)	0.82	(−)	0.76	(+)
ja9847	0.63	(−)	0.65	(+)	0.9	(−)	0.55	(=)	0.74	(+)	0.73	(−)	0.59	(=)	0.95	(−)	0.67	(−)	0.84	(+)
gr9882	0.58	(=)	0.77	(+)	0.87	(−)	0.52	(=)	0.79	(+)	0.77	(−)	0.57	(=)	0.96	(−)	0.77	(−)	0.86	(+)
kz9976	0.84	(−)	0.88	(+)	0.96	(−)	0.55	(=)	0.96	(+)	0.69	(−)	0.87	(+)	1	(−)	0.82	(−)	0.97	(+)
fi10639	0.7	(−)	0.83	(+)	0.77	(−)	0.66	(+)	0.91	(+)	0.56	(=)	0.81	(+)	0.94	(−)	0.71	(−)	0.87	(+)
ho14473	0.61	(−)	0.82	(+)	0.92	(−)	0.58	(=)	0.85	(+)	0.82	(−)	0.53	(=)	0.98	(−)	0.83	(−)	0.85	(+)
mo14185	0.77	(−)	0.87	(+)	0.86	(−)	0.64	(+)	0.96	(+)	0.57	(=)	0.86	(+)	0.98	(−)	0.81	(−)	0.93	(+)
it16862	0.68	(−)	0.7	(+)	0.68	(−)	0.5	(=)	0.8	(+)	0.53	(=)	0.68	(+)	0.84	(−)	0.68	(−)	0.68	(+)
vm22775	0.55	(+)	0.8	(+)	0.86	(+)	0.7	(+)	0.75	(+)	0.8	(+)	0.62	(+)	0.58	(=)	0.66	(−)	0.72	(−)
sw24978	0.72	(−)	0.82	(+)	0.95	(+)	0.52	(=)	0.9	(+)	0.97	(+)	0.74	(+)	0.84	(+)	0.8	(−)	0.94	(−)
bm33708	0.76	(−)	0.83	(+)	0.97	(+)	0.55	(=)	0.94	(+)	0.98	(+)	0.73	(+)	0.87	(+)	0.86	(−)	0.97	(−)

Table B.26: Statistical comparison of tours obtained by the generated solver TSP-A and the generated solvers TSP-[F-Q]

	TSP-A vs TSP-F	TSP-A vs TSP-G	TSP-A vs TSP-H	TSP-A vs TSP-I	TSP-A vs TSP-J	TSP-A vs TSP-K	TSP-A vs TSP-L	TSP-A vs TSP-M	TSP-A vs TSP-N	TSP-A vs TSP-O	TSP-A vs TSP-P	TSP-A vs TSP-Q
u2152	0.71 (−)	0.56 (=)	0.88 (−)	0.86 (−)	0.85 (−)	0.61 (−)	0.66 (+)	0.65 (−)	0.58 (=)	0.9 (+)	0.8 (+)	0.52 (=)
usa13509	0.66 (−)	0.52 (=)	0.73 (−)	0.77 (−)	0.7 (−)	0.55 (=)	0.67 (+)	0.54 (=)	0.62 (+)	0.86 (+)	0.76 (+)	0.58 (=)
d18512	0.56 (=)	0.56 (=)	0.81 (−)	0.81 (−)	0.79 (−)	0.61 (+)	0.82 (+)	0.51 (=)	0.69 (+)	0.98 (+)	0.86 (+)	0.57 (=)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.55 (=)	0.62 (+)	0.51 (=)	0.51 (=)	0.56 (=)	0.62 (+)	0.65 (+)	0.51 (=)	0.57 (=)	1 (+)	0.61 (+)	0.5 (=)
zi929	0.6 (=)	0.59 (=)	0.72 (−)	0.81 (−)	0.73 (−)	0.61 (+)	0.77 (+)	0.5 (=)	0.63 (+)	0.99 (+)	0.82 (+)	0.65 (+)
lu980	0.69 (−)	0.51 (=)	0.86 (−)	0.85 (−)	0.85 (−)	0.52 (=)	0.66 (+)	0.61 (−)	0.59 (=)	0.99 (+)	0.85 (+)	0.54 (=)
rw1621	0.81 (−)	0.52 (=)	0.78 (−)	0.8 (−)	0.76 (−)	0.73 (−)	0.5 (=)	0.74 (−)	0.51 (=)	0.71 (+)	0.66 (+)	0.71 (−)
nu3496	0.66 (−)	0.52 (=)	0.62 (−)	0.72 (−)	0.72 (−)	0.57 (=)	0.6 (=)	0.64 (−)	0.52 (=)	0.85 (+)	0.64 (+)	0.56 (=)
ca4663	0.71 (−)	0.51 (=)	0.72 (−)	0.81 (−)	0.83 (−)	0.62 (−)	0.59 (=)	0.7 (−)	0.57 (=)	0.92 (+)	0.78 (+)	0.68 (−)
tz6117	0.75 (−)	0.56 (=)	0.84 (−)	0.84 (−)	0.89 (−)	0.6 (−)	0.62 (+)	0.68 (−)	0.56 (=)	0.89 (+)	0.82 (+)	0.68 (−)
eg7146	0.52 (=)	0.58 (=)	0.57 (=)	0.67 (−)	0.62 (−)	0.55 (=)	0.74 (+)	0.51 (=)	0.65 (+)	0.87 (+)	0.62 (+)	0.53 (=)
ym7663	0.63 (−)	0.5 (=)	0.61 (−)	0.61 (−)	0.65 (−)	0.58 (=)	0.64 (+)	0.61 (−)	0.54 (=)	0.71 (+)	0.67 (+)	0.66 (−)
ei8246	0.56 (=)	0.56 (=)	0.69 (−)	0.78 (−)	0.69 (−)	0.63 (+)	0.72 (+)	0.5 (=)	0.65 (+)	0.93 (+)	0.84 (+)	0.55 (=)
ar9152	0.61 (−)	0.52 (=)	0.6 (=)	0.65 (−)	0.6 (=)	0.55 (=)	0.67 (+)	0.57 (=)	0.56 (=)	0.77 (+)	0.73 (+)	0.53 (=)
ja9847	0.71 (−)	0.59 (=)	0.59 (=)	0.78 (−)	0.67 (−)	0.73 (−)	0.54 (=)	0.73 (−)	0.55 (=)	0.63 (−)	0.56 (=)	0.73 (−)
gr9882	0.73 (−)	0.53 (=)	0.61 (−)	0.73 (−)	0.59 (=)	0.68 (−)	0.61 (+)	0.68 (−)	0.55 (=)	0.52 (=)	0.65 (+)	0.7 (−)
kz9976	0.7 (−)	0.51 (=)	0.77 (−)	0.8 (−)	0.74 (−)	0.59 (−)	0.69 (+)	0.68 (−)	0.62 (+)	0.82 (+)	0.82 (+)	0.68 (−)
fi10639	0.54 (=)	0.6 (=)	0.65 (−)	0.7 (−)	0.7 (−)	0.53 (=)	0.7 (+)	0.51 (=)	0.63 (+)	0.97 (+)	0.87 (+)	0.52 (=)
ho14473	0.82 (−)	0.59 (=)	0.64 (−)	0.75 (−)	0.62 (−)	0.79 (−)	0.58 (=)	0.78 (−)	0.54 (=)	0.75 (−)	0.65 (+)	0.79 (−)
mo14185	0.61 (−)	0.51 (=)	0.75 (−)	0.8 (−)	0.71 (−)	0.52 (=)	0.74 (+)	0.54 (=)	0.66 (+)	0.9 (+)	0.87 (+)	0.52 (=)
it16862	0.65 (−)	0.53 (=)	0.61 (−)	0.73 (−)	0.62 (−)	0.55 (=)	0.58 (=)	0.59 (=)	0.57 (=)	0.72 (+)	0.64 (+)	0.61 (−)
vm22775	0.61 (−)	0.55 (=)	0.57 (=)	0.68 (−)	0.53 (=)	0.64 (−)	0.7 (+)	0.66 (−)	0.61 (+)	0.5 (=)	0.75 (+)	0.66 (−)
sw24978	0.83 (−)	0.57 (=)	0.75 (−)	0.8 (−)	0.74 (−)	0.68 (−)	0.58 (=)	0.71 (−)	0.53 (=)	0.66 (+)	0.79 (+)	0.72 (−)
bm33708	0.66 (−)	0.57 (=)	0.64 (−)	0.71 (−)	0.64 (−)	0.59 (=)	0.67 (+)	0.62 (−)	0.56 (=)	0.7 (+)	0.85 (+)	0.57 (=)

Table B.27: Statistical comparison of tours obtained by the generated solver TSP-B and the generated solvers TSP-[F-Q]

	TSP-B vs TSP-F	TSP-B vs TSP-G	TSP-B vs TSP-H	TSP-B vs TSP-I	TSP-B vs TSP-J	TSP-B vs TSP-K	TSP-B vs TSP-L	TSP-B vs TSP-M	TSP-B vs TSP-N	TSP-B vs TSP-O	TSP-B vs TSP-P	TSP-B vs TSP-Q
u2152	0.67 (+)	0.81 (+)	0.58 (=)	0.54 (=)	0.54 (=)	0.7 (+)	0.92 (+)	0.76 (+)	0.89 (+)	0.98 (+)	0.96 (+)	0.82 (+)
usa13509	0.66 (+)	0.77 (+)	0.52 (=)	0.51 (=)	0.59 (=)	0.78 (+)	0.86 (+)	0.73 (+)	0.84 (+)	0.94 (+)	0.91 (+)	0.71 (+)
d18512	0.8 (+)	0.81 (+)	0.5 (=)	0.54 (=)	0.59 (=)	0.86 (+)	0.96 (+)	0.84 (+)	0.93 (+)	0.99 (+)	0.98 (+)	0.88 (+)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.56 (=)	0.73 (+)	0.62 (−)	0.61 (−)	0.55 (=)	0.72 (+)	0.76 (+)	0.62 (−)	0.69 (+)	1 (+)	0.72 (+)	0.61 (−)
zi929	0.77 (+)	0.9 (+)	0.64 (+)	0.55 (=)	0.6 (=)	0.86 (+)	0.97 (+)	0.86 (+)	0.91 (+)	0.99 (+)	0.98 (+)	0.91 (+)
lu980	0.73 (+)	0.86 (+)	0.54 (=)	0.56 (=)	0.56 (=)	0.8 (+)	0.93 (+)	0.82 (+)	0.9 (+)	1 (+)	0.98 (+)	0.88 (+)
rw1621	0.63 (+)	0.85 (+)	0.65 (+)	0.59 (=)	0.65 (+)	0.68 (+)	0.89 (+)	0.7 (+)	0.85 (+)	0.97 (+)	0.95 (+)	0.74 (+)
nu3496	0.63 (+)	0.73 (+)	0.64 (+)	0.57 (=)	0.57 (=)	0.69 (+)	0.81 (+)	0.64 (+)	0.76 (+)	0.94 (+)	0.84 (+)	0.71 (+)
ca4663	0.65 (+)	0.8 (+)	0.58 (=)	0.53 (=)	0.53 (=)	0.69 (+)	0.87 (+)	0.71 (+)	0.85 (+)	0.98 (+)	0.95 (+)	0.7 (+)
tz6117	0.72 (+)	0.82 (+)	0.59 (+)	0.57 (=)	0.52 (=)	0.77 (+)	0.92 (+)	0.78 (+)	0.89 (+)	0.98 (+)	0.96 (+)	0.78 (+)
eg7146	0.66 (+)	0.73 (+)	0.58 (=)	0.53 (=)	0.53 (=)	0.7 (+)	0.84 (+)	0.66 (+)	0.78 (+)	0.92 (+)	0.77 (+)	0.7 (+)
ym7663	0.6 (+)	0.69 (+)	0.6 (=)	0.58 (=)	0.56 (=)	0.63 (+)	0.78 (+)	0.63 (+)	0.71 (+)	0.82 (+)	0.8 (+)	0.58 (+)
ei8246	0.7 (+)	0.78 (+)	0.56 (=)	0.52 (=)	0.57 (=)	0.82 (+)	0.88 (+)	0.75 (+)	0.84 (+)	0.98 (+)	0.94 (+)	0.78 (+)
ar9152	0.67 (+)	0.74 (+)	0.64 (+)	0.59 (=)	0.66 (+)	0.78 (+)	0.86 (+)	0.69 (+)	0.78 (+)	0.91 (+)	0.88 (+)	0.71 (+)
ja9847	0.53 (=)	0.57 (=)	0.56 (=)	0.59 (=)	0.5 (=)	0.54 (=)	0.65 (+)	0.54 (=)	0.66 (+)	0.57 (+)	0.68 (+)	0.54 (=)
gr9882	0.6 (−)	0.55 (=)	0.51 (=)	0.62 (−)	0.5 (=)	0.55 (+)	0.67 (+)	0.55 (+)	0.62 (+)	0.6 (+)	0.69 (+)	0.56 (−)
kz9976	0.7 (+)	0.83 (+)	0.61 (+)	0.55 (=)	0.63 (+)	0.74 (+)	0.9 (+)	0.73 (+)	0.87 (+)	0.94 (+)	0.94 (+)	0.75 (+)
fi10639	0.68 (+)	0.77 (+)	0.57 (=)	0.52 (=)	0.52 (=)	0.71 (+)	0.84 (+)	0.71 (+)	0.81 (+)	0.98 (+)	0.94 (+)	0.71 (+)
ho14473	0.68 (−)	0.54 (=)	0.52 (=)	0.6 (−)	0.52 (=)	0.65 (−)	0.68 (+)	0.64 (−)	0.64 (+)	0.61 (−)	0.73 (+)	0.66 (−)
mo14185	0.71 (+)	0.77 (+)	0.56 (=)	0.54 (=)	0.59 (=)	0.76 (+)	0.9 (+)	0.75 (+)	0.86 (+)	0.97 (+)	0.96 (+)	0.77 (+)
it16862	0.56 (=)	0.7 (+)	0.57 (=)	0.53 (=)	0.58 (=)	0.65 (+)	0.73 (+)	0.6 (+)	0.73 (+)	0.82 (+)	0.77 (+)	0.61 (+)
vm22775	0.63 (−)	0.52 (=)	0.6 (−)	0.67 (−)	0.57 (−)	0.64 (−)	0.63 (+)	0.65 (−)	0.55 (=)	0.57 (−)	0.63 (+)	0.65 (−)
sw24978	0.53 (=)	0.67 (+)	0.5 (=)	0.57 (=)	0.52 (=)	0.58 (=)	0.77 (+)	0.57 (=)	0.73 (+)	0.82 (+)	0.89 (+)	0.56 (=)
bm33708	0.65 (+)	0.79 (+)	0.63 (+)	0.56 (=)	0.63 (+)	0.7 (+)	0.85 (+)	0.67 (+)	0.8 (+)	0.88 (+)	0.94 (+)	0.71 (+)

Table B.28: Statistical comparison of tours obtained by the generated solver TSP-C and the generated solvers TSP-[F-Q]

	TSP-C vs TSP-F	TSP-G	TSP-H	TSP-I	TSP-J	TSP-K	TSP-L	TSP-M	TSP-N	TSP-O	TSP-P	TSP-Q
u2152	0.99 (−)	0.96 (−)	1 (−)	1 (−)	1 (−)	0.94 (−)	0.89 (−)	0.98 (−)	0.9 (−)	0.52 (=)	0.77 (−)	0.96 (−)
usa13509	0.91 (−)	0.75 (−)	0.9 (−)	0.92 (−)	0.86 (−)	0.71 (−)	0.59 (−)	0.81 (−)	0.68 (−)	0.65 (+)	0.52 (=)	0.84 (−)
d18512	0.94 (−)	0.93 (−)	0.98 (−)	0.99 (−)	0.99 (−)	0.81 (−)	0.72 (−)	0.91 (−)	0.83 (−)	0.7 (+)	0.65 (−)	0.88 (−)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.63 (−)	0.55 (=)	0.57 (−)	0.61 (−)	0.63 (−)	0.56 (=)	0.58 (=)	0.56 (−)	0.51 (=)	0.99 (+)	0.56 (=)	0.57 (=)
zi929	0.87 (−)	0.74 (−)	0.93 (−)	0.97 (−)	0.93 (−)	0.64 (−)	0.55 (=)	0.82 (−)	0.7 (−)	0.98 (+)	0.54 (=)	0.68 (−)
lu980	0.96 (−)	0.89 (−)	0.99 (−)	0.98 (−)	0.99 (−)	0.77 (−)	0.79 (−)	0.93 (−)	0.82 (−)	0.89 (+)	0.6 (=)	0.87 (−)
rw1621	0.97 (−)	0.77 (−)	0.93 (−)	0.94 (−)	0.92 (−)	0.92 (−)	0.77 (−)	0.94 (−)	0.74 (−)	0.55 (=)	0.64 (−)	0.93 (−)
nu3496	0.79 (−)	0.66 (−)	0.75 (−)	0.83 (−)	0.83 (−)	0.7 (−)	0.54 (=)	0.77 (−)	0.61 (−)	0.75 (+)	0.5 (=)	0.7 (−)
ca4663	0.95 (−)	0.77 (−)	0.89 (−)	0.97 (−)	0.96 (−)	0.87 (−)	0.71 (−)	0.95 (−)	0.75 (−)	0.77 (+)	0.53 (=)	0.95 (−)
tz6117	0.96 (−)	0.88 (−)	0.98 (−)	0.98 (−)	0.98 (−)	0.88 (−)	0.8 (−)	0.96 (−)	0.83 (−)	0.53 (=)	0.66 (−)	0.96 (−)
eg7146	0.82 (−)	0.71 (−)	0.79 (−)	0.87 (−)	0.82 (−)	0.73 (−)	0.53 (=)	0.8 (−)	0.61 (−)	0.64 (+)	0.68 (−)	0.77 (−)
ym7663	0.84 (−)	0.74 (−)	0.81 (−)	0.81 (−)	0.84 (−)	0.81 (−)	0.6 (=)	0.83 (−)	0.7 (−)	0.57 (−)	0.6 (−)	0.86 (−)
ei8246	0.86 (−)	0.76 (−)	0.93 (−)	0.97 (−)	0.93 (−)	0.73 (−)	0.68 (−)	0.87 (−)	0.75 (−)	0.72 (+)	0.51 (=)	0.8 (−)
ar9152	0.89 (−)	0.78 (−)	0.86 (−)	0.88 (−)	0.86 (−)	0.77 (−)	0.65 (−)	0.86 (−)	0.75 (−)	0.57 (=)	0.59 (=)	0.83 (−)
ja9847	0.82 (−)	0.72 (−)	0.72 (−)	0.87 (−)	0.78 (−)	0.84 (−)	0.59 (=)	0.83 (−)	0.59 (=)	0.76 (−)	0.6 (−)	0.83 (−)
gr9882	0.92 (−)	0.79 (−)	0.84 (−)	0.9 (−)	0.83 (−)	0.89 (−)	0.66 (−)	0.89 (−)	0.7 (−)	0.77 (−)	0.66 (−)	0.91 (−)
kz9976	0.94 (−)	0.84 (−)	0.95 (−)	0.96 (−)	0.95 (−)	0.88 (−)	0.71 (−)	0.94 (−)	0.79 (−)	0.62 (−)	0.6 (=)	0.95 (−)
fi10639	0.86 (−)	0.75 (−)	0.9 (−)	0.93 (−)	0.93 (−)	0.77 (−)	0.66 (−)	0.84 (−)	0.77 (−)	0.82 (+)	0.56 (=)	0.82 (−)
ho14473	0.97 (−)	0.87 (−)	0.9 (−)	0.95 (−)	0.89 (−)	0.96 (−)	0.74 (−)	0.96 (−)	0.79 (−)	0.95 (−)	0.74 (−)	0.96 (−)
mo14185	0.96 (−)	0.87 (−)	0.96 (−)	0.98 (−)	0.94 (−)	0.88 (−)	0.66 (−)	0.92 (−)	0.8 (−)	0.57 (=)	0.51 (=)	0.9 (−)
it16862	0.81 (−)	0.67 (−)	0.78 (−)	0.86 (−)	0.79 (−)	0.75 (−)	0.62 (−)	0.78 (−)	0.63 (−)	0.51 (=)	0.57 (=)	0.79 (−)
vm22775	0.86 (−)	0.78 (−)	0.83 (−)	0.88 (−)	0.81 (−)	0.86 (−)	0.66 (−)	0.88 (−)	0.72 (−)	0.81 (−)	0.67 (−)	0.87 (−)
sw24978	0.98 (−)	0.87 (−)	0.94 (−)	0.95 (−)	0.94 (−)	0.92 (−)	0.74 (−)	0.93 (−)	0.78 (−)	0.73 (−)	0.58 (=)	0.94 (−)
bm33708	0.94 (−)	0.77 (−)	0.91 (−)	0.94 (−)	0.88 (−)	0.9 (−)	0.68 (−)	0.91 (−)	0.76 (−)	0.69 (−)	0.52 (=)	0.89 (−)

Table B.29: Statistical comparison of tours obtained by the generated solver TSP-D and the generated solvers TSP-[F-Q]

	TSP-D vs TSP-F		TSP-G		TSP-H		TSP-I		TSP-J		TSP-K		TSP-L		TSP-M		TSP-N		TSP-O		TSP-P		TSP-Q	
u2152	0.82	(+)	0.92	(+)	0.59	(=)	0.65	(+)	0.62	(+)	0.83	(+)	0.97	(+)	0.89	(+)	0.96	(+)	0.99	(+)	0.98	(+)	0.92	(+)
usa13509	0.78	(+)	0.85	(+)	0.63	(+)	0.63	(+)	0.7	(+)	0.86	(+)	0.91	(+)	0.83	(+)	0.9	(+)	0.96	(+)	0.94	(+)	0.81	(+)
d18512	0.75	(+)	0.76	(+)	0.54	(=)	0.51	(=)	0.54	(=)	0.83	(+)	0.95	(+)	0.8	(+)	0.9	(+)	0.99	(+)	0.96	(+)	0.84	(+)
dj38	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.68	(−)	0.5	(=)	0.5	(=)
qa194	0.56	(=)	0.72	(+)	0.62	(−)	0.6	(−)	0.55	(=)	0.71	(+)	0.76	(+)	0.62	(−)	0.69	(+)	0.99	(+)	0.71	(+)	0.61	(−)
zi929	0.81	(+)	0.91	(+)	0.68	(+)	0.61	(+)	0.64	(+)	0.87	(+)	0.97	(+)	0.89	(+)	0.92	(+)	0.99	(+)	0.98	(+)	0.92	(+)
lu980	0.83	(+)	0.92	(+)	0.66	(+)	0.68	(+)	0.69	(+)	0.86	(+)	0.97	(+)	0.91	(+)	0.96	(+)	0.99	(+)	0.99	(+)	0.95	(+)
rw1621	0.85	(+)	0.95	(+)	0.87	(+)	0.79	(+)	0.86	(+)	0.86	(+)	0.97	(+)	0.89	(+)	0.95	(+)	0.99	(+)	0.98	(+)	0.91	(+)
nu3496	0.76	(+)	0.86	(+)	0.75	(+)	0.69	(+)	0.68	(+)	0.81	(+)	0.92	(+)	0.76	(+)	0.87	(+)	0.99	(+)	0.95	(+)	0.84	(+)
ca4663	0.81	(+)	0.91	(+)	0.71	(+)	0.68	(+)	0.61	(+)	0.83	(+)	0.97	(+)	0.88	(+)	0.95	(+)	0.99	(+)	0.99	(+)	0.87	(+)
tz6117	0.85	(+)	0.92	(+)	0.74	(+)	0.72	(+)	0.68	(+)	0.87	(+)	0.98	(+)	0.9	(+)	0.96	(+)	0.99	(+)	0.99	(+)	0.9	(+)
eg7146	0.83	(+)	0.87	(+)	0.71	(+)	0.6	(+)	0.66	(+)	0.84	(+)	0.95	(+)	0.82	(+)	0.9	(+)	0.99	(+)	0.91	(+)	0.86	(+)
ym7663	0.79	(+)	0.87	(+)	0.77	(+)	0.74	(+)	0.73	(+)	0.82	(+)	0.93	(+)	0.81	(+)	0.88	(+)	0.97	(+)	0.96	(+)	0.77	(+)
ei8246	0.82	(+)	0.87	(+)	0.68	(+)	0.61	(=)	0.69	(+)	0.9	(+)	0.94	(+)	0.87	(+)	0.92	(+)	0.98	(+)	0.97	(+)	0.88	(+)
ar9152	0.73	(+)	0.79	(+)	0.68	(+)	0.63	(+)	0.7	(+)	0.84	(+)	0.91	(+)	0.74	(+)	0.83	(+)	0.95	(+)	0.93	(+)	0.76	(+)
ja9847	0.75	(+)	0.85	(+)	0.82	(+)	0.68	(+)	0.79	(+)	0.74	(+)	0.87	(+)	0.75	(+)	0.9	(+)	0.87	(+)	0.95	(+)	0.73	(+)
gr9882	0.74	(+)	0.85	(+)	0.8	(+)	0.7	(+)	0.81	(+)	0.79	(+)	0.91	(+)	0.8	(+)	0.87	(+)	0.93	(+)	0.94	(+)	0.79	(+)
kz9976	0.86	(+)	0.94	(+)	0.79	(+)	0.74	(+)	0.8	(+)	0.89	(+)	0.98	(+)	0.89	(+)	0.97	(+)	1	(+)	1	(+)	0.91	(+)
fi10639	0.75	(+)	0.83	(+)	0.64	(+)	0.58	(=)	0.59	(=)	0.77	(+)	0.89	(+)	0.77	(+)	0.87	(+)	0.98	(+)	0.96	(+)	0.77	(+)
ho14473	0.7	(+)	0.88	(+)	0.83	(+)	0.78	(+)	0.87	(+)	0.74	(+)	0.94	(+)	0.78	(+)	0.93	(+)	0.79	(+)	0.97	(+)	0.72	(+)
mo14185	0.81	(+)	0.85	(+)	0.64	(+)	0.53	(=)	0.67	(+)	0.85	(+)	0.95	(+)	0.85	(+)	0.93	(+)	0.98	(+)	0.97	(+)	0.86	(+)
it16862	0.54	(=)	0.72	(+)	0.56	(=)	0.58	(=)	0.56	(=)	0.65	(+)	0.75	(+)	0.6	(=)	0.74	(+)	0.89	(+)	0.8	(+)	0.59	(=)
vm22775	0.9	(−)	0.84	(−)	0.88	(−)	0.92	(−)	0.87	(−)	0.91	(−)	0.72	(−)	0.92	(−)	0.77	(−)	0.86	(−)	0.73	(−)	0.92	(−)
sw24978	1	(−)	0.97	(−)	0.98	(−)	0.99	(−)	0.99	(−)	0.98	(−)	0.93	(−)	0.99	(−)	0.93	(−)	0.92	(−)	0.87	(−)	0.99	(−)
bm33708	0.99	(−)	0.95	(−)	0.98	(−)	0.99	(−)	0.97	(−)	0.98	(−)	0.93	(−)	0.98	(−)	0.94	(−)	0.94	(−)	0.89	(−)	0.98	(−)

Table B.30: Statistical comparison of tours obtained by the generated solver TSP-E and the generated solvers TSP-[F-Q]

	TSP-E vs TSP-F	TSP-G	TSP-H	TSP-I	TSP-J	TSP-K	TSP-L	TSP-M	TSP-N	TSP-O	TSP-P	TSP-Q
u2152	0.78 (−)	0.63 (−)	0.91 (−)	0.9 (−)	0.89 (−)	0.67 (−)	0.59 (=)	0.72 (−)	0.51 (=)	0.87 (+)	0.75 (+)	0.59 (=)
usa13509	0.69 (−)	0.51 (=)	0.76 (−)	0.79 (−)	0.72 (−)	0.52 (=)	0.64 (+)	0.58 (=)	0.58 (=)	0.84 (+)	0.73 (+)	0.61 (−)
d18512	0.7 (−)	0.7 (−)	0.88 (−)	0.89 (−)	0.88 (−)	0.51 (=)	0.68 (+)	0.64 (−)	0.53 (=)	0.94 (+)	0.75 (+)	0.58 (=)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.63 (−)	0.55 (=)	0.57 (=)	0.6 (−)	0.63 (−)	0.55 (=)	0.57 (=)	0.56 (=)	0.5 (=)	0.99 (+)	0.55 (=)	0.57 (=)
zi929	0.67 (−)	0.51 (=)	0.78 (−)	0.86 (−)	0.79 (−)	0.55 (=)	0.71 (+)	0.58 (=)	0.56 (=)	0.99 (+)	0.77 (+)	0.58 (=)
lu980	0.73 (−)	0.56 (=)	0.89 (−)	0.88 (−)	0.88 (−)	0.52 (=)	0.6 (+)	0.65 (−)	0.54 (=)	0.99 (+)	0.8 (+)	0.52 (=)
rw1621	0.76 (−)	0.55 (=)	0.73 (−)	0.76 (−)	0.71 (−)	0.68 (−)	0.57 (=)	0.69 (−)	0.56 (=)	0.78 (+)	0.72 (+)	0.66 (−)
nu3496	0.68 (−)	0.54 (=)	0.64 (−)	0.74 (−)	0.74 (−)	0.59 (−)	0.58 (=)	0.66 (−)	0.5 (=)	0.84 (+)	0.62 (+)	0.57 (=)
ca4663	0.75 (−)	0.52 (=)	0.74 (−)	0.84 (−)	0.86 (−)	0.65 (−)	0.58 (=)	0.74 (−)	0.55 (=)	0.95 (+)	0.78 (+)	0.73 (−)
tz6117	0.71 (−)	0.54 (=)	0.81 (−)	0.82 (−)	0.86 (−)	0.58 (=)	0.63 (+)	0.65 (−)	0.56 (=)	0.87 (+)	0.81 (+)	0.65 (−)
eg7146	0.67 (−)	0.55 (=)	0.68 (−)	0.78 (−)	0.73 (−)	0.58 (=)	0.63 (+)	0.66 (−)	0.53 (=)	0.81 (+)	0.51 (=)	0.61 (−)
ym7663	0.65 (−)	0.51 (=)	0.63 (−)	0.64 (−)	0.68 (−)	0.6 (=)	0.62 (+)	0.62 (−)	0.52 (=)	0.7 (+)	0.66 (+)	0.68 (−)
ei8246	0.66 (−)	0.53 (=)	0.77 (−)	0.84 (−)	0.76 (−)	0.52 (=)	0.6 (=)	0.61 (−)	0.53 (=)	0.86 (+)	0.74 (+)	0.55 (=)
ar9152	0.57 (=)	0.56 (=)	0.57 (=)	0.62 (−)	0.55 (=)	0.59 (−)	0.71 (+)	0.52 (=)	0.6 (=)	0.81 (+)	0.76 (+)	0.51 (=)
ja9847	0.63 (−)	0.52 (=)	0.53 (=)	0.71 (−)	0.6 (=)	0.65 (−)	0.57 (=)	0.65 (−)	0.59 (−)	0.54 (=)	0.61 (+)	0.65 (−)
gr9882	0.7 (−)	0.51 (=)	0.58 (=)	0.71 (−)	0.56 (=)	0.65 (−)	0.63 (+)	0.64 (−)	0.57 (=)	0.57 (+)	0.68 (+)	0.68 (−)
kz9976	0.75 (−)	0.55 (=)	0.8 (−)	0.84 (−)	0.78 (−)	0.64 (−)	0.63 (+)	0.73 (−)	0.56 (=)	0.76 (+)	0.76 (+)	0.73 (−)
fi10639	0.7 (−)	0.56 (=)	0.78 (−)	0.82 (−)	0.83 (−)	0.62 (−)	0.55 (=)	0.66 (−)	0.54 (=)	0.92 (+)	0.76 (+)	0.64 (−)
ho14473	0.72 (−)	0.5 (=)	0.55 (=)	0.63 (−)	0.52 (=)	0.68 (−)	0.64 (+)	0.66 (−)	0.61 (+)	0.64 (−)	0.69 (+)	0.69 (−)
mo14185	0.76 (−)	0.63 (−)	0.85 (−)	0.89 (−)	0.81 (−)	0.66 (−)	0.63 (+)	0.69 (−)	0.52 (=)	0.86 (+)	0.8 (+)	0.67 (−)
it16862	0.65 (−)	0.53 (=)	0.61 (−)	0.73 (−)	0.61 (−)	0.54 (=)	0.57 (=)	0.59 (−)	0.56 (=)	0.69 (+)	0.62 (+)	0.59 (−)
vm22775	0.79 (−)	0.66 (−)	0.75 (−)	0.83 (−)	0.72 (−)	0.8 (−)	0.51 (=)	0.82 (−)	0.58 (=)	0.7 (−)	0.51 (=)	0.81 (−)
sw24978	0.84 (−)	0.59 (−)	0.76 (−)	0.81 (−)	0.75 (−)	0.7 (−)	0.56 (=)	0.71 (−)	0.51 (=)	0.63 (+)	0.76 (+)	0.74 (−)
bm33708	0.61 (−)	0.62 (+)	0.6 (−)	0.68 (−)	0.6 (=)	0.54 (=)	0.71 (+)	0.57 (=)	0.6 (=)	0.74 (+)	0.86 (+)	0.53 (=)

Table B.31: Statistical comparison of tours obtained by the generated solver TSP-F and the generated solvers TSP-[G-Q]

	TSP-F vs TSP-G	TSP-F vs TSP-H	TSP-F vs TSP-I	TSP-F vs TSP-J	TSP-F vs TSP-K	TSP-F vs TSP-L	TSP-F vs TSP-M	TSP-F vs TSP-N	TSP-F vs TSP-O	TSP-F vs TSP-P	TSP-F vs TSP-Q
u2152	0.66 (+)	0.74 (−)	0.71 (−)	0.69 (−)	0.57 (=)	0.84 (+)	0.59 (−)	0.78 (+)	0.97 (+)	0.93 (+)	0.7 (+)
usa13509	0.67 (+)	0.63 (−)	0.67 (−)	0.58 (−)	0.69 (+)	0.8 (+)	0.6 (=)	0.76 (+)	0.94 (+)	0.89 (+)	0.57 (=)
d18512	0.51 (=)	0.78 (−)	0.76 (−)	0.73 (−)	0.64 (+)	0.83 (+)	0.56 (=)	0.73 (+)	0.98 (+)	0.87 (+)	0.62 (+)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.67 (+)	0.56 (=)	0.54 (=)	0.5 (=)	0.67 (+)	0.71 (+)	0.56 (=)	0.64 (+)	1 (+)	0.67 (+)	0.55 (=)
zi929	0.69 (+)	0.64 (−)	0.74 (−)	0.66 (−)	0.67 (+)	0.84 (+)	0.61 (+)	0.72 (+)	0.99 (+)	0.89 (+)	0.74 (+)
lu980	0.69 (+)	0.72 (−)	0.7 (−)	0.7 (−)	0.66 (+)	0.84 (+)	0.6 (=)	0.77 (+)	1 (+)	0.96 (+)	0.73 (+)
rw1621	0.8 (+)	0.52 (=)	0.53 (=)	0.53 (=)	0.58 (=)	0.84 (+)	0.59 (−)	0.8 (+)	0.96 (+)	0.94 (+)	0.63 (+)
nu3496	0.64 (+)	0.53 (=)	0.57 (=)	0.58 (=)	0.58 (=)	0.75 (+)	0.52 (=)	0.67 (+)	0.93 (+)	0.8 (+)	0.62 (+)
ca4663	0.7 (+)	0.55 (=)	0.64 (−)	0.69 (−)	0.58 (=)	0.8 (+)	0.53 (=)	0.78 (+)	1 (+)	0.94 (+)	0.54 (=)
tz6117	0.68 (+)	0.64 (−)	0.66 (−)	0.72 (−)	0.61 (+)	0.84 (+)	0.58 (=)	0.78 (+)	0.97 (+)	0.95 (+)	0.59 (−)
eg7146	0.62 (+)	0.55 (−)	0.67 (−)	0.61 (−)	0.59 (−)	0.8 (+)	0.51 (=)	0.69 (+)	0.97 (+)	0.68 (+)	0.57 (=)
ym7663	0.65 (+)	0.51 (=)	0.52 (=)	0.55 (=)	0.56 (=)	0.77 (+)	0.53 (=)	0.67 (+)	0.88 (+)	0.82 (+)	0.54 (=)
ei8246	0.62 (+)	0.64 (−)	0.74 (−)	0.64 (−)	0.68 (+)	0.77 (+)	0.57 (=)	0.7 (+)	0.94 (+)	0.87 (+)	0.62 (+)
ar9152	0.63 (+)	0.51 (=)	0.57 (−)	0.51 (=)	0.68 (+)	0.79 (+)	0.55 (=)	0.67 (+)	0.88 (+)	0.84 (+)	0.58 (=)
ja9847	0.63 (+)	0.61 (+)	0.59 (=)	0.54 (=)	0.52 (=)	0.7 (+)	0.51 (=)	0.73 (+)	0.62 (+)	0.77 (+)	0.52 (=)
gr9882	0.7 (+)	0.62 (+)	0.53 (=)	0.64 (+)	0.57 (=)	0.82 (+)	0.58 (=)	0.75 (+)	0.83 (+)	0.88 (+)	0.54 (=)
kz9976	0.69 (+)	0.59 (=)	0.65 (−)	0.56 (=)	0.58 (=)	0.83 (+)	0.54 (=)	0.78 (+)	0.91 (+)	0.91 (+)	0.54 (=)
fi10639	0.64 (+)	0.62 (−)	0.67 (−)	0.67 (−)	0.56 (=)	0.74 (+)	0.54 (=)	0.68 (+)	0.97 (+)	0.9 (+)	0.55 (=)
ho14473	0.75 (+)	0.69 (+)	0.6 (=)	0.73 (+)	0.54 (=)	0.86 (+)	0.57 (=)	0.84 (+)	0.6 (=)	0.93 (+)	0.53 (=)
mo14185	0.62 (+)	0.68 (−)	0.74 (−)	0.64 (−)	0.58 (=)	0.83 (+)	0.57 (=)	0.77 (+)	0.98 (+)	0.96 (+)	0.59 (=)
it16862	0.68 (+)	0.52 (=)	0.6 (=)	0.52 (=)	0.61 (+)	0.72 (+)	0.56 (=)	0.7 (+)	0.85 (+)	0.77 (+)	0.55 (=)
vm22775	0.67 (+)	0.53 (=)	0.59 (=)	0.57 (=)	0.53 (=)	0.78 (+)	0.56 (=)	0.7 (+)	0.62 (+)	0.84 (+)	0.55 (=)
sw24978	0.77 (+)	0.55 (=)	0.53 (=)	0.57 (=)	0.65 (+)	0.87 (+)	0.65 (+)	0.83 (+)	0.93 (+)	0.98 (+)	0.63 (+)
bm33708	0.72 (+)	0.5 (=)	0.59 (=)	0.51 (=)	0.57 (=)	0.8 (+)	0.54 (=)	0.71 (+)	0.85 (+)	0.95 (+)	0.59 (=)

Table B.32: Statistical comparison of tours obtained by the generated solver TSP-G and the generated solvers TSP-[H-Q]

	TSP-G vs TSP-H	TSP-G vs TSP-I	TSP-G vs TSP-J	TSP-G vs TSP-K	TSP-G vs TSP-L	TSP-G vs TSP-M	TSP-G vs TSP-N	TSP-G vs TSP-O	TSP-G vs TSP-P	TSP-G vs TSP-Q
u2152	0.86 (−)	0.84 (−)	0.82 (−)	0.57 (=)	0.72 (+)	0.59 (=)	0.64 (+)	0.92 (+)	0.85 (+)	0.54 (=)
usa13509	0.74 (−)	0.77 (−)	0.7 (−)	0.53 (=)	0.64 (+)	0.56 (=)	0.59 (=)	0.82 (+)	0.73 (+)	0.59 (=)
d18512	0.79 (−)	0.77 (−)	0.74 (−)	0.63 (+)	0.83 (+)	0.56 (=)	0.73 (+)	0.97 (+)	0.87 (+)	0.62 (+)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.62 (−)	0.65 (−)	0.68 (−)	0.5 (=)	0.53 (=)	0.61 (−)	0.54 (=)	0.99 (+)	0.5 (=)	0.62 (−)
zi929	0.79 (−)	0.88 (−)	0.8 (−)	0.54 (=)	0.7 (+)	0.59 (=)	0.55 (=)	0.99 (+)	0.77 (+)	0.57 (=)
lu980	0.86 (−)	0.85 (−)	0.85 (−)	0.53 (+)	0.69 (+)	0.61 (−)	0.6 (=)	1 (+)	0.88 (+)	0.55 (=)
rw1621	0.76 (−)	0.79 (−)	0.74 (−)	0.72 (−)	0.52 (=)	0.73 (−)	0.51 (=)	0.74 (+)	0.68 (+)	0.7 (−)
nu3496	0.6 (=)	0.7 (−)	0.71 (−)	0.55 (=)	0.62 (+)	0.63 (−)	0.54 (=)	0.85 (+)	0.66 (+)	0.53 (=)
ca4663	0.71 (−)	0.8 (−)	0.82 (−)	0.62 (−)	0.59 (=)	0.68 (−)	0.56 (=)	0.9 (+)	0.75 (+)	0.67 (−)
tz6117	0.78 (−)	0.79 (−)	0.84 (−)	0.55 (=)	0.66 (+)	0.61 (−)	0.61 (+)	0.89 (+)	0.83 (+)	0.61 (−)
eg7146	0.64 (−)	0.74 (−)	0.68 (−)	0.53 (=)	0.68 (+)	0.61 (−)	0.58 (=)	0.87 (+)	0.55 (=)	0.56 (=)
ym7663	0.62 (−)	0.63 (−)	0.68 (−)	0.59 (=)	0.64 (+)	0.62 (−)	0.53 (=)	0.74 (+)	0.69 (+)	0.68 (−)
ei8246	0.74 (−)	0.82 (−)	0.73 (−)	0.55 (=)	0.64 (+)	0.57 (−)	0.57 (=)	0.87 (+)	0.76 (+)	0.51 (=)
ar9152	0.63 (−)	0.66 (−)	0.61 (−)	0.53 (=)	0.66 (+)	0.58 (=)	0.54 (=)	0.75 (+)	0.71 (+)	0.55 (=)
ja9847	0.51 (=)	0.71 (−)	0.58 (=)	0.65 (−)	0.6 (+)	0.65 (−)	0.62 (+)	0.53 (=)	0.65 (+)	0.64 (−)
gr9882	0.57 (=)	0.7 (−)	0.56 (=)	0.64 (−)	0.64 (+)	0.64 (−)	0.58 (=)	0.56 (+)	0.69 (+)	0.67 (−)
kz9976	0.75 (−)	0.79 (−)	0.73 (−)	0.6 (=)	0.66 (+)	0.67 (−)	0.59 (=)	0.77 (+)	0.78 (+)	0.67 (−)
fi10639	0.74 (−)	0.77 (−)	0.78 (−)	0.57 (=)	0.61 (+)	0.6 (=)	0.52 (=)	0.92 (+)	0.79 (+)	0.58 (=)
ho14473	0.56 (=)	0.67 (−)	0.53 (=)	0.72 (−)	0.66 (+)	0.71 (−)	0.62 (+)	0.68 (−)	0.74 (+)	0.72 (−)
mo14185	0.74 (−)	0.8 (−)	0.71 (−)	0.53 (=)	0.73 (+)	0.55 (=)	0.64 (+)	0.91 (+)	0.86 (+)	0.53 (=)
it16862	0.65 (−)	0.75 (−)	0.66 (−)	0.6 (−)	0.55 (=)	0.63 (−)	0.53 (=)	0.69 (+)	0.61 (+)	0.65 (−)
vm22775	0.62 (−)	0.73 (−)	0.59 (=)	0.69 (−)	0.67 (+)	0.71 (−)	0.57 (=)	0.56 (=)	0.7 (+)	0.71 (−)
sw24978	0.69 (−)	0.75 (−)	0.68 (−)	0.62 (−)	0.65 (+)	0.64 (−)	0.6 (+)	0.74 (+)	0.85 (+)	0.66 (−)
bm33708	0.7 (−)	0.76 (−)	0.68 (−)	0.66 (−)	0.6 (=)	0.69 (−)	0.51 (=)	0.62 (+)	0.78 (+)	0.65 (−)

Table B.33: Statistical comparison of tours obtained by the generated solver TSP-H and the generated solvers TSP-[I-Q]

	TSP-H vs TSP-I	TSP-H vs TSP-J	TSP-H vs TSP-K	TSP-H vs TSP-L	TSP-H vs TSP-M	TSP-H vs TSP-N	TSP-H vs TSP-O	TSP-H vs TSP-P	TSP-H vs TSP-Q
u2152	0.55 (=)	0.54 (=)	0.76 (+)	0.95 (+)	0.82 (+)	0.92 (+)	0.99 (+)	0.98 (+)	0.87 (+)
usa13509	0.51 (=)	0.57 (=)	0.76 (+)	0.83 (+)	0.7 (+)	0.81 (+)	0.93 (+)	0.89 (+)	0.68 (+)
d18512	0.53 (=)	0.59 (+)	0.84 (+)	0.93 (+)	0.81 (+)	0.89 (+)	0.99 (+)	0.95 (+)	0.85 (+)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.52 (=)	0.57 (=)	0.61 (+)	0.65 (+)	0.5 (=)	0.57 (=)	1 (+)	0.61 (+)	0.51 (=)
zi929	0.59 (=)	0.53 (=)	0.77 (+)	0.91 (+)	0.74 (+)	0.82 (+)	0.99 (+)	0.94 (+)	0.83 (+)
lu980	0.53 (=)	0.53 (=)	0.79 (+)	0.94 (+)	0.82 (+)	0.91 (+)	1 (+)	0.99 (+)	0.89 (+)
rw1621	0.56 (=)	0.5 (=)	0.55 (=)	0.8 (+)	0.56 (=)	0.77 (+)	0.92 (+)	0.9 (+)	0.61 (+)
nu3496	0.59 (=)	0.59 (=)	0.55 (=)	0.71 (+)	0.51 (=)	0.64 (+)	0.91 (+)	0.75 (+)	0.58 (=)
ca4663	0.56 (=)	0.61 (−)	0.61 (+)	0.79 (+)	0.61 (+)	0.76 (+)	0.96 (+)	0.88 (+)	0.6 (+)
tz6117	0.52 (=)	0.58 (=)	0.71 (+)	0.91 (+)	0.71 (+)	0.87 (+)	0.99 (+)	0.98 (+)	0.72 (+)
eg7146	0.6 (=)	0.54 (=)	0.61 (+)	0.77 (+)	0.56 (=)	0.7 (+)	0.88 (+)	0.67 (+)	0.59 (+)
ym7663	0.52 (=)	0.55 (=)	0.54 (=)	0.74 (+)	0.52 (=)	0.65 (+)	0.82 (+)	0.78 (+)	0.54 (=)
ei8246	0.59 (=)	0.51 (=)	0.79 (+)	0.86 (+)	0.71 (+)	0.81 (+)	0.97 (+)	0.93 (+)	0.74 (+)
ar9152	0.55 (=)	0.51 (=)	0.65 (+)	0.76 (+)	0.54 (=)	0.66 (+)	0.85 (+)	0.81 (+)	0.57 (=)
ja9847	0.68 (−)	0.57 (=)	0.63 (−)	0.6 (=)	0.63 (−)	0.62 (+)	0.51 (=)	0.65 (+)	0.63 (−)
gr9882	0.63 (−)	0.52 (=)	0.56 (=)	0.7 (+)	0.56 (=)	0.65 (+)	0.66 (+)	0.77 (+)	0.6 (−)
kz9976	0.56 (=)	0.53 (=)	0.66 (+)	0.86 (+)	0.62 (+)	0.82 (+)	0.93 (+)	0.93 (+)	0.63 (+)
fi10639	0.56 (=)	0.55 (=)	0.66 (+)	0.82 (+)	0.66 (+)	0.78 (+)	0.98 (+)	0.93 (+)	0.66 (+)
ho14473	0.61 (−)	0.53 (=)	0.66 (−)	0.71 (+)	0.64 (−)	0.68 (+)	0.61 (−)	0.78 (+)	0.66 (−)
mo14185	0.59 (=)	0.53 (=)	0.73 (+)	0.89 (+)	0.72 (+)	0.85 (+)	0.97 (+)	0.95 (+)	0.74 (+)
it16862	0.61 (−)	0.5 (=)	0.57 (=)	0.68 (+)	0.53 (=)	0.67 (+)	0.81 (+)	0.73 (+)	0.52 (=)
vm22775	0.61 (−)	0.55 (=)	0.55 (=)	0.75 (+)	0.57 (=)	0.67 (+)	0.55 (=)	0.8 (+)	0.57 (=)
sw24978	0.57 (=)	0.52 (=)	0.59 (=)	0.79 (+)	0.58 (=)	0.76 (+)	0.86 (+)	0.92 (+)	0.56 (=)
bm33708	0.58 (=)	0.51 (=)	0.56 (=)	0.78 (+)	0.53 (=)	0.69 (+)	0.81 (+)	0.92 (+)	0.57 (=)

Table B.34: Statistical comparison of tours obtained by the generated solver TSP-I and the generated solvers TSP-[J-Q]

	TSP-I vs TSP-J		TSP-I vs TSP-K		TSP-I vs TSP-L		TSP-I vs TSP-M		TSP-I vs TSP-N		TSP-I vs TSP-O		TSP-I vs TSP-P		TSP-I vs TSP-Q	
u2152	0.51	(=)	0.74	(+)	0.93	(+)	0.8	(+)	0.91	(+)	0.99	(+)	0.97	(+)	0.85	(+)
usa13509	0.59	(=)	0.79	(+)	0.86	(+)	0.74	(+)	0.84	(+)	0.94	(+)	0.91	(+)	0.71	(+)
d18512	0.55	(=)	0.84	(+)	0.96	(+)	0.8	(+)	0.9	(+)	1	(+)	0.97	(+)	0.85	(+)
dj38	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.68	(−)	0.5	(=)	0.5	(=)
qa194	0.55	(=)	0.64	(+)	0.68	(+)	0.53	(=)	0.61	(+)	1	(+)	0.65	(+)	0.52	(=)
zi929	0.55	(=)	0.83	(+)	0.96	(+)	0.84	(+)	0.89	(+)	0.99	(+)	0.97	(+)	0.9	(+)
lu980	0.51	(=)	0.78	(+)	0.94	(+)	0.81	(+)	0.9	(+)	1	(+)	0.98	(+)	0.88	(+)
rw1621	0.56	(=)	0.6	(=)	0.82	(+)	0.6	(=)	0.79	(+)	0.94	(+)	0.92	(+)	0.64	(+)
nu3496	0.51	(=)	0.64	(+)	0.8	(+)	0.59	(=)	0.73	(+)	0.94	(+)	0.84	(+)	0.68	(+)
ca4663	0.56	(=)	0.69	(+)	0.88	(+)	0.7	(+)	0.86	(+)	1	(+)	0.97	(+)	0.69	(+)
tz6117	0.56	(=)	0.73	(+)	0.91	(+)	0.72	(+)	0.87	(+)	0.98	(+)	0.97	(+)	0.73	(+)
eg7146	0.55	(=)	0.72	(+)	0.85	(+)	0.68	(+)	0.79	(+)	0.94	(+)	0.78	(+)	0.71	(+)
ym7663	0.53	(=)	0.55	(=)	0.74	(+)	0.54	(=)	0.66	(+)	0.79	(+)	0.77	(+)	0.51	(=)
ei8246	0.6	(=)	0.86	(+)	0.92	(+)	0.8	(+)	0.88	(+)	0.99	(+)	0.97	(+)	0.82	(+)
ar9152	0.56	(=)	0.69	(+)	0.8	(+)	0.59	(+)	0.7	(+)	0.87	(+)	0.84	(+)	0.62	(+)
ja9847	0.63	(+)	0.57	(=)	0.76	(+)	0.58	(=)	0.79	(+)	0.72	(+)	0.84	(+)	0.57	(=)
gr9882	0.65	(+)	0.59	(+)	0.81	(+)	0.59	(=)	0.75	(+)	0.79	(+)	0.86	(+)	0.57	(=)
kz9976	0.58	(=)	0.71	(+)	0.89	(+)	0.68	(+)	0.85	(+)	0.94	(+)	0.94	(+)	0.69	(+)
fi10639	0.51	(=)	0.7	(+)	0.85	(+)	0.7	(+)	0.81	(+)	0.99	(+)	0.95	(+)	0.7	(+)
ho14473	0.65	(+)	0.56	(=)	0.81	(+)	0.54	(=)	0.78	(+)	0.51	(=)	0.89	(+)	0.57	(=)
mo14185	0.62	(+)	0.79	(+)	0.92	(+)	0.78	(+)	0.89	(+)	0.99	(+)	0.98	(+)	0.8	(+)
it16862	0.62	(+)	0.71	(+)	0.79	(+)	0.66	(+)	0.78	(+)	0.89	(+)	0.83	(+)	0.66	(+)
vm22775	0.64	(+)	0.56	(=)	0.81	(+)	0.54	(=)	0.74	(+)	0.68	(+)	0.88	(+)	0.54	(=)
sw24978	0.59	(=)	0.65	(+)	0.84	(+)	0.64	(+)	0.8	(+)	0.9	(+)	0.95	(+)	0.63	(+)
bm33708	0.57	(=)	0.64	(+)	0.83	(+)	0.61	(+)	0.75	(+)	0.86	(+)	0.94	(+)	0.65	(+)

Table B.35: Statistical comparison of tours obtained by the generated solver TSP-J and the generated solvers TSP-[K-Q]

	TSP-J vs TSP-K		TSP-J vs TSP-L		TSP-J vs TSP-M		TSP-J vs TSP-N		TSP-J vs TSP-O		TSP-J vs TSP-P		TSP-J vs TSP-Q	
u2152	0.73	(+)	0.93	(+)	0.78	(+)	0.9	(+)	0.99	(+)	0.97	(+)	0.84	(+)
usa13509	0.71	(+)	0.8	(+)	0.66	(+)	0.77	(+)	0.89	(+)	0.85	(+)	0.63	(+)
d18512	0.81	(+)	0.95	(+)	0.78	(+)	0.9	(+)	0.99	(+)	0.96	(+)	0.82	(+)
dj38	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.68	(−)	0.5	(=)	0.5	(=)
qa194	0.67	(+)	0.71	(+)	0.57	(=)	0.64	(+)	1	(+)	0.67	(+)	0.56	(=)
zi929	0.78	(+)	0.91	(+)	0.76	(+)	0.83	(+)	0.99	(+)	0.94	(+)	0.83	(+)
lu980	0.78	(+)	0.94	(+)	0.81	(+)	0.91	(+)	1	(+)	0.99	(+)	0.89	(+)
rw1621	0.54	(=)	0.77	(+)	0.54	(=)	0.75	(+)	0.91	(+)	0.88	(+)	0.59	(=)
nu3496	0.65	(+)	0.8	(+)	0.59	(=)	0.73	(+)	0.94	(+)	0.84	(+)	0.68	(+)
ca4663	0.72	(+)	0.89	(+)	0.73	(+)	0.88	(+)	0.98	(+)	0.96	(+)	0.73	(+)
tz6117	0.77	(+)	0.94	(+)	0.78	(+)	0.91	(+)	0.99	(+)	0.98	(+)	0.79	(+)
eg7146	0.66	(+)	0.81	(+)	0.61	(+)	0.74	(+)	0.91	(+)	0.72	(+)	0.65	(+)
ym7663	0.6	(+)	0.78	(+)	0.58	(+)	0.7	(+)	0.86	(+)	0.82	(+)	0.52	(=)
ei8246	0.78	(+)	0.85	(+)	0.7	(+)	0.8	(+)	0.97	(+)	0.93	(+)	0.74	(+)
ar9152	0.65	(+)	0.77	(+)	0.53	(=)	0.66	(+)	0.86	(+)	0.81	(+)	0.56	(=)
ja9847	0.57	(=)	0.67	(+)	0.56	(=)	0.7	(+)	0.57	(=)	0.73	(+)	0.57	(=)
gr9882	0.59	(−)	0.69	(+)	0.59	(−)	0.64	(+)	0.62	(+)	0.75	(+)	0.61	(−)
kz9976	0.63	(+)	0.85	(+)	0.59	(=)	0.8	(+)	0.93	(+)	0.92	(+)	0.6	(+)
fi10639	0.7	(+)	0.85	(+)	0.7	(+)	0.82	(+)	0.99	(+)	0.95	(+)	0.7	(+)
ho14473	0.7	(−)	0.7	(+)	0.69	(−)	0.66	(+)	0.66	(−)	0.78	(+)	0.7	(−)
mo14185	0.7	(+)	0.87	(+)	0.69	(+)	0.82	(+)	0.96	(+)	0.93	(+)	0.7	(+)
it16862	0.58	(=)	0.7	(+)	0.53	(=)	0.67	(+)	0.83	(+)	0.74	(+)	0.53	(=)
vm22775	0.6	(=)	0.73	(+)	0.6	(−)	0.64	(+)	0.51	(=)	0.76	(+)	0.61	(−)
sw24978	0.57	(=)	0.79	(+)	0.56	(=)	0.75	(+)	0.86	(+)	0.93	(+)	0.54	(=)
bm33708	0.56	(=)	0.76	(+)	0.53	(=)	0.68	(+)	0.79	(+)	0.89	(+)	0.58	(=)

Table B.36: Statistical comparison of tours obtained by the generated solvers TSP-[K-L] and the generated solvers TSP-[M-Q]

	TSP-K vs						**TSP-L vs**				
	TSP-L	**TSP-M**	**TSP-N**	**TSP-O**	**TSP-P**	**TSP-Q**	**TSP-M**	**TSP-N**	**TSP-O**	**TSP-P**	**TSP-Q**
textbfu2152	0.73 (+)	0.52 (=)	0.68 (+)	0.91 (+)	0.83 (+)	0.6 (+)	0.8 (−)	0.57 (=)	0.84 (+)	0.69 (+)	0.69 (−)
usa13509	0.61 (+)	0.59 (=)	0.56 (=)	0.78 (+)	0.69 (+)	0.62 (−)	0.7 (−)	0.56 (=)	0.7 (+)	0.57 (=)	0.73 (−)
d18512	0.66 (+)	0.59 (=)	0.54 (=)	0.9 (+)	0.71 (+)	0.54 (=)	0.78 (−)	0.65 (−)	0.86 (+)	0.57 (=)	0.74 (−)
dj38	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)
qa194	0.52 (=)	0.61 (−)	0.55 (=)	0.99 (+)	0.51 (=)	0.62 (−)	0.64 (−)	0.57 (=)	0.99 (+)	0.52 (=)	0.65 (−)
zi929	0.61 (+)	0.61 (−)	0.51 (=)	0.96 (+)	0.66 (+)	0.5 (=)	0.78 (−)	0.65 (−)	0.98 (+)	0.58 (=)	0.64 (−)
lu980	0.58 (+)	0.59 (−)	0.54 (=)	0.94 (+)	0.73 (+)	0.5 (=)	0.78 (−)	0.57 (=)	0.98 (+)	0.75 (+)	0.63 (−)
rw1621	0.75 (+)	0.5 (=)	0.72 (+)	0.91 (+)	0.88 (+)	0.54 (=)	0.77 (−)	0.51 (=)	0.74 (+)	0.68 (+)	0.74 (−)
nu3496	0.66 (+)	0.56 (=)	0.58 (=)	0.86 (+)	0.7 (+)	0.52 (=)	0.74 (−)	0.58 (=)	0.76 (+)	0.53 (=)	0.65 (−)
ca4663	0.71 (+)	0.55 (=)	0.68 (+)	0.96 (+)	0.86 (+)	0.54 (=)	0.79 (−)	0.53 (=)	0.87 (+)	0.69 (+)	0.78 (−)
tz6117	0.69 (+)	0.54 (=)	0.64 (+)	0.88 (+)	0.82 (+)	0.54 (−)	0.79 (−)	0.56 (=)	0.81 (+)	0.73 (+)	0.8 (−)
eg7146	0.71 (+)	0.58 (=)	0.61 (+)	0.9 (+)	0.58 (=)	0.53 (=)	0.78 (−)	0.58 (=)	0.69 (+)	0.65 (−)	0.75 (−)
ym7663	0.72 (+)	0.53 (=)	0.62 (+)	0.84 (+)	0.78 (+)	0.6 (=)	0.75 (−)	0.61 (−)	0.57 (+)	0.52 (=)	0.8 (−)
ei8246	0.58 (=)	0.64 (−)	0.51 (=)	0.86 (+)	0.72 (+)	0.58 (=)	0.73 (−)	0.58 (=)	0.82 (+)	0.67 (+)	0.67 (−)
ar9152	0.64 (+)	0.62 (−)	0.52 (=)	0.75 (+)	0.7 (+)	0.58 (=)	0.75 (−)	0.62 (−)	0.6 (+)	0.56 (=)	0.71 (−)
ja9847	0.72 (+)	0.51 (=)	0.75 (+)	0.65 (+)	0.8 (+)	0.5 (=)	0.71 (−)	0.51 (=)	0.62 (−)	0.51 (+)	0.72 (−)
gr9882	0.77 (+)	0.5 (=)	0.71 (+)	0.76 (+)	0.84 (+)	0.53 (=)	0.77 (−)	0.55 (=)	0.6 (−)	0.54 (=)	0.8 (−)
kz9976	0.74 (+)	0.56 (=)	0.68 (+)	0.84 (+)	0.84 (+)	0.56 (=)	0.82 (−)	0.56 (=)	0.62 (+)	0.63 (+)	0.83 (−)
fi10639	0.65 (+)	0.52 (=)	0.59 (+)	0.93 (+)	0.81 (+)	0.51 (=)	0.71 (−)	0.6 (−)	0.91 (+)	0.72 (+)	0.69 (−)
ho14473	0.84 (+)	0.53 (=)	0.82 (+)	0.56 (=)	0.91 (+)	0.51 (=)	0.84 (−)	0.55 (=)	0.81 (−)	0.55 (=)	0.84 (−)
mo14185	0.75 (+)	0.52 (=)	0.67 (+)	0.91 (+)	0.87 (+)	0.5 (=)	0.78 (−)	0.61 (−)	0.72 (+)	0.65 (+)	0.76 (−)
it16862	0.64 (+)	0.55 (=)	0.62 (+)	0.8 (+)	0.7 (+)	0.56 (=)	0.67 (−)	0.52 (=)	0.63 (+)	0.56 (=)	0.69 (−)
vm22775	0.79 (+)	0.52 (=)	0.71 (+)	0.63 (+)	0.86 (+)	0.52 (=)	0.81 (−)	0.59 (=)	0.7 (−)	0.51 (=)	0.8 (−)
sw24978	0.74 (+)	0.51 (=)	0.7 (+)	0.83 (+)	0.91 (+)	0.54 (=)	0.76 (−)	0.55 (=)	0.56 (=)	0.7 (+)	0.78 (−)
bm33708	0.75 (+)	0.53 (=)	0.64 (+)	0.79 (+)	0.91 (+)	0.52 (=)	0.77 (−)	0.6 (=)	0.51 (=)	0.68 (+)	0.73 (−)

Table B.37: Statistical comparison of tours obtained by the generated solvers TSP-[M-Q]

	TSP-M vs				**TSP-N vs**			**TSP-O vs**		**TSP-P vs**
	TSP-N	**TSP-O**	**TSP-P**	**TSP-Q**	**TSP-O**	**TSP-P**	**TSP-Q**	**TSP-P**	**TSP-Q**	**TSP-Q**
u2152	0.73 (+)	0.96 (+)	0.91 (+)	0.63 (+)	0.86 (+)	0.74 (+)	0.61 (−)	0.74 (−)	0.92 (−)	0.84 (−)
usa13509	0.66 (+)	0.86 (+)	0.79 (+)	0.53 (=)	0.77 (+)	0.65 (+)	0.69 (−)	0.66 (−)	0.89 (−)	0.82 (−)
d18512	0.67 (+)	0.96 (+)	0.83 (+)	0.56 (=)	0.92 (+)	0.71 (+)	0.61 (−)	0.81 (−)	0.95 (−)	0.79 (−)
dj38	0.5 (=)	0.68 (−)	0.5 (=)	0.5 (=)	0.68 (−)	0.5 (=)	0.50 (=)	0.68 (−)	0.68 (−)	0.5 (=)
qa194	0.56 (=)	1 (+)	0.61 (+)	0.51 (=)	0.99 (+)	0.55 (=)	0.58 (−)	0.99 (−)	1 (−)	0.62 (−)
zi929	0.64 (+)	0.99 (+)	0.84 (+)	0.66 (+)	0.99 (+)	0.72 (+)	0.52 (=)	0.98 (−)	0.99 (−)	0.71 (−)
lu980	0.7 (+)	1 (+)	0.93 (+)	0.66 (+)	0.99 (+)	0.78 (+)	0.56 (−)	0.95 (−)	1 (−)	0.85 (−)
rw1621	0.74 (+)	0.93 (+)	0.91 (+)	0.54 (=)	0.71 (+)	0.66 (+)	0.7 (−)	0.6 (=)	0.92 (−)	0.89 (−)
nu3496	0.66 (+)	0.93 (+)	0.78 (+)	0.6 (=)	0.83 (+)	0.62 (+)	0.58 (−)	0.78 (−)	0.9 (−)	0.71 (−)
ca4663	0.77 (+)	1 (+)	0.95 (+)	0.51 (=)	0.9 (+)	0.73 (+)	0.76 (−)	0.8 (−)	0.99 (−)	0.94 (−)
tz6117	0.73 (+)	0.98 (+)	0.94 (+)	0.51 (=)	0.84 (+)	0.76 (+)	0.73 (−)	0.65 (−)	0.98 (−)	0.95 (−)
eg7146	0.68 (+)	0.95 (+)	0.66 (+)	0.56 (=)	0.74 (+)	0.54 (=)	0.64 (−)	0.85 (−)	0.95 (−)	0.62 (−)
ym7663	0.65 (+)	0.87 (+)	0.82 (+)	0.56 (=)	0.7 (+)	0.65 (+)	0.7 (−)	0.55 (=)	0.9 (−)	0.85 (−)
ei8246	0.66 (+)	0.95 (+)	0.87 (+)	0.55 (=)	0.87 (+)	0.74 (+)	0.59 (=)	0.73 (−)	0.89 (−)	0.8 (−)
ar9152	0.63 (+)	0.85 (+)	0.8 (+)	0.54 (=)	0.72 (+)	0.68 (+)	0.6 (=)	0.53 (=)	0.81 (−)	0.77 (−)
ja9847	0.74 (+)	0.64 (+)	0.79 (+)	0.51 (=)	0.65 (−)	0.5 (−)	0.74 (−)	0.7 (+)	0.65 (−)	0.79 (−)
gr9882	0.71 (+)	0.76 (+)	0.85 (+)	0.54 (=)	0.54 (−)	0.58 (−)	0.73 (−)	0.69 (+)	0.82 (−)	0.88 (−)
kz9976	0.76 (+)	0.92 (+)	0.92 (+)	0.5 (=)	0.7 (+)	0.71 (+)	0.77 (−)	0.53 (=)	0.93 (−)	0.92 (−)
fi10639	0.63 (+)	0.97 (+)	0.88 (+)	0.51 (=)	0.96 (+)	0.81 (+)	0.61 (−)	0.78 (−)	0.97 (−)	0.86 (−)
ho14473	0.81 (+)	0.53 (=)	0.91 (+)	0.54 (=)	0.79 (−)	0.6 (+)	0.82 (−)	0.89 (+)	0.56 (=)	0.91 (−)
mo14185	0.7 (+)	0.95 (+)	0.91 (+)	0.52 (=)	0.85 (+)	0.79 (+)	0.68 (−)	0.58 (−)	0.94 (−)	0.9 (−)
it16862	0.66 (+)	0.82 (+)	0.73 (+)	0.51 (=)	0.65 (+)	0.56 (=)	0.67 (−)	0.56 (=)	0.84 (−)	0.74 (−)
vm22775	0.73 (+)	0.67 (+)	0.88 (+)	0.5 (=)	0.61 (−)	0.59 (+)	0.73 (−)	0.76 (+)	0.65 (−)	0.87 (−)
sw24978	0.72 (+)	0.85 (+)	0.92 (+)	0.52 (=)	0.61 (+)	0.74 (+)	0.73 (−)	0.67 (+)	0.86 (−)	0.93 (−)
bm33708	0.67 (+)	0.81 (+)	0.92 (+)	0.55 (=)	0.62 (+)	0.76 (+)	0.63 (−)	0.69 (+)	0.77 (−)	0.9 (−)

The Nurse-Rostering problem

The statistical analysis provided in this section summarises the rosters found over a 100 independent runs with 3,000 problem evaluations. The instances detail are given in table B.38. More information can be found in [3].

Table B.38: Definition of a nurse rostering instances, with the number of nurses, types of shift and days.

Instance	No of Nurses	No of Types	No of days
BCV-1.8.1	8	5	28
BCV-1.8.2	8	5	28
BCV-1.8.3	8	5	28
BCV-1.8.4	8	5	28
BCV-2.46.1	46	4	28
BCV-3.46.1	46	3	28
BCV-3.46.2	46	3	28
BCV-4.13.1	13	4	29
BCV-4.13.2	13	4	28
BCV-5.4.1	13	4	28
BCV-6.13.1	13	5	30
BCV-6.12.2	13	5	30
BCV-7.10.1	10	6	28
BCV-8.13.1	13	5	28
BCV-8.13.2	13	5	28
BCV-A.12.1	12	5	31
BCV-A.12.2	12	5	31
Instance 1	8	1	14
Instance 2	14	2	14
Instance 3	20	3	14
Instance 4	10	2	28
Instance 5	16	2	28
Instance 6	18	3	28
Instance 7	18	3	28
Instance 9	36	4	28
Instance 10	40	5	28
ORTEC01	16	4	31
ORTEC02	16	4	31
GPOST	8	2	28
GPOST-B	8	2	28
Ikegami-2Shift-DATA1	28	2	30
Ikegami-3Shift-DATA.1[1]	25	3	30

Table B.39: Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-1.8.1, BCV-1.8.2, BCV-1.8.3, BCV-1.8.4 and BCV-2.46.1.

		BCV-1.8.1	BCV-1.8.2	BCV-1.8.3	BCV-1.8.4	BCV-2.46.1
NRP-A	**mean**	4.878e-04	1.585e-02	4.634e-03	-1.195e+00	4.506e+00
	std	(3.4e-03)	(1.2e-02)	(9.6e-03)	(1.6e-15)	(1.3e+00)
	median	0.000e+00	2.439e-02	0.000e+00	-1.195e+00	4.494e+00
	IQR	(0.0e+00)	(2.4e-02)	(0.0e+00)	(0.0e+00)	(1.4e+00)
NRP-B	**mean**	0.000e+00	4.146e-03	2.439e-04	-7.734e-01	2.925e+00
	std	(0.0e+00)	(1.6e-02)	(2.4e-03)	(2.3e-01)	(9.5e-01)
	median	0.000e+00	0.000e+00	0.000e+00	-8.415e-01	2.763e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(2.9e-01)	(1.2e+00)
NRP-C	**mean**	0.000e+00	6.098e-03	0.000e+00	-7.868e-01	5.035e+00
	std	(0.0e+00)	(1.1e-02)	(0.0e+00)	(2.1e-01)	(1.4e+00)
	median	0.000e+00	0.000e+00	0.000e+00	-8.171e-01	5.026e+00
	IQR	(0.0e+00)	(1.2e-02)	(0.0e+00)	(2.7e-01)	(2.0e+00)
NRP-D	**mean**	0.000e+00	7.805e-03	9.756e-04	-7.898e-01	4.187e+00
	std	(0.0e+00)	(1.1e-02)	(4.8e-03)	(2.2e-01)	(1.2e+00)
	median	0.000e+00	0.000e+00	0.000e+00	-8.049e-01	4.141e+00
	IQR	(0.0e+00)	(2.4e-02)	(0.0e+00)	(3.2e-01)	(1.5e+00)
NRP-E	**mean**	0.000e+00	3.659e-03	0.000e+00	2.172e+00	5.576e+00
	std	(0.0e+00)	(8.8e-03)	(0.0e+00)	(3.0e+01)	(1.6e+00)
	median	0.000e+00	0.000e+00	0.000e+00	-8.537e-01	5.615e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(3.0e-01)	(2.0e+00)
NRP-F	**mean**	2.374e+00	7.561e-03	7.317e-04	-7.480e-01	5.845e+00
	std	(2.4e+01)	(1.1e-02)	(4.2e-03)	(2.2e-01)	(1.6e+00)
	median	0.000e+00	0.000e+00	0.000e+00	-7.927e-01	5.724e+00
	IQR	(0.0e+00)	(2.4e-02)	(0.0e+00)	(2.4e-01)	(1.9e+00)
NRP-G	**mean**	0.000e+00	5.366e-03	7.317e-04	-1.180e-01	2.932e+00
	std	(0.0e+00)	(1.0e-02)	(4.2e-03)	(7.2e+00)	(1.0e+00)
	median	0.000e+00	0.000e+00	0.000e+00	-8.537e-01	2.929e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(4.9e-01)	(1.3e+00)
NRP-H	**mean**	0.000e+00	1.073e-02	2.683e-03	-1.195e+00	4.063e+00
	std	(0.0e+00)	(1.2e-02)	(7.7e-03)	(1.6e-15)	(1.1e+00)
	median	0.000e+00	0.000e+00	0.000e+00	-1.195e+00	4.071e+00
	IQR	(0.0e+00)	(2.4e-02)	(0.0e+00)	(0.0e+00)	(1.6e+00)
NRP-I	**mean**	7.829e-02	4.237e-01	2.293e-02	-1.029e+00	4.019e+00
	std	(9.7e-02)	(2.7e-01)	(1.1e-02)	(1.7e-01)	(3.2e+00)
	median	2.439e-02	3.902e-01	2.439e-02	-9.024e-01	2.968e+00
	IQR	(1.1e-01)	(4.1e-01)	(0.0e+00)	(3.2e-01)	(3.5e+00)
NRP-J	**mean**	1.473e-01	4.827e-01	1.166e-01	-6.741e-01	3.020e+01
	std	(4.5e-01)	(3.9e-01)	(2.9e-01)	(2.6e-01)	(2.7e+01)
	median	4.878e-02	4.146e-01	2.439e-02	-6.341e-01	2.065e+01
	IQR	(9.8e-02)	(3.9e-01)	(4.9e-02)	(3.2e-01)	(3.0e+01)

Table B.40: Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-3.46.1, BCV-3.46.2, BCV-4.13.2, BCV-5.4.1.

		BCV-3.46.1	BCV-3.46.2	BCV-4.13.2	BCV-5.4.1
NRP-A	**mean**	4.258e-01	0.000e+00	0.000e+00	0.000e+00
	std	(6.8e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	4.286e-01	0.000e+00	0.000e+00	0.000e+00
	iRQ	(9.1e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-B	**mean**	2.655e-01	0.000e+00	0.000e+00	0.000e+00
	std	(5.0e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	2.597e-01	0.000e+00	0.000e+00	0.000e+00
	iRQ	(6.5e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-C	**mean**	6.406e-01	0.000e+00	0.000e+00	0.000e+00
	std	(1.1e-01)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	6.364e-01	0.000e+00	0.000e+00	0.000e+00
	iRQ	(1.7e-01)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-D	**mean**	5.087e-01	0.000e+00	0.000e+00	0.000e+00
	std	(8.9e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	5.065e-01	0.000e+00	0.000e+00	0.000e+00
	iRQ	(1.3e-01)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-E	**mean**	6.882e-01	0.000e+00	1.178e+02	1.044e+00
	std	(1.1e-01)	(0.0e+00)	(2.1e+02)	(2.7e-15)
	median	6.883e-01	0.000e+00	0.000e+00	1.044e+00
	iRQ	(1.4e-01)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-F	**mean**	8.452e-01	0.000e+00	1.144e+02	1.044e+00
	std	(1.1e-01)	(0.0e+00)	(2.5e+02)	(2.7e-15)
	median	8.442e-01	0.000e+00	0.000e+00	1.044e+00
	iRQ	(1.5e-01)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-G	**mean**	3.825e-01	0.000e+00	1.223e+02	1.044e+00
	std	(6.9e-02)	(0.0e+00)	(2.1e+02)	(2.7e-15)
	median	3.766e-01	0.000e+00	0.000e+00	1.044e+00
	iRQ	(1.0e-01)	(0.0e+00)	(2.4e+02)	(0.0e+00)
NRP-H	**mean**	3.817e-01	0.000e+00	0.000e+00	0.000e+00
	std	(5.9e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	3.896e-01	0.000e+00	0.000e+00	0.000e+00
	iRQ	(7.8e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-I	**mean**	5.803e-01	1.455e-02	2.667e-03	0.000e+00
	std	(2.5e-01)	(1.0e-02)	(7.3e-03)	(0.0e+00)
	median	5.844e-01	1.299e-02	0.000e+00	0.000e+00
	iRQ	(3.6e-01)	(1.3e-02)	(0.0e+00)	(0.0e+00)
NRP-J	**mean**	1.584e+00	3.623e-02	1.468e+02	1.044e+00
	std	(8.3e-01)	(6.0e-02)	(2.5e+02)	(2.7e-15)
	median	1.558e+00	1.299e-02	4.444e-02	1.044e+00
	iRQ	(9.9e-01)	(1.3e-02)	(4.7e+02)	(0.0e+00)

Table B.41: Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-6.13.1, BCV-6.12.2, BCV-7.10.1, BCV-8.13.1 and BCV-8.13.2.

		BCV-6.13.1	BCV-6.12.2	BCV-7.10.1	BCV-8.13.1	BCV-8.13.2
NRP-A	**mean**	7.750e-02	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	std	(8.3e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	1.042e-02	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(1.7e-01)	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-B	**mean**	3.200e-01	0.000e+00	5.455e-02	0.000e+00	0.000e+00
	std	(6.1e-02)	(0.0e+00)	(5.7e-02)	(0.0e+00)	(0.0e+00)
	median	3.333e-01	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(1.1e-01)	(0.0e+00)	(0.0e+00)
NRP-C	**mean**	1.350e-01	0.000e+00	7.955e-03	0.000e+00	0.000e+00
	std	(6.6e-02)	(0.0e+00)	(2.9e-02)	(0.0e+00)	(0.0e+00)
	median	1.667e-01	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-D	**mean**	1.598e-01	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	std	(6.4e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	1.667e-01	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-E	**mean**	1.925e-01	5.253e+01	4.166e+00	1.766e+02	1.280e+02
	std	(8.3e-02)	(1.2e+02)	(2.9e+01)	(5.7e+01)	(7.1e+01)
	median	1.667e-01	0.000e+00	0.000e+00	1.916e+02	1.611e+02
	IQR	(0.0e+00)	(9.2e+01)	(0.0e+00)	(1.4e+01)	(2.1e+01)
NRP-F	**mean**	1.844e+00	5.978e+01	8.156e+00	1.617e+02	1.378e+02
	std	(1.8e+01)	(1.5e+02)	(4.0e+01)	(7.1e+01)	(6.3e+01)
	median	0.000e+00	0.000e+00	0.000e+00	1.907e+02	1.627e+02
	IQR	(1.7e-01)	(9.4e+01)	(0.0e+00)	(1.5e+01)	(1.9e+01)
NRP-G	**mean**	8.354e-02	3.831e+01	0.000e+00	1.747e+02	1.436e+02
	std	(9.6e-02)	(1.1e+02)	(0.0e+00)	(6.0e+01)	(6.1e+01)
	median	0.000e+00	0.000e+00	0.000e+00	1.908e+02	1.671e+02
	IQR	(1.7e-01)	(9.0e+01)	(0.0e+00)	(1.6e+01)	(1.6e+01)
NRP-H	**mean**	4.417e-02	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	std	(7.3e-02)	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
	median	0.000e+00	0.000e+00	0.000e+00	0.000e+00	0.000e+00
	IQR	(1.7e-01)	(0.0e+00)	(0.0e+00)	(0.0e+00)	(0.0e+00)
NRP-I	**mean**	3.565e-01	0.000e+00	9.648e-01	0.000e+00	0.000e+00
	std	(2.6e-01)	(0.0e+00)	(9.9e-01)	(0.0e+00)	(0.0e+00)
	median	3.333e-01	0.000e+00	6.591e-01	0.000e+00	0.000e+00
	IQR	(0.0e+00)	(0.0e+00)	(1.9e+00)	(0.0e+00)	(0.0e+00)
NRP-J	**mean**	4.167e-01	0.000e+00	9.798e-01	1.751e+02	1.383e+02
	std	(6.7e-16)	(0.0e+00)	(2.3e+00)	(5.9e+01)	(6.5e+01)
	median	4.167e-01	0.000e+00	1.136e-01	1.920e+02	1.658e+02
	IQR	(0.0e+00)	(0.0e+00)	(7.3e-01)	(1.4e+01)	(1.3e+01)

Table B.42: Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances BCV-A.12.1, BCV-A.12.2, Instance 2, Instance 3, and Instance 4.

		BCV-A.12.1	BCV-A.12.2	Instance 2	Instance 3	Instance 4
NRP-A	**mean**	6.046e+01	6.320e+01	1.573e+00	1.609e+00	1.635e+00
	std	(9.6e+00)	(9.9e+00)	(8.9e+00)	(8.9e+00)	(8.9e+00)
	median	5.986e+01	6.417e+01	0.000e+00	5.405e-02	7.500e-02
	IQR	(1.4e+01)	(1.5e+01)	(0.0e+00)	(0.0e+00)	(5.0e-02)
NRP-B	**mean**	3.420e+00	2.963e+00	1.961e-01	3.044e-01	1.848e-01
	std	(1.0e+00)	(1.3e+00)	(6.6e-01)	(6.4e-01)	(6.6e-01)
	median	3.427e+00	3.010e+00	8.333e-02	1.892e-01	7.500e-02
	IQR	(1.5e+00)	(1.8e+00)	(6.7e-02)	(5.4e-02)	(5.0e-02)
NRP-C	**mean**	4.057e+00	3.456e+00	5.429e-02	9.175e-02	1.020e-01
	std	(8.2e-01)	(1.1e+00)	(3.2e-01)	(3.2e-01)	(3.2e-01)
	median	4.031e+00	3.438e+00	0.000e+00	5.405e-02	7.500e-02
	IQR	(1.2e+00)	(1.5e+00)	(3.3e-02)	(2.7e-02)	(2.5e-02)
NRP-D	**mean**	1.512e+01	1.531e+01	3.677e-01	4.558e-01	3.975e-01
	std	(2.4e+00)	(2.6e+00)	(2.3e+00)	(2.3e+00)	(2.3e+00)
	median	1.504e+01	1.560e+01	3.333e-02	1.351e-01	7.500e-02
	IQR	(3.6e+00)	(3.4e+00)	(1.0e-01)	(5.4e-02)	(1.2e-02)
NRP-E	**mean**	4.846e+00	4.601e+00	1.402e-01	2.366e-01	8.750e-02
	std	(1.0e+00)	(1.2e+00)	(7.0e-01)	(6.8e-01)	(2.8e-02)
	median	4.844e+00	4.615e+00	3.333e-02	1.351e-01	7.500e-02
	IQR	(1.3e+00)	(1.9e+00)	(1.0e-01)	(6.8e-02)	(3.8e-02)
NRP-F	**mean**	2.714e+01	2.956e+01	4.142e-01	4.587e-01	4.735e-01
	std	(5.5e+00)	(5.5e+00)	(4.1e+00)	(4.1e+00)	(4.1e+00)
	median	2.776e+01	2.865e+01	0.000e+00	5.405e-02	5.000e-02
	IQR	(7.8e+00)	(8.2e+00)	(0.0e+00)	(0.0e+00)	(2.5e-02)
NRP-G	**mean**	1.130e+01	1.129e+01	3.500e-02	1.005e-01	7.000e-02
	std	(3.6e+00)	(3.6e+00)	(4.4e-02)	(5.1e-02)	(2.6e-02)
	median	1.056e+01	1.042e+01	0.000e+00	8.108e-02	7.500e-02
	IQR	(5.8e+00)	(5.8e+00)	(6.7e-02)	(8.1e-02)	(2.5e-02)
NRP-H	**mean**	5.800e+01	5.905e+01	1.083e+00	1.120e+00	5.202e-01
	std	(8.6e+00)	(9.7e+00)	(7.7e+00)	(7.7e+00)	(4.6e+00)
	median	5.846e+01	6.047e+01	0.000e+00	5.405e-02	5.000e-02
	IQR	(1.2e+01)	(1.3e+01)	(0.0e+00)	(2.7e-02)	(2.5e-02)
NRP-I	**mean**	1.887e+01	3.599e+01	8.211e+00	4.690e+01	3.125e+00
	std	(9.4e+00)	(1.5e+02)	(8.5e+00)	(4.4e+02)	(2.3e+00)
	median	1.615e+01	1.753e+01	6.467e+00	2.784e+00	2.562e+00
	IQR	(8.4e+00)	(1.6e+01)	(1.3e-01)	(1.8e+00)	(1.7e+00)
NRP-J	**mean**	1.713e+02	1.544e+02	3.728e+00	3.404e+00	4.853e+00
	std	(2.6e+02)	(2.3e+02)	(1.4e+00)	(2.1e+00)	(2.6e+00)
	median	9.276e+01	8.894e+01	3.333e+00	2.892e+00	5.000e+00
	IQR	(9.0e+01)	(8.1e+01)	(2.8e-01)	(2.8e+00)	(4.8e+00)

Table B.43: Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances Instance 5, Instance 6, Instance 7, Instance 9 and Instance 10.

		Instance 5	Instance 6	Instance 7	Instance 9	Instance 10
NRP-A	**mean**	5.638e+00	8.482e+00	7.875e+00	2.699e+00	1.050e+01
	std	(8.3e+00)	(7.8e+00)	(7.9e+00)	(8.8e+00)	(7.7e+00)
	median	4.283e+00	7.918e+00	6.275e+00	7.059e-01	9.808e+00
	IQR	(2.0e-01)	(1.9e+00)	(1.8e+00)	(8.8e-02)	(2.5e+00)
NRP-B	**mean**	1.761e+00	3.793e+00	3.793e+00	1.290e+00	1.541e+00
	std	(9.0e-01)	(1.2e+00)	(1.2e+00)	(6.0e-01)	(7.3e-01)
	median	2.109e+00	4.041e+00	4.041e+00	1.074e+00	1.164e+00
	IQR	(1.7e-01)	(1.9e+00)	(1.9e+00)	(1.3e-01)	(1.2e+00)
NRP-C	**mean**	3.255e+00	6.220e+00	4.434e+00	1.175e+00	7.364e+00
	std	(1.0e+00)	(1.9e+00)	(1.3e+00)	(9.4e-01)	(2.6e+00)
	median	4.130e+00	6.102e+00	4.320e+00	7.941e-01	8.493e+00
	IQR	(2.0e+00)	(2.0e+00)	(1.8e+00)	(1.0e-01)	(1.7e+00)
NRP-D	**mean**	2.498e+00	5.770e+00	2.422e+00	1.496e+00	4.621e+00
	std	(2.0e+00)	(2.2e+00)	(2.2e+00)	(2.2e+00)	(2.1e+00)
	median	2.217e+00	6.041e+00	2.392e+00	1.029e+00	4.829e+00
	IQR	(1.7e-01)	(2.4e-01)	(2.3e-01)	(9.6e-02)	(1.5e+00)
NRP-E	**mean**	2.144e+00	5.386e+00	2.250e+00	1.469e+00	5.149e+00
	std	(6.6e-01)	(1.6e+00)	(9.4e-01)	(7.0e-01)	(1.6e+00)
	median	2.152e+00	6.010e+00	2.412e+00	1.221e+00	5.219e+00
	IQR	(1.1e-01)	(2.7e-01)	(3.2e-01)	(1.2e-01)	(1.8e+00)
NRP-F	**mean**	4.248e+00	7.306e+00	6.482e+00	1.512e+00	9.786e+00
	std	(3.8e+00)	(3.8e+00)	(3.7e+00)	(4.1e+00)	(4.1e+00)
	median	4.239e+00	7.918e+00	6.196e+00	7.941e-01	9.966e+00
	IQR	(1.8e+00)	(1.9e+00)	(5.3e-01)	(1.2e-01)	(1.5e+00)
NRP-G	**mean**	2.609e+00	6.376e+00	3.161e+00	1.274e+03	5.228e+00
	std	(8.5e-01)	(1.1e+00)	(1.2e+00)	(3.4e+02)	(1.5e+00)
	median	2.239e+00	6.102e+00	2.569e+00	1.312e+03	4.863e+00
	IQR	(2.4e-01)	(1.8e+00)	(1.9e+00)	(7.2e+00)	(1.3e+00)
NRP-H	**mean**	4.771e+00	7.819e+00	7.198e+00	2.015e+00	9.734e+00
	std	(7.2e+00)	(6.9e+00)	(6.9e+00)	(7.6e+00)	(6.8e+00)
	median	4.217e+00	7.867e+00	6.235e+00	6.912e-01	9.664e+00
	IQR	(1.9e+00)	(1.9e+00)	(5.3e-01)	(5.9e-02)	(1.5e+00)
NRP-I	**mean**	1.036e+02	7.064e+02	3.189e+02	2.015e+00	2.682e+01
	std	(5.3e+02)	(1.7e+03)	(5.4e+02)	(7.5e+00)	(2.2e+01)
	median	1.066e+01	1.435e+01	1.796e+01	6.911e-01	1.973e+01
	IQR	(4.2e+00)	(6.4e+00)	(1.9e+01)	(1.6e+00)	(7.5e+00)
NRP-J	**mean**	1.129e+01	6.543e+00	1.377e+01	3.343e+00	1.737e+01
	std	(2.9e+00)	(1.7e+00)	(4.4e+00)	(1.6e+00)	(4.9e+00)
	median	1.096e+01	6.041e+00	1.416e+01	3.713e+00	1.805e+01
	IQR	(3.9e+00)	(2.0e+00)	(4.3e+00)	(1.7e+00)	(4.5e+00)

Table B.44: Statistical comparison of rosters obtained by NRP solvers NRP-[A-J] for the instances ORTECO1, ORTECO2, G-Post, G-Post-B, and Ikegami-3Shift-Data.1.

		ORTEC01	ORTEC02	G-Post	G-Post-B	Ikegami 3 Shift-Data.1
NRP-A	**mean**	9.888e-01	2.031e+01	1.230e+01	9.348e+00	3.919e-01
	std	(1.6e-01)	(1.8e+01)	(1.3e+01)	(1.2e+01)	(7.0e-02)
	median	9.804e-01	6.569e+00	5.645e+00	5.382e+00	3.966e-01
	IQR	(2.0e-01)	(2.7e+01)	(2.6e+01)	(2.1e+01)	(9.5e-02)
NRP-B	**mean**	6.916e-01	2.504e-01	1.361e-01	4.395e-02	1.848e-01
	std	(1.1e-01)	(2.2e-01)	(5.3e-01)	(3.1e-02)	(3.9e-01)
	median	6.863e-01	1.961e-01	7.895e-02	5.263e-02	1.379e-01
	IQR	(2.0e-01)	(3.9e-01)	(2.6e-02)	(2.6e-02)	(6.0e-02)
NRP-C	**mean**	8.600e-01	9.000e-01	6.000e-02	3.711e-02	1.281e-01
	std	(1.3e-01)	(1.4e-01)	(3.8e-02)	(2.5e-02)	(3.3e-02)
	median	8.824e-01	8.824e-01	7.895e-02	5.263e-02	1.207e-01
	IQR	(2.0e-01)	(2.0e-01)	(5.3e-02)	(5.3e-02)	(4.3e-02)
NRP-D	**mean**	9.800e-01	7.401e-01	7.026e-02	5.526e-02	1.767e-01
	std	(1.2e-01)	(1.8e-01)	(2.6e-02)	(3.0e-02)	(3.9e-01)
	median	9.804e-01	7.843e-01	7.895e-02	5.263e-02	1.207e-01
	IQR	(2.0e-01)	(2.9e-01)	(2.6e-02)	(2.6e-02)	(5.2e-02)
NRP-E	**mean**	8.078e-01	3.883e-01	7.526e-02	4.132e-02	1.405e-01
	std	(1.3e-01)	(2.0e-01)	(2.6e-02)	(2.8e-02)	(3.8e-02)
	median	7.843e-01	3.922e-01	7.895e-02	5.263e-02	1.379e-01
	IQR	(9.8e-02)	(2.9e-01)	(0.0e+00)	(2.6e-02)	(5.2e-02)
NRP-F	**mean**	9.580e-01	1.729e+01	1.043e+01	9.137e+00	4.371e-01
	std	(1.4e-01)	(1.8e+01)	(1.2e+01)	(1.3e+01)	(7.9e-02)
	median	9.804e-01	5.686e+00	5.500e+00	5.355e+00	4.310e-01
	IQR	(2.0e-01)	(2.6e+01)	(2.6e+01)	(1.1e+01)	(1.0e-01)
NRP-G	**mean**	8.688e-01	1.348e+01	6.178e+00	4.688e+00	2.257e-01
	std	(1.7e-01)	(1.7e+01)	(1.0e+01)	(8.2e+00)	(1.1e-01)
	median	8.824e-01	3.922e+00	3.421e-01	1.711e-01	2.328e-01
	IQR	(2.0e-01)	(2.9e+01)	(8.2e+00)	(5.5e+00)	(1.9e-01)
NRP-H	**mean**	9.475e-01	4.164e-01	6.105e-02	2.895e-02	1.176e-01
	std	(1.2e-01)	(1.4e-01)	(2.7e-02)	(1.9e-02)	(3.5e-02)
	median	9.804e-01	3.922e-01	7.895e-02	2.632e-02	1.207e-01
	IQR	(2.0e-01)	(9.8e-02)	(2.6e-02)	(2.6e-02)	(5.2e-02)
NRP-I	**mean**	1.461e+01	1.701e+01	2.245e+01	1.907e+01	4.928e-01
	std	(1.9e+01)	(1.6e+01)	(3.1e+01)	(2.7e+01)	(1.9e-01)
	median	3.431e+00	6.029e+00	1.176e+01	1.088e+01	4.741e-01
	IQR	(2.6e+01)	(2.6e+01)	(2.9e+01)	(2.6e+01)	(1.3e-01)
NRP-J	**mean**	1.347e+01	1.981e+01	1.538e+01	1.657e+01	4.310e-01
	std	(1.4e+01)	(1.9e+01)	(1.5e+01)	(1.5e+01)	(2.2e-16)
	median	3.824e+00	6.471e+00	6.066e+00	1.367e+01	4.310e-01
	IQR	(2.6e+01)	(2.7e+01)	(2.6e+01)	(2.6e+01)	(0.0e+00)

Table B.45: Statistical comparison of the generated solver NRP-A and the generated solvers NRP[A-J].

	NRP-A vs								
Instance	**NRP-B**	**NRP-C**	**NRP-D**	**NRP-E**	**NRP-F**	**NRP-G**	**NRP-H**	**NRP-I**	**NRP-J**
BCV-1.8.1	0.51 (=)	0.51 (=)	0.51 (=)	0.51 (=)	0.5 (=)	0.51 (=)	0.51 (=)	0.91 (+)	0.99 (+)
BCV-1.8.2	0.76 (−)	0.7 (−)	0.66 (−)	0.75 (−)	0.67 (−)	0.71 (−)	0.6 (−)	1 (+)	0.99 (+)
BCV-1.8.3	0.59 (−)	0.59 (−)	0.57 (−)	0.59 (−)	0.58 (−)	0.58 (−)	0.54 (=)	0.85 (+)	0.88 (+)
BCV-1.8.4	0.98 (+)	0.99 (+)	0.99 (+)	0.98 (+)	0.98 (+)	0.96 (+)	0.5 (=)	0.76 (+)	1 (+)
BCV-2.46.1	0.84 (−)	0.61 (+)	0.58 (=)	0.7 (+)	0.76 (+)	0.83 (−)	0.6 (=)	0.66 (−)	0.94 (+)
BCV-3.46.1	0.97 (−)	0.95 (+)	0.77 (+)	0.98 (+)	1 (+)	0.68 (−)	0.69 (−)	0.68 (+)	0.94 (+)
BCV-3.46.2	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.9 (+)	1 (+)
BCV-4.13.1	0.5 (=)	0.5 (=)	0.5 (=)	0.59 (−)	0.59 (−)	0.58 (−)	0.5 (=)	0.54 (−)	1 (+)
BCV-4.13.2	0.5 (=)	0.5 (=)	0.5 (=)	0.62 (−)	0.6 (−)	0.62 (−)	0.5 (=)	0.56 (−)	1 (+)
BCV-5.4.1	0.5 (=)	0.5 (=)	0.5 (=)	1 (+)	1 (+)	1 (+)	0.5 (=)	0.5 (=)	1 (+)
BCV-6.13.1	0.97 (+)	0.67 (+)	0.74 (+)	0.79 (+)	0.56 (=)	0.5 (=)	0.61 (−)	0.94 (+)	1 (+)
BCV-6.12.2	0.5 (=)	0.5 (=)	0.5 (=)	0.71 (−)	0.69 (−)	0.65 (−)	0.5 (=)	0.5 (=)	0.5 (=)
BCV-7.10.1	0.74 (−)	0.53 (−)	0.5 (=)	0.51 (=)	0.52 (=)	0.5 (=)	0.5 (=)	0.98 (+)	1 (+)
BCV-8.13.1	0.5 (=)	0.5 (=)	0.5 (=)	0.95 (+)	0.92 (+)	0.95 (+)	0.5 (=)	0.5 (=)	1 (+)
BCV-8.13.2	0.5 (=)	0.5 (=)	0.5 (=)	0.89 (+)	0.92 (+)	0.93 (+)	0.5 (=)	0.5 (=)	1 (+)
BCV-A.12.1	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	0.57 (=)	0.99 (−)	0.76 (+)
BCV-A.12.2	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	1 (−)	0.61 (−)	0.99 (−)	0.72 (+)
Instance 2	0.79 (+)	0.55 (=)	0.67 (+)	0.68 (+)	0.52 (=)	0.61 (−)	0.54 (=)	0.97 (+)	0.96 (+)
Instance 3	0.97 (+)	0.61 (−)	0.94 (+)	0.95 (+)	0.52 (=)	0.78 (+)	0.58 (=)	0.96 (+)	0.96 (+)
Instance 4	0.58 (−)	0.54 (=)	0.51 (=)	0.6 (−)	0.59 (=)	0.55 (=)	0.61 (−)	0.97 (+)	0.96 (+)
Instance 5	0.96 (−)	0.78 (−)	0.92 (−)	0.95 (−)	0.58 (=)	0.87 (−)	0.63 (−)	0.95 (+)	0.96 (+)
Instance 6	0.96 (−)	0.69 (−)	0.78 (−)	0.81 (−)	0.54 (=)	0.7 (−)	0.6 (=)	0.96 (+)	0.66 (−)
Instance 7	0.95 (−)	0.88 (−)	0.97 (−)	0.99 (−)	0.57 (=)	0.95 (−)	0.55 (=)	0.97 (+)	0.86 (+)
Instance 9	0.83 (+)	0.76 (+)	0.84 (+)	0.84 (+)	0.77 (+)	1 (+)	0.6 (=)	0.87 (−)	0.85 (+)
Instance 10	1 (−)	0.76 (−)	0.91 (−)	0.9 (−)	0.61 (+)	0.89 (−)	0.62 (−)	0.97 (+)	0.88 (+)

Table B.46: Statistical comparison of the generated solver NRP-A and the generated solvers NRP[A-J].

	NRP-A vs									
Instance	**NRP-B**	**NRP-C**	**NRP-D**	**NRP-E**	**NRP-F**	**NRP-G**	**NRP-H**	**NRP-I**	**NRP-J**
ORTEC01	0.9 $(-)$	0.74 $(-)$	0.53 $(=)$	0.81 $(-)$	0.57 $(=)$	0.7 $(-)$	0.59 $(=)$	1 $(+)$	1 $(+)$
ORTEC02	0.91 $(-)$	0.91 $(-)$	0.91 $(-)$	0.91 $(-)$	0.5 $(=)$	0.63 $(-)$	0.91 $(-)$	0.51 $(=)$	0.54 $(=)$
G-Post	0.99 $(-)$	1 $(-)$	1 $(-)$	1 $(-)$	0.52 $(=)$	0.7 $(-)$	1 $(-)$	0.63 $(+)$	0.61 $(+)$
G-Post-B	0.99 $(-)$	1 $(-)$	0.99 $(-)$	1 $(-)$	0.51 $(=)$	0.7 $(-)$	1 $(-)$	0.68 $(+)$	0.69 $(+)$
Ikegami	0.98 $(-)$	1 $(-)$	0.98 $(-)$	1 $(-)$	0.66 $(+)$	0.89 $(-)$	1 $(-)$	0.78 $(+)$	0.71 $(+)$

Table B.47: Statistical comparison of the generated solver NRP-B and the generated solvers NRP[C-J].

	NRP-B vs													
	NRP-C		**NRP-D**		**NRP-E**		**NRP-F**		**NRP-G**		**NRP-H**		**NRP-I**		**NRP-J**	
BCV-1.8.1	0.5	(=)	0.5	(=)	0.5	(=)	0.51	(=)	0.5	(=)	0.5	(=)	0.92	(+)	1	(+)
BCV-1.8.2	0.57	(=)	0.6	(−)	0.52	(=)	0.6	(−)	0.55	(=)	0.66	(−)	1	(+)	1	(+)
BCV-1.8.3	0.51	(=)	0.52	(=)	0.51	(=)	0.51	(=)	0.51	(=)	0.55	(−)	0.93	(+)	0.99	(+)
BCV-1.8.4	0.5	(=)	0.5	(=)	0.54	(=)	0.55	(=)	0.57	(=)	0.98	(−)	0.83	(−)	0.63	(+)
BCV-2.46.1	0.89	(+)	0.8	(+)	0.92	(+)	0.94	(+)	0.51	(=)	0.79	(+)	0.53	(+)	0.97	(+)
BCV-3.46.1	1	(+)	0.99	(+)	1	(+)	1	(+)	0.92	(+)	0.94	(+)	0.9	(+)	0.99	(+)
BCV-3.46.2	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.9	(+)	1	(+)
BCV-4.13.1	0.5	(=)	0.5	(=)	0.59	(−)	0.59	(−)	0.58	(−)	0.5	(=)	0.54	(−)	1	(+)
BCV-4.13.2	0.5	(=)	0.5	(=)	0.62	(−)	0.6	(−)	0.62	(−)	0.5	(=)	0.56	(−)	1	(+)
BCV-5.4.1	0.5	(=)	0.5	(=)	1	(+)	1	(+)	1	(+)	0.5	(=)	0.5	(=)	1	(+)
BCV-6.13.1	0.96	(−)	0.94	(−)	0.86	(−)	0.97	(−)	0.95	(−)	0.98	(−)	0.53	(=)	1	(+)
BCV-6.12.2	0.5	(=)	0.5	(=)	0.71	(−)	0.69	(−)	0.65	(−)	0.5	(=)	0.5	(=)	0.5	(=)
BCV-7.10.1	0.7	(−)	0.74	(−)	0.73	(−)	0.71	(−)	0.74	(−)	0.74	(−)	0.87	(+)	0.84	(+)
BCV-8.13.1	0.5	(=)	0.5	(=)	0.95	(+)	0.92	(+)	0.95	(+)	0.5	(=)	0.5	(=)	1	(+)
BCV-8.13.2	0.5	(=)	0.5	(=)	0.89	(+)	0.92	(+)	0.93	(+)	0.5	(=)	0.5	(=)	1	(+)
BCV-A.12.1	0.69	(+)	1	(+)	0.84	(+)	1	(+)	1	(+)	1	(+)	1	(+)	1	(+)
BCV-A.12.2	0.63	(+)	1	(+)	0.82	(+)	1	(+)	0.99	(+)	1	(+)	1	(+)	1	(+)
Instance 2	0.76	(−)	0.64	(−)	0.64	(−)	0.83	(−)	0.7	(−)	0.83	(−)	1	(+)	0.97	(+)
Instance 3	0.98	(−)	0.78	(−)	0.73	(−)	0.99	(−)	0.88	(−)	0.98	(−)	0.83	(+)	0.91	(+)
Instance 4	0.52	(−)	0.57	(−)	0.66	(−)	0.51	(−)	0.51	(−)	0.53	(−)	0.98	(+)	0.98	(+)
Instance 5	0.87	(+)	0.73	(+)	0.64	(+)	0.93	(+)	0.78	(+)	0.92	(+)	1	(+)	1	(+)
Instance 6	0.84	(+)	0.81	(+)	0.78	(+)	0.91	(+)	0.92	(+)	0.93	(+)	0.99	(+)	0.9	(+)
Instance 7	0.73	(+)	0.77	(−)	0.77	(−)	0.91	(+)	0.54	(−)	0.93	(+)	0.99	(+)	0.97	(+)
Instance 9	0.82	(−)	0.62	(−)	0.78	(+)	0.86	(−)	1	(+)	0.88	(−)	0.87	(−)	0.86	(+)
Instance 10	0.96	(+)	0.97	(+)	0.96	(+)	0.98	(+)	0.99	(+)	0.99	(+)	1	(+)	1	(+)

Table B.48: Statistical comparison of the generated solver NRP-B and the generated solvers NRP[C-J].

	NRP-B vs													
	NRP-C		**NRP-D**		**NRP-E**		**NRP-F**		**NRP-G**		**NRP-H**		**NRP-I**		**NRP-J**	
ORTEC01	0.81	$(+)$	0.94	$(+)$	0.71	$(+)$	0.91	$(+)$	0.78	$(+)$	0.9	$(+)$	1	$(+)$	1	$(+)$
ORTEC02	1	$(+)$	0.98	$(+)$	0.72	$(+)$	1	$(+)$	1	$(+)$	0.77	$(+)$	1	$(+)$	1	$(+)$
G-Post	0.66	$(-)$	0.62	$(-)$	0.59	$(+)$	0.99	$(+)$	0.74	$(+)$	0.73	$(-)$	1	$(+)$	1	$(+)$
G-Post-B	0.56	$(=)$	0.56	$(=)$	0.52	$(=)$	0.99	$(+)$	0.78	$(+)$	0.66	$(-)$	1	$(+)$	1	$(+)$
Ikegami	0.56	$(=)$	0.56	$(=)$	0.53	$(=)$	0.98	$(+)$	0.71	$(+)$	0.62	$(-)$	0.98	$(+)$	0.98	$(+)$

Table B.49: Statistical comparison of the generated solver NRP-C and the generated solvers NRP[D-J].

	NRP-C vs NRP-D		**NRP-C vs NRP-E**		**NRP-C vs NRP-F**		**NRP-C vs NRP-G**		**NRP-C vs NRP-H**		**NRP-C vs NRP-I**		**NRP-C vs NRP-J**	
BCV-1.8.1	0.5	(=)	0.5	(=)	0.51	(=)	0.5	(=)	0.5	(=)	0.92	(+)	1	(+)
BCV-1.8.2	0.53	(=)	0.55	(=)	0.53	(=)	0.52	(=)	0.59	(−)	1	(+)	0.99	(+)
BCV-1.8.3	0.52	(=)	0.5	(=)	0.52	(=)	0.52	(=)	0.55	(−)	0.94	(+)	1	(+)
BCV-1.8.4	0.5	(=)	0.54	(=)	0.56	(=)	0.57	(=)	0.99	(−)	0.85	(−)	0.64	(+)
BCV-2.46.1	0.67	(−)	0.6	(=)	0.65	(+)	0.88	(−)	0.7	(−)	0.7	(−)	0.92	(+)
BCV-3.46.1	0.82	(−)	0.61	(+)	0.91	(+)	0.98	(−)	0.99	(−)	0.61	(−)	0.9	(+)
BCV-3.46.2	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.5	(=)	0.9	(+)	1	(+)
BCV-4.13.1	0.5	(=)	0.59	(−)	0.59	(−)	0.58	(−)	0.5	(=)	0.54	(−)	1	(+)
BCV-4.13.2	0.5	(=)	0.62	(−)	0.6	(−)	0.62	(−)	0.5	(=)	0.56	(−)	1	(+)
BCV-5.4.1	0.5	(=)	1	(+)	1	(+)	1	(+)	0.5	(=)	0.5	(=)	1	(+)
BCV-6.13.1	0.59	(−)	0.66	(−)	0.73	(−)	0.65	(−)	0.77	(−)	0.92	(+)	1	(+)
BCV-6.12.2	0.5	(=)	0.71	(−)	0.69	(−)	0.65	(−)	0.5	(=)	0.5	(=)	0.5	(=)
BCV-7.10.1	0.54	(−)	0.52	(=)	0.51	(=)	0.54	(−)	0.54	(−)	0.97	(+)	0.98	(+)
BCV-8.13.1	0.5	(=)	0.95	(+)	0.92	(+)	0.95	(+)	0.5	(=)	0.5	(=)	1	(+)
BCV-8.13.2	0.5	(=)	0.89	(+)	0.92	(+)	0.93	(+)	0.5	(=)	0.5	(=)	1	(+)
BCV-A.12.1	1	(+)	0.73	(+)	1	(+)	1	(+)	1	(+)	1	(+)	1	(+)
BCV-A.12.2	1	(+)	0.75	(+)	1	(+)	0.99	(+)	1	(+)	1	(+)	1	(+)
Instance 2	0.63	(+)	0.63	(+)	0.58	(=)	0.57	(=)	0.59	(−)	1	(+)	0.99	(+)
Instance 3	0.94	(+)	0.94	(+)	0.6	(−)	0.72	(+)	0.69	(−)	0.98	(+)	0.98	(+)
Instance 4	0.56	(=)	0.67	(−)	0.56	(=)	0.51	(=)	0.58	(=)	0.99	(+)	0.99	(+)
Instance 5	0.74	(−)	0.81	(−)	0.7	(+)	0.65	(−)	0.66	(+)	0.99	(+)	1	(+)
Instance 6	0.59	(−)	0.63	(−)	0.65	(+)	0.51	(=)	0.61	(+)	0.99	(+)	0.51	(=)
Instance 7	0.88	(−)	0.89	(−)	0.83	(+)	0.72	(−)	0.87	(+)	0.99	(+)	0.94	(+)
Instance 9	0.83	(+)	0.83	(+)	0.5	(=)	1	(+)	0.84	(−)	0.88	(−)	0.89	(+)
Instance 10	0.81	(−)	0.8	(−)	0.81	(+)	0.76	(−)	0.67	(+)	1	(+)	0.95	(+)

Table B.50: Statistical comparison of the generated solver NRP-C and the generated solvers NRP[D-J].

	NRP-C vs													
	NRP-D		**NRP-E**		**NRP-F**		**NRP-G**		**NRP-H**		**NRP-I**		**NRP-J**	
ORTEC01	0.75	(+)	0.6	(=)	0.7	(+)	0.54	(=)	0.69	(+)	1	(+)	1	(+)
ORTEC02	0.69	(−)	0.88	(−)	1	(−)	0.77	(−)	0.9	(−)	1	(−)	1	(−)
G-Post	0.56	(=)	0.75	(+)	1	(+)	0.81	(+)	0.53	(=)	1	(+)	1	(+)
G-Post-B	0.68	(−)	0.54	(=)	0.99	(+)	0.8	(+)	0.61	(−)	1	(+)	1	(+)
Ikegami	0.5	(=)	0.6	(=)	1	(+)	0.74	(+)	0.58	(=)	1	(+)	1	(+)

Table B.51: Statistical comparison of the generated solvers NRP-[D-E] and the generated solvers NRP[F-J].

	NRP-D vs						**NRP-E vs**				
	NRP-E	**NRP-F**	**NRP-G**	**NRP-H**	**NRP-I**	**NRP-J**	**NRP-F**	**NRP-G**	**NRP-H**	**NRP-I**	**NRP-J**
BCV-1.8.1	0.5 (=)	0.51 (=)	0.5 (=)	0.5 (=)	0.92 (+)	1 (+)	0.51 (=)	0.5 (=)	0.5 (=)	0.92 (+)	1 (+)
BCV-1.8.2	0.58 (−)	0.51 (=)	0.55 (=)	0.56 (=)	1 (+)	0.99 (+)	0.58 (−)	0.53 (=)	0.65 (−)	1 (+)	1 (+)
BCV-1.8.3	0.52 (=)	0.51 (=)	0.51 (=)	0.53 (=)	0.92 (+)	0.98 (+)	0.52 (=)	0.52 (=)	0.55 (−)	0.94 (+)	1 (+)
BCV-1.8.4	0.54 (=)	0.55 (=)	0.57 (=)	0.99 (−)	0.84 (−)	0.64 (+)	0.6 (=)	0.53 (=)	0.98 (−)	0.82 (−)	0.67 (+)
BCV-2.46.1	0.75 (+)	0.8 (+)	0.79 (−)	0.53 (=)	0.62 (−)	0.95 (+)	0.55 (=)	0.92 (−)	0.78 (−)	0.74 (−)	0.91 (+)
BCV-3.46.1	0.9 (+)	0.99 (+)	0.87 (−)	0.88 (−)	0.58 (+)	0.93 (+)	0.84 (+)	0.99 (−)	0.99 (−)	0.67 (−)	0.88 (+)
BCV-3.46.2	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.9 (+)	1 (+)	0.5 (=)	0.5 (=)	0.5 (=)	0.9 (+)	1 (+)
BCV-4.13.1	0.59 (−)	0.59 (−)	0.58 (−)	0.5 (=)	0.54 (−)	1 (+)	0.5 (=)	0.5 (=)	0.59 (−)	0.56 (=)	0.84 (+)
BCV-4.13.2	0.62 (−)	0.6 (−)	0.62 (−)	0.5 (=)	0.56 (−)	1 (+)	0.51 (=)	0.5 (=)	0.62 (−)	0.57 (−)	0.79 (+)
BCV-5.4.1	1 (+)	1 (+)	1 (+)	0.5 (=)	0.5 (=)	1 (+)	0.5 (=)	0.5 (=)	1 (−)	1 (−)	0.5 (=)
BCV-6.13.1	0.58 (−)	0.8 (−)	0.72 (−)	0.84 (−)	0.89 (+)	1 (+)	0.83 (−)	0.77 (−)	0.87 (−)	0.81 (+)	1 (+)
BCV-6.12.2	0.71 (−)	0.69 (−)	0.65 (−)	0.5 (=)	0.5 (=)	0.5 (=)	0.5 (=)	0.55 (=)	0.7 (−)	0.7 (−)	0.7 (−)
BCV-7.10.1	0.51 (=)	0.52 (=)	0.5 (=)	0.5 (=)	0.98 (+)	1 (+)	0.51 (=)	0.51 (=)	0.51 (=)	0.97 (+)	0.98 (+)
BCV-8.13.1	0.95 (+)	0.92 (+)	0.95 (+)	0.5 (=)	0.5 (=)	1 (+)	0.56 (=)	0.52 (=)	0.95 (−)	0.95 (−)	0.51 (=)
BCV-8.13.2	0.89 (+)	0.92 (+)	0.93 (+)	0.5 (=)	0.5 (=)	1 (+)	0.53 (=)	0.59 (=)	0.89 (−)	0.89 (−)	0.58 (+)
BCV-A.12.1	1 (−)	0.98 (+)	0.79 (−)	1 (+)	0.58 (+)	1 (+)	1 (+)	0.99 (+)	1 (+)	1 (+)	1 (+)
BCV-A.12.2	1 (−)	0.99 (+)	0.81 (−)	1 (+)	0.66 (+)	0.99 (+)	1 (+)	0.98 (+)	1 (+)	0.99 (+)	1 (+)
Instance 2	0.5 (=)	0.71 (−)	0.56 (=)	0.72 (−)	0.98 (+)	0.97 (+)	0.71 (−)	0.56 (=)	0.72 (−)	1 (+)	0.98 (+)
Instance 3	0.55 (=)	0.96 (−)	0.7 (−)	0.96 (−)	0.86 (+)	0.94 (+)	0.96 (−)	0.73 (−)	0.97 (−)	0.86 (+)	0.93 (+)
Instance 4	0.62 (−)	0.62 (−)	0.58 (=)	0.65 (−)	0.98 (+)	0.97 (+)	0.71 (−)	0.68 (−)	0.74 (−)	1 (+)	0.99 (+)
Instance 5	0.61 (−)	0.87 (+)	0.58 (=)	0.85 (+)	0.98 (+)	0.98 (+)	0.91 (+)	0.69 (+)	0.89 (+)	1 (+)	1 (+)
Instance 6	0.55 (=)	0.75 (+)	0.62 (+)	0.72 (+)	0.98 (+)	0.56 (=)	0.78 (+)	0.67 (+)	0.76 (+)	0.99 (+)	0.6 (+)
Instance 7	0.51 (=)	0.93 (+)	0.75 (+)	0.95 (+)	0.99 (+)	0.98 (+)	0.95 (+)	0.75 (+)	0.97 (+)	1 (+)	0.99 (+)
Instance 9	0.84 (+)	0.87 (−)	1 (+)	0.89 (−)	0.86 (−)	0.86 (+)	0.87 (−)	1 (+)	0.89 (−)	0.85 (−)	0.84 (+)
Instance 10	0.68 (+)	0.88 (+)	0.59 (=)	0.91 (+)	0.99 (+)	0.99 (+)	0.89 (+)	0.58 (−)	0.91 (+)	1 (+)	0.99 (+)

Table B.52: Statistical comparison of the generated solvers NRP-[D-E] and the generated solvers NRP[F-J].

	NRP-D vs						**NRP-E vs**				
	NRP-E	**NRP-F**	**NRP-G**	**NRP-H**	**NRP-I**	**NRP-J**	**NRP-F**	**NRP-G**	**NRP-H**	**NRP-I**	**NRP-J**
ORTEC01	0.83 $(-)$	0.56 $(=)$	0.7 $(-)$	0.57 $(=)$	1 $(+)$	1 $(+)$	0.78 $(+)$	0.62 $(+)$	0.77 $(+)$	1 $(+)$	1 $(+)$
ORTEC02	0.83 $(-)$	1 $(-)$	0.85 $(-)$	0.84 $(-)$	1 $(-)$	1 $(-)$	1 $(-)$	0.98 $(-)$	0.63 $(=)$	1 $(-)$	1 $(-)$
G-Post	0.72 $(+)$	1 $(+)$	0.79 $(+)$	0.6 $(-)$	1 $(+)$	1 $(+)$	1 $(+)$	0.72 $(+)$	0.83 $(-)$	1 $(+)$	1 $(+)$
G-Post-B	0.65 $(-)$	0.98 $(+)$	0.73 $(+)$	0.77 $(-)$	1 $(+)$	1 $(+)$	0.99 $(+)$	0.78 $(+)$	0.65 $(-)$	1 $(+)$	1 $(+)$
Ikegami	0.59 $(=)$	0.98 $(+)$	0.73 $(+)$	0.57 $(=)$	0.98 $(+)$	0.98 $(+)$	1 $(+)$	0.7 $(+)$	0.66 $(-)$	1 $(+)$	1 $(+)$

Table B.53: Statistical comparison of the generated solvers NRP[F-J].

	NRP-F vs NRP-G		NRP-F vs NRP-H		NRP-F vs NRP-I		NRP-F vs NRP-J		NRP-G vs NRP-H		NRP-G vs NRP-I		NRP-G vs NRP-J		NRP-H vs NRP-I		NRP-H vs NRP-J		NRP-I vs NRP-J	
BCV-1.8.1	0.51	(=)	0.51	(=)	0.91	(+)	0.99	(+)	0.5	(=)	0.92	(+)	1	(+)	0.92	(+)	1	(+)	0.52	(+)
BCV-1.8.2	0.55	(=)	0.56	(=)	1	(+)	0.99	(+)	0.61	(−)	1	(+)	1	(+)	1	(+)	0.99	(+)	0.51	(=)
BCV-1.8.3	0.5	(=)	0.54	(=)	0.92	(+)	0.98	(+)	0.54	(=)	0.92	(+)	0.98	(+)	0.88	(+)	0.93	(+)	0.55	(−)
BCV-1.8.4	0.61	(−)	0.98	(−)	0.88	(−)	0.59	(+)	0.96	(−)	0.77	(−)	0.68	(+)	0.76	(+)	1	(+)	0.9	(+)
BCV-2.46.1	0.94	(−)	0.83	(−)	0.76	(−)	0.9	(+)	0.77	(+)	0.53	(+)	0.97	(+)	0.62	(−)	0.95	(+)	0.94	(+)
BCV-3.46.1	1	(−)	1	(−)	0.84	(−)	0.81	(+)	0.51	(=)	0.74	(+)	0.96	(+)	0.74	(+)	0.96	(+)	0.9	(+)
BCV-3.46.2	0.5	(=)	0.5	(=)	0.9	(+)	1	(+)	0.5	(=)	0.9	(+)	1	(+)	0.9	(+)	1	(+)	0.64	(−)
BCV-4.13.1	0.5	(=)	0.59	(−)	0.56	(=)	0.84	(+)	0.58	(−)	0.55	(=)	0.84	(+)	0.54	(−)	1	(+)	0.96	(+)
BCV-4.13.2	0.52	(=)	0.6	(−)	0.56	(=)	0.81	(+)	0.62	(−)	0.58	(−)	0.78	(+)	0.56	(−)	1	(+)	0.98	(+)
BCV-5.4.1	0.5	(=)	1	(−)	1	(−)	0.5	(=)	1	(−)	1	(−)	0.5	(=)	0.5	(=)	1	(+)	1	(+)
BCV-6.13.1	0.55	(=)	0.55	(=)	0.94	(+)	0.99	(+)	0.6	(−)	0.93	(+)	1	(+)	0.96	(+)	1	(+)	0.95	(+)
BCV-6.12.2	0.54	(=)	0.69	(−)	0.69	(−)	0.69	(−)	0.65	(−)	0.65	(−)	0.65	(−)	0.5	(=)	0.5	(=)	0.5	(=)
BCV-7.10.1	0.52	(=)	0.52	(=)	0.95	(+)	0.96	(+)	0.5	(=)	0.98	(+)	1	(+)	0.98	(+)	1	(+)	0.59	(−)
BCV-8.13.2	0.57	(=)	0.92	(−)	0.92	(−)	0.54	(=)	0.93	(−)	0.93	(−)	0.53	(=)	0.5	(=)	1	(+)	1	(+)
BCV-A.12.1	0.99	(−)	1	(+)	0.82	(−)	0.96	(+)	1	(+)	0.79	(+)	1	(+)	0.99	(−)	0.77	(+)	0.98	(+)
BCV-A.12.2	1	(−)	0.99	(+)	0.75	(−)	0.93	(+)	1	(+)	0.85	(+)	1	(+)	0.98	(−)	0.75	(+)	0.95	(+)
Instance 2	0.65	(−)	0.52	(=)	0.99	(+)	0.98	(+)	0.66	(−)	1	(+)	0.99	(+)	0.98	(+)	0.97	(+)	0.84	(−)
Instance 3	0.78	(+)	0.6	(−)	0.98	(+)	0.98	(+)	0.83	(−)	0.93	(+)	0.97	(+)	0.98	(+)	0.97	(+)	0.66	(+)
Instance 4	0.54	(=)	0.51	(=)	0.99	(+)	0.98	(+)	0.56	(=)	1	(+)	0.99	(+)	0.99	(+)	0.98	(+)	0.76	(+)
Instance 5	0.81	(−)	0.54	(=)	0.98	(+)	0.98	(+)	0.77	(+)	1	(+)	1	(+)	0.97	(+)	0.97	(+)	0.6	(=)
Instance 6	0.68	(−)	0.56	(=)	0.98	(+)	0.63	(−)	0.63	(+)	0.99	(+)	0.53	(=)	0.97	(+)	0.6	(−)	0.98	(−)
Instance 7	0.9	(−)	0.52	(=)	0.98	(+)	0.89	(+)	0.94	(+)	1	(+)	0.97	(+)	0.98	(+)	0.88	(+)	0.74	(−)
Instance 9	1	(+)	0.85	(−)	0.89	(−)	0.88	(+)	1	(−)	0.85	(−)	1	(−)	0.93	(−)	0.91	(+)	0.83	(−)
Instance 10	0.88	(−)	0.71	(−)	0.97	(+)	0.88	(+)	0.88	(+)	1	(+)	0.99	(+)	0.98	(+)	0.9	(+)	0.65	(−)

Table B.54: Statistical comparison of the generated solvers NRP[F-J].

	NRP-F vs NRP-G	NRP-F vs NRP-H	NRP-F vs NRP-I	NRP-F vs NRP-J	NRP-G vs NRP-H	NRP-G vs NRP-I	NRP-G vs NRP-J	NRP-H vs NRP-I	NRP-H vs NRP-J	NRP-I vs NRP-J
ORTEC01	0.64 $(-)$	0.52 $(=)$	1 $(+)$	1 $(+)$	0.63 $(+)$	1 $(+)$	1 $(+)$	1 $(+)$	1 $(+)$	0.55 $(=)$
ORTEC02	0.6 $(-)$	0.91 $(-)$	0.57 $(=)$	0.6 $(=)$	0.87 $(-)$	0.72 $(-)$	0.74 $(-)$	1 $(-)$	1 $(-)$	0.57 $(=)$
G-Post	0.68 $(-)$	1 $(-)$	0.65 $(+)$	0.63 $(+)$	0.82 $(-)$	0.79 $(+)$	0.77 $(+)$	1 $(+)$	1 $(+)$	0.54 $(=)$
G-Post-B	0.7 $(-)$	1 $(-)$	0.68 $(+)$	0.68 $(+)$	0.84 $(-)$	0.82 $(+)$	0.82 $(+)$	1 $(+)$	1 $(+)$	0.5 $(=)$
Ikegami	0.94 $(-)$	1 $(-)$	0.64 $(+)$	0.53 $(-)$	0.77 $(-)$	0.97 $(+)$	0.98 $(+)$	1 $(+)$	1 $(+)$	0.69 $(-)$

Abbreviations

Table B.55: List of abbreviations

Abbreviations	Description
A-CHS	Adaptive Harmony Search
CGP	Cartesian Genetic Programming
GP	Genetic Programming
GRAPE	Graph-Structure Program Evolution
IC-CGP	Implicit Cartesian Genetic Programming
LGP	Linear Genetic Programming
MC	Mimicry problem
NRP	Nurse-rostering problem
PADO	Parallel Algorithm Discovery and Orchestration
PDPG	Parallel distributed Genetic Programming
TSP	Traveling Salesman problem
SEL-HH	Selective hyper-heuristic

Glossary

Algorithms are sequences of primitives that are executed in a certain order. The control of the flow can repeat a set of primitives several times or execute a set of operators only if a condition is met.

Algorithm encoding scheme represents an algorithm, using a data structure.

Algorithm fitness functions assesses the quality of an algorithm. Its purpose is to provide a numeral value that can help predict the algorithm abilities to solve unseen instances.

Algorithm optimisation processes produce algorithms that should efficiently solve a problem. This process does not guarantee to find the optimum algorithms.

Algorithm representation A set of symbols adopted by a group of people used similarly to write an algorithm.

Algorithm search space or algorithm space. A set was representing all possible algorithms for a certain problem, using a well-defined set of operations.

Algorithm Selection Problem finds an algorithm from an algorithm space, such that this algorithm maximises the problem fitness value. It has been formalised by [267].

Algorithm-solution is a sequence of instructions that have been produced by a learning mechanism. An algorithm-solution belong in the algorithm search space.

Bloom's taxonomy classifies educational goals, in six objectives; those are referred as knowledge, comprehension, application, analysis, synthesis, evaluation. It is often represented as a pyramid, with the knowledge at the base and the evaluation at the top.

Coefficient of variation shows how the data is dispersed near a central of tendency. It is calculated by the formulate $C_v = \frac{\sigma}{\mu}$

Complexity represents the amount of work required to find a solution for a problem.

Crossover is a type of reproductive operator that breaks the genetic material of an individual and recombined it with the genetic material of another individual.

Directed acyclic graph represents pair-wise relationships between two objects. The edges are directed to show the direction of flow. They have no cycle.

Directed graphs or directed "cyclic" graphs represent pair-wise relationships between two objects. The edges are directed to show the direction of flow. These graphs can have cycles. In this document we refer *directed graphs* and *directed cyclic graphs* interchangeably.

Effort is a metric used to measure the effort to understand an algorithm [127].

Flowchart depicts some algorithms steps by steps. They often use boxes and arrows to diagrammatically shows the flows of operations from the start to an end.

Generative hyper-heuristics generates a sequence of primitives using a given set of stochastic operators. The product can be algorithm-solutions as well as problem-solutions.

Graph-based Genetic Programming is a form a genetic programming that encodes programs with a directed graph.

Heuristics A non-deterministic search method that offers an alternative approach to exhaustive search. Those can find solutions to difficult computational problems in a reasonable amount of time. These methods guarantee to find a solution at any time, but it may not be optimum.

Human learnability is a concept that readers can easily and quickly familiarise themselves with new solvers. This concept is often applied in user interaction design.

Hyper-heuristics is a search methodology that selects or generates heuristics to find solutions of hard computational problems [59, 65] In some part of the literature, this term is also spelt without the hyphen.

Inductive bias represents some assumptions a learning algorithm uses to predict an output from an input.

Imperative programming uses statements that change program states. In the context of this work, a permanent and temporary population represents the states. The operators change the states of these two populations of problem solutions.

Instance is often considered as a concrete representation of a problem. It should have some features that differentiate them from the others.

Knowledge can be defined as the information stored in the computer system or facts, information, and skills acquired through experience or education.

Language system A linguistic system that combines elements into patterned expressions, that can be used to accomplish specific tasks in specific contexts [101].

Length is metric that estimate a program length [127].

Lexicon is the vocabulary of a language. In the context of this document, the language is used to code some metaheuristics using some symbols representing some variables, constants, and operators. The latter includes assignment, boolean, logical, problem specific and population operators.

Local search apply some local changes until a limit of time is elapsed.

Meta-learning improves algorithms performance through experience. A meta-level optimises the performance of an algorithm, and a base-level specialises in a problem to solve.

Metaheuristics The purpose of such approaches is to find, generate, or select a method or algorithm to solve a problem; their search space is now the collection of all possible heuristics and the outcome can be formulae or algorithms together with a solution of the problem it solves.

Mutation changes some genes in a problem solutions to produce a new offspring solution.

No of independent path is measure that indicates the number of possible paths that exists in a program. It indicates the number of tests and the level of maintenance required for a program [215].

Objectives are soft constraints in the context of constraint-satisfaction problem. Those add values to a solution. The purpose of the nurse rostering problem is to lower its cost. Therefore, a significant score would indicate none or few objectives are met by a roster. Otherwise, the cost is low.

Path is a sequence of edges which connect a sequence of vertices. In a directed graph (acyclic and cyclic), all the edges must be in the same direction.

Primitives are segments of the code that can be used to construct programs.

Problem - A general statement describing problem an objective to achieve.

Problem algorithm refers to a property that characterises the problem domain and should affect its output.

Problem domain is a component that includes the operators used to find solutions, a problem encoding scheme, a problem fitness function and a problem search space.

Problem encoding scheme represents a solution of a problem, using a data structure.

Problem fitness functions assess the quality of a solution, by providing a numerical value based on a solution.

Problem optimisation processes search for a solution for a problem.

Problem-solutions refers to a solution of a problem that has been found by a heuristic.

Programs are often considered to be mathematical expressions in genetic programming. It can also be a sequence of instructions or subroutines.

Programming languages are made of symbols and keywords that can be used by programmers to give instructions to a computer.

Prototype is a patterned of bitstring that needs to be imitated by another. (Mimicry problem)

Pseudocode is thought as a simplified programming language. It is used in program design. The pseudocode applied in this thesis is based on imperative programming.

Quality of an algorithm A measure that helps to determine the ability to solve a problem. In machine learning, it can be referred as an *objective function* or a *fitness function*. In this document, it is referred as *an algorithm fitness function*.

Recombination see crossover

Ruin-and-Recreate is a variety of operators that removes or mutate some part of the genetic code (ruin) and then repair the damaged solution (recreate).

Selective hyper-heuristics build an algorithm from an empty state, and stochastic operators are incrementally added to produce a complete sequence of operators gradually.

Vocabulary is a metric that measure the size of a symbols used in a programs [127].

References

[1] National traveling salesman problems. http://www.math.uwaterloo.ca /tsp/world/countries.html. Accessed: 2016-09-30.

[2] The online encyclopedi of integer sequences. https://oeis.org/A003087/list. Accessed:2017-02-25.

[3] XML FORMAT FOR AUTOROSTER PROBLEM INSTANCES, howpublished = `http://www.schedulingbenchmarks.org/documentation.html`, note = Accessed: 2016-10-11.

[4] Ajith Abraham, Nadia Nedjah, and Luiza de Macedo Mourelle. Evolutionary computation: from genetic algorithms to genetic programming. In *Genetic Systems Programming*, pages 1–20. Springer, 2006.

[5] Alexandros Agapitos, Michael O'Neill, Ahmed Kattan, and Simon M. Lucas. Recursion in tree-based genetic programming. *Genetic Programming and Evolvable Machines*, 18(2):149–183, 2017.

[6] Alexandros Agapitos, Michael ONeill, Ahmed Kattan, and Simon M Lucas. Recursion in tree-based genetic programming. *Genetic Programming and Evolvable Machines*, pages 1–35, 2016.

[7] Fardin Ahmadizar, Khabat Soltanian, Fardin AkhlaghianTab, and Ioannis Tsoulos. Artificial neural network development by means of a novel combination of grammatical evolution and genetic algorithm. *Engineering Applications of Artificial Intelligence*, 39:1–13, 2015.

[8] Uwe Aickelin and Kathryn Dowsland. Exploiting problem structure in a genetic algorithm approach to a nurse rostering problem. *arXiv preprint arXiv:0802.2001*, 2008.

[9] Uwe Aickelin and Kathryn A Dowsland. An indirect genetic algorithm for a nurse-scheduling problem. *Computers & Operations Research*, 31(5):761–778, 2004.

[10] Bahriye Akay and Dervis Karaboga. Parameter tuning for the artificial bee colony algorithm. In *International Conference on Computational Collective Intelligence*, pages 608–619. Springer, 2009.

[11] Brad Alexander and Brad Zacher. Boosting search for recursive functions using partial call-trees. In *International Conference on Parallel Problem Solving from Nature*, pages 384–393. Springer, 2014.

[12] Shawkat Ali and Kate A Smith. On learning algorithm selection for classification. *Applied Soft Computing*, 6(2):119–138, 2006.

[13] Alejandro Reyes Amaro, Eric Monfroy, and Florian Richoux. Un langage orienté parallèle pour modéliser des solveurs de contraintes. In *Onzièmes Journées Francophones de Programmation par Contraintes (JFPC)*, 2015.

[14] Peter J Angeline. Adaptive and self-adaptive evolutionary computations. In *Computational intelligence: a dynamic systems perspective*. Citeseer, 1995.

[15] David Applegate, R Bixby, Vasek Chvátal, and W Cook. Cutting planes and the traveling salesman problem. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, page 429. Society for Industrial and Applied Mathematics, 2000.

[16] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[17] Aliasghar Arab and Alireza Alfi. An adaptive gradient descent-based local search in memetic algorithm applied to optimal controller design. *Information Sciences*, 299:117–142, 2015.

[18] Alejandro Arbelaez, Youssef Hamadi, and Michele Sebag. Building portfolios for the protein structure prediction problem. In *Workshop on Constraint Based Methods for Bioinformatics*, 2010.

[19] Dirk V Arnold and Nikolaus Hansen. A (1+ 1)-cma-es for constrained optimisation. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 297–304. ACM, 2012.

[20] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[21] Shahriar Asta and Ender Ozcan. A tensor-based approach to nurse rostering.

[22] M Naceur Azaiez and SS Al Sharif. A 0-1 goal programming model for nurse scheduling. *Computers & Operations Research*, 32(3):491–507, 2005.

[23] Zalilah Abd Aziz. Ant colony hyper-heuristics for travelling salesman problem. *Procedia Computer Science*, 76:534–538, 2015.

[24] Thomas Bäck and Frank Hoffmeister. Basic aspects of evolution strategies. *Statistics and Computing*, 4(2):51–63, 1994.

[25] Mohamed Bader-El-Den and Riccardo Poli. Generating sat local-search heuristics using a gp hyper-heuristic framework. In *Artificial evolution*, pages 37–49. Springer, 2007.

[26] Mohamed Bader-El-Den, Riccardo Poli, and Shaheen Fatima. Evolving timetabling heuristics using a grammar-based genetic programming hyper-heuristic framework. *Memetic Computing*, 1(3):205–219, 2009.

[27] Alampallam V Balakrishnan. *Introduction to random processes in engineering*. Wiley New York, 1995.

[28] Wolfgang Banzhaf. The "molecular" traveling salesman. *Biological Cybernetics*, 64(1):7–14, 1990.

[29] Rodrigo C Barros, Márcio P Basgalupp, André CPLF de Carvalho, and Alex A Freitas. Towards the automatic design of decision tree induction algorithms. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 567–574. ACM, 2011.

[30] Rodrigo C Barros, Márcio P Basgalupp, André CPLF de Carvalho, and Alex A Freitas. Automatic design of decision-tree algorithms with evolutionary algorithms. *Evolutionary computation*, 21(4):659–684, 2013.

[31] Rodrigo C Barros, André CPLF de Carvalho, and Alex A Freitas. Evolutionary algorithms and hyper-heuristics. In *Automatic Design of Decision-Tree Induction Algorithms*, pages 47–58. Springer, 2015.

[32] Aisha Batool, Muhammad Habib ur Rehman, Aihab Khan, and Amsa Azeem. Impact and comparison of programming constructs on java and c# source code readability. 2015.

[33] Thomas BDack, Frank Hoffmeister, and Hans-Paul Schwefel. A survey of evolution strategies. In *Proceedings of the 4th international conference on genetic algorithms*, pages 2–9, 1991.

[34] J Christopher Beck and Eugene C Freuder. Simple rules for low-knowledge algorithm selection. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 50–64. Springer, 2004.

[35] Gareth Beddoe, Sanja Petrovic, and Jingpeng Li. A hybrid metaheuristic case-based reasoning system for nurse rostering. *Journal of Scheduling*, 12(2):99–119, 2009.

[36] Hans-Georg Beyer. Evolution strategies. *Scholarpedia*, 2(8):1965, 2007.

[37] Leonardo CT Bezerra, Manuel López-Ibánez, and Thomas Stützle. Automatic design of evolutionary algorithms for multi-objective combinatorial optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 508–517. Springer, 2014.

[38] Benjamin S Bloom et al. Taxonomy of educational objectives. vol. 1: Cognitive domain. *New York: McKay*, pages 20–24, 1956.

[39] Scott Brave. Evolving recusive programs for tree search. 1996.

[40] Pavel Brazdil and Lars Kotthoff. Metalearning & algorithm selection. 2015.

[41] Pavel B Brazdil, Ricardo Vilalta, Carlos Soares, and Christophe Giraud-Carrier. Meta-learning. In *Encyclopedia of the Sciences of Learning*, pages 2239–2243. Springer, 2012.

[42] Esther Bron, Marion Smits, John van Swieten, Wiro Niessen, and Stefan Klein. Feature selection based on svm significance maps for classification of dementia. In *International Workshop on Machine Learning in Medical Imaging*, pages 272–279. Springer, 2014.

[43] David C Brown and Balakrishnan Chandrasekaran. *Design problem solving: knowledge structures and control strategies*. Morgan Kaufmann, 2014.

[44] Giuseppe Bruno, Andrea Genovese, and Gennaro Improta. Routing problems: a historical perspective. *BSHM Bulletin*, 26(2):118–127, 2011.

[45] Edmund Burke, Peter Cowling, Patrick De Causmaecker, and Greet Vanden Berghe. A memetic approach to the nurse rostering problem. *Applied Intelligence*, 15(3):199–214, 2001.

[46] Edmund Burke, Patrick De Causmaecker, Sanja Petrovic, and Greet Vanden Berghe. Variable neighborhood search for nurse rostering problems. In *Metaheuristics: computer decision-making*, pages 153–172. Springer, 2003.

[47] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of metaheuristics*, pages 457–474. Springer, 2003.

[48] Edmund K Burke, Timothy Curtois, Gerhard Post, Rong Qu, and Bart Veltman. A hybrid heuristic ordering and variable neighbourhood search for the nurse rostering problem. *European Journal of Operational Research*, 188(2):330–341, 2008.

[49] Edmund K Burke, Timothy Curtois, Rong Qu, and G Vanden Berghe. A time predefined variable depth search for nurse rostering. *INFORMS Journal on Computing*, 2007.

[50] Edmund K Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem. The state of the art of nurse rostering. *Journal of scheduling*, 7(6):441–499, 2004.

[51] Edmund K Burke, Patrick De Causmaecker, Sanja Petrovic, and G Vanden Berghe. Fitness evaluation for nurse scheduling problems. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 1139–1146. IEEE, 2001.

[52] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[53] Edmund K Burke, Mathew R Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R Woodward. Exploring hyper-heuristic methodologies with genetic programming. In *Computational intelligence*, pages 177–201. Springer, 2009.

[54] Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. A classification of hyper-heuristic approaches. In *Handbook of metaheuristics*, pages 449–468. Springer, 2010.

[55] Edmund K Burke, Matthew Hyde, Graham Kendall, and John Woodward. A genetic programming hyper-heuristic approach for evolving 2-d strip packing heuristics. *IEEE Transactions on Evolutionary Computation*, 14(6):942–958, 2010.

[56] Edmund K Burke, Matthew R Hyde, and Graham Kendall. Evolving bin packing heuristics with genetic programming. In *Parallel Problem Solving from Nature-PPSN IX*, pages 860–869. Springer, 2006.

[57] Edmund K Burke, Jingpeng Li, and Rong Qu. A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems. *European Journal of Operational Research*, 203(2):484–493, 2010.

[58] Edmund K Burke, Barry McCollum, Amnon Meisels, Sanja Petrovic, and Rong Qu. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of Operational Research*, 176(1):177–192, 2007.

[59] EK Burke, M Hyde, G Kendall, G Ochoa, E Ozcan, and J Woodward. Handbook of metaheuristics, volume 146 of international series in operations research & management science, chapter a classification of hyper-heuristic approaches, 2010.

[60] Raymond PL Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008.

[61] Xinye Cai, Stephen L Smith, and Andy M Tyrrell. Positional independence and recombination in cartesian genetic programming. In *European Conference on Genetic Programming*, pages 351–360. Springer, 2006.

[62] Jaime G Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. Technical report, DTIC Document, 1985.

[63] Jorge Cardoso, Jan Mendling, Gustaf Neumann, and Hajo A Reijers. A discourse on complexity of process models. In *Business process management workshops*, pages 117–128. Springer, 2006.

[64] Professor Cayley. Desiderata and suggestions: No. 2. the theory of groups: Graphical representation. *American Journal of Mathematics*, 1(2):174–176, 1878.

[65] Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics: recent developments. In *Adaptive and multilevel metaheuristics*, pages 3–29. Springer, 2008.

[66] Ching-Yuen Chan, Fan Xue, WH Ip, and CF Cheung. A hyper-heuristic inspired by pearl hunting. In *Learning and intelligent optimization*, pages 349–353. Springer, 2012.

[67] Edward C Chang, Thomas J D'Zurilla, and Lawrence J Sanna. *Social problem solving: Theory, research, and training*. American Psychological Association, 2004.

[68] Brenda Cheang, Haibing Li, Andrew Lim, and Brian Rodrigues. Nurse rostering problems—-a bibliographic survey. *European Journal of Operational Research*, 151(3):447–460, 2003.

[69] Guang Chen and Mengjie Zhang. Evolving while-loop structures in genetic programming for factorial and ant problems. *AI 2005: Advances in Artificial Intelligence*, pages 1079–1085, 2005.

[70] Yujie Chen, Philip Mourdjis, Fiona Polack, Peter Cowling, and Stephen Remde. Evaluating hyperheuristics and local search operators for periodic routing problems. In *Evolutionary Computation in Combinatorial Optimization*, pages 104–120. Springer, 2016.

[71] BMW Cheng, Jimmy Ho-Man Lee, and JCK Wu. A nurse rostering system using constraint programming and redundant modeling. *IEEE Transactions on information technology in biomedicine*, 1(1):44–54, 1997.

[72] Gopinath Chennupati, R Azad, and Conor Ryan. Performance optimization of multi-core grammatical evolution generated parallel recursive programs. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1007–1014. ACM, 2015.

[73] Gopinath Chennupati, Jeannie Fitzgerald, and Colan Ryan. On the efficiency of multi-core grammatical evolution (mcge) evolving multi-core parallel programs. In *Nature and Biologically Inspired Computing (NaBIC), 2014 Sixth World Congress on*, pages 238–243. IEEE, 2014.

[74] Vic Ciesielski and Xiang Li. Experiments with explicit for-loops in genetic programming. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 1, pages 494–501. IEEE, 2004.

[75] Carlos A Coello Coello, Alan D Christiansen, and Arturo Hernández Aguirre. Use of evolutionary techniques to automate the design of combinational circuits. *International Journal of Smart Engineering System Design*, 2:299–314, 2000.

[76] William Comisky, Jessen Yu, and John Koza. Automatic synthesis of a wire antenna using genetic programming. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference, Las Vegas, Nevada*, pages 179–186. Citeseer, 2000.

[77] William Cook. National travelling salesman problems, 2009.

[78] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *Practice and theory of automated timetabling III*, pages 176–190. Springer, 2000.

[79] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *Practice and Theory of Automated Timetabling III*, pages 176–190. Springer, 2001.

[80] Broderick Crawford, Ricardo Soto, Eric Monfroy, Wenceslao Palma, Carlos Castro, and Fernando Paredes. Parameter tuning of a choice-function based hyperheuristic using particle swarm optimization. *Expert Systems with Applications*, 40(5):1690–1695, 2013.

[81] Laura Cruz-Reyes, Claudia Gómez-Santillán, Norberto Castillo-García, Marcela Quiroz, Alberto Ochoa, and Paula Hernández-Hernández. A visualization tool for heuristic algorithms analysis. In *7th International Conference on Knowledge Management in Organizations: Service and Cloud Computing*, pages 515–524. Springer, 2013.

[82] Laura Cruz-Reyes, Claudia Gómez-Santillán, Joaquín Pérez-Ortega, Vanesa Landero, Marcela Quiroz, and Alberto Ochoa. Algorithm selection: From meta-learning to hyper-heuristics.

[83] Tim Curtois, Gabriela Ochoa, Matthew Hyde, and José Antonio Vázquez-Rodríguez. A hyflex module for the personnel scheduling problem. *School of Computer Science, University of Nottingham, Tech. Rep*, 2010.

[84] George B Dantzig. Letter to the editor-a comment on edie's traffic delays at toll booths. *Journal of the Operations Research Society of America*, 2(3):339–341, 1954.

[85] Charles Darwin and William F Bynum. The origin of species by means of natural selection: or, the preservation of favored races in the struggle for life. 2009.

[86] Lawrence Davis et al. *Handbook of genetic algorithms*, volume 115. Van Nostrand Reinhold New York, 1991.

[87] J Arjan GM de Visser and Joachim Krug. Empirical fitness landscapes and the predictability of evolution. *Nature Reviews Genetics*, 15(7):480–490, 2014.

[88] Paul Dempster and John H Drake. Two frameworks for cross-domain heuristic and parameter selection using harmony search. In *Harmony Search Algorithm*, pages 83–94. Springer, 2016.

[89] Jörg Denzinger, Marc Fuchs, and Matthias Fuchs. *High performance ATP systems by combining several AI methods*. Citeseer.

[90] Shifei Ding, Hui Li, Chunyang Su, Junzhao Yu, and Fengxiang Jin. Evolutionary artificial neural networks: a review. *Artificial Intelligence Review*, 39(3):251–260, 2013.

[91] Laura Silvia Diosan and Mihai Oltean. Evolving evolutionary algorithms using evolutionary algorithms. In *Proceedings of the 9th annual conference companion on Genetic and evolutionary computation*, pages 2442–2449. ACM, 2007.

[92] Bei Dong, Licheng Jiao, and Jianshe Wu. Graph-based hybrid hyper-heuristic channel scheduling algorithm in multicell networks. *Transactions on Emerging Telecommunications Technologies*, 28(1), 2017.

[93] Xingye Dong, Maciek Nowak, Ping Chen, and Youfang Lin. A self-adaptive iterated local search algorithm on the permutation flow shop scheduling problem. In *Informatics in Control, Automation and Robotics (ICINCO), 2014 11th International Conference on*, volume 1, pages 378–384. IEEE, 2014.

[94] Xingye Dong, Maciek Nowak, Ping Chen, and Youfang Lin. Self-adaptive perturbation and multi-neighborhood search for iterated local search on the permutation flow shop problem. *Computers & Industrial Engineering*, 87:176–185, 2015.

[95] John H Drake, Ender Özcan, and Edmund K Burke. An improved choice function heuristic selection for cross domain heuristic search. In *International Conference on Parallel Problem Solving from Nature*, pages 307–316. Springer, 2012.

[96] Leslie C Edie. Traffic delays at toll booths. *Journal of the operations research society of America*, 2(2):107–138, 1954.

[97] Agoston E Eiben and Selmar K Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[98] Agoston Endre Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, 3(2):124–141, 1999.

[99] Andoni Elola, Javier Del Ser, Miren Nekane Bilbao, Cristina Perfecto, Enrique Alexandre, and Sancho Salcedo-Sanz. Hybridizing cartesian genetic programming and harmony search for adaptive feature construction in supervised learning problems. *Applied Soft Computing*, 2016.

[100] Andreas T Ernst, Houyuan Jiang, Mohan Krishnamoorthy, and David Sier. Staff scheduling and rostering: A review of applications, methods and models. *European journal of operational research*, 153(1):3–27, 2004.

[101] Edward Finegan. *Language: Its structure and use*. Cengage Learning, 2014.

[102] David B Fogel. An evolutionary approach to the traveling salesman problem. *Biological Cybernetics*, 60(2):139–144, 1988.

[103] Richard Forsyth. Beagle-a darwinian approach to pattern recognition. *Kybernetes*, 10(3):159–166, 1981.

[104] Richard M Friedberg. A learning machine: Part i. *IBM Journal of Research and Development*, 2(1):2–13, 1958.

[105] Alex S Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, 2008.

[106] Matteo Gagliolo and Jürgen Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, 2006.

[107] Edgar Galvan-Lopez. Efficient graph-based genetic programming representation with multiple outputs. *International Journal of Automation and Computing*, 5(1):81–89, 2008.

[108] Cormac Gebruers, Alessio Guerri, Brahim Hnich, and Michela Milano. Making choices using structure at the instance level within a case based reasoning framework. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, pages 380–386. Springer, 2004.

[109] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. A portfolio solver for answer set programming: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning*, pages 352–357. Springer, 2011.

[110] Celia A Glass and Roger A Knight. The nurse rostering problem: A critical appraisal of the problem structure. *European Journal of Operational Research*, 202(2):379–389, 2010.

[111] Fred Glover and Claude McMillan. The general employee scheduling problem. an integration of ms and ai. *Computers & operations research*, 13(5):563–573, 1986.

[112] Fred W Glover and Gary A Kochenberger. *Handbook of metaheuristics*, volume 57. Springer Science & Business Media, 2006.

[113] David E Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, volume 154. Lawrence Erlbaum, Hillsdale, NJ, 1985.

[114] Brian W Goldman and William F Punch. Reducing wasted evaluations in cartesian genetic programming. In *European Conference on Genetic Programming*, pages 61–72. Springer, 2013.

[115] Juan Carlos Gomez and Hugo Terashima-Marín. Evolutionary hyper-heuristics for tackling bi-objective 2d bin packing problems. *Genetic Programming and Evolvable Machines*, pages 1–31, 2017.

[116] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum, New Jersey (160-168), 1985.

[117] John J Grefenstette. Incorporating problem specific knowledge into genetic algorithms. *Genetic algorithms and simulated annealing*, 4:42–60, 1987.

[118] Jacomine Grobler and Andries Engelbrecht. Metric-based heuristic space diversity management in a meta-hyper-heuristic framework. In *Computational Intelligence, 2015 IEEE Symposium Series on*, pages 1665–1672. IEEE, 2015.

[119] Jacomine Grobler, Andries Engelbrecht, Graham Kendall, and VSS Yadavalli. The entity-to-algorithm allocation problem: extending the analysis. In *Computational Intelligence in Ensemble Learning (CIEL), 2014 IEEE Symposium on*, pages 1–8. IEEE, 2014.

[120] Jacomine Grobler, Andries P Engelbrecht, Graham Kendall, and VSS Yadavalli. Heuristic space diversity control for improved meta-hyper-heuristic performance. *Information Sciences*, 300:49–62, 2015.

[121] Jacomine Grobler, Andries Petrus Engelbrecht, Graham Kendall, and VSS Yadavalli. Multi-method algorithms: Investigating the entity-to-algorithm allocation problem. In *IEEE Congress on Evolutionary Computation*, pages 570–577, 2013.

[122] Sumit Gulwani. Technical perspective: Program synthesis using stochastic techniques. *Communications of the ACM*, 59(2):113–113, 2016.

[123] Haipeng Guo. *Algorithm selection for sorting and probabilistic inference: a machine learning-based approach*. PhD thesis, Citeseer, 2003.

[124] Gregory Gutin and Daniel Karapetyan. A memetic algorithm for the generalized traveling salesman problem. *Natural Computing*, 9(1):47–60, 2010.

[125] Mohammed Hadwan, Masri Ayob, Nasser R Sabar, and Roug Qu. A harmony search algorithm for nurse rostering problems. *Information Sciences*, 233:126–140, 2013.

[126] Michael Hahsler and Kurt Hornik. Tsp-infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 23(2):1–21, 2007.

[127] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[128] Nikolaus Hansen. An analysis of mutative σ-self-adaptation on linear fitness functions. *Evolutionary Computation*, 14(3):255–275, 2006.

[129] Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

[130] Nikolaus Hansen, Asma Atamna, and Anne Auger. How to assess step-size adaptation mechanisms in randomised search. In *International Conference on Parallel Problem Solving from Nature*, pages 60–69. Springer, 2014.

[131] Nikolaus Hansen and Stefan Kern. Evaluating the cma evolution strategy on multimodal test functions. In *International Conference on Parallel Problem Solving from Nature*, pages 282–291. Springer, 2004.

[132] Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es). *Evolutionary computation*, 11(1):1–18, 2003.

[133] Pierre Hansen and Nenad Mladenović. First vs. best improvement: An empirical study. *Discrete Applied Mathematics*, 154(5):802–817, 2006.

[134] Simon Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 1921–1928. IEEE, 2008.

[135] Simon Harding, Vincent Graziano, Jürgen Leitner, and Jürgen Schmidhuber. Mt-cgp: Mixed type cartesian genetic programming. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 751–758. ACM, 2012.

[136] Simon Harding, Juergen Leitner, and Juergen Schmidhuber. Cartesian genetic programming for image processing. In *Genetic programming theory and practice X*, pages 31–44. Springer, 2013.

[137] Simon Harding, Julian F Miller, and Wolfgang Banzhaf. Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing. In *European Conference on Genetic Programming*, pages 133–144. Springer, 2009.

[138] Simon Harding, Julian F Miller, and Wolfgang Banzhaf. Developments in cartesian genetic programming: self-modifying cgp. *Genetic Programming and Evolvable Machines*, 11(3-4):397–439, 2010.

[139] Simon L Harding, Julian F Miller, and Wolfgang Banzhaf. Self-modifying cartesian genetic programming. In *Cartesian Genetic Programming*, pages 101–124. Springer, 2011.

[140] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2012.

[141] Kyle I Harrington, Lee Spector, Jordan B Pollack, and Una-May O'Reilly. Autoconstructive evolution for structural problems. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 75–82. ACM, 2012.

[142] Sean Harris, Travis Bueter, and Daniel R Tauritz. A comparison of genetic programming variants for hyper-heuristics. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1043–1050. ACM, 2015.

[143] Emma Hart and Kevin Sim. On constructing ensembles for combinatorial optimisation. *Evolutionary Computation*, 2017.

[144] Pei He, Zelin Deng, Chongzhi Gao, Liang Chang, and Achun Hu. Analyzing grammatical evolution and\ pi grammatical evolution with grammar model. In *Information Technology and Intelligent Transportation Systems*, pages 483–489. Springer, 2017.

[145] Keld Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic. *Mathematical Programming Computation*, 1(2-3):119–163, 2009.

[146] Michael Herdy. Application of the 'evolutionsstrategie' to discrete optimization problems. In *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, PPSN I, pages 188–192, London, UK, UK, 1991. Springer-Verlag.

[147] Holger Hoos, Marius Lindauer, and Torsten Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *arXiv preprint arXiv:1405.1520*, 2014.

[148] Holger H Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80, 2012.

[149] Radek Hrbácek, Michaela Sikulova, Pietro Lio, Orazio Miglino, Giuseppe Nicosia, Stefano Nolfi, and Mario Pavone. Coevolutionary cartesian genetic programming in. *Advances in Artificial Life, ECAL 2013*, pages 431–438, 2013.

[150] Alois Huning, Ingo Rechenberg, and Manfred Eigen. Evolutionsstrategie. optimierung technischer systeme nach prinzipien der biologischen evolution, 1976.

[151] Frank Hutter, Manuel López-Ibáñez, Chris Fawcett, Marius Lindauer, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Aclib: A benchmark library for algorithm configuration. In *Learning and Intelligent Optimization*, pages 36–40. Springer, 2014.

[152] Jens Jägersküpper. Rigorous runtime analysis of the (1+ 1) es: 1/5-rule and ellipsoidal fitness landscapes. In *International Workshop on Foundations of Genetic Algorithms*, pages 260–281. Springer, 2005.

[153] David S Johnson and Lyle A McGeoch. The traveling salesman problem: A case study in local optimization. *Local search in combinatorial optimization*, 1:215–310, 1997.

[154] David H Jonassen. Toward a design theory of problem solving. *Educational technology research and development*, 48(4):63–85, 2000.

[155] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming–CP 2011*, pages 454–469. Springer, 2011.

[156] Jérôme Henri Kämpf and Darren Robinson. A hybrid cma-es and hde optimisation algorithm with application to solar energy potential. *Applied Soft Computing*, 9(2):738–745, 2009.

[157] Yanfei Kang, Rob J Hyndman, Kate Smith-Miles, et al. Visualising forecasting algorithm performance using time series instance spaces. Technical report, Monash University, Department of Econometrics and Business Statistics, 2016.

[158] Karthik Kannappan, Lee Spector, Moshe Sipper, Thomas Helmuth, William La Cava, Jake Wisdom, and Omri Bernstein. Analyzing a decade of human-competitive (?humie?) winners: What can we learn? In *Genetic Programming Theory and Practice XII*, pages 149–166. Springer, 2015.

[159] Elaine Kant. Understanding and automating algorithm design. *Software Engineering, IEEE Transactions on*, (11):1361–1374, 1985.

[160] Wolfgang Kantschik and Wolfgang Banzhaf. Linear-graph gp-a new gp structure. In *European Conference on Genetic Programming*, pages 83–92. Springer, 2002.

[161] Wolfgang Kantschik, Peter Dittrich, Markus Brameier, and Wolfgang Banzhaf. Empirical analysis of different levels of meta-evolution. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.

[162] Sheng Liang Kao, Tse Hsine Liao, Chih Hua Ke, Wan Chen Wang, Yen Chu Lu, Jih Ping Chi, Chieh Fu Chung, et al. Method and system for parameter configuration, June 30 2015. US Patent 9,069,581.

[163] K Katayama, H Sakamoto, and H Narihisa. The efficiency of hybrid mutation genetic algorithm for the travelling salesman problem. *Mathematical and Computer Modelling*, 31(10):197–203, 2000.

[164] S Kazarlis, John Kalomiros, Anastasios Balouktsis, and Vassilios Kalaitzis. Evolving optimal digital circuits using cartesian genetic programming with solution repair methods. In *Proceedings of the 2015 International Conference on Systems, Control, Signal Processing and Informatics (SCSI 2015), Barcelona, Spain*, pages 39–44, 2015.

[165] Gul Muhammad Khan, Julian Francis Miller, and David M Halliday. Coevolution of intelligent agents using cartesian genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 269–276. ACM, 2007.

[166] Maryam Mahsal Khan, Arbab Masood Ahmad, Gul Muhammad Khan, and Julian F Miller. Fast learning neural networks using cartesian genetic programming. *Neurocomputing*, 121:274–289, 2013.

[167] Ahmed Kheiri, Mustafa Mısır, and Ender Özcan. Ensemble move acceptance in selection hyper-heuristics. In *International Symposium on Computer and Information Sciences*, pages 21–29. Springer, 2016.

[168] Kenneth E Kinnear. Fitness landscapes and difficulty in genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 142–147. IEEE, 1994.

[169] George R Klare. Assessing readability. *Reading research quarterly*, pages 62–102, 1974.

[170] Lars Kotthoff. Algorithm selection for combinatorial search problems: A survey. *arXiv preprint arXiv:1210.7959*, 2012.

[171] John R Koza. *Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems*. Stanford University, Department of Computer Science, 1990.

[172] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[173] John R Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):251–284, 2010.

[174] John R Koza and David Andre. Evolution of iteration in genetic programming. In *Evolutionary Programming*, pages 469–478, 1996.

[175] John R Koza, Martin A Keane, Matthew J Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic programming IV: Routine human-competitive machine intelligence*, volume 5. Springer Science & Business Media, 2006.

[176] Oliver Kramer. Iterated local search. In *A Brief Introduction to Continuous Evolutionary Optimization*, pages 45–54. Springer, 2014.

[177] Natalio Krasnogor, Jim Smith, et al. A memetic algorithm with self-adaptive local search: Tsp as a case study. In *GECCO*, pages 987–994, 2000.

[178] Stuart E Lacy, Michael A Lones, Stephen L Smith, Jane E Alty, DR Jamieson, Katherine L Possin, and Norbert Schuff. Characterisation of movement disorder in parkinson's disease using evolutionary algorithms. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 1479–1486. ACM, 2013.

[179] Timothy Lai. Discovery of understandable math formulas using genetic programming.

[180] William Benjamin Langdon. *Genetic programming and data structures*. PhD thesis, University College London, 1996.

[181] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[182] Jan Larres, Mengjie Zhang, and Will N Browne. Using unrestricted loops in genetic programming for image classification. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[183] J Leitner, S Harding, A Förster, and J Schmidhuber. Mars terrain image classification using cartesian genetic programming. In *International Symposium on Artificial Intelligence, Robotics & Automation in Space*, 2012.

[184] Benjamin Lenz, Bernd Barak, Julia Mührwald, and Carolin Leicht. Virtual metrology in semiconductor manufacturing by means of predictive machine learning models. In *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, volume 2, pages 174–177. IEEE, 2013.

[185] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. Boosting as a metaphor for algorithm design. In *Principles and Practice of Constraint Programming–CP 2003*, pages 899–903. Springer, 2003.

[186] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, and Yoav Shoham. A portfolio approach to algorithm selection. In *IJCAI*, volume 1543, page 2003, 2003.

[187] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Principles and Practice of Constraint Programming-CP 2002*, pages 556–572. Springer, 2002.

[188] Lin Li, Jonathan M Garibaldi, and Natalio Krasnogor. Automated self-assembly programming paradigm: The impact of network topology. *Int. J. Intell. Syst.*, 24(7):793–817, 2009.

[189] Ling Li, Qiuhua Tang, Peng Zheng, Liping Zhang, and CA Floudas. An improved self-adaptive genetic algorithm for scheduling steel-making continuous casting production. In *Proceedings of the 6th International Asia Conference on Industrial Engineering and Management Innovation*, pages 399–410. Springer, 2016.

[190] Wenwen Li, Ender zcan, and Robert John. Multi-objective evolutionary algorithms and hyper-heuristics for wind farm layout optimisation. *Renewable Energy*, 105:473 – 482, 2017.

[191] Xiang Li. *Utilising restricted for-loops in genetic programming*. PhD thesis, School of Computer Science and Information Technology, Faculty of Applied Science, Royal Melbourne Institute of Technology, Melbourne, 2007.

[192] Xiang Li and Victor Ciesielski. An analysis of explicit loops in genetic programming. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2522–2529. IEEE, 2005.

[193] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. Program synthesis from natural language using recurrent neural networks. Technical report, Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, 2017.

[194] M Lindauer, H Hoos, and F Hutter. From sequential algorithm selection to parallel portfolio selection. In *Learning and Intelligent Optimization*, pages 1–16. Springer, 2015.

[195] Marius Lindauer, Holger Hoos, Kevin Leyton-Brown, and Torsten Schaub. Automatic construction of parallel portfolios via algorithm configuration. *Artificial Intelligence*, 2016.

[196] Marius Lindauer, Holger Hoos, Kevin Leyton-Brown, and Torsten Schaub. Automatic construction of parallel portfolios via algorithm configuration. *Artificial Intelligence*, 244:272–290, 2017.

[197] Yong Liu, Huifeng Wang, Hong Zhang, and Karsten Liber. A comprehensive support vector machine-based classification model for soil quality assessment. *Soil and Tillage Research*, 155:19–26, 2016.

[198] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search: Framework and applications. In *Handbook of Metaheuristics*, pages 363–397. Springer, 2010.

[199] Nuno Lourenço, Francisco Pereira, and Ernesto Costa. Evolving evolutionary algorithms. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 51–58. ACM, 2012.

[200] Nuno Lourenço, Francisco B Pereira, and Ernesto Costa. Sge: a structured representation for grammatical evolution. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 136–148. Springer, 2015.

[201] Nuno Lourenço, Francisco B Pereira, and Ernesto Costa. Studying the properties of structured grammatical evolution. 2015.

[202] Nuno Lourenço, Francisco Baptista Pereira, and Ernesto Costa. The importance of the learning conditions in hyper-heuristics. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1525–1532. ACM, 2013.

[203] Cristian Loyola, Mauricio Sepúlveda, Mauricio Solar, Pierre Lopez, and Victor Parada. Automatic design of algorithms for the traveling salesman problem. *Cogent Engineering*, 3(1):1255165, 2016.

[204] Hai-liang Lu, Xi-shan Wen, Lei Lan, Yun-zhu An, and Xiao-ping Li. A self-adaptive genetic algorithm to estimate ja model parameters considering minor loops. *Journal of Magnetism and Magnetic Materials*, 374:502–507, 2015.

[205] Sean Luke. *Essentials of metaheuristics*. Lulu Com, 2013.

[206] QingLian Ma, Yu-an Zhang, Kiminobu Koga, Kunihito Yamamori, Makoto Sakamoto, and Hiroshi Furutani. Stochastic analysis of onemax problem using markov chain. *Artificial Life and Robotics*, 17(3-4):395–399, 2013.

[207] Mashael Maashi, Graham Kendall, and Ender Özcan. Choice function based hyper-heuristics for multi-objective optimization. *Applied Soft Computing*, 28:312–326, 2015.

[208] Broos Maenhout and Mario Vanhoucke. An electromagnetic meta-heuristic for the nurse scheduling problem. *Journal of Heuristics*, 13(4):359–385, 2007.

[209] Andrea Maesani and Dario Floreano. Viability principles for constrained optimization using a (1+ 1)-cma-es. In *International Conference on Parallel Problem Solving from Nature*, pages 272–281. Springer, 2014.

[210] Francisco AL Manfrini, Heder S Bernardino, and Helio JC Barbosa. On heuristics for seeding the initial population of cartesian genetic programming applied to combinational logic circuits. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 105–106. ACM, 2016.

[211] Harry M Markowitz. Meanvariance analysis. In *Finance*, pages 194–198. Springer, 1989.

[212] Richard J Marshall, Mark Johnston, and Mengjie Zhang. Hyper-heuristics, grammatical evolution and the capacitated vehicle routing problem. In *Proceedings of the 2014 conference companion on Genetic and evolutionary computation companion*, pages 71–72. ACM, 2014.

[213] Matthew A Martin and Daniel R Tauritz. Hyper-heuristics: A study on increasing primitive-space. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1051–1058. ACM, 2015.

[214] Franco Mascia, Manuel López-Ibáñez, Jérémie Dubois-Lacoste, and Thomas Stützle. Grammar-based generation of stochastic local search heuristics through automatic algorithm configuration tools. *Computers & operations research*, 51:190–199, 2014.

[215] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[216] Brendan D McKay, Frédérique E Oggier, Gordon F Royle, NJA Sloane, Ian M Wanless, and Herbert S Wilf. Acyclic digraphs and eigenvalues of (0, 1)-matrices. *Journal of Integer Sequences*, 7(2):3, 2004.

[217] Eric Medvet, Fabio Daolio, and Danny Tagliapietra. Evolvability in grammatical evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO*, 2017.

[218] Andreas Meier, Mark Gonter, and Rudolf Kruse. Accelerating convergence in cartesian genetic programming by using a new genetic operator. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 981–988. ACM, 2013.

[219] H Miihlenbein and J Kindermann. The dynamics of evolution and learning-towards genetic neural networks. *Connectionism in perspective*, pages 173–197, 1989.

[220] Bradley N Miller and David L Ranum. *Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION*. Franklin, Beedle & Associates Inc., 2011.

[221] Julian F Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*, pages 1135–1142. Morgan Kaufmann Publishers Inc., 1999.

[222] Julian F Miller. *Cartesian genetic programming*. Springer, 2011.

[223] Julian F Miller, Dominic Job, and Vesselin K Vassilev. Principles in the evolutionary design of digital circuitspart i. *Genetic programming and evolvable machines*, 1(1-2):7–35, 2000.

[224] Julian F Miller, Dominic Job, and Vesselin K Vassilev. Principles in the evolutionary design of digital circuitspart ii. *Genetic programming and evolvable machines*, 1(3):259–288, 2000.

[225] Julian F Miller and Peter Thomson. Cartesian genetic programming. In *European Conference on Genetic Programming*, pages 121–132. Springer, 2000.

[226] Julian F Miller and Peter Thomson. Cartesian genetic programming. In *European Conference on Genetic Programming*, pages 121–132. Springer, 2000.

[227] Péricles BC Miranda, Ricardo BC Prudêncio, and Gisele L Pappa. H3ad: A hybrid hyper-heuristic for algorithm design. *Information Sciences*, 2017.

[228] Mustafa Mısır. Matrix factorization based benchmark set analysis: A case study on hyflex.

[229] Mustafa Misir and Michele Sebag. Algorithm selection as a collaborative filtering problem. 2013.

[230] Tom M Mitchell. *The need for biases in learning generalizations*. Department of Computer Science, Laboratory for Computer Science Research, Rutgers Univ. New Jersey, 1980.

[231] Tom M Mitchell et al. Machine learning. wcb, 1997.

[232] Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[233] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[234] Krista R Muis, Cynthia Psaradellis, Susanne P Lajoie, Ivana Di Leo, and Marianne Chevrier. The role of epistemic emotions in mathematics problem solving. *Contemporary Educational Psychology*, 42:172–185, 2015.

[235] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian sketch learning for program synthesis. *arXiv preprint arXiv:1703.05698*, 2017.

[236] Yuichi Nagata and Shigenobu Kobayashi. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. *INFORMS Journal on Computing*, 25(2):346–363, 2013.

[237] Allen Newell, John C Shaw, and Herbert A Simon. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA, 1959.

[238] Allen Newell and Herbert Simon. The logic theory machine–a complex information processing system. *IRE Transactions on information theory*, 2(3):61–79, 1956.

[239] Allen Newell and Herbert A Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

[240] Mohd Razali Noraini and John Geraghty. Genetic algorithm performance with different selection strategies in solving tsp. 2011.

[241] Nomzamo Ntombela and Nelishia Pillay. Evolving construction heuristics for the symmetric travelling salesman problem. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, page 30. ACM, 2016.

[242] Gabriela Ochoa, Matthew Hyde, Tim Curtois, Jose A Vazquez-Rodriguez, James Walker, Michel Gendreau, Graham Kendall, Barry McCollum, Andrew J Parkes, Sanja Petrovic, et al. Hyflex: A benchmark framework for cross-domain heuristic search. In *Evolutionary computation in combinatorial optimization*, pages 136–147. Springer, 2012.

[243] Gabriela Ochoa and Nadarajen Veerapen. Deconstructing the big valley search space hypothesis. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 58–73. Springer, 2016.

[244] Mihai Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.

[245] Mihai Oltean and Crina Groşan. Evolving evolutionary algorithms using multi expression programming. In *Advances in Artificial Life*, pages 651–658. Springer, 2003.

[246] Ender Özcan. Memetic algorithms for nurse rostering. In *International Symposium on Computer and Information Sciences*, pages 482–492. Springer, 2005.

[247] Ender Ozcan and Murat Erenturk. A brief review of memetic algorithms for solving euclidean 2d traveling salesrep problem. In *Proc. of the 13th Turkish Symposium on Artificial Intelligence and Neural Networks*, pages 99–108, 2004.

[248] Miriam Padberg. Harmony search algorithms for binary optimization problems. In *Operations Research Proceedings 2011*, pages 343–348. Springer, 2012.

[249] Gisele L Pappa and Alex A Freitas. *Automating the design of data mining algorithms: an evolutionary computation approach*. Springer Science & Business Media, 2009.

[250] Gisele L Pappa, Gabriela Ochoa, Matthew R Hyde, Alex A Freitas, John Woodward, and Jerry Swan. Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(1):3–35, 2014.

[251] Helena MM Patching, Laurence M Hudson, Warrick Cooke, Andres J Garcia, Simon I Hay, Mark Roberts, and Catherine L Moyes. A supervised learning process to validate online disease reports for use in predictive models. *Big data*, 3(4):230–237, 2015.

[252] Ed Pegg Jr. The icosian game, revisited. *Mathematica J*, pages 310–314, 2009.

[253] Tessa Phillips, Mengjie Zhang, and Bing Xue. Genetic programming for evolving programs with recursive structures. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 5044–5051. IEEE, 2016.

[254] Nelishia Pillay. A generative hyper-heuristic for deriving heuristics for classical artificial intelligence problems. In *Advances in Nature and Biologically Inspired Computing*, pages 337–346. Springer, 2016.

[255] Riccardo Poli. Parallel distributed genetic programming. *COGNITIVE SCIENCE RESEARCH PAPERS-UNIVERSITY OF BIRMINGHAM CSRP*, 1996.

[256] Riccardo Poli et al. Evolution of graph-like programs with parallel distributed genetic programming. In *ICGA*, pages 346–353. Citeseer, 1997.

[257] Riccardo Poli and Mario Graff. There is a free lunch for hyper-heuristics, genetic programming and computer scientists. In *European Conference on Genetic Programming*, pages 195–207. Springer, 2009.

[258] Riccardo Poli, Leonardo Vanneschi, William B Langdon, and Nicholas Freitag McPhee. Theoretical results in genetic programming: the next ten years? *Genetic Programming and Evolvable Machines*, 11(3-4):285–320, 2010.

[259] Thomas D Pollard, William C Earnshaw, and Jennifer Lippincott-Schwartz. *Cell biology*. Elsevier Health Sciences, 2007.

[260] V William Porto and David B Fogel. Evolving artificial neural networks. *Clinical applications of artificial neural networks*, page 223, 2001.

[261] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*, pages 73–82. ACM, 2011.

[262] Marcela Quiroz, Laura Cruz-Reyes, José Torres-Jiménez, Patricia Melin, et al. Improving the performance of heuristic algorithms based on exploratory data analysis. In *Recent Advances on Hybrid Intelligent Systems*, pages 361–375. Springer, 2013.

[263] Abdur Rais and Ana Viana. Operations research in healthcare: a survey. *International transactions in operational research*, 18(1):1–31, 2011.

[264] Gerhard Reinelt. Tsplib95. *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg*, 1995.

[265] Z. Ren, H. Jiang, J. Xuan, Y. Hu, and Z. Luo. New insights into diversification of hyper-heuristics. *IEEE Transactions on Cybernetics*, 44(10):1747–1761, Oct 2014.

[266] Joel Ribeiro, Josep Carmona, Mustafa Mısır, and Michele Sebag. A recommender system for process discovery. In *Business Process Management*, pages 67–83. Springer, 2014.

[267] John R Rice. The algorithm selection problem. 1975.

[268] Robert W Robinson. Counting unlabeled acyclic digraphs. In *Combinatorial mathematics V*, pages 28–43. Springer, 1977.

[269] Raymond Ros and Nikolaus Hansen. A simple modification in cma-es achieving linear time and space complexity. In *International Conference on Parallel Problem Solving from Nature*, pages 296–305. Springer, 2008.

[270] Peter Ross. Hyper-heuristics. In *Search Methodologies*, pages 611–638. Springer, 2014.

[271] Andrew Runka. Evolving an edge selection formula for ant colony optimization. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1075–1082. ACM, 2009.

[272] Stuart Jonathan Russell, Peter Norvig, John F Canny, Jitendra M Malik, and Douglas D Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River, 2003.

[273] Patricia Ryser-Welch, Julian F Miller, and Shahriar Asta. Generating human-readable algorithms for the travelling salesman problem using hyper-heuristics. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1067–1074. ACM, 2015.

[274] Patricia Ryser-Welch, Julian F Miller, Jerry Swan, and Martin A Trefzer. Iterative cartesian genetic programming: Creating general algorithms for solving travelling salesman problems. In *European Conference on Genetic Programming*, pages 294–310. Springer, 2016.

[275] Nasser R Sabar, Masri Ayob, Graham Kendall, and Rong Qu. Automatic design of a hyper-heuristic framework with gene expression programming for combinatorial optimization problems. *IEEE Transactions on Evolutionary Computation*, 19(3):309–325, 2015.

[276] Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. Snappy: A simple algorithm portfolio. In *Theory and Applications of Satisfiability Testing–SAT 2013*, pages 422–428. Springer, 2013.

[277] Alexander Schrijver. On the history of combinatorial optimization (till 1960). *Handbooks in operations research and management science*, 12:1–68, 2005.

[278] Robert Sedgewick and Philippe Flajolet. *An introduction to the analysis of algorithms*. Addison-Wesley, 2013.

[279] Lukas Sekanina. *Evolvable components: from theory to hardware implementations*. Springer Science & Business Media, 2012.

[280] Lukas Sekanina and Vlastimil Kapusta. Visualisation and analysis of genetic records produced by cartesian genetic programming. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1411–1418. ACM, 2016.

[281] PM Sheppard. The evolution of mimicry; a problem in ecology and genetics. In *Cold Spring Harbor Symposia on Quantitative Biology*, volume 24, pages 131–140. Cold Spring Harbor Laboratory Press, 1959.

[282] Shinichi Shirakawa and Tomoharu Nagao. Evolution of sorting algorithm using graph structured program evolution. In *2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 1256–1261. IEEE, 2007.

[283] Shinichi Shirakawa and Tomoharu Nagao. Graph structured program evolution with automatically defined nodes. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1107–1114. ACM, 2009.

[284] Shinichi Shirakawa and Tomoharu Nagao. Graph structured program evolution: Evolution of loop structures. In *Genetic Programming Theory and Practice VII*, pages 177–194. Springer, 2010.

[285] Shinichi Shirakawa, Shintaro Ogino, and Tomoharu Nagao. Graph structured program evolution. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1686–1693. ACM, 2007.

[286] Kevin Sim and Emma Hart. A combined generative and selective hyper-heuristic for the vehicle routing problem. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 1093–1100, New York, NY, USA, 2016. ACM.

[287] Kevin Sim and Emma Hart. A combined generative and selective hyper-heuristic for the vehicle routing problem. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 1093–1100. ACM, 2016.

[288] Herbert A Simon and Allen Newell. Human problem solving: The state of the theory in 1970. *American Psychologist*, 26(2):145, 1971.

[289] Bryan Singer and Manuela Veloso. Learning to predict performance from formula modeling and training data. In *ICML*, pages 887–894, 2000.

[290] Selmar K Smit and AE Eiben. Parameter tuning of evolutionary algorithms: Generalist vs. specialist. In *European Conference on the Applications of Evolutionary Computation*, pages 542–551. Springer, 2010.

[291] Selmar K Smit and Agoston E Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *2009 IEEE congress on evolutionary computation*, pages 399–406. IEEE, 2009.

[292] Mike U Smith. *Toward a unified theory of problem solving: Views from the content domains*. Routledge, 2012.

[293] Stephen L Smith and Michael A Lones. Implicit context representation cartesian genetic programming for the assessment of visuo-spatial ability. In *2009 IEEE Congress on Evolutionary Computation*, pages 1072–1078. IEEE, 2009.

[294] Kate Smith-Miles and Jano van Hemert. Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence*, 61(2):87–104, 2011.

[295] Kate Smith-Miles, Brendan Wreford, Leo Lopes, and Nur Insani. Predicting metaheuristic performance on graph coloring problems using data mining. In *Hybrid Metaheuristics*, pages 417–432. Springer, 2013.

[296] Kate A. Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1):6:1–6:25, January 2009.

[297] Kate A Smith-Miles, Ross JW James, John W Giffin, and Yiqing Tu. A knowledge discovery approach to understanding relationships between scheduling problem structure and heuristic performance. In *Learning and intelligent optimization*, pages 89–103. Springer, 2009.

[298] Kenneth Sörensen. Metaheuristicsthe metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18, 2015.

[299] Kenneth Sorensen, Marc Sevaux, and Fred Glover. A history of metaheuristics. *arXiv preprint arXiv:1704.00853*, 2017.

[300] Jorge A Soria-Alcaraz, Gabriela Ochoa, Marco A Sotelo-Figeroa, and Edmund K Burke. A methodology for determining an effective subset of heuristics in selection hyper-heuristics. *European Journal of Operational Research*, 260(3):972–983, 2017.

[301] William M Spears et al. Crossover or mutation. *Foundations of genetic algorithms*, 2:221–237, 1992.

[302] Lee Spector. Autoconstructive evolution: Push, pushgp, and pushpop. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, volume 137, 2001.

[303] Lee Spector, Nicholas Freitag McPhee, Thomas Helmuth, Maggie M Casale, and Julian Oks. Evolution evolves with autoconstruction. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1349–1356. ACM, 2016.

[304] Susan Stepney. Statistics with confidence. *ionosphere*, 93:91–67, 2009.

[305] Jerry Swan and Nathan Burles. Templar–a framework for template-method hyper-heuristics. In *Genetic Programming*, pages 205–216. Springer, 2015.

[306] Jerry Swan, Ender Özcan, and Graham Kendall. Hyperion–a recursive hyper-heuristic framework. In *Learning and intelligent optimization*, pages 616–630. Springer, 2011.

[307] Jerry Swan, John Woodward, Ender Özcan, Graham Kendall, and Edmund Burke. Searching the hyper-heuristic design space. *Cognitive Computation*, 6(1):66–73, 2014.

[308] Jorge Tavares, Penousal Machado, Amilcar Cardoso, Francisco B Pereira, and Ernesto Costa. On the evolution of evolutionary algorithms. In *Genetic Programming*, pages 389–398. Springer, 2004.

[309] Astro Teller and Manuela Veloso. Pado: Learning tree structured algorithms for orchestration into an object recognition system. Technical report, DTIC Document, 1995.

[310] Astro Teller and Manuela Veloso. Pado: A new learning architecture for object recognition. *Symbolic visual learning*, pages 77–112, 1997.

[311] Germán Terrazas, Marian Gheorghe, Graham Kendall, and Natalio Krasnogor. Evolving tiles for automated self-assembly design. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 2001–2008. IEEE, 2007.

[312] Germán Terrazas, Hector Zenil, and Natalio Krasnogor. Exploring programmable self-assembly in non-dna based molecular computing. *Natural Computing*, 12(4):499–515, 2013.

[313] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in stapl. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 277–288. ACM, 2005.

[314] Yanfei Tian, Liwen Huang, and Yong Xiong. A general technical route for parameter optimization of ship motion controller based on artificial bee colony algorithm. *International Journal of Engineering and Technology*, 9(2):133, 2017.

[315] Murchhana Tripathy and Anita Panda. A study of algorithm selection in data mining using meta-learning. *Journal of Engineering Science and Technology Review*, 10(2):51–64, 2017.

[316] Yuri R Tsoy. The influence of population size and search time limit on genetic algorithm. In *Science and Technology, 2003. Proceedings KORUS 2003. The 7th Korea-Russia International Symposium on*, volume 3, pages 181–187. IEEE, 2003.

[317] Andrew James Turner and Julian Francis Miller. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1005–1012. ACM, 2013.

[318] Andrew James Turner and Julian Francis Miller. Neuroevolution: Evolving heterogeneous artificial neural networks. *Evolutionary Intelligence*, 7(3):135–154, 2014.

[319] NLJ Ulder, E Pesch, PJM van Laarhoven, HJ Bandelt, and EHL Aarts. Improving tsp exchange heuristics by population genetics. *Parallel Problem Solving from Nature*, pages 109–116, 1991.

[320] unknown. Netflix prizes, 2015.

[321] Enrique Urra, Daniel Cabrera-Paniagua, and Claudio Cubillos. Towards an object-oriented pattern proposal for heuristic structures of diverse abstraction levels. *XXI Jornadas Chilenas de Computación*, 2013.

[322] Christos Valouxis, Christos Gogos, George Goulas, Panayiotis Alefragis, and Efthymios Housos. A systematic two phase approach for the nurse rostering problem. *European Journal of Operational Research*, 219(2):425–433, 2012.

[323] Jorne Van den Bergh, Jeroen Beliën, Philippe De Bruecker, Erik Demeulemeester, and Liesje De Boeck. Personnel scheduling: A literature review. *European Journal of Operational Research*, 226(3):367–385, 2013.

[324] Willem Van Onsem and Bart Demoen. Parhyflex: A framework for parallel hyper-heuristics. In *BNAIC 2013: Proceedings of the 25th Benelux Conference on Artificial Intelligence, Delft, The Netherlands, November 7-8, 2013*. Delft University of Technology (TU Delft); under the auspices of the Benelux Association for Artificial Intelligence (BNVKI) and the Dutch Research School for Information and Knowledge Systems (SIKS), 2013.

[325] Sander van Rijn, Hao Wang, Matthijs van Leeuwen, and Thomas Bäck. Evolving the structure of evolution strategies. *arXiv preprint arXiv:1610.05231*, 2016.

[326] Greet Vanden Berghe. An advanced model and novel meta-heuristic solution methods to personnel scheduling in healthcare. 2002.

[327] Zdenek Vasicek and Lukas Sekanina. Hardware accelerators for cartesian genetic programming. In *European Conference on Genetic Programming*, pages 230–241. Springer, 2008.

[328] Zdeněk Vašíček and Karel Slanỳ. Efficient phenotype evaluation in cartesian genetic programming. In *European Conference on Genetic Programming*, pages 266–278. Springer, 2012.

[329] Vesselin K Vassilev and Julian F Miller. The advantages of landscape neutrality in digital circuit evolution. In *International Conference on Evolvable Systems*, pages 252–263. Springer, 2000.

[330] Alan Vella et al. *Hyper-heuristic decision tree induction*. PhD thesis, Heriot-Watt University, 2012.

[331] Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

[332] James Alfred Walker and Julian Francis Miller. Evolution and acquisition of modules in cartesian genetic programming. In *Genetic Programming*, pages 187–197. Springer, 2004.

[333] James Alfred Walker and Julian Francis Miller. Investigating the performance of module acquisition in cartesian genetic programming. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1649–1656. ACM, 2005.

[334] James Alfred Walker and Julian Francis Miller. Embedded cartesian genetic programming and the lawnmower and hierarchical-if-and-only-if problems. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 911–918. ACM, 2006.

[335] James Alfred Walker and Julian Francis Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, 2008.

[336] James Alfred Walker, Julian Francis Miller, and Rachel Cavill. A multi-chromosome approach to standard and embedded cartesian genetic programming. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 903–910. ACM, 2006.

[337] James Alfred Walker, Katharina Völk, Stephen L Smith, and Julian Francis Miller. Parallel evolution using multi-chromosome cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 10(4):417–445, 2009.

[338] Yingxu Wang and Vincent Chiew. On the cognitive process of human problem solving. *Cognitive Systems Research*, 11(1):81–92, 2010.

[339] Yingxu Wang, Ying Wang, S. Patel, and D. Patel. A layered reference model of the brain (lrmb). *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(2):124–133, March 2006.

[340] D Michael Warner and Juan Prawda. A mathematical programming model for scheduling nursing personnel in a hospital. *Management Science*, 19(4-part-1):411–422, 1972.

[341] Karsten Weicker and Nicole Weicker. On evolution strategy optimization in dynamic environments. In *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, volume 3. IEEE, 1999.

[342] Robert W Weisberg. Toward an integrated theory of insight in problem solving. *Thinking & Reasoning*, 21(1):5–39, 2015.

[343] Gayan Wijesinghe and Vic Ciesielski. Parameterised indexed for-loops in genetic programming and regular binary pattern strings. In *Asia-Pacific Conference on Simulated Evolution and Learning*, pages 524–533. Springer, 2008.

[344] Gayan Wijesinghe and Vic Ciesielski. Evolving programs with parameters and loops. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

[345] David H Wolpert. What the no free lunch theorems really mean; how to improve search algorithms.

[346] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[347] David Harlan Wood, Junghuei Chen, Eugene Antipov, Bertrand Lemieux, and Walter Cedeño. A design for dna computation of the onemax problem. *Soft Computing*, 5(1):19–24, 2001.

[348] John R. Woodward and Jerry Swan. The automatic generation of mutation operators for genetic algorithms. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 67–74, New York, NY, USA, 2012. ACM.

[349] John Robert Woodward and Jerry Swan. Automatically designing selection heuristics. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 583–590. ACM, 2011.

[350] John Robert Woodward and Jerry Swan. Automatically designing selection heuristics. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '11, pages 583–590, New York, NY, USA, 2011. ACM.

[351] Sewall Wright. *The roles of mutation, inbreeding, crossbreeding, and selection in evolution*, volume 1. na, 1932.

[352] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of Artificial Intelligence Research*, pages 565–606, 2008.

[353] Mutsunori Yagiura, Toshihide Ibaraki, and Fred Glover. An ejection chain approach for the generalized assignment problem. *INFORMS Journal on Computing*, 16(2):133–151, 2004.

[354] Fan Yang, Zhilin Yang, and William W Cohen. Differentiable learning of logical rules for knowledge base completion. *arXiv preprint arXiv:1702.08367*, 2017.

[355] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[356] W Yates and EC Keedwell. Clustering of hyper-heuristic selections using the smith-waterman algorithm for off line learning. GECCO, 2017.

[357] Jianbo Yu, Shijin Wang, and Lifeng Xi. Evolving artificial neural networks using an improved pso and dpso. *Neurocomputing*, 71(4):1054–1060, 2008.

[358] Tina Yu and Chris Clark. Recursion, lambda-abstractions and genetic programming. *Cognitive Science Research Papers-University Of Birmingham CSRP*, pages 26–30, 1998.

[359] Shiu Yin Yuen, Chi Kin Chow, and Xin Zhang. Which algorithm should i choose at any point of the search: an evolutionary portfolio approach. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 567–574. ACM, 2013.

[360] Shiu Yin Yuen and Xin Zhang. On composing an (evolutionary) algorithm portfolio. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*, pages 83–84. ACM, 2013.

[361] Shiu Yin Yuen and Xin Zhang. On composing an algorithm portfolio. *Memetic Computing*, 7(3):203–214, 2015.