An Investigation into the Suitability of Genetic Programming for Computing Visibility Areas for Sensor Planning

by

Michael Sean Grant, M.A. Hons (Cantab), M.Sc.

Thesis submitted in candidature
for the degree of Doctor of Philosophy

Heriot-Watt University
Department of Computing and Electrical Engineering
May 2000

Contents

List of Figures iv
List of Tables viii
List of Programs x

1 Introduction **1**
1.1 Meta-introduction 1
1.2 Genetic Programming 1
1.3 Computer Vision and Sensor Planning 2
1.4 Objectives 3
1.5 Motivation 3
1.6 Structure of this thesis 4
1.7 Contributions 5

2 Literature Survey **6**
2.1 Introduction 6
2.2 Genetic Programming 7
2.2.1 Introduction to Evolutionary Computation 7
2.2.2 Evolutionary Computation and Biology 10
2.2.3 Genetic Programming as an EA 11
2.2.4 Representation In GP 12
2.2.5 Program Creation 16
2.2.6 Program Execution in GP 16
2.2.7 Fitness Evaluation in GP 20
2.2.8 Selection in GP 22
2.2.9 Genetic Operators 23
2.2.10 Turnover in GP 26
2.2.11 Relative Utility of Genetic Operators 29
2.2.12 A Simple Example 31
2.2.13 Applications of Genetic Programming 36
2.2.14 Summary 42
2.3 Sensor Planning 42
2.3.1 Introduction to Sensor Planning 42
2.3.2 Visibility Space Analysis 46
2.3.3 Summary 53
2.4 Conclusions 54

3 The Untyped System **55**
3.1 Introduction . . . 55
3.2 Problem Specification . . . 55
3.3 Overview of the Untyped System . . . 57
3.4 System Specifications . . . 59
3.4.1 Model Representation . . . 60
3.4.2 The Genetic Virtual Machine . . . 62
3.5 Fitness Cases . . . 78
3.6 Fitness Function . . . 80
3.6.1 Fitness By Vector Difference in Arithmetic Parameter Space . . . 82
3.6.2 Fitness By Vector Difference in Geometric Parameter Space . . . 84
3.7 Experiments and Results . . . 85
3.8 Conclusions . . . 88

4 The First Typed System **90**
4.1 Introduction . . . 90
4.2 Rationale . . . 91
4.2.1 Problems with an Untyped System . . . 91
4.2.2 Strongly Typed GP . . . 92
4.2.3 Limitations of Flat Typing . . . 94
4.2.4 Other Consequences of Flat Typing . . . 97
4.3 Implementing The Typed System: A Nodes Possibilities Table . . . 97
4.4 System Specifications . . . 104
4.4.1 Genetic Machine . . . 104
4.4.2 Fitness Function and Fitness Cases . . . 117
4.4.3 Seeding the Population . . . 118
4.5 Complexity Analysis . . . 121
4.6 Implementational Issues . . . 124
4.6.1 Redesign of the Function and Terminal Sets . . . 124
4.6.2 Evaluation Caching . . . 125
4.7 Experiments and Results . . . 127
4.7.1 Altering the Fitness Function . . . 131
4.7.2 Altering the Fitness Cases . . . 133
4.7.3 Parameter Tuning . . . 136
4.7.4 Altering the Function Set . . . 143
4.7.5 Dynamic Specifications . . . 148
4.7.6 Templates . . . 154
4.8 A Simpler System to Model Evolution . . . 160
4.8.1 Rationale . . . 160
4.8.2 System specifications . . . 162
4.8.3 Experiments and Results . . . 164
4.9 Conclusions . . . 170

5 The Second Typed System **173**
5.1 Introduction . . . 173
5.2 Rationale . . . 173
5.2.1 Objectives of the New System . . . 173
5.2.2 Approaches Considered . . . 174
5.3 System Specifications . . . 178
5.3.1 Fitness Function . . . 189
5.3.2 Seeded Programs . . . 191
5.3.3 Implementational Issues . . . 195
5.3.4 Graphical User Interface . . . 197
5.4 Results . . . 202
5.4.1 Unseeded experiments . . . 203
5.4.2 Seeded experiments . . . 208
5.4.3 Variations in GP . . . 222
5.5 Conclusions . . . 231

6 Discussion and Conclusions **234**
6.1 Review . . . 234
6.2 Future Work . . . 238
6.3 Conclusions . . . 239

A Algorithms **255**
A.1 Nodes Possibilities Table . . . 255
A.2 Seeded programs . . . 260
A.2.1 The Untyped System . . . 260
A.2.2 The First Typed System . . . 262
A.2.3 The Second Typed System . . . 267
A.3 Evaluation Caching . . . 272
A.3.1 Sorting and Aligning Halfplanes . . . 275

List of Figures

2.1 The mechanics of the genetic operations in Evolutionary Computation. 7
2.2 Summary of the Evolutionary Computation paradigm. 8
2.3 Evolutionary exploration of a fitness landscape. 9
2.4 Representation of genetic programs by parse trees. 13
2.5 Structure of graph-based genetic programs. 15
2.6 Full and Grow creation types. 16
2.7 The *compress* operator in Module Acquisition. 19
2.8 Roulette-wheel selection. 22
2.9 Crossover for tree-based GP. 24
2.10 Mutation for tree-based GP. 26
2.11 Evolutionary equivalence of population size and number of generations 27
2.12 Sample evolved program from the example system. 34
2.13 Graph of evolution in a run of the example system. 34
2.14 Default hierarchies. 41
2.15 A typical machine vision setup. 43
2.16 Three stages in the tessellation of a sphere 49
2.17 Example object with visibility space shown for a feature on the upper surface. . . 50
2.18 Illustration of the resolution, field-of-view and depth-of-field constraints for a square, and combined admissibility space for all of them. 51
2.19 Sensor parameters. 52

3.1 Correct solution to the 2D polygons visibility problem. 56
3.4 Architecture of the genetic machine in the untyped system. 64
3.5 Hierarchy of ADFs in the untyped system. 65
3.6 Relationship between elevation of a point above the boundary of a halfplane, as measured by evaluation of the inequality defining the halfplane, and by geometric angles. 78
3.7 Fitness cases for the untyped system. 79
3.8 How improvement in fitness can compensate for length penalisation. 81
3.9 Arithmetic alignment of sorted halfplanes. 83
3.10 Definition of the geometric representation of halfplanes. 84
3.11 Typical result of the untyped system. 87

4.1 Illustration of codon slippage errors. 91
4.2 Probabilities of selecting a particular variable during program creation using lvalued symbols and using indexed memory. 96

4.3 Rebinding of `start`, `prev` and `next` in the clockwise and anticlockwise iterations. 109
4.4 Illustration of the operation of `halfplane` and `elevation`. 111
4.5 On distinguishing x° from $-(360-x)^\circ$. 115
4.6 Original fitness cases for the first typed system. 118
4.7 Fitnesses of the seeds with the original fitness measure. 121
4.8 Sample early run of the first typed system. 128
4.9 Execution of Program 4.6 using the original fitness cases for the first typed system contrasted with that of the correct solution. 129
4.10 Fitness of the static solutions compared to that of the seeds. 130
4.11 Fitness measures for the first typed system. 131
4.12 Comparison of fitness measures. 132
4.13 Execution of the programs in Figure 4.15b/4.14b. 134
4.14 Fitness of the seeds and static solutions with the original fitness cases and with the revised fitness cases, using the fitness base $100 \times \frac{F \cup G - F \cap G}{F \cap G}$. 135
4.15 Fitness of the seeds and static solutions with the original fitness cases and with the revised fitness cases, using the fitness base $100 \times \frac{F \cup G - F \cap G}{F \cup G}$. 135
4.16 Execution of seed 1, the program on p. 136 and seed 3, using the original fitness cases. 136
4.17 More difficult fitness cases for the first typed system. 137
4.18 Evolution with the fitness cases shown in Figure 4.17a. 137
4.19 Evolution with the old fitness cases, before and after the removal of `prev` from the function set; also, runs with successively decreased crossover and increased mutation rates. 139
4.20 Fitness graphs with the raw fitness squared. 140
4.21 Fitness graphs with the revised fitness cases with an extra penalty added for incorrect solutions. 141
4.22 Fitness graphs with an extra reward for correct solutions. 142
4.23 Seed fitness graphs for the fitness evaluation parameters given in Table 4.17. . . . 145
4.24 Effect of removing `next` from the function set. 146
4.25 Illustration of the impossibility of predicting, whilst traversing around a model, whether the point subtending the highest elevation has yet been reached. 147
4.26 The two alternate sets of fitness cases used for dynamic fitness. 150
4.27 Fitness of the seeds and static solutions with the dynamically switching fitness cases. 150
4.28 Simple fitness cases for the single dynamically switched fitness case system. . . . 150
4.29 Fitness of the seeds and static solutions with two simple dynamically-switched fitness cases. 151
4.30 Evolution with dynamically altered node sets. 153
4.31 Execution of product of evolution with function and terminal set reduced to that necessary for finding seed 7. 155
4.32 Evolution with dynamically altered node sets. 155
4.33 Execution of best program of run in Figure 4.32. 156
4.34 Comparison of the use of maximum and minimum elevations in constructing an answer halfplane. 156
4.35 An unusual evolved result. 156
4.36 Execution of result of evolution with a mostly frozen template. 157

4.37 Discovery of a seed 2 hidden inside a seed 6. 159
4.38 Clonal analysis of the array average problem. 165
4.39 Clonal analysis of a successful solution of the average-of-array problem. 168
4.40 Clonal analysis of the array average problem with extraneous terminals. 169
4.41 Clonal analysis of the sensor planning problem. 169
4.42 A fitness landscape which GP is poorly equipped to explore. 172

5.1 Example points distribution for the first fitness measure considered for the new system. 176
5.2 One iteration in execution in the second fitness measure considered for the new system. 176
5.3 Architecture of the second typed system. 179
5.4 Illustration of the semantics of +=. 184
5.5 Fitness cases for the second typed system. 191
5.6 Format of diagrams used in this chapter. 193
5.7 Execution of seeds for the second typed system. 194
5.7 (continued) Execution of seed 1a 195
5.8 Graphical interface developed for the second typed system. 198
5.8 (continued) 199
5.9 Increase of amplitude of oscillations of the average fitness of the population. . . . 203
5.10 A run using unseeded evolution. 204
5.11 Execution of evolved programs from the unseeded run in Figure 5.10. 204
5.12 A run using unseeded evolution. 205
5.13 Execution of best evolved program from the unseeded run shown in Figure 5.12. 207
5.14 Performance comparison of the best evolved program from unseeded runs with seed 1 on unseen input. 208
5.15 A run using evolution from seed 1. 209
5.16 Close-up of the beginning of Figure 5.15 above, with seed 1's average fitness indicated. 209
5.17 Performance comparison from the run illustrated in Figure 5.15, of seed 1, which was used to seed the run, and the best evolved program. 210
5.18 A run using evolution from seed 1. 210
5.19 A run using evolution from seed 1a. 213
5.20 A run using evolution from seed 2. 214
5.21 A run using evolution from seed 2. 214
5.22 Performance comparison from the run illustrated in Figure 5.21, of the best evolved program and seed 2 (which was used to seed the run). 216
5.23 A run using evolution from seed 3. 217
5.24 A run using evolution from seed 3. 218
5.25 Best individual of generation 272 from the run using evolution from seed 3 shown in Figure 5.24. 218
5.26 A run using evolution from seed 3a. 220
5.27 A run using evolution from mixed seeds. 221
5.28 Best individual of generation 4 from the run using mixed seeds shown in Figure 5.27. 222
5.29 Stages in the execution of Program 5.8. 222
5.30 Typical result of evolution without crossover. 223

5.31 Run displaying an unusual collapse of genetic diversity. 225
5.32 An unusual style of execution. 226
5.33 Untyped evolution seeded from seed 1a. 228
5.34 Summary of the best results of the second typed system. 232

6.1 Illustration of a disjoint visibility space in three dimensions. 239

List of Tables

2.1 Fitness cases for the example system. 32
2.2 Tableau for the example system 33
2.3 Fitness calculation of execution of the program in Figure 2.12. 34
2.4 Results of evolution of the example system. 35

3.1 Definition of the untyped system's genetic machine in Backus Naur Form. . . . 66
3.2 Functions and terminals used in the untyped system's genetic machine 67
3.3 Contents of state memories at the end of execution of Program 3.1. 75
3.4 Tableau for the untyped system 86

4.1 Simple genetic machine for illustrating typing. 92
4.2 Table illustrating how typing can cut down search spaces. 93
4.3 Illustration of the use of hierarchical typing to implement lvalues and rvalues. . . 95
4.4 Specification of a simple Roman numeral system. 98
4.5 Types possibilities table for the Roman numeral system. 99
4.6 Use of a types possibilities table. 99
4.7 Nodes possibilities table for the Roman numeral system. 103
4.8 Architecture of the genetic machine of the first typed system. 105
4.9 Definition of the first typed system's genetic machine in Backus Naur Form. . . . 108
4.10 Functions and terminals in the first typed system, listed by functionality. 110
4.11 Description of the seeds for the first typed system. 120
4.12 Complexity analysis of programs of INDIFFERENT type root up to the maximum depth for crossover 123
4.13 Comparison of the complexities of different problems tackled by GP 123
4.14 Tableau for the first typed system 126
4.15 Fitnesses of hand-crafted and evolved programs. 129
4.16 Configurations of genetic parameters investigated. 138
4.17 Alterations made to the fitness evaluation parameters. 144
4.18 Genetic machine for the array manipulation problems. 162
4.19 Tableau for the array problems. 163
4.20 Table showing size of the program tree generation space for the array manipulation system. 165
4.21 Nodes Possibilities Table for the array problems system. 165
4.22 Nodes Possibilities Table for the Visibility Space Problem. 166
4.23 Nodes usage tables for the array problems system, in a population of 1000. . . . 166

4.24 Nodes usage table for the array problems system for trees only of depth three or greater. 168

5.1 Definition of the second typed system in Backus Naur Form. 181
5.2 Definition of the second typed system's genetic machine. 182
5.3 Description of the seeds for the second typed system. 192
5.4 Fitnesses of the hand-crafted programs for the second typed system. 193
5.5 Tableau for the second typed system 197
5.6 Comparison of best-of-generation individuals from generations 15 and 21 (seed 6 equivalent) from Figure 5.12. 207
5.7 Comparative performance of runs seeded with seeds 1 and 1a. 213
5.8 Results of evolution without crossover. 224
5.9 Summary of the best results of the second typed system. 232

List of Programs

3.1 Sample evolved program solving the visibility space problem for convex polygons. 73
3.2 Hand-crafted solution for the untyped system. 76
3.2 (continued) 77
3.3 Best of run individual from run shown in Figure 3.11. 87
4.1 Example program from the untyped system. (repeated) 100
4.2 Algorithm for construction of a nodes possibilities table. 102
4.3 Framework for evolution in the first typed system. 107
4.4 Hand-crafted perfect solution for the first typed system. 114
4.5 Original conception of model solution before the implementation of the system. . 121
4.6 Result of an early run. 128
4.7 Hand-crafted partial solution for the first typed system. 152
4.8 Result of evolution with function and terminal set reduced to that necessary for finding seed 6. 153
4.9 Best program of the run with dynamically altered node sets shown in Figure 4.32. 155
4.10 An unusual evolved result. 156
4.11 Attempt to evolve a correct solution from a mostly frozen template. 158
4.12 Result of a run seeded with seed 2 in which correct use of memory locations led to incremental freezing of the program. 161
4.13 Correct solution to the largest-in-an-array problem. 163
4.14 Programs from the average-of-an-array run shown in Figure 4.39. 168
5.1 Hand-crafted solution for the second typed system. 187
5.2 Best individual of generation 21 from the unseeded run shown in Figure 5.12. . . 205
5.3 Best of run individual from the unseeded run shown in Figure 5.12. 206
5.4 Best of run individual from the unseeded run shown in Figure 5.12, edited for human readability. 206
5.5 Best of run individual from the run seeded with seed 1 shown in Figure 5.12. . . 212
5.6 Best evolved program from the run using seed 2 shown in Figure 5.21. 215
5.7 Best individual of generation 272 from the run using evolution from seed 3 shown in Figure 5.24. 219
5.8 Best individual of generation 4 from the run using mixed seeds shown in Figure 5.27. 221
5.9 A program with an unusual style of execution (See Figure 5.32.) 227
5.10 Best evolved program of unseeded runs with no typing. 228
5.11 Best evolved program from an untyped run seeded with seed 1. 229
5.12 Best evolved program from an untyped run seeded with seed 1a. 230

Acknowledgements

I would like to thank here my supervisors Emanuele Trucco and Greg Michaelson for all the help they have given me over the last three and a half years. I would also like to thank my parents for giving me a first-rate secondary education and supporting me as an undergraduate, without either of which I would not have got as far as embarking upon a doctoral degree.

I would like to thank the Engineering and Physical Sciences Research Council for remunerating the year of funding I had previously used up doing my M.Sc. Without this third year, I would not have had the financial resources to maintain my living costs until the end of my Ph.D.

And finally, I would like to thank my mother and Eric Volk, for selling me respectively her car at a price I could afford, and his mountain bike, thus between them making a long daily commute bearable.

Abstract

This thesis considers the application of Genetic Programming to visibility space calculation, for Sensor Planning in Machine Vision. This is a problem considerably more complex than most for which GP has been used; no closed-form algorithm for it yet exists in the most general case.

The main contributions and results are the application of GP to a new field, and the conclusion that GP is better suited to solve this complex problem by a generate-and-test approach than an analytic one.

Three systems were implemented to evolve programs for calculating visibility spaces. The first used untyped GP and low-level operations, for maximum flexibility in evolution, but could solve the problem only for trivial cases.

The second used high-level geometric operations and typed GP, but tended to get trapped in local optima. Approaches used, unsuccessfully, to obviate this included altering the fitness cases and function set both statically and dynamically, parameter tuning, seeding the population, using program templates, and using a simpler system for modelling evolution.

The third system, which used a generate-and-test approach, evolved useful solutions. When seeded with hand-crafted partial solutions, it was able to improve them considerably.

The work shows the potential of GP to evolve or refine a region-growing generate-and-test algorithm for calculating visibility spaces, a problem not hitherto approached by the GP community.

"On the sixth day, G-d created the platypus. And G-d said: let's see the evolutionists try and figure this one out."

Chapter 1

Introduction

1.1 Meta-introduction

This thesis describes a study into the suitability of Genetic Programming for computing visibility spaces, for offline sensor planning in the field of Machine Vision. These fields are described briefly below, following which the problem tackled in this study is described in more detail, and a synopsis of the results achieved given. The structure of the thesis is then outlined. The chapter concludes by summarising the results and contributions that this work has achieved.

1.2 Genetic Programming

Since the 1950s, research has been undertaken with the aim of getting computers to program themselves. As the volume of code that is generated rises, the ability of human programmers to write it is stretched ever further. Ideally it would be possible to tell a computer *what* to do to solve a task, rather than *how* to do it, with the computer capable of determining the precise mechanics of carrying out this task on its own.

Various approaches in the field of Artificial Intelligence have been used in tackling this problem; the term Genetic Programming[83,84] (GP) defines one that is based on biological evolution. In GP, a computer constructs programs to solve a prespecified task, with no need for human input after the initial problem specification.

The field is an offshoot of that of Genetic Algorithms (GAs). In GAs the genetic operators of *reproduction*, *recombination* and *mutation* are applied to a randomly generated population of chromosomes, which encode potential solutions to the problem in hand. In GP the chromosomes take the form of programs; generally parse trees, though linear,[120] graph-based[152] and indirectly encoded programs[136] have also been used. These programs are selected for reproduction and recombination according to their *fitness* at solving the task in hand; over the course of many generations this selection results in the programs becoming better at solving this task, until eventually a perfect solution may be found.

As an AI method, GP may be characterised as follows:

- It is a "weak" method: It does not require the incorporation of domain-dependent knowledge about the task in hand, other than the low-level choice of programming primitives to be used.
- It is a symbolic method: Unlike neural nets, its output takes the form of symbolic algorithms which can then be analysed by humans.
- It is a beam search algorithm, which operates on a population of trial solutions simultaneously.
- It is a new field, the limitations of which have yet to be fully explored.

1.3 Computer Vision and Sensor Planning

Machine Vision describes the process by which a computer is able to analyse images and interpret what is described by them.[17,70,76,156] The image may be obtained by a number of different types of sensor; two of the most widely used are video camera and laser-stripes.[22,50,51] Image interpretation as described here subsumes a variety of specific tasks, such as object recognition, shape reconstruction, motion reconstruction and object location.

In many tasks, for the image to be of the maximum utility, the sensors must be configured to guarantee optimal sensing conditions. Achieving this configuration is the subject of the field

of Sensor Planning.[148] Traditionally this task has been done by hand, in an iterative, labour-intensive manner. This may even result in the cost of configuring the system for just one set-up exceeding that of the rest of the system put together.[148] The goal of Sensor Planning is to automate this process.

The most basic task tackled by Sensor Planning is that of calculating *visibility spaces*: ensuring that a direct line-of-sight exists between both the sensor and sources of illumination, and the features to be examined on the object to be viewed. This includes checking that they are not obscured by the body of the object, or protrusions on the object, or other objects in the vicinity, such as (robot) arms on which the sensor and sources of illumination are mounted.

The two main approaches used for calculating visibility spaces are those of *generate-and-test*,[137,138,155] in which a volume is hierarchically classified by the lines of sight available from each of its parts, and of *synthesis*,[27,41,149] in which the visibility area is analytically calculated.

1.4 Objectives

The objectives of this work may be stated as follows:

- To assess the suitability of Genetic Programming for use in this area.
- To evolve a program capable of solving the visibility space problem for two-dimensional polygonal objects.
- To determine which of the generate-and-test and synthesis approaches is more appropriate for use in this, and to investigate the reasons why.

1.5 Motivation

Though a new field, only seven years old at the commencement of this work, Genetic Programming has already managed to surpass human performance in certain areas, such as classification of transmembrane segments in protein primary structure sequences.[89] Furthermore, it often discovers solution strategies different in approach to those used in human-coded solutions.[84]

Genetic Programming has shown itself to be applicable to a wide variety of problems,[84] in fields as diverse as planning,[29,63,130] programming,[62] design[90] and classification;[10,12,84,88] and its applicability in the field of Machine Vision has already been demonstrated.[11,43,64,77]

As described above, visibility space planning is an area within Machine Vision which would be greatly aided by automation. This is a task which GP has not until now been used to solve; its potential for use in this area has been an unknown.

Automated inspection and sensor planning[154,155] constitute an area in which work has previously been carried out in this Department. Therefore it was decided to carry out this study to determine to what extent, if any, GP is able to evolve programs to solve the visibility space problem.

It was decided to study the ability of GP to solve the visibility space problem in two dimensions, as a simple model for more complex visibility space problems.

1.6 Structure of this thesis

The structure of this thesis is as follows: Chapter 2 describes the Genetic Programming paradigm and sensor planning in detail, and provides a survey of work done in both fields.

Chapter 3 commences by describing how Genetic Programming is to be used to attempt to solve the visibility space problem. The remainder of Chapter 3, along with Chapters 4 and 5, describe the three successive GP systems implemented to carry this out.

Chapter 3 describes a system using untyped GP to attempt to solve the visibility space problem through a synthesis approach. The chapter concludes from the results obtained that this system is insufficiently powerful or focused to be able to evolve a solution.

Chapter 4 describes a system which tackles the same problem using typed GP and higher-level operations. The chapter includes a discussion of the issues that arose in implementing a typed GP system. A variety of approaches to try and overcome the obstacles to evolving a complete solution are described. It is concluded that the components of the system used—both the programming primitives and the fitness function—are not conducive enough to the evolution of progressively more complex programs to permit the evolution of a program solving

the visibility space problem.

Chapter 5 commences with a discussion of how to design a system immune to the weaknesses of the previous two. The chapter then describes the system, using the generate-and-test approach, that was as a result implemented, and the positive results that were obtained. The chapter concludes with an investigation into the evolutionary mechanisms responsible for the production of such solutions.

Finally, Chapter 6 draws together the lessons learned from this study and presents the conclusions that were drawn.

1.7 Contributions

The contributions of this study may be summarised as follows:

- Genetic Programming was applied to a problem area not previously tackled by it.
- The two-dimensional visibility space problem was solved by use of Genetic Programming.
- This problem was shown to be a hard one for GP to solve. To evolve a good solution, typed GP was needed, along with the use of an "Anytime algorithm" approach.
- The use of seeded programs to guide evolution was demonstrated.
- The generate-and-test approach to sensor planning was shown to be more appropriate than the synthesis one for solving this problem by means of GP.

Some results were also shown demonstrating the utility of various genetic operators and methods in solving this problem.

Chapter 2

Literature Survey

2.1 Introduction

The previous chapter explained the motivation behind applying Genetic Programming to Sensor Planning. In this chapter, an introduction is given in turn to each of these, surveying the current state of both fields.

Genetic Programming is introduced in Section 2.2 as a branch of Evolutionary Computation. Sections 2.2.4 to 2.2.11 then discuss each in turn of the parts that make up a Genetic Programming System. Section 2.2.12 illustrates these by providing an example of a simple GP system, for carrying out symbolic regression.

Section 2.2.13 describes the uses to which GP has been put, with particular reference to its uses within the field of Machine Vision.

Section 2.3 then provides a survey of the field of Sensor Planning. Section 2.3.1 gives an introduction to the field, and Section 2.3.2 describes in more detail each of the two main techniques, generate-and-test and synthesis, used to carry out visibility space analysis, the task to which GP will be applied in this study.

2.2 Genetic Programming

2.2.1 Introduction to Evolutionary Computation

The term Evolutionary Computation (EC) defines a paradigm of Artificial Intelligence, in which solutions to a problem are refined in a process based on biological evolution.[127] In EC, a population of trial solutions to the problem in hand is randomly generated, and each of them assigned a *fitness* indicating the quality of that solution. Each individual is typically referred to as a *chromosome*, after the structures which encode an organism's genetic makeup in biology.[6,99] In certain Evolutionary Algorithms (EAs)—the different types of Evolutionary Computation—this chromosome itself comprises a trial solution to the problem; in others the chromosome contains information which is used to construct the trial solution. In the latter case, the chromosome is called the *genotype*, and the solution constructed from it the *phenotype*.

Individuals are then selected from the population and genetic operators are applied to them. This selection is done such that the more fit are more likely to be selected. The most common genetic operators are as follows:

- *Reproduction* consists of the individual's survival to the next generation unaltered.
- *Recombination*, also referred to as *crossover*, denotes the production of a new individual by combining portions of two or more parent chromosomes from the current generation.

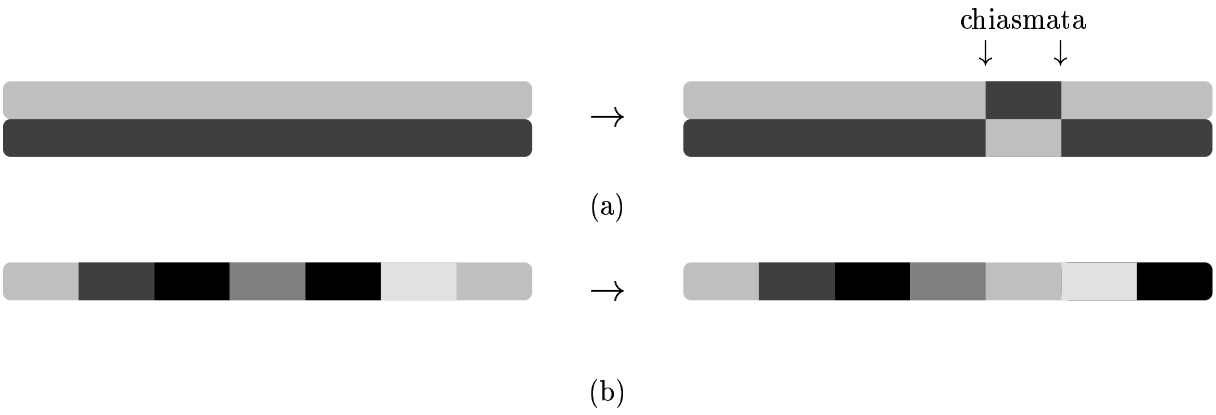


Figure 2.1: The mechanics of the genetic operations: (a) crossover, (b) inversion mutation.

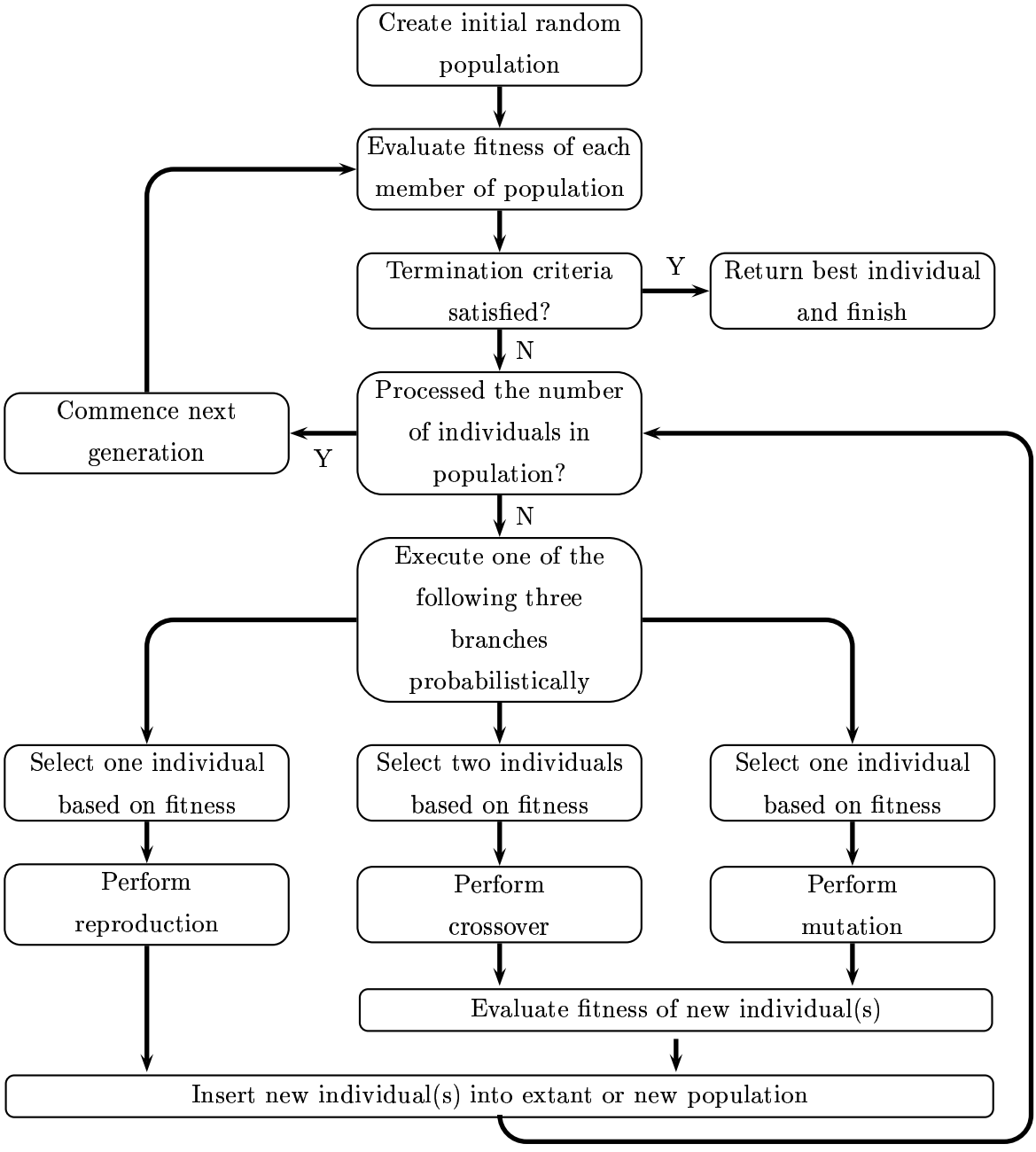


Figure 2.2: Summary of the Evolutionary Computation paradigm.

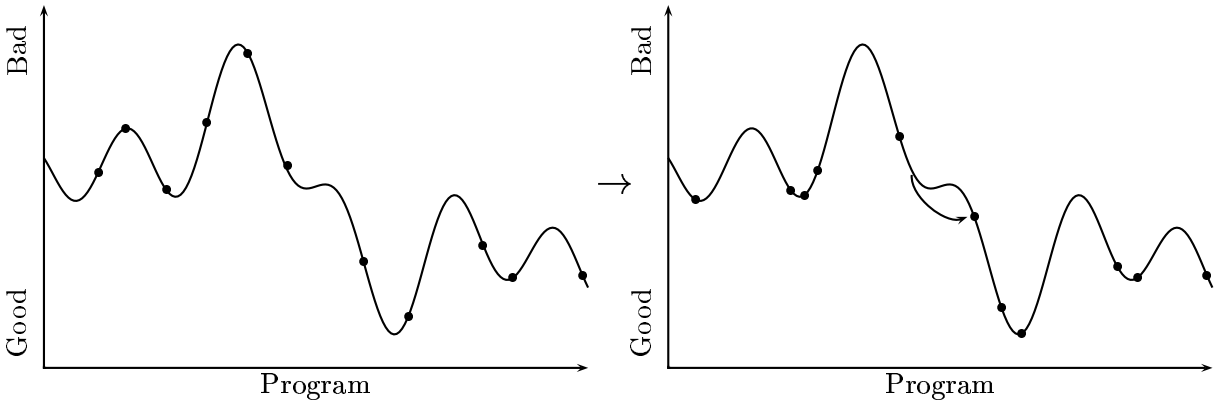


Figure 2.3: Evolutionary exploration of a fitness landscape.

This is modelled on the manner by which biological chromosomes recombine during sexual reproduction (see Figure 2.1a).

- *Mutation* operators perform various changes on a single individual, such as altering, shrinking or inverting component parts of it (Figure 2.1b).

The individuals produced by these operations are inserted back into the same, or a different, population. Over many iterations of the genetic process across the entire population, known as *generations*, the chromosomes come to model successively better solutions to the problem, and may solve it entirely. This process is summarised in Figure 2.2.

Evolutionary Computation constitutes a form of *beam search*,[18] in that it explores a search space neither by a single step-by-step exploration, nor by an exhaustive search, but by an in between approach of considering several data points at a time. By considering a population of data points strewn randomly across a fitness landscape (Figure 2.3), EC is able to ensure that even if some individuals become trapped in local optima of the landscape, others will escape them, hence ensuring local optima do not prevent further improvement of the population. Furthermore, the use of crossover between different individuals allows the possibility of their progeny skipping over obstacles in that fitness landscape, which would not be possible with a simple hill-climbing* algorithm (highlighted transition in the figure).

*The term "hill-climbing algorithm" normally refers to one that maximises some value. However, when standardised fitness is used in GP, optimisation corresponds instead to minimising fitness. For consistency with the

A key feature of EAs is that of partial credit: individuals that fail to completely solve the problem are rewarded for how close they manage to get. If the fitness of an individual is not implicit in the solution itself, fitness is typically measured by testing the individual against a fixed-size set of input data, known as *fitness cases*.

An individual's measured fitness is then used as its likelihood of being selected for one of the genetic operators. Less fit individuals fall by the wayside and, over the course of many generations, natural selection[44]—"survival of the fittest"—results in the population becoming better able to solve the problem in hand.

The various different Evolutionary Algorithms include Genetic Algorithms,[58,68] Genetic Programming,[18,84,87] Evolutionary Programming, Evolutionstrategie and Classifier Systems.

2.2.2 Evolutionary Computation and Biology

Though biological evolution normally takes place over the course of millions of years, EC shows much the same principles. In particular, experiments have been done evolving ribonucleic acid (RNA) in a test tube[18,118] which have resulted in some remarkable resemblances to EC. In these, the enzyme $Q\beta$ replicase was used to replicate strands of RNA. Every thirty minutes one drop of the solution containing the experiment would be transferred to a fresh test tube of the experimental reagents: the enzyme and the nucleotide monomers used to construct RNA. Resistance to antibiotics was used to impose a form of selection. The results of these experiments showed the following resemblances to Evolutionary Computation:

- Rapid evolution.
- Convergence of the population onto a stable end state.
- Dependence of the result of evolution on initial conditions and the parameters of the system, including the fitness function used.
- The production of individuals with complex structures that would be highly unlikely to be discovered through random chance alone.

rest of this thesis, this is the scheme used in Figure 2.3, but the term "hill-descending" is not used to describe it, because of the connotations this has of finding the worst value.

Two additional data which further blur the division between biological and computational evolution are worth mentioning: It has been shown possible to carry out evolution of self-reproductive programs which undergo processes similar to biotic life, such as speciation, competition, cooperation and parasitism;[127] and it has also been shown possible to carry out universal computation using a cellular automaton—a paradigm of computation for which Genetic Programming has also been applied[5, 10, 12]—made out of DNA.[159]

2.2.3 Genetic Programming as an EA

Genetic Programming (GP),[18, 83, 84] devised by John Koza in 1989, is an Evolutionary Algorithm that has emerged as an offshoot of Genetic Algorithms. The chromosomes in GP take the form of computer programs, and fitness evaluation consists of running those programs and measuring the quality of their output.

As an AI method, GP is fairly weak, which is to say that it is domain-independent.[85] It also violates almost all of the seven principles of AI[84] enumerated by Koza: correctness, consistency, justifiability, certainty, orderliness, parsimony and decisiveness. That is to say, the answers it delivers are almost always approximate; they are reached by simultaneously pursuing many different solution strategies; they have no logical chain of reasoning justifying their constituent parts, and they are frequently long-winded, in contravention of Occam's Razor, which states that the best solution is likely to be the simplest one fitting the facts. Moreover, the evolutionary process, being probabilistic, does not guarantee that any answer will be found, and there is frequently no point at which evolution can be guaranteed not to proceed any further. Not all of these characteristics are unique to GP, though—some other paradigms of AIs, such as heuristic search, also have no certainty of finding an answer, for example.

Despite the characteristics given above, GP has been shown capable of outperforming both humans and other paradigms of AI in certain fields[89] such as, for example, design of a cellular automaton rule for solving the majority classification problem.[12] Furthermore, it is robust,[84] able to find solutions even in the presence of noise and incomplete information. GP frequently comes up with solutions very different from the way humans would solve these problems. For example,[84] problem for which π was not necessary to construct an answer, hence was not included

in the terminal set, GP came up with a solution using $\pi/2$ (1.579 to three decimal places), which the program approximated by:

```
2 - sin (sin (sin (sin (sin (sin (sin (sin (1)) * sin (sin (1))))))))
```

which evaluates to 1.567, to three decimal places.

Because GP is a symbolic approach, the answers it delivers can be analysed to see how they work, and insights can be garnered in this manner that would be considerably more difficult with non-symbolic methods such as neural networks.[146]

The following sections describe the GP paradigm in more detail.

2.2.4 Representation In GP

In GAs the chromosome is typically a fixed-length bit or character string. Genetic Programming represents an extension to this concept, in which the chromosome consists instead of a computer program of arbitrary length, and the solution to the problem is obtained by executing this program. Therefore whereas GAs represent a search through the problem's solution space, GP represents a search through program space.

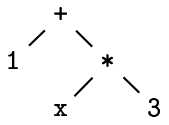
Genetic programs are expressed in a language of the user's choice. As in other paradigms of symbolic Artificial Intelligence,[101] this requires having to choose the best representation for the task in hand.

Genetic programs typically take the form of *parse trees**, each node of which is known as a *gene*. Parse trees are an intermediate stage in the compilation of most languages, and the native representation of programs in some, such as Lisp.[84] Figure 2.4 shows a simple program represented both as a parse tree and in Lisp.

A genetic program is put together from two types of primitive: *functions*, which are intermediate nodes of the parse tree, and *terminals*, which are leaf nodes. In Figure 2.4, `1`, `3` and `x` are terminals, and `+` and `*` are functions.

The representation of a problem through the functions and terminals used in genetic programs is extremely important; a poor choice can hinder evolution. Their choice determines the

*This refers to the program's internal representation; a routine can of course be written to output the programs in a more sequentially-oriented format.



(a)

```
(+ 1 (* x 3))
```

(b)

Figure 2.4: Representation of genetic programs by parse trees. (a) Parse tree; (b) The same program as written in Lisp.

level at which evolution will take place: For example, both a system with `push` and `pop` in its function set, and one which instead possesses read and write functions for an indexed memory, have the potential to evolve complex functions which require stack manipulation. However, the latter system will have to evolve the stack management routines first, and will therefore have to process many more individuals to achieve the same result. Moreover, if there is no fitness benefit for evolving the stack management functions alone, the latter system may be unable to solve this problem.

Though use of low-level functions and terminals will increase the size of the search space, it is more important that a representation be used that is well-suited to GP evolution: Though it has been shown that the traditional representation of programs as parse trees can actually increase the size of the search space,[87] the same study showed this to be without adverse effect.

Languages used for Genetic Programming vary from the extremely simple, comprising just Boolean or arithmetical operations on integers, barely above the level of Genetic Algorithms, to highly complex ones, including loops and subroutines.

Traditionally Genetic Programming languages have been untyped.[84] This leads to the need for a phenomenon known as *genetic closure*, since all function elements of the language need to be able to accept all outputs of every function and terminal as possible inputs. For instance, the division operator `/` needs to be redefined to accept zero as a second argument without causing an error. This is normally done by replacing it with the "protected division function" `%`, which returns 1 in such a case.[84]

Applications

Many early GP systems were written in Lisp. The advantage of Lisp is its lack of discrimination between program and data; thus genetic programs can be constructed during system execution then simply fed through the native interpreter. This frees the GP programmer from having to implement an entire language; all that has to be written then is any specialised program elements, or any that need to be redefined for genetic closure.

For GP systems written in languages lacking this capability, two options are available. The more common is to implement a simple virtual machine capable of interpreting programs written in a language of the programmer's choice. The alternative is to represent genetic programs in an extant computer language such as C[119] or machine code.[112] The advantage of this is that the resulting programs execute fast, which can be crucial in cases where execution takes a very long time,[114] and they do not need rewriting for stand-alone execution, as is the case for hand-tailored interpreters. The disadvantages are that genetic programs must either be compiled, which slows overall execution down greatly, or interpreted, in which case the advantage of using machine code is lost. Furthermore, the resulting programs will not be portable between platforms. Greater care must also be taken than with interpreted programs to ensure evolved programs cannot cause run time errors, for example by executing code that divides by zero or writes to arrays beyond their bounds.

Variations in representation of genetic programs include the following:

- *Use of typing.* Whilst untyped GP is acceptable for simple problems, for more complex ones it can be too unfocused. Montana added type information to the program elements,[107] thus not only rendering the search more focused, but reducing the need for genetic closure too. (However, it has been demonstrated that the addition of type information to problems that can easily be expressed without the need for types does not necessarily improve evolutionary performance.[106])

  In strongly typed GP, a return type is specified for every function and terminal, and argument types for the functions. This constrains the ways they can be put together during program creation, crossover and mutation. Flat typing such as this can, however,

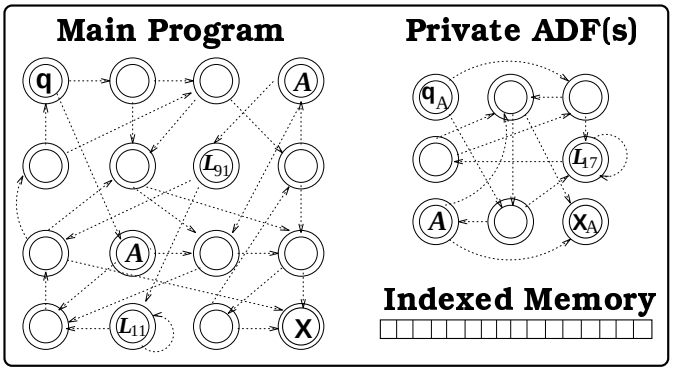


Figure 2.5:[152] Structure of a graph-based genetic program in PADO.

be insufficient to focus the search. For example, in a system with types INTEGER, REAL, CHARACTER and STRING it would be advantageous to specify that operations such as * can only take arguments of one of the numerical types. Haynes *et al.* extended GP typing to allow this by means of using a hierarchy of types.[66]

- *Linear and graph-based programs.* Whilst most GP systems represent programs as trees, linear chromosomes have also been used. Examples include GP for extant languages with a linear structure, such as C[119] or assembly language,[112] as discussed above; and stack-based GP,[120] discussed further below under program execution.

  A third alternative is graph-based programs. PADO, by Teller and Veloso[152] is one such system. Program execution in this resembles the traversal of a finite state transition machine (see Figure 2.5).

- *Non-programmatic genotypes.* Grammatical Evolution[136] is a variant on GP which specifies programs by Backus Naur Form grammars rather than parse trees. GE individuals consist of a linear chromosome containing a string of numbers. These encode which choices are made in the descent of a program generation tree. This is a flexible approach which allows for the encoding of linear, tree-based and graph-based programs.

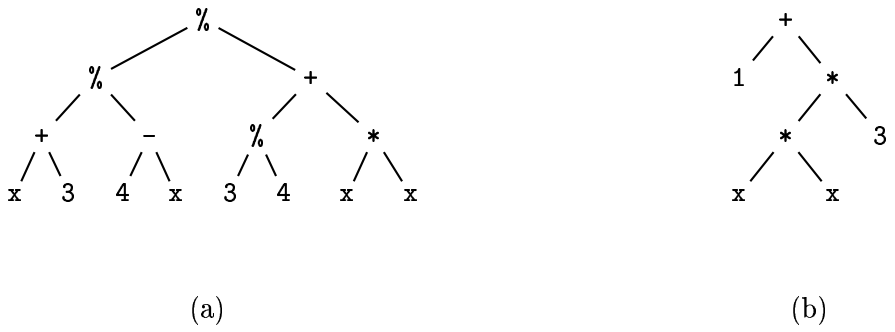


Figure 2.6: Creation types: (a) Full, (b) Grow.

2.2.5 Program Creation

The initial population of random programs in a GP run is put together by building trees out of randomly selected functions and terminals.

There are two primary methods of constructing tree-based genetic programs. In the *Full* method (Figure 2.6a), all branches of the program tree reach the maximum depth used for creation, whereas in the *Grow* method (Figure 2.6b), program branches may bottom out at any depth up to this maximum. The method of program creation shown to be most efficacious[84] is the *ramped half-and-half* method. This consists of distributing the initial random population evenly between the minimum and maximum tree depths, with half of the trees at each depth being generated by the Full method, and half by the Grow method.

Very large programs often contain large amounts of code which are not executed. Such regions of code are termed *introns*. To prevent this wastage of resources, a limit is generally set on the maximum size that programs may be created.

2.2.6 Program Execution in GP

Execution of traditional tree-based genetic programs operates on the same principle as that of Lisp s-expressions: Each function or terminal returns its *evaluation*, which is the result of carrying out its operation upon the evaluated results of its argument subtrees (if any).

For example, consider execution of the program `(* 3 (+ 1 x))` when $x = 10$. The left column shows the node being evaluated, with the indentation indicating depth in the parse tree;

the right column shows the return values of each subtree.

*	
3	`3` $\Longrightarrow 3$
+	
1	`1` $\Longrightarrow 1$
x	`x` $\Longrightarrow 10$
	`(+ 1 x)` $\Longrightarrow 11$
	`(* 3 (+ 1 x))` $\Longrightarrow 33$

The return value of the root node of the program is taken as the genetic program's answer. This value often requires some form of postprocessing, which is carried out by a function termed a *wrapper*.[84] In highly complex systems of which the genetic program forms only a part, the wrapper may determine how the genetic program is to be executed. For example, in some of the image analysis programs discussed below, the genetic program is run on the values of a small mask—seven pixels square, for example—convolved over the image used for input; in others the genetic program is used to determine the values of each pixel in the kernel which is then convolved over the image.

In such situations, the distinction between Genetic Programming and Genetic Algorithms can grow rather blurry. To take a simple example, a system to maximise the value of y in the relation $y = x^2$ is clearly a Genetic Algorithm, whereas one to evolve a symbolic equation $y = f(x)$ for any set of (x, y) data is clearly Genetic Programming; but a system to evolve numbers (a, b, c) to feed into the equation $y = ax^2 + bx + c$ falls somewhere in between.

It could be argued that Genetic Programming is an extension of Genetic Algorithms, capable of evolving solutions to domains of problems, rather than to individual problems, as in the case above—but the argument could also be made that Genetic Programming is a subset of Genetic Algorithms, used only to evolve data that is executed as a program.

Another potential distinction hinges upon the interpretation of the results which GAs and GP yield. For example in the case of GAs, numeric answers may require conversion from a genotypic Gray code. (This is a form of binary representation in which each successive number differs from

the last by only one bit, thus avoiding the "Hamming cliff" between, for instance, 00111111 and 01000000.[58]) However, such a conversion is merely a mapping between representations. GP, by contrast, yields executable programs which can give outputs for many sets of input data, without the need for running the evolutionary process again.

Evolvable subroutines

Only simple problems can be solved without the use of subroutines. If problems complex enough to require the repeated use of subtasks in their solution are to be solved, the GP paradigm must be supplemented with evolvable subroutines, since the chance of a program evolving the same structure every time it is needed is very low.

There are two main methods of incorporating subroutines into GP:[82] Koza's ADFs (Automatically Defined Functions)[86,87] and Angeline's Module Acquisition (MA).[15,16]

Automatically Defined Functions are functions with a fixed architecture, the contents of which evolve along with the main body of genetic programs. In the canonical version, each program has a predefined number of ADFs associated with it, each of which takes a predefined number of arguments. Each program branch—the result-producing branch (RPB) and the ADFs—possesses its own function and terminal sets. Each ADF's terminal set includes that ADF's arguments; the ADFs themselves are included as functions within the main body's function set. During evolution, crossover is only permitted to occur between the same program branches—i.e. for a system with two ADFs, the main program body can only cross with another main program body, ADF 0 can only cross with an ADF 0 and ADF 1 can only cross with an ADF 1.

In Module Acquisition no subroutines are initially provided. Instead, a new genetic operator, *compress*, randomly selects a subtree in an individual and replaces it to a given depth with a call to a module, to which the compressed code has been transferred. Any subtrees extending beyond that depth become arguments to the module, thus resulting in an abstraction of the original code.[117] This is illustrated in Figure 2.7. A second operation, *expand*, replaces a module reference in a program with the module's code. Modules allow a freer subroutine architecture, and once acquired, a module may be accessed by any member of the population by means of

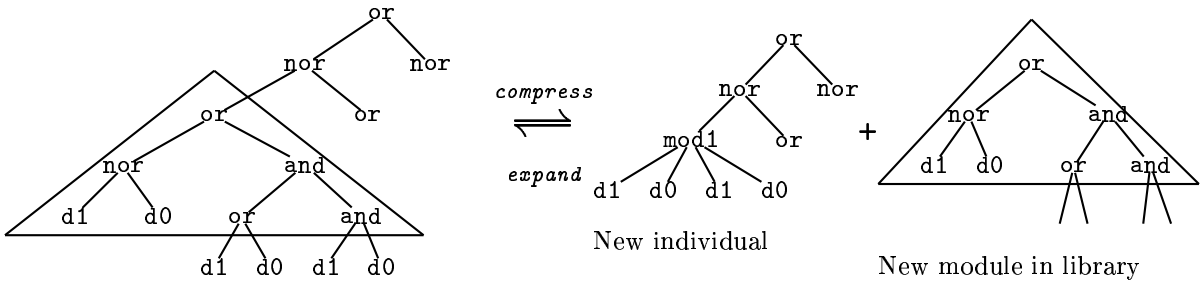


Figure 2.7: The *compress* operator in Module Acquisition.

crossover.

Unlike an ADF, once a module is acquired its contents are no longer free to evolve. Moreover, no modules are present in the initial population. Perhaps because of this headstart ADFs have at the start of a run, in a comparison of the two on the Boolean even-4-parity problem, ADFs outperformed MA.[82] When the two were compared for evolving a sorting algorithm, MA had little effect, though in this case ADFs degraded performance. If program subtrees are to be regarded as the GP version of schemata (see Section 2.2.9), then module acquisition results in the breakup of extant building blocks.

Variations on these include a system in which private ADFs are accompanied by a public ADF library,[152] the use of new genetic operators for altering the architecture of ADFs,[88] and a technique similar to Module Acquisition, named Adaptive Representation through Learning (ARL)[132] in which modules are created from subtrees which are heavily used, rather than randomly.

State memory

Without a way of representing the internal state of a program's execution, the genetic program is merely a finite-state machine of limited capacity. Simple systems use terminals to represent bound variables, but this becomes unwieldy with more than a few such, and does not easily allow both read and write operations to be performed on them. Addition of an indexed state memory, as pioneered by Teller,[150, 151] makes the genetic machine potentially Turing complete

and allows it to evolve solutions to much more complex problems.

Ordinarily such a state memory would be cleared every time a program is executed; however it has been shown that if this is not done, thus enabling information to be transmitted between individuals "culturally" rather than genetically, this can decrease the computational effort required to solve some problems.[142]

Execution of non-tree-based GP

In Perkis' stack-based GP,[120] programs are represented as linear chromosomes, as is the case in both biology and GAs. Stack-based genetic programs have no functions and terminals; instead program elements take their arguments from an internal stack and return their results to that stack.

The top value on the stack is used as the result of the program; the remainder of the stack is discarded. If parsimony of stack usage is desired, this can be selected for by penalising excess stack length at the termination of program execution.

One advantage of using a results stack is that it allows genetic programs to use stack blocks as data structures, without the necessity of explicitly providing data structures of any prespecified architecture in the function or terminal sets.

The use of linear chromosomes for stack GP partially restores the GA phenomenon of lexical convergence (see Section 2.2.9). This could lead to lower genetic diversity in the later stages of a run than with tree-based GP.

Use of a program stack is not limited to linear programs; it could be used with tree-shaped programs too. The PADO graph-based system[152] also uses a program stack for execution.

2.2.7 Fitness Evaluation in GP

Fitness evaluation in GP consists of a quantitative comparison of the answer delivered by a program, and the ground truth for a series of inputs, the output values for which are known in advance. These are known as *fitness cases*. For the symbolic regression system from which examples have been drawn in the preceding section, in which the objective is to discover a relationship $y = f(x)$ describing a series of (x, y) input data, the fitness cases would consist

of a variety of values of x for which y is known. Possible fitness measures would include the summation of the absolute values,[84] or the root mean square,[18] of the discrepancies between the value returned by the program and the correct answer, over these values of x. These conform to the concept of *standardised fitness*, in which more fit individuals have lower fitness, and the perfect solution has a fitness of zero.

The fitness cases are generally chosen to broadly cover the domain of input and output data. If this range is sufficiently diverse, programs will evolve which are *robust*, and able to cope well with input data outside of the range on which they have been trained.

Normally, the number of fitness cases is very much less than the total size of the domain search space. The danger in this is of solutions becoming *brittle* or overfitted, that is, capable of delivering correct answers to the fitness case input data, but with poor performance for other input data. As an example, consider a symbolic regression system to discover the relationship $y = sin(x)$. Given only data in the range $-40^\circ \ldots 40^\circ$ as fitness cases, the system might come up with $y = 0.0161x$ as the best solution. This equation works quite well within this range, but performs increasingly poorly outside it.

One method for preventing such brittleness is the addition of stochastic noise to the input data.[129,130] Another is to use dynamic fitness cases,[87,97] that is, to alter the fitness cases from generation to generation, generally by drawing the ones used from a larger potential set.

Other variations in measurement of fitness that have been used include the following:

- Use of a dynamic fitness measure[13,102] for problems complex enough that evolution of a complete answer *ex nihilo* is unlikely. In this case, a fitness function assessing performance in a subtask necessary for the solution of the entire problem can at first be used. Once this subtask has been solved, the fitness function would be changed to assess performance in the entire problem.

- Competitive fitness measures.[84,128] Though in most EC systems the fitness is explicit, in most biological systems it is implicit: organisms compete against each other. In certain situations, such as game-playing, it is possible for genetic programs to compete directly against each other, thus obviating any need for an external, possibly arbitrary, criterion

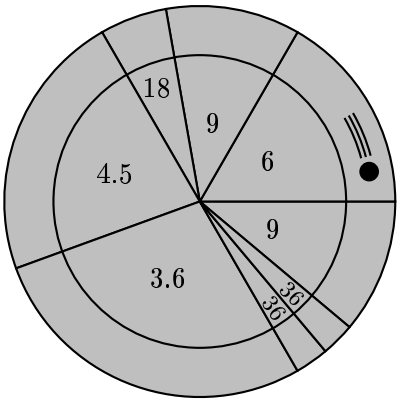


Figure 2.8: Roulette-wheel selection—numbers indicate individuals' fitness.

as an index of success.

2.2.8 Selection in GP

The two main methods of selection of individuals for reproduction and crossover in EC are *fitness-proportionate* selection and *tournament* selection.[84]

In fitness-proportionate selection, also known as the roulette-wheel strategy, an imaginary roulette wheel is divided into sections like a pie chart, according to each individual's fitness. The better the fitness, the higher the chance an individual has of being selected. This is illustrated in Figure 2.8. Note that with the use of standardised fitness (see above), each individual's share of the roulette wheel is inversely proportional to its fitness.

In tournament selection, a number of individuals are picked randomly from the population, and the fittest one selected for the genetic operation. The size of the tournament constitutes a parameter to the system.

When it is required to replace an individual in the population, methods in use include replacing the worst-of-population, reverse fitness-proportionate selection and reverse tournament selection.

Structured populations

The basic GP paradigm is *panmictic* in that all individuals are able to breed with all other individuals. However in nature it is the separation of communities from each other that leads to speciation. This can be modelled in GP by the use of *demes*,[147] small communities breeding mostly within themselves, but with a small, user-specified amount of migration between them.

This is normally done with a linear geometry,[147] such that demetic migration can only occur with the deme that comes immediately before or after an individual's in the population, though other geometries, such as a two-dimensional grid of demes[5] can also be used.

Disassortive mating is a variation on the theme of demes. A common flaw of GP systems is that of *premature convergence*, in which the population loses genetic diversity once an individual with moderately good fitness is discovered. Once this has happened, the population is unable to evolve better solutions, and becomes filled with functional clones of the mediocre solution.

One way of overcoming this is to use phenotypically different parents for recombination. Ryan introduced a scheme[134,135] of using two breeding pools, one of individuals which are good solutions but are bulky, and one of short but less efficient individuals. When recombination is carried out, one parent is chosen from each of these. It has been shown[60] that for a certain complexity of problem the use of this technique, called the "Pygmy Algorithm", works in synergy with ADFs to help solve the problem.

A variation on this approach, devised by Aler,[7] is to maintain ADFs as a separate population from the main program bodies, evaluating main bodies with the best ADF in the ADF population, and ADFs with the best main body in the main body population. This allows ADFs to be selected for their fitnesses separately from the rest of the program. Some good results have been shown using this for simple boolean problems.

2.2.9 Genetic Operators

Basic crossover

Crossover for tree-based chromosomes cannot utilise the mechanism described in Section 2.2.1 without modification. Instead, tree-based crossover works by swapping whole subtrees between

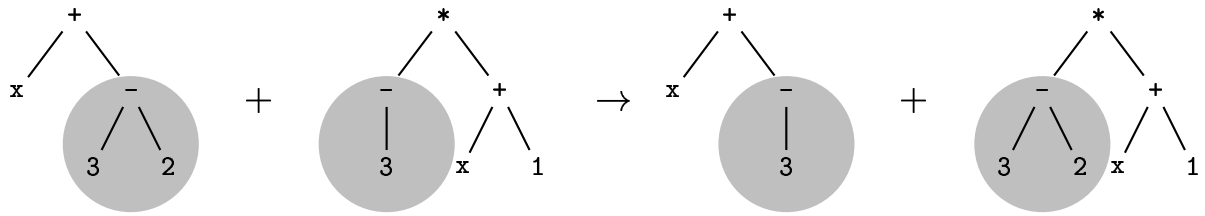


Figure 2.9: Crossover for tree-based GP.

individuals, as illustrated in Figure 2.9.

Crossover between terminals is equivalent to point mutation; to encourage the use of crossover for exchanging programmatic building blocks, crossover may be biased toward using function points in the program tree. However, in experiments in which the frequency of selection at different points in program trees was permitted to evolve along with the trees themselves, crossover point selection gravitated to an optimum of between 45% and 55% terminal crossover.[14]

It has been observed that during the course of evolution, programs tend to increase in size ("bloat").[96] To prevent this from getting out of hand, a maximum depth for crossover is normally imposed.

A theory, the Schema Hypothesis,[58] has become established for how GAs operate: namely, by the accumulation of building blocks named *schemata*. Attempts have been made to apply this to GP and identify schemata in program trees, but success in this regard has been limited.[122,123] Subtrees are often identified as schemata, but this identification does not agree well with the mathematics of the schema hypothesis.

One consequence of using tree-shaped rather than linear chromosomes is the loss of the phenomenon of *lexical convergence*. With linear chromosomes of fixed length, crossing two identical individuals yields the same individuals again,[46] whereas with tree-shaped chromosomes, it generally leads to two different progeny. Though towards the end of a GP run populations tend toward functional convergence,[46] the population retains larger genetic diversity than will be found in an equivalent run using linear chromosomes.

Variations on crossover

A phenomenon frequently seen in GP is for subtrees to become duplicated many times. D'haeseleer has alleged[46] that the reason for this is crossover's tendency to destroy subtrees, which are the building blocks of correct solutions. By becoming duplicated, subtrees are protected against destruction. This builds on the concept of the selfish gene:[45] that natural selection selects for genes rather than individuals, and sometimes genes will be favoured over the individual that hosts them. ("Genes" in this context must be interpreted as useful building blocks, rather than individual nodes of a parse tree.)

Context-preserving crossover,[46] developed by D'haeseleer, is a form of crossover which prevents subtree destruction in this manner, by only allowing recombination between two subtrees in the same relative positions in their parents. Two types of context-preserving crossover have been presented, of which the best results were obtained[46,111] with a mixture of the more restrictive type and the normal, unrestricted crossover.

Depth-dependent crossover, developed by Ito *et al.*,[75] also works to prevent destruction of subtrees, by only allowing subtrees of the same tree depth to recombine. Results for this do not clearly show improvement over unrestricted crossover.

A third approach is the use of Explicitly Defined Introns, as developed by Nordin *et al.*,[115] to bias crossover toward locations in between subtrees. This approach has been shown[24] not to work for problems such as symbolic regression in which code is only rarely entirely intronic.

Poli and Langdon[121] argue that standard crossover is too local, because most of an offspring's genetic material comes from one parent, and also too biased towards terminals. They present experimental results favouring uniform crossover instead, which not only overcomes these difficulties, but also automatically progresses during the course of a run from making major changes to making slight ones.

Mutation

The two most common mutation operations used in GP are swap mutation and shrink mutation. Swap mutation consists of swapping a function or terminal for another of the same arity (Fig-

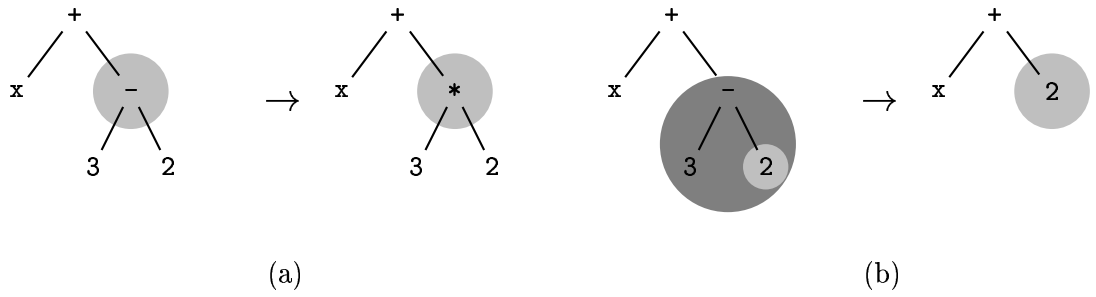


Figure 2.10: Mutation for tree-based GP: (a) swap mutation, (b) shrink mutation.

ure 2.10a). Shrink mutation consists of replacing a subtree with a smaller subtree from within it (Figure 2.10b).

Other mutation operators have also been proposed, such as replacing a subtree with a freshly generated random subtree.[80,81]

Other genetic operators

Various other genetic operators have been proposed for GP, such as Kim Kinnear's *hoist* and *create*.[80,81] Hoist takes a function point within a program, i.e. the root node of a subtree of depth greater than one, and makes that subtree into a new individual. This was developed to help combat bloat. Create creates a new individual as in the initial random population; this can be useful for helping to prevent premature convergence of the population.

2.2.10 Turnover in GP

Population size versus number of generations

It is generally recommended to use the largest population allowed by resources;[84] this maximises the size of the gene pool, which is essential for evolution to proceed well.† Earlier work carried out by myself[60] bore this out, in that evolution proceeded at approximately the same pace for

†This is illustrated in biology by the evolution of mammals: In New Zealand, isolated from the rest of the world, mammalian evolution reached only as far as egg-laying monotremes. In the island continents of Australia and South America, the larger gene pools allowed the evolution of marsupials, which carry their developing young in a pouch. The remaining continents shared a common gene pool, and on them marsupials further evolved into eutherian mammals, the young of which develop inside their mothers until capable of independent existence.

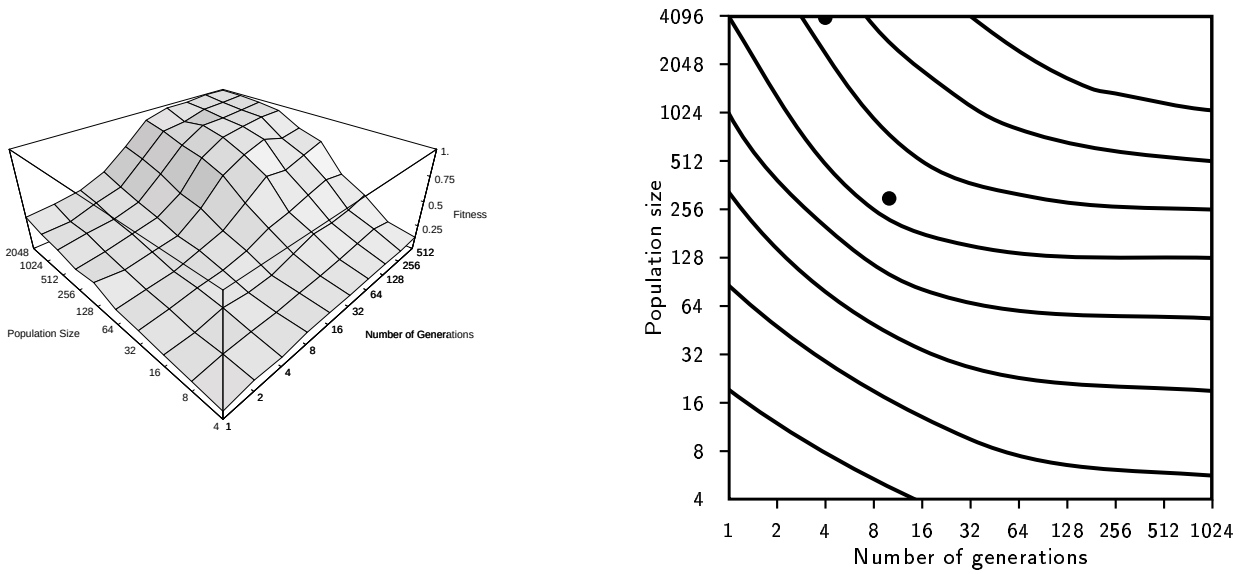


Figure 2.11: Evolutionary equivalence of population size and number of generations. (a) In Luke and Spector's work.[104] (b) Hypothesised relationship in my own work (contours indicate equal fitness).

runs with different population sizes, with respect to number of individuals processed, but the improvement in fitness plateaued earlier, at a higher (less good) fitness, for the run with the smaller population.

Work by Luke and Spector presents dissenting evidence which shows an approximate symmetry between increased population size and increased number of generations[104] (Figure 2.11a). For some problems this was skewed in favour of large populations, however for others it was skewed the other way. Figure 2.11b shows a hypothesised relationship of this type to explain my findings in my earlier work, with the two dark circles showing the population sizes and number of generations used for experiments. For the run with the smaller population, improvement has levelled off by the time the run has completed: allowing the experiment to run for further generations will not result in further improvement (crossing of contour lines on the graph). For the run with the larger population size, by contrast, allowing the run to continue for longer will result in more contour lines being crossed; and in general, increasing the population size of a run will lead to more contour lines being crossed.

Gathercole and Ross[55] have demonstrated the existence of problems for which small pop-

ulations over many generations performed better than large populations over few; the reason they suggest is because shorter generations lead to more fitness evaluations, allowing evolution to proceed in smaller, and hence easier to accomplish, increments. (They do not give sufficient data in their work to allow construction of a graph of the form in Figure 2.11.) This could explain the discrepancy with the results from biology, because biological individuals are assessed for their fitness continuously, rather than at the single points before their reproduction.

However, the main conclusion to be drawn from these various differing results is that whether it is more advantageous to use large populations or to let runs with smaller ones last for more generations, is problem-specific, and cannot be predicted in advance. In this work, Koza's original advice of using a large population size is followed, in order to maximise the size of the gene pool.

Method of replacement

In GP as originally devised, the system iterates through the population, applying the genetic operators probabilistically to each individual in turn. The new individuals thus produced are written to a fresh population, resulting in the whole population being effectively turned over in one go. This is termed *generational* evolution.[84]

In most biological systems, however, breeding takes place all the time. This is modelled in EC by *steady state* evolution,[145] in which the population is updated continuously, each newly produced individual replacing one selected for poor fitness.

Elitism consists of allowing individuals to persist from generation to generation without having to be selected on account of their fitness. It can be helpful to allow the best individual of each generation to persist through to the next in this manner; since even being the best in a generation is not sufficient to guarantee survival in a probabilistic selection system.

With tasks which have a clear criterion of success, the EA run can be programmed to terminate when this criterion has been reached. In general, there comes a point beyond which evolution is unlikely to make any more improvements (see above), but it makes sense to carry out EA runs with a large number of generations in order that this point may be well established.

2.2.11 Relative Utility of Genetic Operators

This section discusses the relative advantages of crossover and mutation in biology, GAs and GP.

In vivo

Most of the genetic variation between parents and their offspring in nature is a result of crossover;[6] mutations are more often deleterious than beneficial.[99]

However, crossover by itself is not sufficient to account for speciation; moreover, bacteria, which comprise the largest biomass on Earth, largely reproduce without any use of recombination.[18]

The question of which of these two scenarios is of greatest relevance to EC comes down to whether EC models microevolution (alterations in individual genes over tens of generations) or macroevolution (generation of entire new species over hundreds of thousands, or more, of generations).

One factor which may play a role in which of these operators is of more benefit to an individual is whether its reproductive strategy is *r-selective* or *K-selective*.[79]

The propagation of an r-selective species is determined by the rate of population growth, r. Each individual produces many offspring—up to hundreds or thousands—only a few of which survive predation to adulthood. The propagation of a K-selective species is limited by the carrying capacity, K, of the ecosystem. Only a small number of offspring are produced, but by caring for their progeny, the parents ensure that most of these survive to maturity.

As a result of this, an r-selective species can try out novel genetic combinations, by utilising mutation in its reproduction, without endangering the production of sufficient viable offspring to produce the following generation.

Whilst GP individuals do not, of course, care for their offspring before they are ready to reproduce, GP is K-selective, in that each individual generally only gives rise to two offspring per reproduction, and this results in the genetic operator used to produce the next generation being highly important, as each individual may only get the one chance to reproduce.

In numeris

In GAs, theoretical and empirical evidence mostly point to crossover being the most important operator.[103] This is because crossover acts to build up ever more complex schemata. However, there is also evidence that mutation can play a useful role; and the issue of which is more important has yet to be fully resolved.

In programmis

Though initial experiments with GP appeared to agree with the evidence from GAs that crossover is the most important operator, with mutation having little effect,[84] doubt has now been cast upon the generality of this.[54,104] The Schema hypothesis has not been universally accepted for GP, and both the representation of GP trees and the forms of the crossover and mutation operators are greatly different from their counterparts in both GAs and biology.

The initial results cited above were obtained with a system evolving Boolean functions, in which the order in which subtrees were executed was irrelevant. This is clearly not applicable to more complex systems, and still less to those that use some form of global state memory. Banzhaf *et al* suggest[18] that the breaking of this dependency between a subtree and its parent node greatly reduces the utility of crossover; however, experiments by Luke and Spector to test this[104] have not shown this clearly. Instead they show, for systems in which global dependencies do not exist, a slight preference for mutation in smaller populations and crossover in larger ones; and for systems in which they do, no statistically significant preference either way.

In the light of these results, and that of the fact that combinations of the two operators do not perform more successfully than any one of them alone, it has been suggested[18] that subtree crossover in GP functions in effect as a macromutation operator. Chellapilla has reported good results[33] obtained without any use of crossover whatsoever.

Various improvements have been suggested to crossover, to prevent the damage it does in breaking up dependencies; some of these were discussed in the previous section.

In contrast to the abovementioned work, other people[48] have claimed the success of a GP system at solving a problem depends on the optimal configuration of the system's parameters:

the likelihood of each of the genetic operators, the number of ADFs available to each program, etc. This represents the injection of *a priori* knowledge, which, some would argue, defeats the point of letting the computer work the solution out. Koza has shown[87] that, if a further genetic level is added, so that these parameters are allowed to evolve themselves, Genetic Programming systems are still able to solve the problem in hand, and optimise their own parameters. Eiben *et al.* have claimed[48] that this invariably leads to an improvement in evolution; however, although Koza's work shows that GP can optimise its own parameters, it also suggests[87] doing so hinders the evolutionary process, by slowing down the discovery of good solutions.

2.2.12 A Simple Example

To illustrate the principles given above, a simple system is now described.[18] The problem is one of function regression: to come up with a symbolic equation describing the relationship of one set of numbers to another. In this case, the relationship between the two sets of numbers is:

$$f(x) = \frac{x^2}{2}$$

In this simple example, the relationship is known in advance. In a real situation, this would of course not be the case; the relationship is, therefore, treated as unknown in the rest of this section.

The first step in devising a GP system is to define the function and terminal sets that will be used. It is important these be sufficient to solve the problem; it is also important they are no larger than is necessary for this, as superfluous operations have been shown to have a deleterious effect on evolution.[18,84] Since it is not known in advance what form the relationship takes, whether polynomial, trigonometric, or exponential, the function and terminal sets must be sufficiently powerful to solve the problem whichever of these is the case. Consequently, the following function and terminal sets were chosen for this problem, with the arity of each function indicated in subscript:

$$F = \{\texttt{+}_{/2}, \texttt{-}_{/2}, \texttt{*}_{/2}, \texttt{\%}_{/2}, \texttt{sin}_{/1}, \texttt{cos}_{/1}, \texttt{exp}_{/1}, \texttt{rlog}_{/1}\}$$

$$T = \{\texttt{x}, \Re\}$$

x	0.000	0.100	0.200	0.300	0.400	0.500	0.600	0.700	0.800	0.900
$f(x)$	0.000	0.005	0.02	0.045	0.080	0.125	0.180	0.245	0.320	0.405

Table 2.1: Fitness cases for the example system.

To achieve genetic closure, every function must be able to accept all outputs of every other function and terminal as potential inputs. For this reason the following two protected functions are used:

- `(% a b)` returns 1 if $b = 0$ and a/b otherwise.
- `(rlog x)` returns 0 if $x = 0$ and $\log \mid x \mid$ otherwise.

$\Re$ denotes the Ephemeral Random Constant;[84] whenever this terminal is selected during initial program creation, it is replaced by a random floating point value in the range $-5 \ldots 5$.

The architecture of the system must be decided upon here too: how many ADFs there are to be, and what their arities are. In this simple system, ADFs are not necessary and will not be used.

The next step is to define the fitness function and fitness cases. In this case, ten values of x over the interval 0–1 for which the value of $f(x)$ is already known are used as fitness cases (see Table 2.1). The fitness measure used is the root mean square discrepancy between the answer delivered by the genetic program and the fitness case values, for each of the values of x given in the table.[18] This conforms to the definition of *standardised fitness*, in which zero is the best value.[84]

Nextly the evaluation function must be written. Since this system shall be running in Lisp, the native Lisp function `eval` can be used.

For some problems the output of the genetic program must be postprocessed before fitness can be measured. In this system, however, no postprocessing is needed.

The termination criterion for this system would be the evolution of a program scoring *hit*s—correct answers—for all ten fitness cases. For this problem a hit is defined as one for which the raw fitness is less than 0.01.

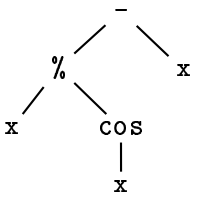
Parameter	Values
Objective	Evolve function fitting the fitness case values for x from -5 to 5
Function Set	`+`, `-`, `*`, `%`, `sin`, `cos`, `exp`, `rlog`
Terminal set	`x`, $\Re_{[-5,5]}$
Hit	Correct value to within 0.01
Termination criterion	Full complement of hits
Maximum number of Generations	100
Size of Population	500
Maximum depth of new individuals	6
Maximum depth of new subtrees for mutants	4
Maximum depth of individuals after crossover	17
Fitness-proportionate reproduction fraction	10%
Crossover at any point fraction	20%
Crossover at function points fraction	70%
Number of fitness cases	10
Selection method	Fitness-proportionate
Generation method	Ramped half-and-half
Randomiser seed	0.435
Population replacement	Generational

Table 2.2: Tableau for the example system

Finally, the various parameters of the genetic programming system must be adjusted such that evolution proceeds optimally. This is generally done by hand (though it can also be done by a meta-GA). It is customary to present the values chosen for these parameters in a tableau, to aid replicability of results. The tableau for this system is given in Table 2.2; the values are based on those that have been used in similar problems.[84]

Figure 2.12 shows an example program evolved with this system, and Table 2.3 shows the result of execution of this program.

The system was run in an Emacs Lisp port[60] of lilgp, the original GP system devised by Koza.[84] A program satisfying the termination criterion was found after thirteen generations. The results are shown in Table 2.4 and Figure 2.13.




```
(- (% x (cos x)) x)
```

$$\frac{x}{\cos x} - x$$

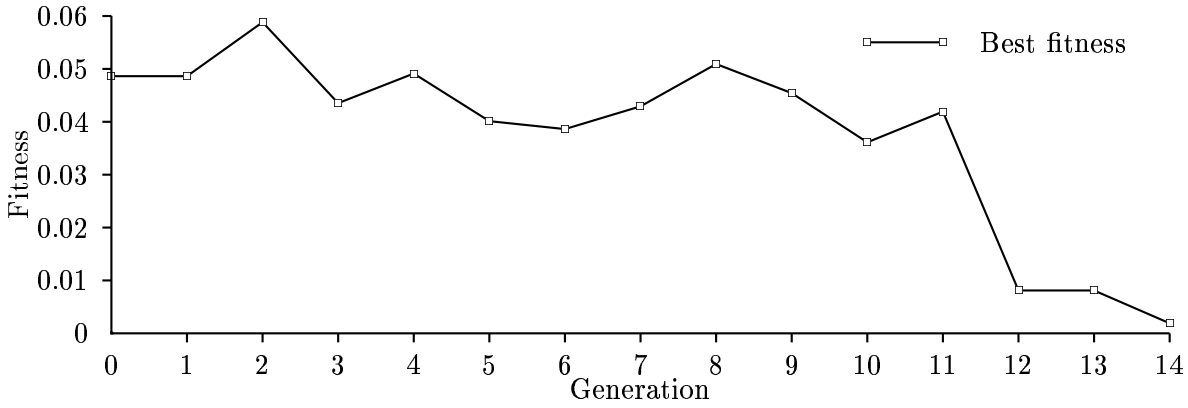
(a) (b) (c)

Figure 2.12: Sample evolved program. (a) Parse tree; (b) The same program as written in Lisp; (c) Relationship encoded by it.

x	0.0	0.1	0.2	0.3	0.400	0.500	0.600	0.700	0.800	0.900
$f(x)$	0.0	0.005	0.020	0.045	0.080	0.125	0.180	0.245	0.320	0.405
$g(x)$	0.0	0.001	0.004	0.014	0.034	0.070	0.127	0.215	0.348	0.548
$(f(x) - g(x))^2$	0	0.00002	0.00025	0.00096	0.00209	0.00305	0.00281	0.00089	0.00080	0.02040

$$\sqrt{\overline{(f(x) - g(x))^2}} = 0.056$$

Table 2.3: Fitness calculation of execution of the program in Figure 2.12.



Random number seed*: 0.435

* lilgp's random number generator is the Park-Miller multiplicative congruential randomiser.[84]

Figure 2.13: Graph showing the results given in Table 2.4.

Generation	Best individual of generation	Fitness	Hits	Average standardised fitness
0	`(% x 2.8070023921438274)`	0.0486	2	off-scale
1	`(% x 2.8070023921438274)`	0.0486	2	off-scale
2	`(% (% x 3.411562179887115) (% x x))`	0.0588	2	3.6234
3	`(* (exp x)` `(rlog (% (cos (* x -0.7703205461388318)) (cos x))))`	0.0435	3	off-scale

4	`(sin (* (* (+ x x) (sin x))`
	`(* (* x x) (exp (% 2.550803688014649`
	`-3.0779435270741553)))))`
		0.0491	3	0.9236

5	`(sin (cos (+ (- (exp x) (- x 2.320061794733075))` `(exp (exp -1.1408096514279298)))))`	0.0401	2	2.7158
6	`(* (+ -0.1279876605126713 x)` `(% 1.078126362110198 2.590542264339417))`	0.0386	2	6.4292
7	`(+ (exp (+ (exp (sin x)) (- -1.2153032627813767 x)))` `(+ -1.6098488655772991 0.8436850436846424))`	0.0429	1	0.7435
8	`(* x 0.3835258980292249)`	0.0509	1	0.7433

9	`(- (% x 1.9862248220096692)`
	`(* (* (sin x) (exp -1.1408096514279298))`
	`(cos -1.0232649110220438)))`
		0.0454	2	5.14×10^{32}

10	`(sin (* x (* x 0.413116779938278)))`	0.0361	4	157.204
11	`(% (* (* x x) (rlog 2.089763885187457))` `(exp (rlog 1.2140761715105386)))`	0.0419	4	0.8053
12	`(* (* x x) (cos -1.0232649110220438))`	0.0081	7	6.2417
13	`(* (* x x) (cos -1.0232649110220438))`	0.0081	7	0.5712
14	`(* (* x x) (exp -0.7026799331286124))`	0.0019	10	0.4446

Table 2.4: Results of evolution of the example system.

Note how in the best-of-run individual, the constant 0.5 is approximated by $e^{-0.7026799331286124}$, which is equal to 0.49525627004081670127.

2.2.13 Applications of Genetic Programming

Like any new field, Genetic Programming will take a while to be established as a means of solving practical problems. At just a decade since its creation, Genetic Programming still tends to be applied to small problems for the purposes of investigative research. Nonetheless applications for GP on a larger scale are beginning to emerge. Uses to which GP has been put include the following:

Natural Language Processing

- *Phone classification* in speech. In computer understanding of spoken speech, one of the first tasks to be carried out is identification of the different speech sounds (phones*) in the input. This is a difficult task to generalise for more than one speaker, and generally requires preprocessing of the input, such as Fourier transforming it. Conrads *et al.*[39] developed a GP approach to this problem, which performed well with more than one speaker, and acted on the raw, untransformed input. This used Nordin's machine-code GP;[112] the genetic language was untyped and possessed no ADFs, and operated by use of simple arithmetic and boolean functions. A population of 10 000 was used.

  Separate classifiers for each sound to be identified were evolved; two to eight recordings of each of the similarly-numbered sounds to be identified were used as fitness cases.

- *Decision tree induction* for disambiguation of natural language: decision trees, being tree-shaped, are readily amenable to being programmed by GP. Siegel[141] used a mixture of GP and GA to program decision trees for disambiguating the usage of words such as "anyway" (example contexts being "They programmed it anyway" and "Anyway, let's move on to the next topic").

*The term *phone* denotes a physical sound, as distinct from the language elements termed phonemes. For example, the 'p' sounds in 'pill' and 'lip' are different phones (allophones) of the same phoneme, /p/.

Planning

- *Corridor following* (obstacle avoidance). A real-time controller for an autonomous robot—planning its movements to avoid collisions with walls—was evolved by Reynolds.[129,130] This work was carried out in a simulated two-dimensional environment, but the input from the robot's sensors and the output to its actuators was rendered noisy in an attempt to make it realistic. This had the additional effect of making the solutions evolved more robust and general.

  The genetic language consisted of arithmetical functions, an `if` and a sensing operation. Populations of 2000 and 10 000 were used, with 64 separate program executions as fitness cases.

- *Planning* the actions of a robot agent. A GP system was developed by Calderoni and Marcenac to evolve candidate plans for a robot agent in a symbolic instruction language, for execution in a simple simulated 2D world.[29] A "moderate" population of 20 was deliberately chosen; possibly as a result of this the best evolved behaviour was not quite as good as that of a hand-crafted solution.

  In another work, Handley developed a Genetic Planner[63] capable of planning for complex goals involving conjunctions and negations. Because the Genetic Planner plans for all components of its goals simultaneously, it is able to cope with goals containing conjuncts which cannot be solved independently. Rather than taking however long is necessary to solve the problem, as in the traditional approach (and the work described in Chapters 3 and 4 of this study), the Genetic Planner uses an "Anytime algorithm"[151] to output a stream of increasingly accurate answers as time passes, so the robot is not held up waiting for a plan to be finalised.

Classification

- *Classification of transmembrane domains of proteins* from their primary sequence. Until a protein's three-dimensional structure is elucidated by X-ray crystallography or NMR spectroscopy, a difficult and in some cases impossible task, one can determine its struc-

ture solely by trying to predict it from its sequence of amino acid residues. For proteins embedded or anchored in biological membranes the first task is to identify the run(s) of residues which cross the membrane; the GP system developed by Koza and Andre to carry out this task has managed to exceed the performance of human-written programs to do this.[88]

This system used arithmetic operations, plus `if` and `or`, predicates for each of the twenty amino acids, a constant returning the length of sequence being examined, and $\Re$. It operated on a one-cell state memory. The language was not typed. This system included the architecture-altering operations discussed above in Section 2.2.6. The population consisted of 64 demes of 2000 each, and there were 496 fitness cases.

- Several groups have used GP to evolve cellular automaton rules for classifying which of two symbols is more common in a string. The work of Andre *et al.*[12] used Boolean operations to combine data from adjacent cells; the GP language included ADFs. A population of 64 demes of 800 was used, on 51 generations; there were a total of 1 001 000 fitness cases.

  Andersson and Nordahl's work[10] by contrast used polynomials to combine the data, with a stack-based representation and execution, using runs of 200 generations on a 100-strong population. There were 1000 fitness cases.

Data analysis

- *Marketing.* Eiben *et al.* used GP for data analysis in the field of marketing, in which it performed comparably with other techniques.[49]

Programming

- *Programming of neural networks.* One approach to this, Cellular Encoding, developed by Gruau,[62] involves storing the network's programming in the form of a grammar tree. Since GP deals with tree-shaped chromosomes, it can therefore be used to program the neural network. An alternative developed by Pujol and Poli[125] uses linear chromosomes to specify a two-dimensional representation.

- *Design of circuits.* Koza *et al.*[90] have developed GP systems able to design a number of different types of electric circuitry, including computational circuits (analogue circuits which carry out mathematical computations), high-gain operational amplifiers, and low-level robot controllers (i.e. made out of transistors, diodes, resistors and power supplies). Their results include the evolution of a computational circuit for calculating a cube root, a task for which they had found no prior circuit in the published literature.

  In the system for constructing computational circuits, programs consisted of a construction branch, with operations for constructing circuits, and an arithmetic-performing branch, with operations +, - and $\Re$. A population of 640 000 was used, with 21 fitness cases.

Machine Vision

- *Edge detection.* Harris and Buxton[64] have evolved a one-dimensional algebraic function for use as an edge detection kernel, based on Canny's criteria for an optimal edge detector.[31] Their representation was similar to that used for symbolic regression, taking the form of arithmetic and trigonometric functions. A 200-strong population was used, with 40 example signals as fitness cases. This system managed to evolve edge detectors outperforming Canny's, but suffered from overfitting due to the low resolution of the sampling of evolved functions.

- *Model description.* Some recent work in the field of Machine Vision has been directed towards the automatic generation of object recognition programs from a model of the object.[73] Sensor Planning for inspection tasks too requires a model of the object as an input to the system. Because of the high diversity of real-world objects and the large amount of time required to produce a model, work was undertaken by Nguyen and Huang[110] towards using GP to evolve descriptions of models for objects to be recognised. This work used a GA-like representation for evolving symbolic descriptions of objects.

- *Handwriting recognition.* A system to do this has been developed by Andre,[11] using a two-dimensional GA to evolve a hit-miss matrix—a kernel for recognising shapes, in which each pixel may be either "hit", "miss" or "don't care". This was combined with GP to evolve a

program for processing the image with the kernel. For use, the kernel is convolved across the image and the genetic program tests how the image window compares with the kernel. Populations used varied between 500 and 1200 individuals, with run lengths between 200 and 500. Other than the use of a sequencing function, the GP portion of the individuals acted as a decision tree, and operated by moving an automaton across the kernel and examining it. This is similar to GP languages for problems as diverse as controlling an artificial ant[84] or robot,[29] and building a maze.[60] The trial system presented in this work evolved individuals capable of recognising very low resolution digits.

- *Pattern Classification.* Adorni *et al.*[5] have developed a cellular genetic algorithm, in which each individual cell of a two-dimensional population divided into demes of size one, is used to read car registration plates from traffic camera output.

  A different approach was used by Belpaeme,[21] who used strongly-typed GP (with types POINT, SCALAR and IMAGE) to evolve a set of feature detectors. This work used filters operating on images, and involved an unusual—and accidental—solution to the problem of code bloat: since most of the functions were image filters, programs that were nested too deeply ended up filtering too much information out to be useful. A population of 100 was used over 500 generations, and there were 59 fitness cases.

- *Image segmentation.* Bhandarkar and Zhang[23] have used a hybrid GA and simulated annealing to carry out image segmentation. This work fell nearer the field of GAs than GP; individuals consisted of segmented images, as represented by an edge image, membership label array, and region adjacency graph associating the two.

- *Feature location.* Examples of this carried out by GP include the following:

  Johnson *et al.*[77] have used GP to develop a real-time interactive video environment, for pinpointing the location of a person's hands in a bitmap silhouette. This system used typed GP; the functions and terminals in it, which consisted of point operators, feature detectors and point list filters, were deliberately chosen to be weak, in order to avoid overfitting of solutions to the fitness cases, of which there were 46. A population of 500 was used. The

```
if X then do A
  else if Y then do B
    else if Z then do C
      else do D
```

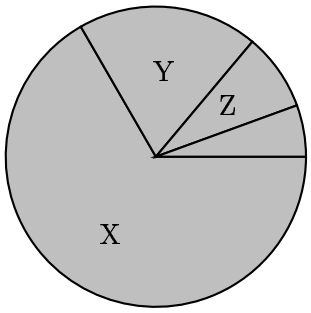


Figure 2.14: Default hierarchies (left): coping with successively less probable contingencies (right).

results obtained with this system were better than those using a hand-coded solution.

Daida *et al.* used GP to identify low-contrast pressure ridges in satellite radar images of Arctic ice.[43] These ridges are difficult for even trained humans to identify, and they take an unacceptably long time to do so. For similar reasons traditional image-processing algorithms have proved inadequate for the task. However, Genetic Programming was able to discover a filter that could identify the features sought. The system used a combination of matrix and real number operations, but was nevertheless untyped; the language consisted of arithmetic operations on 5×5 grids of image data and filtered versions of it (Laplacian, Mean, etc), plus $\Re$. A dynamically increasing number of fitness cases was used—up to 53—to explicitly encourage the formation of default hierarchies (see Figure 2.14), a method by which GP often operates.[84] Runs lasted up to thirty generations, on populations of 357.

A similar system identifying tanks on infrared images, developed by Tackett,[146] achieved better success using GP than using a neural network. This system used simple arithmetic operations on image features and $\Re$, with a population of 500 running for 60 generations, and 2000 fitness cases. An additional advantage of GP over neural nets is that, because they are symbolic, the solutions it delivers may be analysed to see how they work more easily than with neural nets.

Howard *et al.* have developed a GP system for ship identification from satellite images,[71]

for the purposes of monitoring ship movement in the crowded environment of the Dover Straits. This work used genetic programs to process the output of a kernel convolved across the image. The GP language consisted of arithmetic functions acting on perimeter and body variances, and averages of the window moving across the image.

2.2.14 Summary

This section has presented an introduction to the field of Genetic Programming and a survey of applications and techniques that have been used with it. GP has been demonstrated as an efficient search tool in a variety of different fields. Advantages of GP as a method for discovering solutions to problems include:

- It is applicable in a wide variety of different problem areas.
- It is an optimisation method able to skip over obstacles in a fitness landscape.
- It has been used to find solutions which have outperformed those created by humans or discovered by other methods.
- The programs it creates often use different methods to solve problems from those humans use.
- It is symbolic, therefore the programs it produces can be analysed to see how they work.

2.3 Sensor Planning

2.3.1 Introduction to Sensor Planning

Machine Vision describes the process by which a computer is able to analyse an image of some kind and understand what is described by it.[17, 70, 156] The image may be obtained by a number of different types of sensor; two of the most widely used are video cameras and laser stripes.[22, 50, 51] Figure 2.15 shows a typical machine vision setup.

A necessary step before or during the execution of most machine vision tasks is configuring the sensors involved to guarantee optimal sensing conditions. Achieving this configuration is

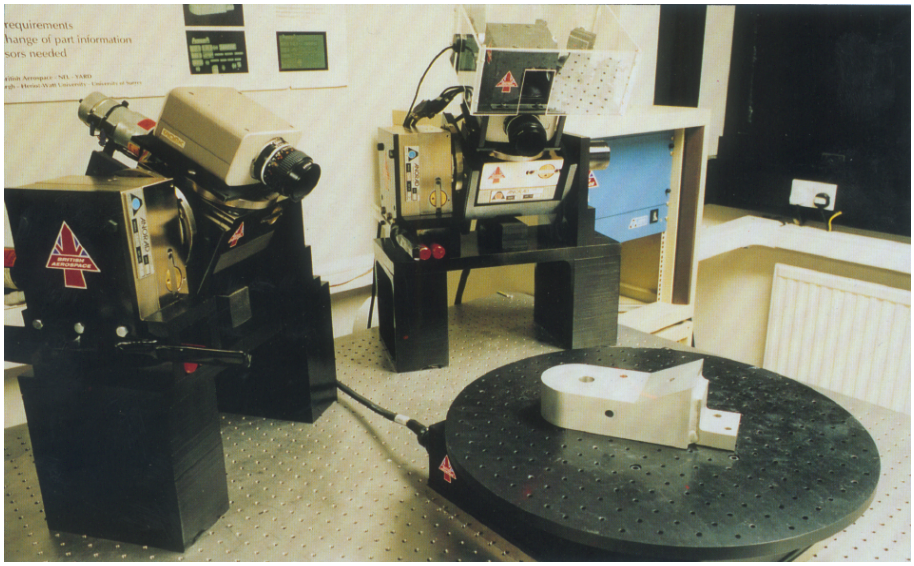


Figure 2.15: A typical machine vision setup.

the subject of the field of Sensor Planning.[148] Configuring these parameters has traditionally been done by hand, in an iterative manner. This is highly labour intensive, and can result in the cost of configuring the system for just one set-up exceeding that of the rest of the system put together.[148] The goal of Sensor Planning is therefore to automate this process.

There are three areas of Machine Vision in which sensor planning plays a part: *scene reconstruction*, *object recognition and localisation*, and *feature detection*.[148]

- Scene reconstruction[92,105] describes the scenario of incrementally constructing a model of an unknown world from observations. Each observation is analysed in order to determine the most useful location for the next in terms of filling in the gaps in the model. This has application in the navigation of autonomous robots, for example.[40]

- Object recognition[30,34,61,67,93,100] and localisation[78,109] takes place in a world consisting of known objects but in unknown poses. Again observations are analysed in order to determine the most useful location for successive observations. Object recognition and localisation systems follow a strategy of hypothesise-and-verify; sensor planning is done in order to verify hypotheses about the sensor's view, to plan where to take a measurement next, and to minimise the number of observations needed to recognise the object.

These techniques can also be used in *dynamic sensor planning*, in which the environment does not remain static throughout the course of the measurements.[1–3]

- Object feature detection by contrast relies exclusively on *a priori* knowledge. Feature detection consists of locating features on known objects in known poses; for example for industrial inspection of parts on an assembly line. For this, sensor and illuminator poses and configurations must be planned in order for the observations to be of the maximum utility. Determining the best location to achieve visibility and good illumination of features also has application in such fields as image synthesis[153] and machining of parts.[148] It is in the field of visibility planning for viewing known objects in known poses that the current study is set, and this area is explored in more detail below.

Sensor planning systems fall into two categories:[4] *dynamic*, in which either the sensor or object to be viewed is in motion, and the viewpoint must be periodically recalculated; and *static*, in which the sensor and object are not in motion, and viewpoint planning is done off-line, before the system is used. This study is concerned with inspection tasks, which fall into the latter category.

The input to the system takes the form of CAD/CAM models of the objects to be viewed. Sub-tasks to be planned include the following:

- *Visibility of target.* In the case of a camera mounted on a robot arm, this includes planning the camera position so its view is not occluded by the arm.[137] Furthermore, the features to be observed on the target have to be pointing at the sensor and not occluded by another part of the object, and have to be aligned relatively close to perpendicular to the sensor's line of sight.[41,154]

- *Illumination of target:*[108,138] The illuminatory devices are also a part of the sensory system. In the case of a video camera, the light sources have to be positioned so the target is adequately lit and shadows are not cast in undesirable places. In the case of a light-stripe sensor, the laser has to be positioned sufficiently far away from the camera so that surface points at different distances from the camera are imaged at different image points, but not so far that parts of the target are visible to the camera but unreachable by the stripe.

From the point of view of visibility, planning is the same whether done for a camera or a light source—for example, the same planning has to be done for two cameras as for a light-stripe and a camera—and one sensor planning system treats sensors and illuminators as undistinguished "generalised sources".[73] However, in addition to visibility there are extra sub-tasks to be achieved with regard to illumination, such as shadow avoidance.[138]

- *Focus*:[41,149] The target has to be in focus. If the vision system is to observe a variety of points in the image, the depth-of-field must be configured so that all of them are within acceptable limits on focus. This is termed the depth of focus of the camera, and is defined by the blur circle to which points off the focal plane project. An acceptable criterion is typically that the diameter of the blur circle is less than that of an image pixel.

- *Field-of-view*:[41,149] The target has to occupy the majority of the field of view. If its projection onto the sensor is too large, not all of the target will be visible at any one time; if it is too small, the usefulness of the image will be restricted.[109,149]

- *Perspective distortion*:[8,9] Many vision systems require a sensor at some distance from the object, so that they may model the image approximately with orthographic projection.

- *Contrast*:[161] Where two object faces meet at a shallow concave angle, it may be difficult for many systems to perceive them separately; for example, shape-by-shading systems might not record a sufficiently high brightness difference either side of the edge to perceive the faces as separate; and if the faces are not close to normal for range-based systems, noise in the signal might overcome the slight change in angle that is recorded.

To achieve these goals, the configurable parameters of the sensor have to be correctly set. These can include:[148]

- *Position*: A sensor's (or light source's) positioning has three translational degrees of freedom, and three (or two in the case of some kinds of light source) rotational degrees of freedom.

- *Focus of camera.*

- *Camera aperture*: The aperture of a camera controls both brightness and depth-of-focus of an image.
- *Focal length*: This can either be a constant property of the lens, in which case it has to be catered for by the planning system, or in the case of zoom lenses a variable parameter of the system.
- *Exposure time.*
- *Geometry of illumination*: The illumination may take the form of a cone of light from a point source; or it may be a stripe or spot of light from a laser. The light distribution may vary in cases where there is more than one source of light.
- *Brightness of illumination* (radiant intensity).
- *Spectral distribution of illumination*: This becomes important when the objects to be viewed have non-uniform colouration.
- *Polarisation of illumination*: For light-stripe sensing of highly specular surfaces (e.g., polished metal), multiple reflection of the incident light-stripe can lead to incorrect detection of the stripe in the image. This can be cured by the use of polarised light;[36, 37, 157] and this, therefore, needs to be planned too.

These parameters are built into sensor models; it is the sensor models along with the object models and task descriptions which are the inputs to a sensor planning system.*

2.3.2 Visibility Space Analysis

The visibility space problem is a complex one and, in its fullest form—that in which the object to be viewed is a generalised three-dimensional shape—one for which no closed-form solution yet exists, though closed-form solutions are known for certain constrained types of three-dimensional object, such as polyhedra[57] or solids of revolution.[47]

*Sensor models must also take into account the performance of sensors under various conditions—for example, range-based detection methods are frequently unable to detect surfaces inclined at a high angle.[73]

Traditionally there have been two approaches taken towards Sensor Planning:[148] *generate-and-test* and *synthesis*. These are considered separately below. Both are forms of search through the space of the parameters described above. Since this parameter space has high dimensionality, both approaches utilise measures to reduce this complexity and render the task feasible. A common measure of this sort is to model the objects to be viewed as polyhedra.

A third approach is that of sensor simulation; this fits in between the spheres of machine vision and computer graphics. It concentrates more on illumination than the other criteria listed above.

An expert systems approach has also been used for sensor planning;[148] however this is not capable of handling the level of spatial detail of the other approaches.

Representing Visibility Spaces

A common representation used for visibility space analysis is that of the *Aspect Graph,*[26,32,52,56,57,72,154] devised by Koenderink and van Doorn, or its close cousin, Characteristic Views.[35] This is a form of viewer-centred representation, the prime advantage of which is its similarity to the data actually detected by sensors.

Aspect graphs carve three-dimensional space up into regions from which line-drawings of observations of the object (aspects) are topologically identical; these regions are separated by boundaries known as visual events. They are generated from boundary representations of the object,[32] in contradistinction to the CAD models often otherwise used,[72,149] which are volumetric. Algorithms have been presented for producing aspect graphs for both 3D spatial (i.e., perspective)[144] and Sphere of View Points (i.e., orthographic) representations[144] (see Section 2.3.2); and for models which are 2D convex and concave polygons,[32] 3D polyhedra,[57,131] solids of revolution[26] and general curved surfaces.[143] Two extensions of aspect graphs have been proposed to take into account both object geometry and sensor position, the *observation graph*[61] and the *asp*.[26]

One disadvantage of aspect graphs is their large size and complexity; they can take some time to calculate and not all of the aspects produced will be useful in practice. For example, some aspects, called *accidental viewpoints*, are visible only along lines, planes or at single points in

space; and others will not be detectable by realistic sensors. An exact aspect graph will include all of these, but algorithms have been presented that avoid calculating such non-utilitarian parts of the aspect graph, and merge aspects which are identical but divided by an accidental viewpoint,[52] to reduce the computational complexity.

Work has also been done on systems which take into account the ability of particular sensors to detect features under various conditions in the construction of aspect graphs.[61,73]

As an example of how aspect graphs may in practice be used, consider the last-cited work, by Ikeuchi and Kanade. In this system, VANTAGE,[74] a CAD model of the object to be viewed is used to construct an aspect graph. Information from the sensor model is used to prune the graph of aspects which are not in practice detectable. The remaining aspects are then arranged hierarchically into an *interpretation tree* according to sequential classification of features of the aspects.

The interpretation tree is then used to automatically generate an object recognition program. The advantages of this approach are that it automates the selection of features to be used for model recognition, a process that is otherwise carried out by hand, and that the burden of computation and thus execution time is placed on the off-line generation of the object recognition program, allowing the on-line execution of that program to be rapid.

Generate-and-Test

The generate-and-test approach consists of quantising three-dimensional space and examining the properties (for instance visibility) of each element of this discretised grid. The overall answer to the problem in hand is then given by grouping points with the same properties.

A common means of doing this is to model sensor positions on the surface of a sphere centred on the object, called the *Sphere of View Points*[26,137,138,155] (SVP). The use of a fixed distance from the object acts to help cut down the high dimensionality of the parameter space (and imposes a resolution constraint). Further constraints are used to simplify the parameter space more; these include modelling the object's surface as being of Lambertian reflectance—i.e. that each surface point appears equally bright from all directions[156]—and assuming that the camera is pointing towards the centre of the sphere, defined as being at the object's centre of gravity.

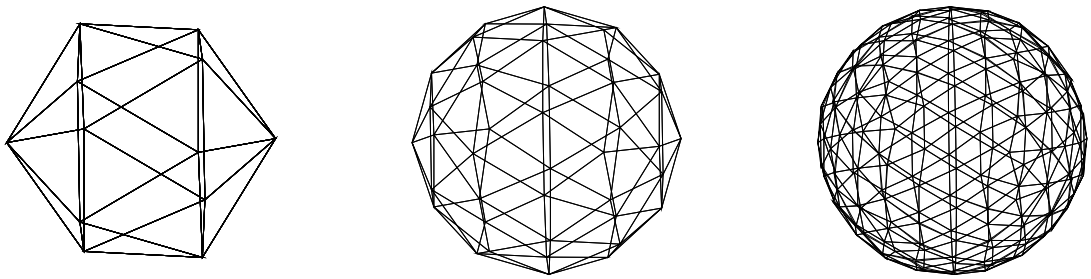
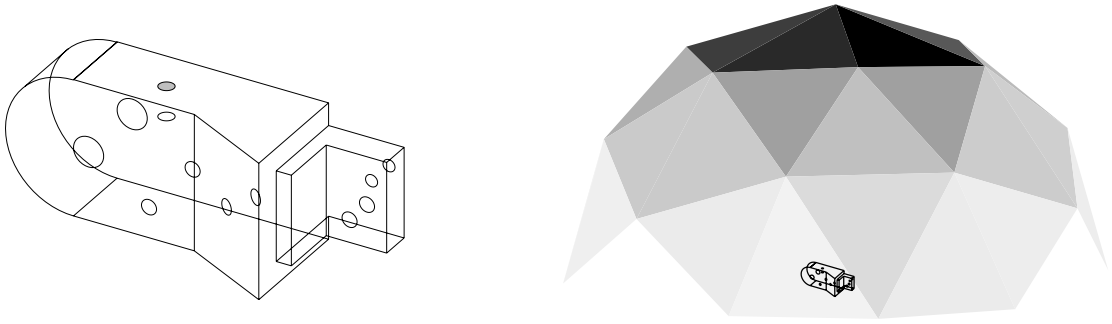


Figure 2.16: Three stages in the tessellation of a sphere

The SVP is implemented by means of a geodesic sphere, created by tessellating an icosahedron hierarchically by means, for example, of a quadtree,[154] to whatever depth of detail is required[137] (see Figure 2.16). Having to know in advance what depth of tessellation is appropriate is one of the drawbacks of the SVP approach. For each facet of the SVP, a ray is projected from the centre of the facet to the centre of the sphere, and this ray is tested for intersection with the objects in the sphere. Again, methods are used to cut down the complexity. The ray is tested for intersection with a bounding box containing the object; if an intersection is found then the facet is split recursively into smaller facets, and each of these is then tested for intersection.

Once all the sphere has been divided in this manner, the facets are grouped into regions according to the task in hand—for instance, regions of visibility and regions of non-visibility (see Figure 2.17). Some systems then rank individual facets on their distance from the group boundary, since facets near this boundary are likely to have less than optimal solutions, which may not be valid positions for viewing under the limitations of real sensors. Positions near the boundary might also suffer from occluded lines of sight, since one of the simplifications involved is for each facet to be represented by the view from its centre, which might not be valid for other regions within that facet.

The advantages of the generate-and-test approach are:[148]



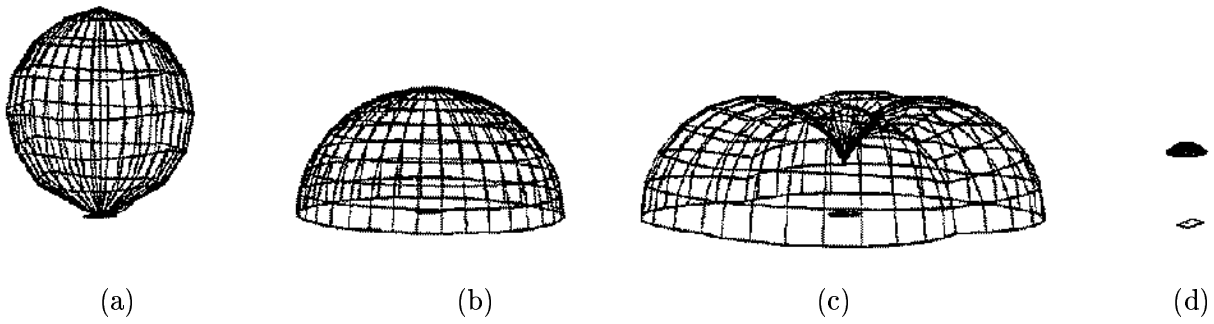
(a) (b)

Figure 2.17:[154] (a) Example object (the British Aerospace widget show in Figure 2.15) with a feature on the upper surface highlighted. (b) Visibility space for this feature, colour-coded according to goodness.

- It is straightforward.
- The hierarchical tessellation of the SVP is an efficient method of performing a search over its surface.
- It can handle planning to view multiple features, by intersecting groups of tessera.
- Illuminability is dealt with using the same representation as visibility.

Its disadvantages are:[148]

- The computational cost.
- The simplifications involved, such as assuming viewpoints to be limited to the centre of facets.
- The computational cost involved in extending the model to multiple features, or features of more complicated geometry than single points.
- The inability to plan for such requirements as resolution or field-of-view with a sphere of fixed size.
- It does not allow for planning of the sensor's internal parameters, and assumes the sensor is pointing directly at the centre of the object.



(a) (b) (c) (d)

Figure 2.18:[41] Illustration of the (a) resolution, (b) field-of-view and (c) depth-of-field constraints for a square, and (d) combined admissibility space for all of them.

Synthesis

The other main approach is synthesis.[27,41,149] This involves finding the exact solution to the problem mathematically rather than following an empirical strategy like the generate-and-test approach. For instance, if the focus constraint is defined in terms of the blur circle, as described above, then the camera must be placed within the distance interval

$$\frac{Daf}{af \pm c(D-f)}$$

where D is the focus distance, a the lens aperture, f the focal length and c the minimum dimension of a camera pixel (as distinct from an image pixel).[41] This then gives rise to a partitioning of three-dimensional space into valid and invalid regions for sensor placement, as depicted in Figure 2.18. Figure 2.18a shows the volume within which resolution of an image taken is acceptably high; Figure 2.18b shows the volume within which the image occupies a large enough field of view. Figure 2.18c shows the volume outside of which both the closest and most distant parts of the object to be viewed fall when imaged within the depth-of-focus as measured in the above equation. Figure 2.18d shows the admissible space for all these criteria combined.

For visibility space planning, each of the potentially occluding model faces can be used to generate an occlusion volume; the inverse of the union of these then gives the visibility space for the feature to be viewed.[148]

In the work of Abrams *et al.*,[4,149] each of the criteria to be planned is expressed in the form

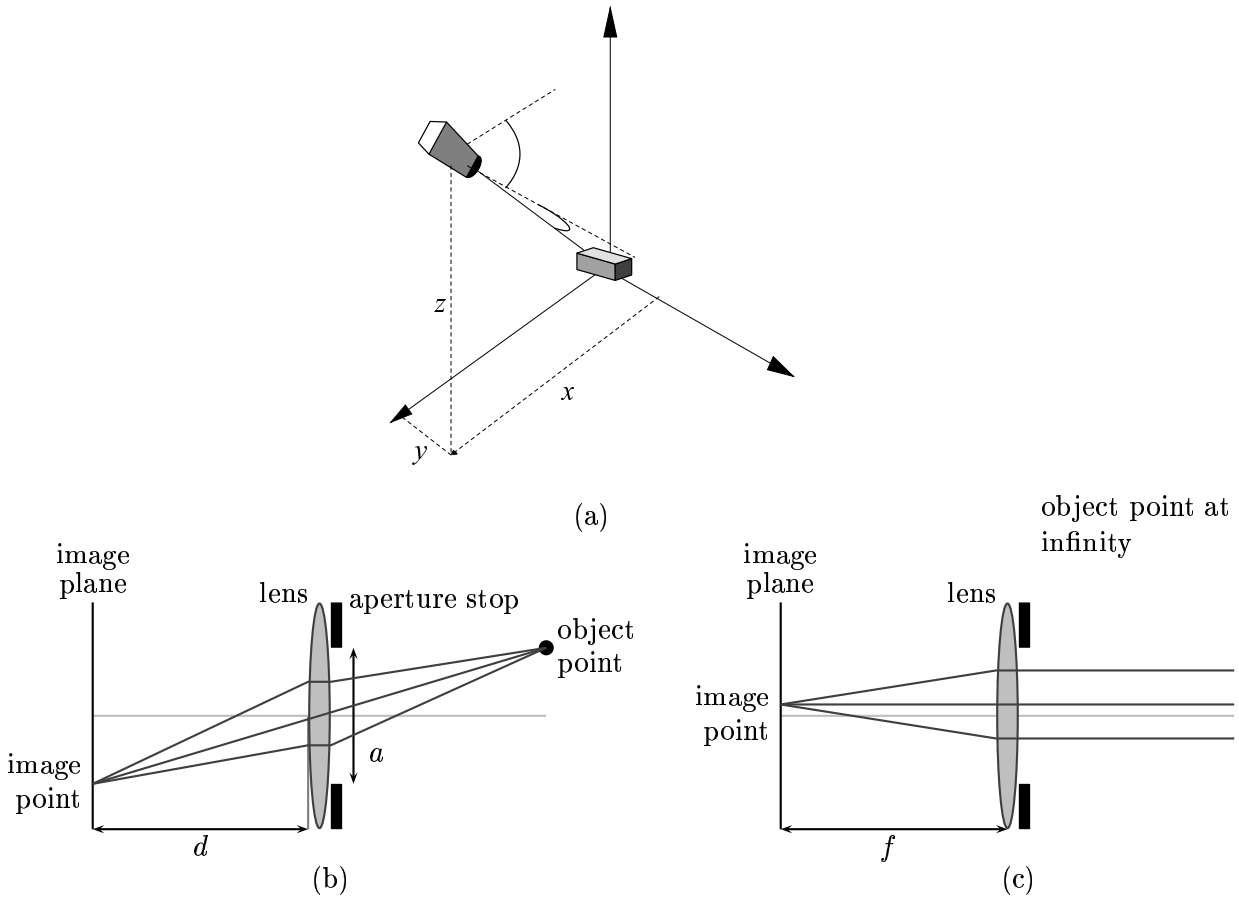


Figure 2.19: Sensor parameters in the generalised viewpoint $(\mathbf{r_0}, \mathbf{v}, d, f, a)$: (a) external parameters; (b), (c) internal parameters.

of an inequality:

$$g_i(x) \geq 0$$

where x is the generalised viewpoint $(\mathbf{r_0}, \mathbf{v}, d, f, a)$ (see Figure 2.19):

$$\begin{aligned}
\mathbf{r_0} &= (x, y, z) \quad \text{(position)} \\
\mathbf{v} &= (\rho, \theta) \quad \text{(orientation)} \\
d &= \text{back nodal point to image plane distance} \\
f &= \text{focal length} \\
a &= \text{aperture}
\end{aligned}$$

Sensor planning is then carried out by optimising the weighted sum of the constraints, i.e.

$$\max_x f(x) \text{ where } f(x) = \sum_{i=0}^{n} \alpha_i g_i(x)$$

A non-linear constrained optimisation routine is used for this.

As with the generate-and-test approach, the robustness of the algorithm is improved by ranking the goodness of visibility spaces according to their distance from the boundaries of acceptable regions.[148]

As when using generate-and-test, it is necessary to reduce the dimensionality of the search space; this is done by constraints and assumptions as before. Some of the parameters may not be considered at all, such as camera orientation,[41, 131] or sensor distance:[131] in the former case, the camera may be assumed to be pointing at the centre of the area of interest, and in the latter, that it is a prespecified distance from the object's centre of gravity, thus leading again to a sphere of viewpoints.

The advantages of the synthesis approach include:

- The ability to use optimisation techniques.
- The ease of extensibility to multiple features.

Its disadvantages include:

- The difficulty of working with function spaces of high dimensionality.
- The formulation of a cost function taking into account all necessary constraints.
- Some of the constraints are non-linear and at points non-differentiable, which can make convergence difficult for general-purpose optimisation algorithms.[4, 149]

2.3.3 Summary

This section has presented an introduction to the field of Sensor Planning. The salient features of this field may be summed up as follows:

Acquisition of useful image or range data for Machine Vision depends on optimal configuration of sensors and illuminators. This can be done both in real-time for active vision systems,

and off-line for systems which are not dynamically responsive. Tasks to be carried out include planning for visibility, illumination, focus, field-of-view, perspective projection and contrast. The main methods of carrying out sensor planning are generate-and-test and synthesis.

2.4 Conclusions

This chapter has presented a review of the fields of Genetic Programming and Sensor Planning. The features of Genetic Programming and visibility space planning which suggest that the former is an appropriate technique for tackling the latter include the following:

- Visibility space planning is a multidimensional optimisation problem; and GP is a technique for efficient exploration of large search spaces.
- Other areas in the surrounding field of Machine Vision have been successfully tackled with GP.[11,43,64,77]
- Though various methods have been used to automatically generate programs to carry out sensor planning, GP has not yet been applied to this problem.
- GP often discovers solutions which are different from those created by humans; and has managed to outperform human-coded solutions in certain areas.[89] Since the identity of these areas cannot be known in advance, the utility of GP in the field of sensor planning can only be determined by carrying out a trial study in this area.

Therefore, it was determined to apply Genetic Programming to the task of visibility space planning: to determine whether the technique was appropriate for the problem, and to investigate to how high a level GP could be used to solve this problem.

Chapter 3

The Untyped System

3.1 Introduction

This chapter describes the first GP system constructed in this study, its objective being the evolution of a program that could calculate visibility areas for two-dimensional polygonal objects.

Section 3.2 describes this problem in more detail. Section 3.3 presents an overview of the system and Section 3.4 describes its component parts in detail. Sections 3.5 and 3.6 discuss in turn the fitness cases and fitness measure that were used, and Section 3.7 describes the results that were obtained with this system.

3.2 Problem Specification

The rationale for investigating whether the visibility space problem could be solved by means of Genetic Programming was given at the end of the previous chapter. Whilst it would theoretically be possible to use a GA to evolve a visibility area for a given set of input data, the objective here was to use GP instead to evolve a program capable of producing an answer for any model and feature specifications.

For an inspection system (a machine vision system intended for checking size and other properties of known objects), the sensor planning system is effectively an algorithm accepting as input a description of the objects to be viewed, a list of the features to be viewed on them and

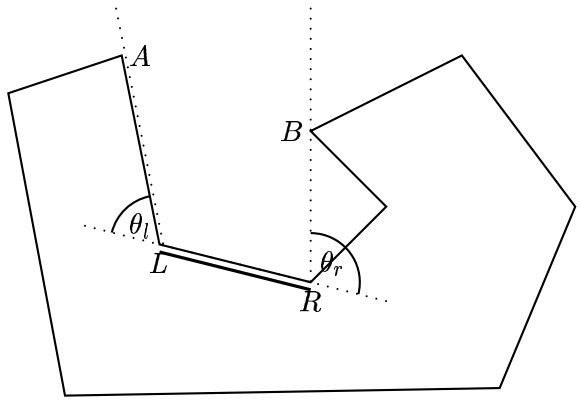


Figure 3.1: Correct solution to the 2D polygons visibility problem. The visibility area is delimited by the feature to be viewed and a line projected from each of its extremities past the model vertex subtending the highest fully occluded elevation in the direction outward from the feature.

a model of the sensor(s). Its output is a description of the volumes of space yielding acceptable sensor configurations.

To investigate the applicability of GP to visibility space analysis, a scaled-down version of this problem was tackled:

To evolve a program capable of solving the visibility space problem for two-dimensional polygonal objects.

That is, given features on a model which are to be examined, to determine the area in which a sensor can be placed in order to view those features. This work can therefore also be applied to prismatic objects in the three-dimensional world: A similar technique has been used to plan positions for security cameras from a two-dimensional room plan.[140]

Example

Figure 3.1 illustrates the correct solution to a 2D visibility problem. LR indicates the model feature for which the visibility space has to be calculated. The region from which this feature can be seen in its entirety is delimited by a partial contour defined by up to four points. The

middle two of these are the extrema of the model feature itself, L and R. The other two are the model vertices which subtend the highest elevations, θ_l and θ_r, with respect to the line LR, as measured in the direction outward from the extremum. Highest elevation here means the maximum angle in which all views to the feature extremum from an arbitrarily large distance are occluded by another part of the model. In this example, the vertex subtending the highest elevation from L is A and not B, because as measured in the outward direction, views to L within $A\hat{L}B$ are not occluded. The highest elevation from L is similarly B.

Depending on whether the lines LA and RB intersect, this space may be finite or infinite. Since in reality machine vision tasks do not have an infinite area to work with, incompletely bounded visibility areas may be rendered finite by closing the contour between LA and RB at the maximum acceptable sensor distance. In a more comprehensive machine vision system, this maximum distance will be further informed by such criteria as resolvability of the feature by the sensor (see Section 2.3.1).

Thus, in general, the visibility space problem for 2D polygons can be broken down into finding the model vertices which subtend the highest angles from the feature extrema.

3.3 Overview of the Untyped System

Genetic Programming systems comprise two parts:

- A problem-independent *kernel*
- Problem-specific code

The kernel deals with the population, creating the programs and applying the genetic operators on the basis of their fitness. The problem-specific code sits on top of this; it defines the fitness function, and the function and terminal sets for the problem. If the system is written in a language where program and data are interchangeable, such as Lisp, these may be no more than a list of operations, such as the ones given in Section 2.2.12. The problem-specific code also provides any other code that might be necessary to execute the genetic programs.

In designing a GP system, the following steps have to be taken:

- Choose the functions and terminals to be used.
- Decide upon the fitness function.
- Decide upon the architecture of the genetic machine.
- Select a suitable kernel.
- Implement a wrapper, if necessary.
- Implement the evaluation routine, if necessary.
- Devise the fitness cases.

The two most important decisions in the design of a system are the choices of functions and terminals, and of the *fitness evaluation* routine. The former of these affects the ability of selection to find a solution; the latter guides the direction that evolution will take. The fitness function can be as simple as summing the program's output across the fitness cases; in more complex systems considerable postprocessing of the output may be necessary. It has been shown[139,160] that a fitness function that does not take into account *a priori* knowledge of the problem can perform worse than a blind random search.

The architecture of the genetic machine describes how many ADFs or other evolvable subroutines there are; and what their arity is.

If the GP system is not implemented in a language where program and data are interchangeable, a simple interpreter must be provided to execute the genetic programs. The *wrapper* is the routine that takes the (usually numerical) output of the genetic program, and transforms it into a form suitable for calculating fitness. This routine is not necessary for simple problems, such as the one described in Section 2.2.12.

For a complex system such as this one, in which a program's fitness is an abstract concept not directly related to the result it returns, a more sophisticated wrapper is required. In this system, the distinction between the wrapper and the fitness evaluation routine is rather blurred.

Finally, the parameters to the GP paradigm, given in Table 2.2 on p. 33, must be optimised.

3.4 System Specifications

The first task in implementing a Genetic Programming system is to select a suitable kernel. This work being primarily an application of Genetic Programming, rather than research into the technique itself, it was felt unnecessary to write a Genetic Programming system from scratch.

There are a variety of kernels available, both public domain and commercial, ranging from specific applications to complete and general GP systems, written in languages including Lisp, Prolog, C, C++, Smalltalk, Mathematica and Java.

Systems considered for use in this work included the following:

- lilgp is Koza's original GP implementation in Common Lisp, given in his book *Genetic Programming*,[84] and extended to include Automatically Defined Functions in the following book.[87] The advantages of using Lisp are that it is itself tree-based, which makes parse trees easy to represent, and that program and data are interchangeable in it. That way programs can first be assembled by list manipulation, then be executed by the built-in function `eval` without need for a specially-written interpreter. The disadvantage of Lisp is the fact it is interpreted rather than compiled, resulting in slow running of the system. For simple problems this is tolerable, but the visibility space problem was expected to be a complex one, with programs that took a while to run; so it was preferred that a compiled language be used.

- lilgp had previously been ported to Emacs Lisp by myself.[60] The advantage of Emacs Lisp is the large number of operations readily available for display and user interfacing, and this system includes a window-based user interface. The disadvantage again is the inability to compile code.

- SGPC (Simple GP in C) has much the same functionality as lilgp, with a few extensions, such as the ability to use demes. Its primary advantage is its speed.

- Nordin has devised a system to use genetic programs in machine code.[112] Whilst such a system would not need interpretation at all, running at the speed of compiled programs,

machine code is restrictive in the operations that can be carried out. For a high-level task such as the one to be carried out here, it was deemed not to be appropriate.

- Regarding public-domain GP systems in other languages: At the time of commencement of this work, I was not fluent in any other language for which a GP system was available. Since the choice of GP system would then be coupled with the choice of language to learn, I decided to choose a system in C++, as that was probably the most widely used of the available languages.

GPC++[158] is a library of routines written in C++ implementing a GP kernel. Being object-oriented, the addition of problem-specific code would be straightforward, as such code could be kept unentangled from the kernel, and full mastery of the kernel code's workings would not be required in order to implement the problem-specific code. This kernel was therefore chosen for this system.

GPC++ allows for tournament and fitness proportionate selection, demetic populations, and swap and shrink mutation operations. Though many other variations on the GP paradigm have been presented, as discussed in the last chapter, most of the applications reported in Section 2.2.13 have used a fairly basic set of operations. Since GPC++ is object-oriented, if additional operations should become necessary, it would be possible to implement them using derived classes with little disruption to the system.

3.4.1 Model Representation

Depending on the task in hand, the object to be viewed in a sensor planning problem can be represented by either a *volumetric* or a *boundary* representation. Any representation facilitates the performing of certain operations, but at the cost of the efficiency of others; for example a Constructive Solid Geometry model in which objects are described by boolean operations on primitives will not be appropriate for recognising human faces, which exhibit a high degree of variation.[156] Typically input will be in the form of a CAD model, and will be preprocessed into a format appropriate for sensor planning.[72,154] Various formats were considered:

- CAD models are often specified using Constructive Solid Geometry (CSG) modelling,[76]

which uses boolean set operations—union, intersection and difference—to combine a few simple primitives—cones, cylinders, cuboids and spheres. The drawback of CSG representation is that arbitrarily curved objects cannot be represented. This was not to be desired, since this 2D system was intended to act as a model for more complex systems. Also, the task of visibility space planning, as shown in Figure 3.1 on p. 56, is primarily concerned with lines of sight past parts of the model. For this a boundary representation would be more appropriate than a volumetric one.

- Spatial occupancy models[76] allow the representation of arbitrary shapes. The simplest spatial occupancy representation is by using voxels in 3D, and pixels in 2D. This is extremely inefficient; a more parsimonious representation is by the use of octrees in 3D or quadtrees in 2D, and such a representation has been used in sensor planning systems.[38] Nevertheless, octrees are still highly inefficient, and do not capture aspects of the geometry explicitly, such as surface gradients, as is necessary for this problem.

- Other work,[131] including work carried out in our Department,[154,155] represents objects by decomposition of a CAD model into planar patches and faceted surfaces. This representation would be appropriate for a 3D system; however it is unnecessarily complex for a 2D one.

For the system under design, it was decided to use a boundary representation consisting of a list of halfplanes (modelling the use of halfspaces in three dimensions). The boundary of the model would thus be specified by the line segments connecting the intersections of successive halfplanes in the list; the use of halfplanes rather than a contour in 2D space would allow a simple way of determining whether a point was inside the model or not.

The representation would be object-centred, as is usual for a system in which the location of the object is known but that of the sensor is not. Since the system would be used to delimit potential sensor locations rather than anticipating the view a sensor would obtain from those positions, this system would use orthogonal geometry rather than perspective projection.

The halfplanes would be specified as (a, b, c) triplets, where for a point (x, y):

$$ax + by + c \geq 0$$

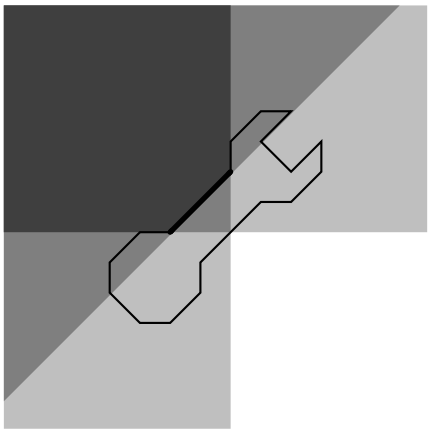


Figure 3.2: Illustration of the use of halfplanes to delimit the answer.

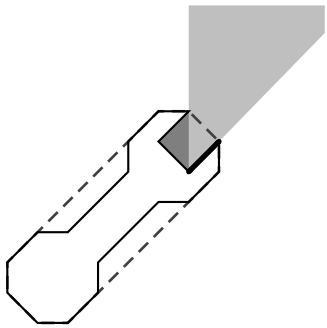


Figure 3.3: Concavities in the visibility space (medium grey) are only to be found within the convex hull of the model (dashed contour).

The output of the system would be a similar series of halfplanes, the intersection of which would describe the visibility space for the required feature(s) (see Figure 3.2). An intersection of halfplanes can only describe a convex figure; and whilst visibility spaces can include concave areas, such concavities can only occur within the convex hull of the object to be viewed (see Figure 3.3). Since in a vision system sensors would not realistically be placed within that convex hull, it seemed reasonable that the output produced should be interpreted as meaningful only outside of the convex hull.

3.4.2 The Genetic Virtual Machine

Overview

Though it is possible to create quite high-level languages for execution of genetic programs, most of the ones presented in Section 2.2.13 were fairly low-level, using untyped GP and a simple set of arithmetic operations acting upon numerical data. Low-level functions and terminals cannot embody *a priori* knowledge about the problem domain; the discovery of solutions can therefore be attributed wholly to the evolutionary process. Furthermore, low-level functions and terminals allow the virtual machine to be more general, as demonstrated by the GP Problem

Solver (GPPS).[91] This was constructed by Koza to counter claims that the success of GP was due to its incorporation of problem-specific knowledge in the choice of functions and terminals. The GPPS has been used to solve diverse problems, using generic low-level function and terminal sets based solely on numerical manipulation.

Therefore, it was decided that the system described here would use a floating-point representation, and low-level, predominantly arithmetical operations.

Many simple problems can be solved using a system with no state memory; however for complex tasks this facility is highly beneficial[126, 150] or even crucial. The simplest GP systems have no ability to represent a program's execution state, but Teller[150] has extended the GP paradigm to include this, in the form of an indexed memory. The visibility space problem was adjudged to be a sufficiently complex task to merit the inclusion of a state memory; it was decided that the system would include flat arrays of floating-point numbers as memories. The intention was that groups of three successive memory elements would be interpreted as triplets (a, b, c) describing the halfplane $ax + by + c \geq 0$, and groups of two successive elements would be interpreted as duplets (x, y) describing a point. Only locations 0–49 would be writable; and to achieve genetic closure, attempts to read locations out of range would return -1.

In order to access an indexed memory by location, the genetic machine's sole type must be a numerical one. A floating-point representation could in theory be used, but since it was expected that most operations would take place on duplets or triplets of numbers, it was decided instead to limit the type of language elements to integers. This thus created a functional separation between the floating-point memories, which would be used for arithmetical operations, and the return values of program subtrees, which would be used for indexing memory.

Various possibilities were considered for how functions might be able to return oligopartite data, such as the triplets and duplets mentioned above, while still only returning a single integer each. These included the use of a memory stack, with each function returning the number of items deposited on the stack. Such a system would need robust stack management since most evolved programs could not be relied upon to manage the stack properly. Bill Langdon has shown[28, 95] that Genetic Programming is capable of evolving routines to manage stacks, queues and other such data structures; however in those studies the evolution of these was the overall

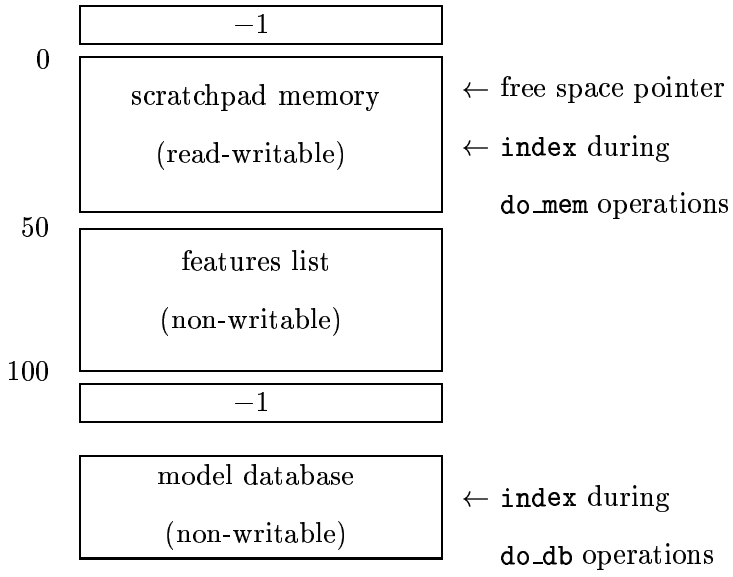


Figure 3.4: Architecture of the genetic machine in the untyped system.

aim of the experiments, and the criterion on which programs' fitness was measured. In the present case, the burden of evolving such routines would be in addition to that of solving the visibility space problem. Consequently a simpler system was implemented using a scratchpad memory, the return value of memory-manipulating operations being a pointer to the data block used by them.

Two iteration routines were provided, one for iterating over the model, and one for iterating over the scratchpad memory. Both of these iteration routines were 'capped' to prevent nested iteration from occurring.[84] This was because the system did not include penalisation for computational complexity, and without the use of these caps the system would have spent a long time running programs containing multiply-nested `dos`, most of which would not have done anything useful.

Architecture

The architecture of the virtual machine on which the genetic programs ran is given in Figure 3.4. The initial design consisted of two numerical memories. One, a read-only memory, contained the model; the second consisted of 100 numbered locations. Locations 0–49 were writable, and

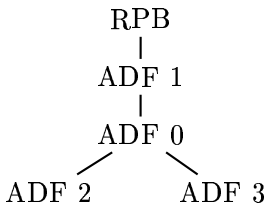


Figure 3.5: Hierarchy of ADFs in the untyped system: Each ADF can call only those beneath it in the tree.

were intended for use as a scratchpad memory; locations 50–99 were read-only and specified the features to be observed on the model. A single floating-point number could be stored in each location.

The choice of these sizes would allow up to 16 triplets in the scratchpad memory both for general workspace and for the program to return the list of answer halfplanes. If this turned out to be a poor choice of number, the size of the memory could be adjusted in the light of run results.

A free-space pointer was included in the genetic machine to ease memory manipulation: Functions would have the option of having their answers written to the free-space pointer, which would be increased by the size of the data block written. Should other functions write to absolute addresses beyond the free-space pointer, this would have the side-effect of resetting that pointer to the address following the highest one written to.

ADFs (Automatically Defined Functions)[87] were implemented in a hierarchical manner, as shown in Figure 3.5. This was intended to encourage hierarchical calling of the ADFs. In addition, the RPB and ADF 1 were provided with full function and terminal sets, ADF 0's function set lacked the operations for carrying out iterations and calling the built-in higher-level functions, and the low-level ADFs had a restricted function and terminal set intended only for doing numerical calculations.

Syntax and semantics

Since this problem would involve arithmetical manipulation of numbers describing halfplanes and points, the function and terminal sets chosen contained the basic arithmetical operations.

```
<genome> ::=
   (defun rpb ()           <subtree>)
   (defun adf0 (arg0)      <subtree>)
   (defun adf1 (arg0)      <subtree>)
   (defun adf2 (arg0 arg1) <subtree>)
   (defun adf3 (arg0 arg1) <subtree>)

<subtree> ::= <terminal>
          | (<function> <arguments>)

<arguments> ::= <argument> | <argument> <arguments>
<argument> ::= <subtree> | #<subtree>

<terminal> ::= <integer> | arg0
             | arg0.a | arg0.b | arg0.c
             | arg1.a | arg1.b | arg1.c
             | fspc.a | fspc.b | fspc.c

<integer> ::= -1 |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |  8 |  9 |
              10 | 12 | 15 | 18 | 21 | 24 | 27 | 30 | 33 | 36 | 39 |
              42 | 45 | 48 | 50 | 53 | 56 | 59 | 62 | 65 | 68 | 71 |
              74 | 77 | 80 | 83 | 86 | 89 | 92 | 95 | 98

<function> ::= free | index | + | - | * | % | = | "<" | and | or | not
             | prog2 | prog3 | copy | if | do_db | do_mem
             | adf0 | adf1 | adf2 | adf3 | evaluate | intersect | in_half_space
```

Table 3.1: Definition of the untyped system's genetic machine in Backus Naur Form.

Programming operations, both sequencing and branching ones, were also used; and boolean operations were included so that complex predicates could be built up. Functions were provided for manipulating the system's memory, and loop operations for iterating over the memory and model. Table 3.1 provides a definition of the genetic machine in Backus Naur Form, and Table 3.2 shows the functions and terminals defining the genetic machine in their syntactic contexts.

Programs are represented in a pseudo-Lisp fashion; this is because Lisp, being an overtly tree-based language, is well-adapted for representing parse trees.

For reasons of genetic closure (see Section 2.2.4), all functions had to be able to handle all possible outputs from all of the functions and terminals as potential inputs. Consequently every operation was made to return an integer within the range $-4 \ldots 99$. Only one operation returned a value less than -1; the values from -4 to -2 are uniquely used by `index`, as shall be described in detail, to access the current iteration, within the loop bodies of the iteration functions.

Terminals:	
-1...10; 12, 15...48; 50, 53...98	
Arithmetic operations:	
+, -, *, % (#VAL, #VAL)	→ VAL
+, -, *, % (ADR, #VAL, #OVERWRITE-FLAG)	→ ADR
+, -, *, % (DEST, SRC, #OVERWRITE-FLAG, #NO.)	→ ADR
Comparison operations:	
and, or (#VAL, #VAL)	→ BOOLEAN
not (#VAL)	→ BOOLEAN
<, = (ADR, ADR)	→ BOOLEAN
Program sequencing operations:	
if (#PREDICATE, ROUTINE1, ROUTINE2)	→ RESULT
prog2 (ROUTINE1, ROUTINE2)	→ ROUTINE2 RESULT
prog3 (ROUTINE1, ROUTINE2, ROUTINE3)	→ ROUTINE3 RESULT
Memory manipulation operations:	
copy (SRC, DEST, #NO)	→ ADR
free (#VAL)	→ VAL
Iteration operations:	
do_mem (ROUTINE, #START, #END, #STEP)	→ −1 or 1
do_db (ROUTINE, #STOPFLAG, #RETURN_ADR)	→ −1 or ADR
index (#OFFSET)	→ −4 . . . − 2
Automatically Defined Functions:	
ADF0 (#arg0)	→ RESULT
ADF1 (#arg0)	→ RESULT
ADF2 (#arg0, #arg1)	→ RESULT
ADF3 (#arg0, #arg1)	→ RESULT
Arguments to the ADFs:	
arg0	→ VAL
arg0.a, arg0.b, arg0.c, arg1.a, arg1.b, arg1.c	→ ADR
fspc.a, fspc.b, fspc.c	→ ADR
Built-in higher-level functions:	
evaluate (HALFPLANE, POINT)	→ ADR
in_half_space (HALFPLANE, POINT)	→ BOOLEAN
intersect (HALFPLANE1, HALFPLANE2)	→ ADR

Program Branch	Node															
	integers, +, -, *, %, and, or, not, <, copy, if, prog2, prog3, =	free	index	do_db	do_mem	ADF0	ADF1	ADF2	ADF3	intersect	in_half_space	eval.	arg0	$arg0_n$	$arg1_n$	$fspc_n$
RPB	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				
ADF 0	✓	✓	✓				✓	✓	✓				✓			
ADF 1	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓		✓			
ADF 2	✓	✓												✓	✓	✓
ADF 3	✓	✓												✓	✓	✓

Table 3.2: Functions and terminals used in the untyped system's genetic machine

Though all operands in this system are integers, some are to be interpreted as absolute values, while others are to be interpreted as memory locations. To distinguish between these, the former are prefixed by a hash symbol (`#`).

For the interpretation of operands as boolean values, all positive values are taken as *true*, and all nonpositive ones as *false*.

The following restrictions apply to all operations:

- Arithmetical Closure:

  if *result* < -1 **then return** -1

  if *result* > 99 **then return** 99

- Result address:

  Certain operations calculate the address to place their result in as follows:

  If the overwrite flag is set, the result is written to the requested number of memory locations commencing at the address specified; if this result block extends beyond the free-space pointer, the free-space pointer is altered to point beyond it.

  If the overwrite flag is not set, the result block is written to the free-space pointer, which is advanced by the requested number of locations.

A description of the semantics of the functions and terminals follows.

- Numerical Terminals: `-1...10; 12, 15...48; 50, 53...98`

  These evaluate to their numerical values. Because superfluous functions and terminals degrade the probability of discovery of correct solutions in a linear fashion,[84] it was decided not to provide terminals to access every location in the indexed memory individually. Consequently, sufficient terminals were provided to access the first ten locations individually, and thereafter every third location up to the end of writable memory (location 49). By only providing every third location, this would, it was hoped, encourage evolution to treat the memory as triplets.

Terminals were also provided to access the (a, b, c) triplets in the read-only memory from location 50 up to 98.

Access to the intermediate values in each triplet would be provided by use of the lower ADFs (q.v.).

- Arithmetic operations: `+`, `-`, `*`, `%`

  `%` signifies a protected division operation:

  if *denominator* $= 0$ **then return** 0 **else return** *numerator* $\div$ *denominator*

  - Two-valued Arithmetic operations: `<op> (#VAL, #VAL)`

    These operate on absolute-valued inputs, and return the result converted to an integer.

  - Three-valued Arithmetic operations: `<op> (ADR, #VAL, #OVERWRITE-FLAG)`

    These operate on the contents of the memory value specified in the first parameter, and the absolute value specified in the second. Depending on the value of the third argument, the result is written to either the same location as the first argument, or to the free-space pointer, which is then incremented, as described above. The value returned is the location of the answer cell.

  - Four-valued Arithmetic operations: `<op> (DEST, SRC, #OVERWRITE-FLAG, #NO.)`

    These operate on two blocks of memory locations, starting at the locations given in the first and second argument respectively, and continuing from there for the number of locations given in the fourth argument. Depending on the value of the third argument, the result is written to either the second data block, or to a data block allocated at the free-space pointer, as described above.

    The block arithmetic is carried out in a manner such that if the two memory blocks overlap, the values used are those present before the operation began.

    The value returned is the location of the start of the answer block.

- Boolean operations:

  `and (#VAL, #VAL)`

```
or (#VAL, #VAL)
not (#VAL)
< (ADR, ADR)
= (ADR, ADR)
```

and, **or** and **not** act in the expected manner on their absolute-valued arguments, interpreting the integers as Boolean values in the manner described above; **<** and **=** compare the contents of the two memory locations referenced in their arguments. These functions return 0 or 1 to signify *false* or *true*.

- Program sequencing operations:

```
if (PREDICATE, ROUTINE1, ROUTINE2)
prog2 (ROUTINE1, ROUTINE2)
prog3 (ROUTINE1, ROUTINE2, ROUTINE3)
```

if evaluates either its second or its third argument depending on the boolean interpretation of its first argument.

prog2 and **prog3** evaluate each of their arguments in turn, discarding the values of all bar the last, which is returned.

- Memory manipulation operations:

```
copy (SRC, DEST, #NO)
free (#VAL)
```

copy copies the number of memory cells specified in its third argument from the location in its first to the location in its second. Like the four-valued arithmetic functions, if the source and destination blocks overlap, all the source locations are read before any are overwritten.

If the source pointer is **index** when no iteration is in progress, a block of -1s is copied; if the destination pointer is -1, the destination block is set to the free space pointer.

free lowers the value of the free-space pointer by its argument, if this argument is positive. The pointer is prevented from moving beneath 0.

- Iteration operations:

  `do_mem (ROUTINE, #START, #END, #STEP)`

  `do_db (ROUTINE, #STOPFLAG, #RETURN_ADR)`

  `index (#OFFSET)`

  `do_mem` iterates over the scratchpad memory. The iteration starts at the value specified in its second argument, and ends at the value specified in its third, proceeding in steps of the size indicated in its fourth. In each iteration, the first argument is evaluated. No iteration takes place if the start memory address is not less than the end one, or if a model iteration is currently in progress. The function returns 1 if the iterations have been carried out, and -1 if they have not.

  `do_db` iterates over the model in steps of three, i.e. each iteration is aligned with the start of a triplet. In each iteration, the first argument is evaluated. The intention of the second argument had been for iteration to cease should it evaluate to *false*; however due to an error in implementation this argument is only evaluated once, before the first iteration. This results in programs iterating the whole way around the model rather than breaking off part-way through; however this was unlikely to have had any effect on the success of evolution. (This error was not discovered until after the work had moved on to the next system.) No iteration takes place if a memory iteration is currently in progress.

  If the iterations have been carried out, the model triplet used in the last iteration is copied to either the address given in the third argument, or the free-space pointer if this address is negative, and the function returns 1. If the iterations have not been carried out, the function returns -1.

  In both these cases, `index` is used to access the loop pointer as follows:

  if *argument* < 0 **then let** *argument* $:= 0$

  if *argument* > 2 **then let** *argument* $:= 2$

  return $-2 -$ *argument*

  Since the values $-4 \ldots -2$ are not produced by any other function or terminal, they can be

used uniquely for this purpose. They access the component parts of the triplet of memory or model indexed by the loop pointer.

- Automatically Defined Functions:

  `ADF0 (arg0)`

  `ADF1 (arg0)`

  `arg0`

  These call the Automatically Defined Functions, assigning to `arg0` the result of evaluating their argument (which is evaluated once, before the ADF is called). The ADF call returns the value returned by the ADF program branch.

- Special ADFs:

  `ADF2 (arg0, arg1)`

  `ADF3 (arg0, arg1)`

  `arg0`, `arg1`, `fspc`

  These call the special low-level ADFs intended for use for doing numerical calculations. Their arguments are evaluated before entering the ADF, then accessed as follows:

  `arg0.a` $\rightarrow$ *arg0*

  `arg0.b` $\rightarrow$ *arg0* + 1

  `arg0.c` $\rightarrow$ *arg0* + 2

  where *arg0* is the value to which `arg0` was bound on entering the ADF. `arg1.a`, etc, work the same way. If these were used in contexts in which they would be interpreted as addresses, they could therefore be used to access the individual parts of an (a, b, c) triplet or (x, y) duplet.

  There was also a terminal `fspc`, which could be accessed in the same way; it referenced the three memory locations commencing at the cell the free-space pointer referenced at the time the ADF was called.

- Built-in higher-level functions:

  `evaluate (HALFPLANE, POINT)`

`in_half_space (HALFPLANE, POINT)`

`intersect (HALFPLANE1, HALFPLANE2)`

These are non-evolvable ADFs written in the language of the system. It was intended that if the system proved able to solve the visibility space problem with the help of these functions, they could be turned into evolvable ADFs to see whether the system was capable of discovering these functionalities on its own. The code for them is given in Appendix A.2.1.

If `arg0` points to a halfplane (a, b, c) and `arg1` to a point (x, y), `evaluate` returns the value of $ax + by + c$.

`in_half_space` returns 1 if $ax + by + c \geq 0$, and zero otherwise.

`intersect` returns a pointer to the start of an answer block allocated at the free-space pointer, giving the intersection coordinates of the two halfplanes pointed at by its arguments.

All three of these functions consume three locations of memory at the free-space pointer; as implemented they do not release that memory afterwards.

Worked example

A sample program evolved by this system is shown in Program 3.1.

```
1.  (defun rpb () (adf1 #37))
2.  (defun adf0 (arg0) (% #72 #51))
3.  (defun adf1 (arg0)
4.    (- (- #87 #62)
5.       (+ 50 #61 #40)
6.       #2
7.       #29))
8.  (defun adf2 (arg0 arg1) (- #arg0.0 #arg1.1))
9.  (defun adf3 (arg0 arg1) (not #arg1.0))
```

Program 3.1: Sample evolved program solving the visibility space problem for convex polygons.

The evaluation of this program proceeds as follows:

Line 1 assigns the value 37 to `arg0` (which is not used by this program) and enters ADF 1. (ADF 0 is not used.)

Lines 4–7 carry out a memory block subtraction. The four arguments, which are evaluated first, are as follows:

- The result of the inner subtraction in line 4: $87 - 62 = 25$. This is used as the address of the destination block.
- The memory addition in line 5 adds 61 to the contents of memory location 50 and, because the third argument (40) is positive, attempts to overwrite location 50 with the result. Because location 50 is read-only, the result is discarded. The addition returns 50, which is used as the source address for the outer subtraction.
- The third argument is a flag indicating whether to write the result of the subtraction to the free-space pointer or the destination block. Its value is positive (2), so the result is written to the destination block.
- The fourth argument indicates that the number of memory cells on which to carry out the subtraction is 29.

Once all the arguments have been evaluated, the subtraction is carried out as follows: The contents of each memory cell from location 50 to location 78 is subtracted from that of the corresponding cell in the range 25 to 53, overwriting the latter if this address falls in read-write memory (locations 0–49).

The subtraction operation returns the value 25. The ADF which encloses it then also returns the value 25, and the program returns the value 25, indicating the location of the answerblock.

Since the read-write memory is initialised to contain zeroes, and location 50 is the start of the features list, the effect of this program is to copy across the feature halfplanes, inverting each component, i.e. converting

$$\{(-3, 5, 0), (-3, -3, 6)\}$$

to:

$$\{(3, -5, 0), (3, 3, -6)\}$$

Memory cell index	0	1	2	3	4	5	6	7	8	9	
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
20	0.00	0.00	0.00	0.00	0.00	⇒ 3.00 ⇐	−5.00	0.00	3.00	3.00	Scratchpad memory (writable)
30	−6.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
40	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
50	−3.00	5.00	0.00	−3.00	−3.00	6.00	0.00	0.00	0.00	0.00	
60	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
70	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	Features list (read-only)
80	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
90	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
	−3.00	−3.00	6.00	2.00	4.00	0.00	−3.00	5.00	0.00	4.00	Model (read-only)
	−4.00	8.00	0.00	−2.00	2.00	0.00	0.00	0.00	0.00	0.00	

Table 3.3: Contents of state memories at the end of execution of Program 3.1; due to space restriction, the linear memories are presented in tabular form. The box indicates the free space pointer; the start of the returned answerblock, memory cell 25, is indicated with arrows.

The result of this is to return the complement of the feature halfplanes, which is the solution to the visibility space problem for convex polygons. This result is, of course, trivial.

Table 3.3 shows the state of the system's memories at the termination of execution.

A hand-crafted solution solving the visibility space problem for a single feature on a concave polygon is given in Program 3.2*. The program operates as follows:

At `A` the inverse of the feature halfplane is added to the answer, then the program iterates around the model between `B1` and `B2` to identify the feature halfplane and record its index. `C1` to `C2` is a second iteration around the model, setting up variables that will be needed for the main iteration in the clockwise direction.

The "main iteration" actually consists of two iterations, one (`D1`–`D2`) from the model feature until the end of the data list comprising the model; and a second (`E1`–`E2`) from the start of the model list back to the feature. During each iteration step, the elevation subtended by the current model point (the intersection of the current and previous halfplanes) is measured, and

*This program has been simplified: the version printed here is unable to deal with feature specifications which lie at the ends of the list of model halfplanes. The full program taking this contingency into account would be considerably larger still.

```
(defun rpb ()
  (prog3
    (prog3
A:    (- 50 33 #1 #3)
      (copy 4 30 1)
      (+ 5 #1 #1))
    (prog3
B1:   (do_db
        (prog3
          (if (and (and (= 18 50) (= (+ 18 +1) 51))
                   (= (+ 18 2) 52))
              (copy (+ 15 2) 8))
          (+ (+ 15 2) #1 #1))
          (copy (index 0) 18 #3) 1 50)
B2:     1 50)
      (prog3
        (prog2
          (copy 4 (+ 15 2) #1)
C1:       (do_db
            (prog2
              (prog2
                (copy 18 21 #3)
                (copy (index 0) 18 #3))
              (prog2
                (if (< (+ 15 2)
                       (prog2 (copy 8 18 #1) (+ 18 #2 #1)))
                  (copy (intersect 18 21) 15 2) 0)
                (+ (+ 15 2) #1 #1) 0))
C2:         1 50))
        (prog2
          (adf0 0)
          (prog3 (- 4 12 #1 #1) (- 4 (+ 12 1) #1 #1)
                 (- 4 (+ 12 2) #1 #1)))
        (prog3
          (prog2
            (copy (copy 4 (+ 15 2) #1) 0)
D1:         (do_db
              (prog3
                (prog2
                  (copy 18 21 #3)
                  (copy (index 0) 18 #3))
F1a:              (if (not (< (+ 15 2)
                              (prog2 (copy 8 18 #1)
                                     (+ 18 #2 #1))))
F1b:                (adf1 0) 0)
                (+ (+ 15 2) #1 #1))
D2:           1 50))
          (prog2
            (copy 4 (+ 15 2) #1)
E1:         (do_db
              (prog3
                (prog2
                  (copy 18 21 #3)
                  (copy (index 0) 18 #3))
F2:               (if (< (+ 15 2) 8) (adf1 0) 0)
                  (+ (+ 15 2) #1 #1))
E2:           1 50))
G:        (if (> 3 0) (copy (adf3 15 6)
                            (+ 30 #3 #1) 3) 0)))
```

```
(prog3
  (prog2
    (copy 4 (+ 15 2) #1)
    (do_db
      (prog2
        (prog2
          (copy 18 21 #3)
          (copy (index 0) 18 #3))
        (prog2
          (if (< (+ 15 2) 8)
                 (copy (intersect 18 21)
                       15 2)
                 0)
          (+ (+ 15 2) #1 #1) 0))
      1 50))
  (adf0 0)
  (prog3
    (prog2
      (- (copy
           (copy
              (copy
                 (copy
                    (copy 4 (+ 15 2) #1)
                    (+ 9 2) #1)
                 3 #1)
              1 #1)
           0 #1)
        0 #2 #1)
      (do_db
        (prog3
          (prog2
            (copy 18 21 #3)
            (copy (index 0) 18 #3))
          (if (not (< (+ 15 2)
                      (prog2 (copy 8 18 #1)
                             (+ 18 #2 #1))))
            (adf1 0) 0)
          (+ (+ 15 2) #1 #1))
        1 50))
      (prog2
        (copy 4 (+ 15 2) #1)
        (do_db
          (prog3
            (prog2
              (copy 18 21 #3)
              (copy (index 0) 18 #3))
            (if (< (+ 15 2) 8) (adf1 0) 0)
            (+ (+ 15 2) #1 #1))
          1 50))
        (if (> 3 0)
            (copy (adf3 6 15) (+ 30 #3 #1) 3)
            0)))
  (if (= (+ 9 2) 5) 95 27)))

(defun adf0 (arg0)
  (prog3
    (- (* (copy 33 12 3) 15 #1 #1)
       (* (+ 12 2) 15 #1 #1))
    (copy (+ 33 1) 12)
    (- 4 (copy 33 (+ 12 1)) #1 #1)))
```

Program 3.2: Annotated hand-crafted solution for the complete visibility problem.

```
(defun adf1 (arg0)
  (prog3
    (prog3
      (prog3 (free 98) (+ 18 #0 #1) (copy (intersect 21 18) 9))
      (prog3 (free 98) (+ 18 #0 #1) (copy (adf2 12 9) 2 #1)))
      (prog3 (free 98) (+ 18 #0 #1)
        (% (adf2 33 9) (copy 2 18 1) #1 1)))
    (prog3
      (if (and (< 2 0) (not (< 1 0)))
        (if (not (< 18 0)) (+ 0 #1 #1) (- 0 #1 #1))
        (if (and (not (< 2 0) (< 1 0)))
          (if (not (< 18 0))
              (- 0 #1 #1)
              (+ 0 #1 #1)) 0))
      (if (or (= 0 5)
              (= 0 (prog2 (copy 4 18 #1) (- 18 #3 #1))))
          (+ (+ 9 2) #1 #1) 0)
      (if (and (not (< 2 3)) (= 0 6))
        (prog2 (copy 2 3 #1) (copy 9 6 #2)) 0))
    (copy 2 1 1)))

(defun adf2 (arg0 arg1)
  (prog2
    (prog2
      (copy arg0.0 fspc0 3)
      (* arg1.0 fspc0 1 2))
    (prog2
      (+ fspc1 fspc0 1 1)
      (+ fspc2 fspc0 1 1))))

(defun adf3 (arg0 arg1)
  (prog3
    (+ (* arg0.1 (- arg0.0 (copy arg1.0 fspc.0) #1 1) #1 #1)
       (* arg0.0 (- arg1.1 (copy arg0.1 fspc.2) #1 1) #1 #1)
       #1 #1)
    (- arg0.0 (copy arg1.0 fspc.1) #1 #1)
    (- arg0.1 (copy arg1.1 fspc.0) #1 #1)))
```

Program 3.2: (continued)

if it proves to be the largest measured yet, a record is taken of that point and its elevation (`F1a`–`F1b` and `F2`).

At the end of these iterations (`G`), if the highest elevation measured is positive, i.e. the feature lies within a concavity, ADF 3 is used to construct a halfplane from the point subtending this elevation, and the proximal end of the model feature (see Figure 3.1 on p. 56). This point is then added to the answer (`G`).

The remainder of the program carries out this process a second time to discover the vertex subtending the highest anticlockwise elevation.

The final line of the RPB gives the return value. In the case that the model feature has no visibility space, it returns the address of a null-triplet located after the portion of memory used to specify the features to be viewed. Otherwise, it returns the address of the answerblock.

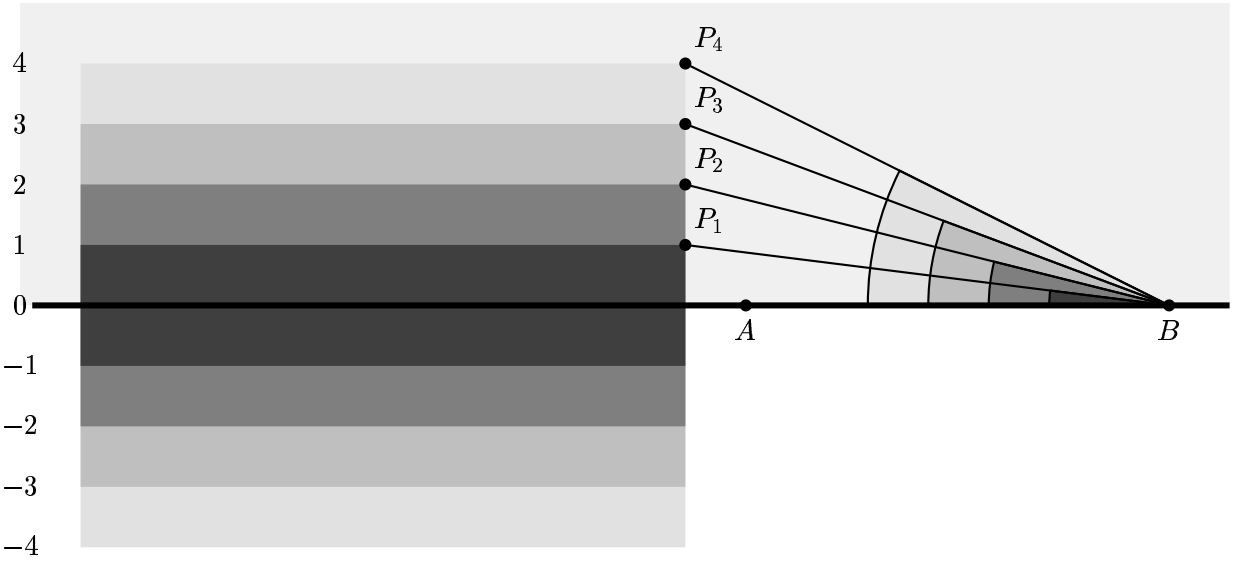


Figure 3.6: Relationship between elevation of a point, P_i, above the boundary of a halfplane $ax + by + c \geq 0$ (large shaded area), as measured by $ax + by + c$ (left) and as measured by the angle $A\hat{B}P$ (right) for any known points A and B on the halfplane boundary.

Instead of calculating angles, the program considers elevation as a point's distance into the inverse of the feature halfplane, since this provides an equivalent and simpler to calculate metric. This is illustrated in Figure 3.6—the value of $ax + by + c$ (grey boxes on the left) increases in the same order as the angle $A\hat{B}P$ (right) for known points A and B on the halfplane boundary. In this application, A and B would be the extrema of the model feature, in which case the line segment AB is the portion of the halfplane boundary delimiting the model boundary between these vertices.

The calculation of vertical distance shown in this figure must, of course, be divided by a likewise measured horizontal distance in order to be meaningful as a measure of elevation.

3.5 Fitness Cases

The fitness of genetic programs is usually determined by assessment against several sets of input data, for which the correct answer is already known. For the visibility space problem, a fitness case would consist of a model and a list of features to be viewed. It was decided to count only

0 1 2 3 4 5 6 7 8

Figure 3.7: Fitness cases for the untyped system.

complete visibility of model features as being within those features' visibility area; i.e. regions of space from which only part of the model feature could be seen would be regarded identically to those from which it could not be seen at all.

The fitness cases were carefully selected to be representative of the whole of the problem space, in order that programs evolved to solve the visibility space problem for these training data would not be overfitted to them, that is, not able to give correct answers to data they were not trained on. Their choice also implements the concept, central to GP, of *partial credit*, such that increasingly good programs have increasingly good fitnesses; whilst trying to avoid awarding good fitness to programs that did not deserve it.

Nine fitness cases were chosen in total. It will be observed that this number is a great deal less than that in many of the examples given in Section 2.2.13, which range between 10^1 and 10^6. However, the information content of the fitness cases in this application is much higher than many of the ones in Section 2.2.13, where the programs' output consisted of a binary classification, of only one bit in information capacity. In this case, the programs' output would be a representation of a two-dimensional region of space, of considerably more complexity. If the comparison is restricted to problems in which the program's answer and fitness evaluation is a complex one, this problem fits in better with the cited examples.

The fitness cases are shown in Figure 3.7, in increasing complexity of the task to be solved from left to right. The features to be viewed in each fitness case are indicated in colour. The diagrams illustrate the halfplanes that make up the answer; the visibility space the answer comprises is the intersection of these, shown on the figures in the deepest shading. In cases in subsequent diagrams where the intersection of all the halfplanes is null, the halfplanes are left unshaded.

Fitness cases 0–2 test the simplest solution to the visibility space problem: calculating the

visibility space of a convex polygon. This consists simply of the feature halfplanes inverted. Fitness case 1 introduces a new model in order to thwart any program which became overfitted to the details of the first model. Fitness cases 2 onwards test the ability to deal with more than one feature to be viewed.

Fitness case 3 introduces an asymmetric model; it is possible that solutions to the earlier fitness cases may have taken advantage of their symmetry. Fitness case 3 tests the generation of a visibility space for non-adjacent sides of the model.

Fitness case 4 includes a side which is not geometrically needed for the solution to the problem; the intention was that this should force the system to come up with an algorithm for calculating the answer rather than blindly copying and inverting the input.

Fitness cases 5–8 test the ability to deal with concavities: Fitness cases 5, 6 and 7 require the use of a halfplane in the answer that is not the inverse of one of the features to be viewed; fitness cases 7 and 8 require the construction of a halfplane that is not in the model at all.

3.6 Fitness Function

The output of the system was a number indicating the start of an answerblock in the scratchpad memory, terminated by three zeroes, which was interpreted as a series of halfplanes

$$\{(a_g, b_g, c_g)_1, \ldots, (a_g, b_g, c_g)_n\}$$

delimiting the calculated visibility space, where (a_g, b_g, c_g) are the parameters of the equation

$$ax + by + c \geq 0$$

These were compared to the correct solution as described in detail below and converted into a numeric measure of the program's fitness. Standardised fitness was used, such that all fitnesses were positive, with zero being ideal.

This measure was then adjusted to penalise poor solutions and reward good ones, both to encourage promising lines of evolution and to discourage simple ones of limited potentiality.

These rewards and penalisations included the following:

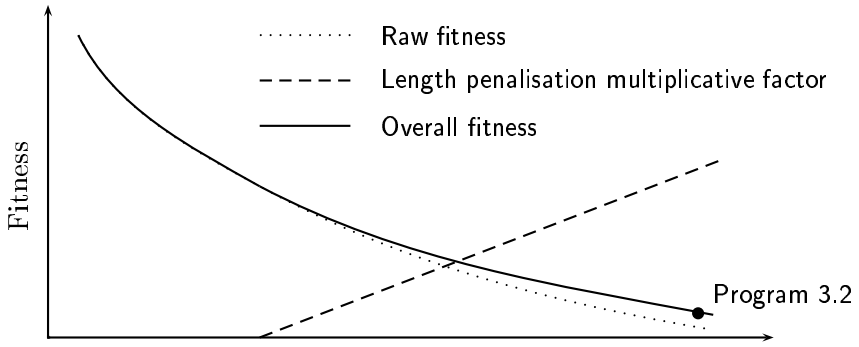


Figure 3.8: How improvement in fitness can compensate for length penalisation.

Length

- A common problem with the results of genetic programming is that programs increase in size, the extra code being either redundant or non-functional (*intronic*). This phenomenon is called *Bloat*,[25,96] and is usually tackled by penalising overlong programs, though other methods have been devised,[134] as described in Section 2.2.9. In this instance, the fitness as averaged across the fitness cases was penalised proportionately to the program's length in genes (nodes of the parse tree, or program elements), as follows:

  if $length > 150$ **then** **let** $fitness := fitness \times \frac{length}{150}$

  Many of the results reported in the field have program lengths reaching only double figures.[64,84,129] While this system, being more complex than most of these, would require longer programs, a free length limit of 150 would allow plenty of scope for the amassing of useful genetic material in partial solutions. Programs which thereafter exceeded this limit would then be penalised and have poorer fitnesses, unless their performance improved to compensate: Since the length penalty was a multiple of, and therefore proportional to, fitness, then the better a program performed, the lower the penalty would be in absolute terms and the longer the program could therefore afford to be. This is illustrated in Figure 3.8: even though as solutions get progressively longer, the length penalisation becomes proportionately ever larger compared to the raw fitness, the raw fitness continues to drop fast enough to keep the overall fitness dropping too.

- Since it was possible to produce halfplanes not entirely dissimilar to the correct answer

by going down wrong evolutionary routes, it was felt necessary to reward *hits*, that is fitness evaluations involving an exactly correct answer for that fitness case. Since correct answers would have a fitness of zero and nearly-correct answers would have a raw fitness not much higher, this could not really be done by applying a fractional multiplication or a subtraction to the correct answer. Instead, the reward was implemented by means of a penalisation of all incorrect answers. This penalisation would result in a fitness gulf opening up between nearly-correct answers and absolutely correct ones.

- Fitness cases in which no visibility space existed for the specified feature to be observed, represented a qualitatively different kind of test. In these cases, programs were penalised if they failed to deliver a null solution when they should, or did so when they should not.

The raw fitnesses for each fitness case were summed to give an overall fitness measure for each genetic program.

3.6.1 Fitness By Vector Difference in Arithmetic Parameter Space

The first approach taken was to compare the genetic and correct answers arithmetically. The reason for this decision was because of the problem of comparing the potentially infinite areas delivered by the system. The genetic program's answer

$$\{(a_g, b_g, c_g)_1, \ldots, (a_g, b_g, c_g)_n\}$$

would be sorted, and then aligned against the correct solution

$$\{(a_f, b_f, c_f)_1, \ldots, (a_f, b_f, c_f)_n\}$$

such that halfplanes in the one list were aligned against similar halfplanes in the other. Where no similar halfplane existed, halfplanes were compared against the null halfplane $(0, 0, 0)$. The methods used for this alignment are summarised in Figure 3.9; the procedure is described in full detail in the Appendices.

Once the halfplanes were aligned, the fitness was taken as the arithmetic difference between the two sets:

$$\sum(\mid a_g - a_f \mid + \mid b_g - b_f \mid + \mid c_g - c_f \mid)$$

Relationship	Interpretation
$a_f < a_g$	$f < g$
$a_f = a_g,\ b_f = b_g,\ c_f = c_g$	$f = g$
$a_f = a_g\ (b_f \neq b_g,\ c_f \neq c_g)$	$f \approx g$
$a_f > a_g$	$f > g$

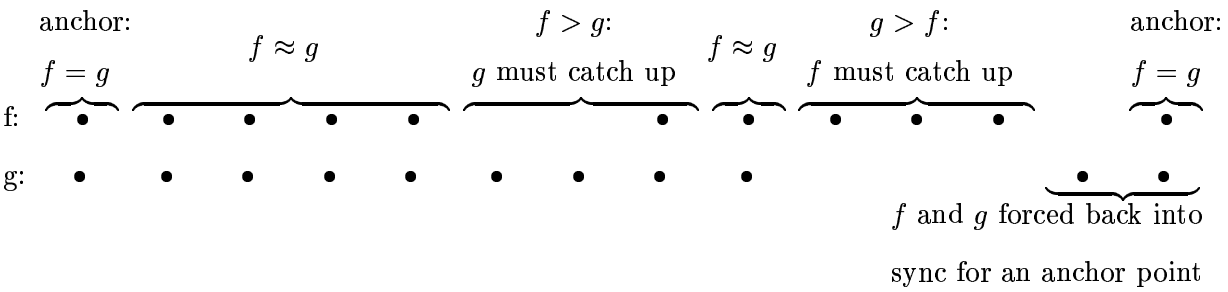
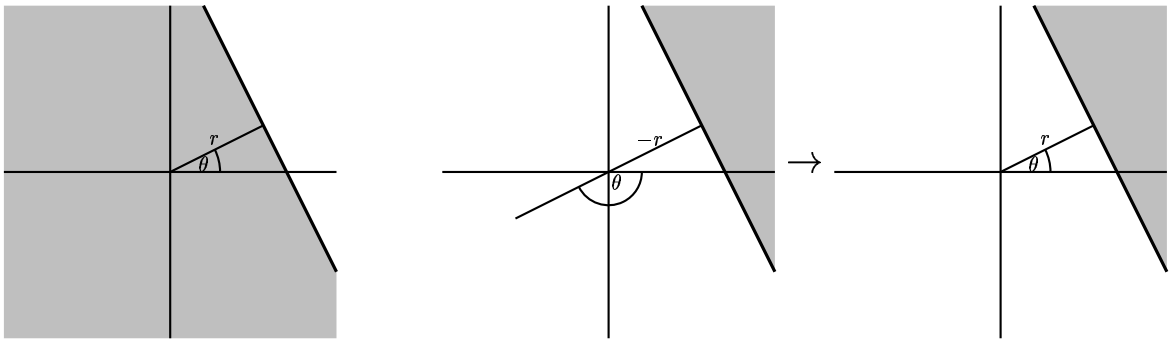


Figure 3.9: Arithmetic alignment of sorted halfplanes.

This was later changed in favour of the sum of the squares of the differences, so that larger differences would have a greater-than-proportional effect on fitness:[114]

$$\sum((a_g - a_f)^2 + (b_g - b_f)^2 + (c_g - c_f)^2)$$

The adjustments made to this raw fitness value included the penalisation of trivial solutions, such as those which simply copied the input, or those which did so and multiplied each parameter by -1. This inverted the halfplanes, thus creating halfplanes which were an essential part of the correct solution. However, because these could be produced with the minimum of knowledge, this could be accomplished by trivially simple programs which would, without penalisation of this kind, lead to evolutionary cul-de-sacs, by premature convergence onto programs of this kind. This is because these simple programs would become the ones most likely to be selected for crossover and reproduction, due to their good fitness.



A halfplane (a, b, c) becomes the geometric $(\frac{c}{\sqrt{a^2+b^2}}, \arctan\frac{b}{a}, 1)$. If the origin is outside the halfplane (that is, $r < 0$), θ is renormalised into the range $-\pi \ldots +\pi$ and the flipped flag set to -1*.

Figure 3.10: Definition of the geometric representation of halfplanes.

3.6.2 Fitness By Vector Difference in Geometric Parameter Space

The problem with the above approach is that arithmetic distance between halfplanes does not correspond well with geometric distance. For example, (a, b, c) and (ka, kb, kc) (where $k > 0$) describe halfplanes with identical boundaries, but have large arithmetic differences for values of k not close to 1.

Consequently, a new fitness measure was devised (see Figure 3.10). This one involved the halfplanes being converted prior to their comparison, to a new measure (r, θ, f), that reflected better the geometry of the halfplane. f was a flag indicating whether or not the halfplane was the one on the side of its boundary including the origin; this allowed the simple transformation from the halfplane (a, b, c) to $(-a, -b, -c)$ to map onto the equally simple transformation from $(r, \theta, \pm 1)$ to $(r, \theta, \mp 1)$ rather than the more complex transformation (r, θ) to $(-r, (\theta + \pi)\ rem\ 2\pi)$.

The new fitness measure compared the geometric measures by sorting, aligning and comparing them arithmetically, as above. This arithmetical comparison now had geometric relevance, and allowed the comparison of both the orientation and the positioning of the halfplanes.

Pareto fitness[53, 94] describes a method for evaluating the fitness of a program on multiple

*The algorithm given is simplified slightly; the real version has to trap division by zeroes.

criteria independently. However, no individual on a Pareto front is deemed better than any other, so an individual which optimises one criterion at the expense of another is assigned equal fitness to one which balances the two. This makes Pareto fitness more appropriate for cases with non-commensurable objectives, i.e. where optimising one objective compromises another,[53] or where weighting of the objectives cannot be assigned in advance. Since in this case it was required to optimise both objectives—orientation and positioning, as measured by the three coordinate components—simultaneously, it was deemed more appropriate to combine the two criteria arithmetically instead.[84]

With this fitness measure, all halfplanes (ka, kb, kc) are equal (provided $k \neq 0$); the measure also retains the ease of algorithmically inverting halfplanes, since equal but opposite halfplanes differ only with respect to the flag f, thus configuring the fitness landscape in such a way as to aid the evolution of (a, b, c) from $(-a, -b, -c)$.

3.7 Experiments and Results

Table 3.4 shows the parameters to the GP paradigm that were used for running the system. As stated in Section 2.2.10, larger populations often correlate with greater ability to evolve, therefore the largest population practical was chosen. The values of the other parameters were chosen from the ranges given in published works in the field.

The very first runs of the system gave trivial solutions. An example is the case of a halfplane that is the inverse of the correct one: this would be awarded near-optimal fitness in many of the fitness cases. This led to the best programs in each run simply returning the number 50, which is the memory address of the input data. This solution could be discovered in the initial random generation of programs. This led to the addition of the penalisations for trivial solutions mentioned in Section 3.6.

Even so, the best the system could manage was to copy the input data across inverted (see Program 3.1 on p. 73).

A variety of approaches were taken to attempt to improve the system's performance. These included carrying out runs with and without demes of different sizes, using fitness proportionate

Parameter	Setting
Population size	5000
Number of generations	≥ 50
Selection type	Tournament, size 10
Demes	0 / 10
Creation Type	Ramped half and half
Crossover Probability	75%
Creation Probability	2%
Maximum Depth For Creation	4
Maximum Depth For Crossover	10
Demetic Migration Probability	100%
Mutation Probability	0 / 0.2% / 2%
Population replacement	Steady State

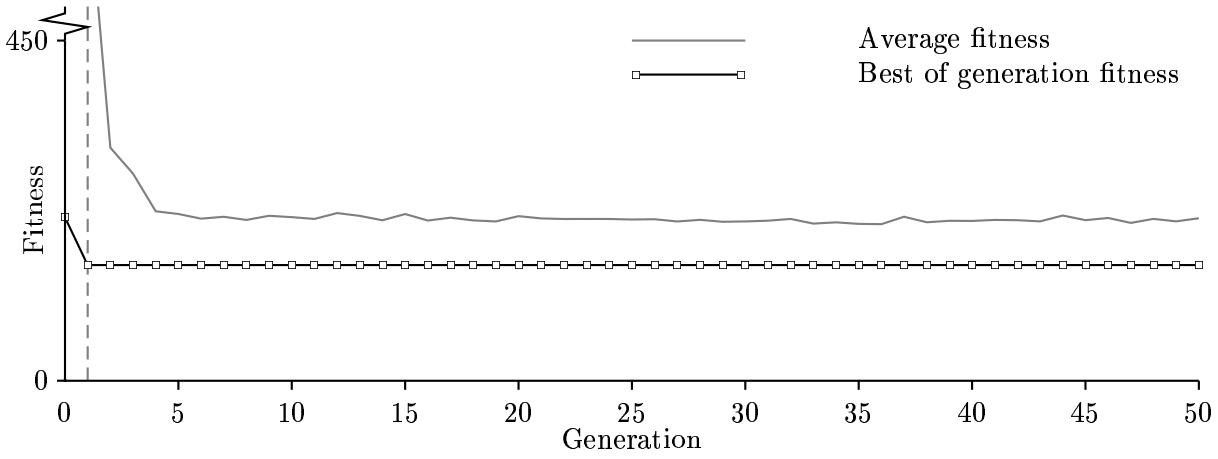
Table 3.4: Tableau for the untyped system

rather than tournament selection, varying the size of the tournament when using tournament selection, and carrying out runs with no mutation, a low mutation rate (0.2%) and a higher mutation rate (2%).

Dynamic fitness cases were also used (see Section 2.2.7), in an attempt to focus evolution. This was done here by introducing a second set of fitness cases in place of the first, once a program in the population had managed to attain a fitness threshold. These fitness cases had quite different solution types to those in the first set, such that any program which was overfitted to solving the fitness cases in the first set would not be able to perform well on solving the second.

Unfortunately, none of these measures had the desired effect. Figure 3.11 shows a typical run of the untyped system, and Program 3.3 the best-of-run individual from it.

One possible cause of problems with the system is illustrated with the hand-crafted correct solution in Program 3.2 on p. 76. Though it was determined that the function and terminal sets would be sufficient to solve the problem, and a hand-crafted solution for convex polygons was constructed using them; a complete hand-crafted solution was not constructed until after the system had been implemented. As a result, the program (or at least this possible solution) is



Random number seed: 866730616*; dynamic fitness cases: threshold 483, changeover indicated.

* Unless otherwise specified, the random number seed is derived from the time of execution, in the Unix format of seconds since 0:00:00, January 1 1970. GPC++'s random number generator is adapted from the one given in *Numerical Recipes in C*.[124]

Figure 3.11: Typical result of the untyped system.

```
(defun rpb  () (adf4 #71))
(defun adf0 (arg0) (- 71 #39 #39))
(defun adf1 (arg0)
  (- 33
     (+ 50
        (free 21)
        #(and #3 #68)
        #18)
     #45
     #30))
(defun adf2 (arg0 arg1)
  (% (or #2 #arg1.2)
     #arg0.2
     #arg1.1))
(defun adf3 (arg0 arg1) (% #3 #arg0.2))
(defun adf4 (arg0) (adf1 #86))
(defun adf5 (arg0)
  (% #(+ (adf2 #89 #62)
         (or #3 #39)
         #2
         #(copy 45 86 #48))
     #18))
```

Program 3.3: Best of run individual from run shown in Figure 3.11.

not always simple or elegant in its use of code.

For example, there are two places where code is duplicated and would be better coded as an ADF and called twice: the double iteration around the model, once to find the highest clockwise elevation and once to find the highest anticlockwise one (see Figure 3.1 on p. 56), and the double iteration necessary within each of these. This inner double iteration is itself due to a poor choice of programming primitive: it is necessary because of the inability to access forward or prior model locations in `do_db`, and the fact that iteration starts at an arbitrary point not related to the model feature.

However, only one of these two encapsulations can be accomplished, due to the facts that ADF 1 is the only ADF to possess iteration routines, and that nested iterations are not permitted. Furthermore, since the RPB is the only program branch with `evaluate` in its function set, the functionality of this primitive must be reimplemented in a lower ADF if it is to be used in ADF 1, which is inefficient and loses the benefit of having `evaluate` in the first place.

The program given also has a parse tree two levels deeper than the maximum depth permitted after crossover, so would not have been able to evolve in that precise form; but this problem could be obviated by a rearrangement of the code.

None of the above points necessarily prevents the evolution of a correct solution to the problem operating along different lines; but should the most likely evolutionary route to discovering a correct solution indeed lie along the same lines as the solution given in Program 3.2, it would be difficult for such a program to evolve.

3.8 Conclusions

The chapter has presented the first attempt to calculate two-dimensional visibility spaces genetically. Though initial thoughts had suggested that this problem might be easily solved by Genetic Programming, allowing progression on to more complex tasks, experimentation failed to evolve a solution. It is apparent from this study that the visibility space problem is a more complex one than had initially been thought, and the system used was insufficiently powerful to solve it.

The following lessons can be derived from the work done on this system:

- Visibility space analysis is confirmed as a complex problem requiring the use of a state memory.
- The low-level representation used in this system resulted in a hand-crafted solution to the problem being considerably longer than most programs GP is capable of discovering; by using a higher-level representation, the correct solution would be shorter and therefore probably easier to discover.
- It is important to craft a fitness measurement according to the result which is required rather than the form of data delivered by the program.
- It is not sufficient merely to verify that the function and terminal sets are sufficient to solve the problem; construction of a full hand-crafted solution may be necessary if it is to be ensured that solutions can be constructed with the minimum of code.

It was apparent that it was now necessary to give some thought as to how the system might be better tailored for the job in hand. This is described at the start of the next chapter.

Chapter 4

The First Typed System

4.1 Introduction

This chapter describes the second GP system written to tackle the two-dimensional visibility space problem. Section 4.2 discusses the reasons why the previous system was inadequate to solve the problem, and describes the reasoning behind the changes that were now made.

Amongst these changes was the adoption of strongly typed GP. Section 4.3 describes the nodes possibilities table that was used to carry this out efficiently.

Section 4.4 describes the architecture and fitness function of the new system, and Section 4.6 describes issues that arose during the implementation of the system.

Section 4.7 reports how evolution with this system tended to get trapped in local optima, in the form of "static solutions", then goes on to describe the various approaches used in attempting to alleviate this problem: altering the fitness function, fitness cases and function set, tuning the parameters of evolution, use of program templates and use of dynamic fitness.

Section 4.8 describes a simple system for solving array manipulation tasks that was used to model the visibility space system, in order to determine the reasons for the observed limits to its performance.

	(a) DNA / polypeptide	(b) numbers / halfplanes
Genotype:	ATCCTAGACGTCCAAAGGGTTTTC	−2 −1 1 0 2 2 2 −1 1 0 0 0
Correct interpretation:	ATC CTA GAC GTC CAA AGC GTT TTC	(−2, −1, 1) (0, 2, 2) (2, −1, 1) (0, 0, 0)
Correct phenotype:	Ile Leu Asp Val Gln Ser Val Phe	
Incorrect interpretation:	A TCC TAG ACG TCC AAA GCG TTT TC	−2) (−1, 1, 0) (2, 2, 2) (−1, 1, 0) (0, 0,
Incorrect phenotype:	Ser **stop** Thr Ser Lys Ala Phe	

Figure 4.1: Illustration of codon slippage errors: (a) codons in DNA; (b) the application of this concept to this system.

4.2 Rationale

Experimentation showed the untyped system to be unable to produce programs solving the visibility space problem for anything other than convex polygons, for which the solution is trivial. However, various points were gleaned from this use of the original system which aided in its redesign.

4.2.1 Problems with an Untyped System

Firstly, it was considered that evolution was unable to accrete the low-level programming instructions together into higher-level constructs. The programs the original system generated manipulated the data solely as numbers, and not as the geometric and algebraic entities those numbers represented. Furthermore, the programs were at risk of "codon-slippage errors", that is, attempting to read the data out of synchronisation with its storage as duplets and triplets (Figure 4.1).

GP has been shown capable of evolving code to manipulate abstract data structures;[28,95] however, in those experiments evolution was explicitly directed at evolving such code and data structures. Here the evolution is directed at solving a different problem, and it had been hoped that the ability to read such structures would arise as a side-product of evolution. It was partly

Functions:	
`if` SUBTREE `then` SUBTREE	
Boolean terminals:	
`okay`	
`not-okay`	
Action terminals:	
`act`	
`stop`	

(a)

Functions:	
`if` BOOLEAN `then` ACTION	→ ACTION
Boolean terminals:	
`okay`	→ BOOLEAN
`not-okay`	→ BOOLEAN
Action terminals:	
`act`	→ ACTION
`stop`	→ ACTION

(b)

Table 4.1: Simple genetic machine for illustrating typing. (a) Untyped, (b) typed.

for this reason that high-level structures had not been added, since to do so would result in the system being constrained to use its indexed memory in one particular manner,[150] ruling out the possibility of the system discovering good evolutionary paths along different approaches.

Since the old system was manifestly unable to discover data-manipulation algorithms on its own, function and terminal sets for the new system were chosen to include predefined higher-level constructs. Though necessarily ruling out potential approaches to solving the visibility space problem, this would considerably lessen the size of the search space in exploring the remainder.

This approach would represent the inclusion of *a priori* knowledge about the domain of the answer; however this is not necessarily a bad thing: it has been shown[160] that search algorithms cannot be guaranteed to efficiently explore large spaces without the inclusion of *a priori* knowledge, a finding known as the No Free Lunch Theorem.

4.2.2 Strongly Typed GP

Secondly, it was considered that strong typing[107] might be necessary to solve the problem. Without typing, the program elements were used indiscriminately, without regard to their semantic meaning. Adding what in effect would be semantic constraints would greatly reduce the size of the search space.

Consider, for example, the simple genetic machine given in Table 4.1a. This system contains two BOOLEAN terminals and two ACTION terminals. In an untyped system, it would be syntactically permissible for an `if` node to take any of these terminals as either of its arguments.

Semantically valid `if` nodes	Semantically meaningless `if` nodes	
`if okay then act`	`if okay then okay`	`if okay then not-okay`
`if okay then stop`	`if not-okay then okay`	`if not-okay then not-okay`
`if not-okay then act`	`if act then act`	`if act then stop`
`if not-okay then stop`	`if act then okay`	`if act then not-okay`
	`if stop then act`	`if stop then stop`
	`if stop then okay`	`if stop then not-okay`

Table 4.2: Table illustrating how typing can cut down search spaces.

However, only four of the sixteen are semantically meaningful, as shown in Table 4.2. Adding typing (Table 4.1b) would prevent the system from wasting its resources exploring the 75% of possibilities that are semantically meaningless, and would furthermore relieve the programmer from the necessity of having to implement sensible behaviour for semantically invalid programs.

If this system included more types that were valid as the second argument to an `if`, then in theory a separate `if` would have to be provided for each of these. In practice, this can be obviated by provision of a GENERIC type. Functions and terminals with a generic argument or return type are instantiated permanently (but independently for each usage) at program creation. Consider, for instance, an extension to this system that possessed also a type REACTION. `if` would now be specified as having a GENERIC second argument; but all `if`s occurring in programs would be of type ACTION or REACTION, and an `if`, once instantiated, would remain the same type: an ACTION `if` would not be able to turn into a REACTION one. The advantage of generic nodes, other than brevity of specification, over explicitly providing, for instance, an integer `if` and a real-number `if`, is that the different versions of `if` do not end up occupying a disproportionate fraction of the function set, which would cause the probability of selecting *any* `if` to be disproportionately high.

4.2.3 Limitations of Flat Typing

A hierarchical typing system for GP has been proposed.[66] In this, for example, a system might possess types STRING and NUMERIC, with NUMERIC further divided into INTEGER and REAL, such that some operations, such as `mod`, could act only on integers, and others on either integers or reals. For this application, it was reasoned that flat typing would suffice.

One consequence of using flat typing is that it would not permit the implementation of variables as terminals, as this would require the ability to distinguish between *lvalues* and *rvalues*. An *lvalue* is yielded by the evaluation of a variable; it combines attributes of value and reference, and altering it leads to alteration of the variable's value. An *rvalue* by contrast has only the attribute of value, and thus may not be altered. The names derive from their positions on the left and right hand sides of statements such as:

```
var := 3
```

where `:=` signifies an operation assigning the value on its right to the variable on its left, then returning that variable.

Operations which return an lvalue can be nested within each other:

```
var1 := var2 := 3
```

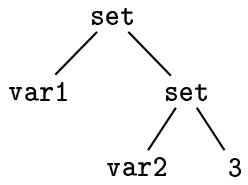
This lends itself naturally to the tree structure of genetic programs; Table 4.3a shows how this would normally be encoded in a genetic program and Table 4.3b shows the parse tree thus encoded. A hierarchically typed GP system (Table 4.3c,d) would allow both `var1` and `3` to be used as the second argument of `set`, whilst forbidding the latter from being used as its first argument, as shown in Table 4.3e.

However, when considering whether to represent variables as lvalued symbols (as in most human-written code) or numbered memory locations (as in most GP code), there is another factor to be taken into consideration, and that is the likelihood of the variable being selected during program creation.

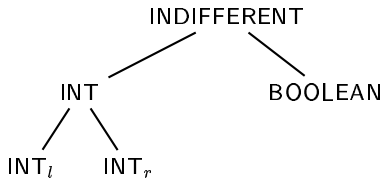
The issue is similar to that in the case of `if` in the previous section. Consider a system in which there are three variables, and six other members of the combined function and terminal

(a) `set (var1, set (var2, 3))`

(b)



(c)



(d)

`var1`	$\rightarrow$ INT$_l$	Integer variable
`var2`	$\rightarrow$ INT$_l$	Integer variable
`set (`INT$_l$`, INT)`	$\rightarrow$ INT$_l$	Sets arg1 to the value of arg2, returning arg1
`clear (`INT$_l$`)`	$\rightarrow$ INT$_l$	Clears an lvalue and returns it
`+ (INT, INT)`	$\rightarrow$ INT$_r$	Addition, returning an rvalue

(e)

Valid	Invalid
`set (var1, 3)`	`set (3, var1)`
`set (var1, var2)`	`set (3, 4)`

Table 4.3: Illustration of the use of hierarchical typing to implement lvalues and rvalues.

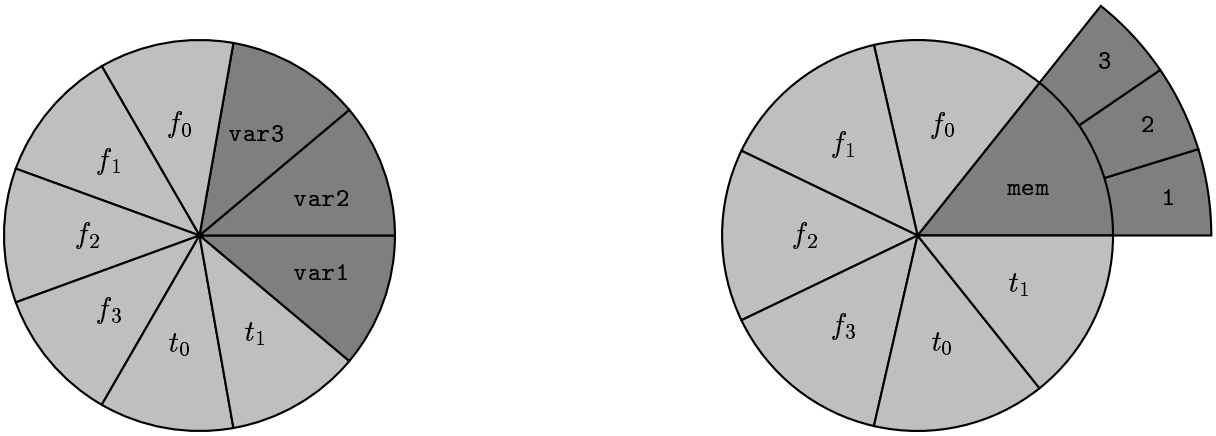


Figure 4.2: Probabilities of selecting a particular variable during program creation using (a) lvalued symbols, and (b) indexed memory.

sets. If variables are represented as lvalued symbols, then (not considering discrimination between the selection of a function and the selection of a terminal) there will be a one in nine chance of picking a particular variable; whereas if variables are represented as numbered memory locations, that value drops to only one in twenty-one (see Figure 4.2), i.e. the one in seven chance of selecting any variable, multiplied by the one in three chance of selecting one particular variable.

However, the chance of a variable being picked at all is now one in seven, equal to the chance of picking any other function or terminal, which is one in nine if variables are represented as symbols. (Both of these values will be altered considerably when taking into account the distribution of Full and Grow methods in program creation (Section 2.2.5), along with the chance of picking a terminal when using the Grow method, and the distribution of function and terminal types in the types or nodes possibilities table (see Section 4.3).)

The qualitative interpretation of these different probabilities of picking a variable with the two different representations, i.e. which of the two is better, leads to further questions: Is it preferred that a particular variable be included in the program, or that *any* variable be included in the program—and is bias towards the selection of a variable to be desired in program generation? This also has repercussions on evolution, as the use of indexed memory allows an extra site in the program tree for crossover and other operations to be applied. The issue of

contextual semantics in the application of genetic operators is a complex and not fully understood one,[59] and is not discussed here, being judged to be beyond the scope of this study.

To answer these questions would need an extensive investigation, taking in the distribution of functions and terminals in initial program generation and in the evolutionary ancestors of evolved correct solutions, and which of the two representations performed better in statistically significant numbers of runs.

To do this would require a well-understood system for which both the answer, and the distribution of correct solutions in the program space, were already known. Consequently, this issue was considered to be beyond the scope of this study, and the system was constructed with simple flat typing.

4.2.4 Other Consequences of Flat Typing

One final point remains: By representing variables as numbered memory, such that the variable must be read by code of the form `mem[1]`, any code that reads a variable will have a parse tree one level deeper than the equivalent code with symbolic variables, read by just `var1`, thus reducing the effective maximum potential complexity of programs of a given depth. Since the system specifies maximum depths for program creation and crossover, to achieve programs of the same complexity with both systems, the one using indexed memory variables must therefore have both of those depths increased by 1.

4.3 Implementing The Typed System: A Nodes Possibilities Table

Since there was no public-domain strongly typed GP system available in C++, I decided to write an extension to the GPC++ kernel to add typing to the system. This therefore necessitated concerning myself with implementational issues of typed GP as well as the semantics of applying it to my system.

A typed GP system needs to keep track of what nodes can be used to construct a valid tree during program creation. For example, consider a simple system for manipulating numerals,

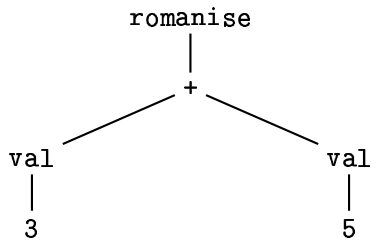
Root Type: ROMAN

`val` (NUMERAL)	→ INTEGER
`+` (INTEGER, INTEGER)	→ INTEGER
`romanise` (INTEGER)	→ ROMAN
`print` (INDIFFERENT, GENERIC)	→ GENERIC
1	→ NUMERAL
2	→ NUMERAL
3	→ NUMERAL

(a)

`romanise (val (1))`

(b)



(c)

Table 4.4: Specification of a simple Roman numeral system.

possessing terminals of type NUMERAL, a conversion operator to type INTEGER, in which type arithmetic operations can be carried out, and a second converter to type ROMAN, which yields a Roman numeral (see Table 4.4a).

To construct a program tree of height 2 in this system, the top node must be either `romanise` or `print`, because these are the only functions which can return type ROMAN. `romanise`'s single argument is of type INTEGER; the arguments of a ROMAN instantiation of `print` are INDIFFERENT (any type) and ROMAN. The subtree height of 1 means a terminal must now be selected, but there are no terminals of type either ROMAN or INTEGER, therefore a tree of this height and root type cannot be constructed. The smallest tree that can be created is of height 3; Table 4.4b shows an example. Table 4.4c shows the program tree for a tree of height 4.

To guide this selection of nodes during program creation, the paper in which STGP was first proposed[107] advocated using a *types possibilities table*, built from the function and terminal sets before program creation. This calculated which return types could be used at each depth, building a separate table for the two basic methods of program creation, Full and Grow (see Section 2.2.5), taking into account generic functions and terminals. The types possibilities table for the above example to a maximum creation depth of five is shown in Table 4.5. Note that the entries for each depth refer to the root nodes of subtrees of that height.

Subtree Height	Type					
	NUMERAL		INTEGER		ROMAN	
	Full	Grow	Full	Grow	Full	Grow
1	✓	✓				
2	✓	✓	✓	✓		
3	✓	✓	✓	✓	✓	✓
4	✓	✓	✓	✓	✓	✓
5	✓	✓	✓	✓	✓	✓

Table 4.5: Types possibilities table for the Roman numeral system.

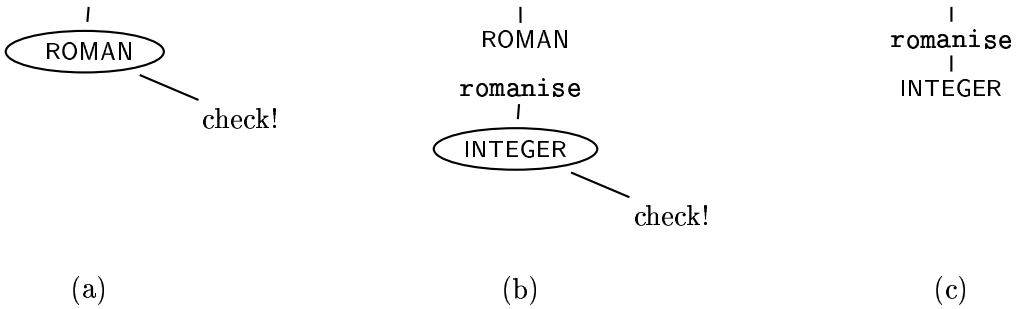


Table 4.6: Use of a types possibilities table. (a) Check ROMAN is in the table for subtrees of height 4. Then select a function (`romanise`). (b) Check INTEGER is in the table for height 3. (c) Add new node, and repeat process for its children.

As an example of how this table would be used, consider the generation of a program of tree depth 4, using the Full mode of creation, in which all subtrees have the maximum depth. The root node has to be of type ROMAN, so the system checks the types possibilities table to see whether ROMAN is permitted for the root node of a subtree of height 4 using Full creation (Figure 4.6a)*. Since it is, a function of this type, `romanise`, is selected. `romanise`'s sole argument is of type INTEGER, so the system checks that INTEGER is in the table for height 3 (Figure 4.6b). It is, so `romanise` can now be used as the root node (Figure 4.6c). The process is then repeated to find a node of type INTEGER for use as its child node. In this example, INTEGER instantiation of the generic function `print` is selected. The first argument of this function is

*Note that this figure shows the types available at varying depths of a program tree and should not be confused with Table 4.3c, which shows the relationship between the types in a hierarchically typed system.

```
(defun rpb  () (adf4 #71))
(defun adf0 (arg0) (- 71 #39 #39))
(defun adf1 (arg0)
  (- 33
     (+ 50
        (free 21)
        #(and #3 #68)
        #18)
     #45
     #30))
(defun adf2 (arg0 arg1)
  (% (or #2 #arg1.2)
     #arg0.2
     #arg1.1))
(defun adf3 (arg0 arg1) (% #3 #arg0.2))
(defun adf4 (arg0) (adf1 #86))
(defun adf5 (arg0)
  (% #(+ (adf2 #89 #62)
         (or #3 #39)
         #2
         #(copy 45 86 #48))
     #18))
```

Program 4.1: Example program from the untyped system.

INDIFFERENT and may be of any type; its second argument is instantiated, along with the return type, to INTEGER. For the first argument, the system checks the types possibilities table to see which types are available for subtrees of height 2. Only types INTEGER and NUMERAL can be used here, so the system picks a function with one of these return types, and checks that the types of its arguments are all included in the table for subtrees of height 1.

In this work, a slightly different approach was used. Use of the types possibilities table as described necessitates checking the presence in the table of a function's argument types every time that function is selected. This slows down program creation, and if the function set is varied enough that the argument types may not all be present in the table, that would slow down program creation even more.

Consider, for example, the system described in the previous chapter and, as a typical program in it, Program 3.3, repeated here as Program 4.1. This program has 43 genes; if it were typed this would require 43 node selections of a specified type, and 36 argument checks. For a population of 5000 individuals, this would mean 215 000 selections and 180 000 checks.

To alleviate this problem, instead of specifying allowable types in the possibilities table, all

allowable nodes were specified in this work. This obviated the requirement for any type checking after the table was generated. The price that had to be paid for this was a considerably larger possibilities table, of size:

$$N \times d \times 2 \times t \times n \times 2$$

where:

N = Number of node sets (one for each branch of the program)

d = Maximum depth of creation

t = Number of data types

n = Maximum number of nodes of any one type

The other two multiplicands are the two methods of creating GP trees (Full and Grow), and the two tables that are needed, one for functions and one for terminals.

In practice, this measure can be reduced somewhat, since there is no function set for subtree height 1, and only one depth is needed for the terminal sets—the table for terminals is identical for all depths using the Grow method, and using the Full method is empty except for subtree height 1, which is identical to the table for the Grow method. This still leaves a fairly large table, though (of size $Nt(2n(d-1)+n')$), where n and n' are the maximum table sizes for the functions and terminals tables respectively). However, considering that the system is dealing with a population of possibly several thousand individuals, each of which has to store a program tree plus its fitness, the use of a few more kilobytes for the nodes table becomes less significant and thus more justifiable.

Construction of the nodes possibilities table is somewhat more complicated than that of the types possibilities table, because it has to take account of generic nodes and children of INDIFFERENT arguments. Instantiation of generics ordinarily progresses down the tree from the top; however when constructing the nodes possibilities table it percolates up the table from the bottom. The algorithm used to construct the nodes possibilities table is given in Program 4.2.

Once constructed, the nodes possibilities table is used as follows:

Program creation: Almost all subtrees to be created already have a root type specified; all the system has to do is pick a node from the ones listed in the appropriate table entry. If the

```
foreach creation-type in {Full, Grow}
  foreach terminal t
    if t.type = GENERIC then
      foreach type T
        add t to table (1, creation-type, T)
    else add t to table (1, creation-type, t.type)
    for height := 2 to maximum-depth-for-creation
      foreach creation-type in {Full, Grow}
        if creation-type = Grow then
          foreach terminal t in table (height − 1, creation-type )
            add t to table (height, creation-type, t.type)
        foreach function f in node-set
          if for every f.argument a
              table (height − 1, creation-type, a) ≠ {}
               or a = INDIFFERENT
               or a = GENERIC then
            if f.return-type ≠ GENERIC then
              add f to table (height, creation-type, f.return-type)
            else
              if there exists f.argument a where a = GENERIC then
                foreach type T in table (height − 1, creation-type)
                  add f to table (height, creation-type, T)
              else
                foreach type T
                  add f to table (height, creation-type, T)
```

Program 4.2: Algorithm for construction of a nodes possibilities table.

node picked is GENERIC, it is instantiated to the specified type.

The subtrees without prespecified root types are those that fit into an INDIFFERENT argument specification, or GENERIC ones not already instantiated by their parent node, such as:

```
prog2 (GENERIC, GENERIC) → BOOLEAN
```

In these cases, the program chooses a random type of those represented in the nodes possibilities table at that depth and creation type, then proceeds to select a random node of that type as normal. In the case of the child of the GENERIC specification, the parent is now instantiated.

Crossover: Once generic nodes are instantiated, they become effectively strongly typed. A subtree can only be crossed with another of the same root type, unless both subtrees are the children of an INDIFFERENT argument specification, in which case the type-checking is suspended. The system must recognise that it may not always be possible to find a subtree in the second

Subtree Height	Type					
	NUMERAL		INTEGER		ROMAN	
	Full	Grow	Full	Grow	Full	Grow
1	`1, 2, 3`	`1, 2, 3`				
2	`print`	`1, 2, 3, print`	`val, print`	`val, print`		
3	`print`	`1, 2, 3, print`	`+, print`	`val, +, print`	`romanise`	`romanise`
4	`print`	`1, 2, 3, print`	`+, print`	`val, +, print`	`romanise, print`	`romanise, print`
5	`print`	`1, 2, 3, print`	`+, print`	`val, +, print`	`romanise, print`	`romanise, print`

Table 4.7: Nodes possibilities table for the Roman numeral system.

program to match that of the one selected in the first*.

Swap Mutation: A node can only be swapped with one of the same signature (child and return types), with the following exceptions:

- The child of an INDIFFERENT specification can match any return type.
- A candidate node with GENERIC specifications in its signature can be used so long as the one instantiation holds up for all GENERIC arguments. If this is so, once the mutation is carried out, the new node becomes permanently instantiated.
- For INDIFFERENT argument specifications on the new node, type-checking may be suspended.
- For INDIFFERENT argument specifications on the original node, for which arguments the new node does not also have INDIFFERENT specifications, type-checking must be done according to the child node's actual type.

The system must recognise that it may not be possible to find a replacement node of the appropriate type.

*As should the user: in his work on evolving feature detectors, Belpaeme[21] recognises that reproduction rates may be higher than specified due to failed crossovers.

Shrink Mutation: A subtree may only be replaced by a subtree from within it of the same return type, unless the (larger) subtree is the child of a gene with an INDIFFERENT child specification, in which case type-checking is suspended.

Table 4.7 gives the nodes possibilities table for the system outlined above, showing the number of occurrences of each node in the table.

4.4 System Specifications

4.4.1 Genetic Machine

Overview

In the redesigned genetic virtual machine, the following types were utilised:

HALFPLANE

POINT

ANGLE

BOOLEAN

INTEGER

[DIRECTION]

GENERIC—instantiated on program generation

INDIFFERENT—for use only in the declaration of functions' arguments

(Bracketed elements in this and following tables indicates items subsequently revised out of the system specification.)

The representation was now of a higher level than before, such that halfplanes and points were now abstract data types rather than sequences of floating-point numbers with no guarantee that programs would interpret them correctly. This change would act to focus evolution; the corollary was that the representation was now highly specific to the problem, and no longer one suitable for generalised arithmetic operations.

The INDIFFERENT type was provided for use by sequencing operations; in such operations only the final argument's type would be of importance, as it would be that argument's return

Points memory	
0	(x, y)
1	(x, y)
2	(x, y)
3	(x, y)
4	(x, y)
5	(x, y)
6	(x, y)

Angles memory	
0	x
1	x
2	x
3	x
4	x
5	x
6	x

Features list
(a, b, c)
...

← current feature

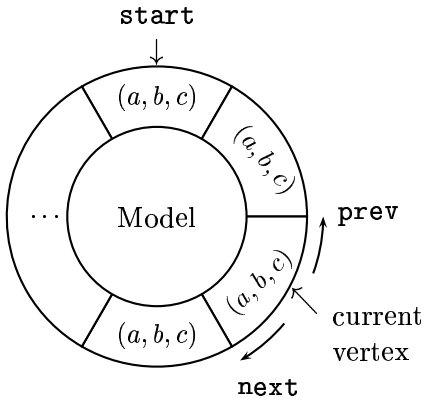


Table 4.8: Architecture of the genetic machine of the first typed system.

value that would then be returned by the sequencing operation. Both the final argument and the sequencing operation's return type would therefore be specified as GENERIC, as described in Section 4.2.

Part of the problem of designing an adequate function set was to decide how free programs should be to traverse the data set in their own way. Constraining the program to solve a particular sub-problem in a particular way might exclude a large part of the solution space from the space searched.

However, considering the inability of the original system to learn to navigate through its memories in an orderly fashion, it was decided, for reasons similar to those discussed above in Section 4.2.1, to explicitly constrain the new system to do this. The evolved code would thus be part of a larger framework, which is described below.

Architecture

In the new system, shown in Table 4.8, the model, features list and working memories are no longer parts of a united whole, but are semantically separate entities addressed in different ways. The two separate working memories, of types POINT and ANGLE, are indexed memories as in the previous system. Navigation of the features list and the model are now taken out of the genetic programs' hands and are handled by the system, as described below. Furthermore, the model now consists of a circular list of points.

Koza has shown[84] that superfluous functions and terminals degrade the probability of discovery of correct solutions in a linear fashion; consequently the function and terminal sets in this system were reduced to the minimum necessary to solve the problem. As part of this, the size of the indexed memory in the new architecture was also reduced to its minimum necessary;[28] that is, seven elements of ANGLE memory, and two of POINT memory. However, in order to allow the function for accessing the points memory to use the same INTEGER arguments as that for accessing the angles memory, equal numbers of memory cells were provided for both types.

The new system architecture broke programs up into several separately evolving branches to guide evolution, as shown in the genome specification in Table 4.9 on p. 108. Capital letters in the specification indicate code that is part of the fixed framework of program execution.

The intention was that, once the visibility space problem had been solved with this framework, the optimal configuration of genetic parameters for doing so could be discovered, and once these were known this framework could be incrementally unfrozen until the system was capable of evolving this framework, or an equivalent, by itself.

Two ADFs were provided, returning types BOOLEAN and ANGLE respectively. These types were chosen primarily because they were used in the hand-crafted correct solution (which is discussed further below). HALFPLANE was also omitted because it is only returned by `halfplane`, `add` and `abort`, all of which operate at a high-level in the program structure and were expected to be found in the RPB.

The other program branches were arranged around a built-in loop for iterating around the model, such that the program branches internal to it would be executed every time around the loop. This is not dissimilar to systems used for other GP problems such as the robot controller problem,[63] in which the evolved code is called at every timestep, or for image processing problems,[11,64] in which the kernel evolved by GP is convolved over every pixel of the image in turn.

This framework was itself then embedded in a larger framework, as shown in Program 4.3, which calls the genetic program twice for each feature to be viewed, once for a clockwise iteration around the model and once for an anticlockwise one. During these iterations, certain of the functions and terminals are re-bound in order to give symmetrical behaviour corresponding to

```
procedure GP :: evaluate
begin
  foreach feature in features
    add (invert (feature))
    run genetic program (clockwise)
    run genetic program (anticlockwise)
end GP :: evaluate

procedure GP :: run genetic program
  in direction
begin
  bind next to direction
  bind prev to direction
  if direction = clockwise then let start := feature.r
  else let start := feature.l
  call program-branch-0
  for points[0] := next(start) to eval (program-branch-1)
    if eval (program-branch-2) then eval (program-branch-3)
  if eval (program-branch-4) then eval (program-branch-5)
end GP :: run-genetic-program
```

Program 4.3: Framework for evolution in the first typed system. (Cf. genetic program specification in Table 4.9)

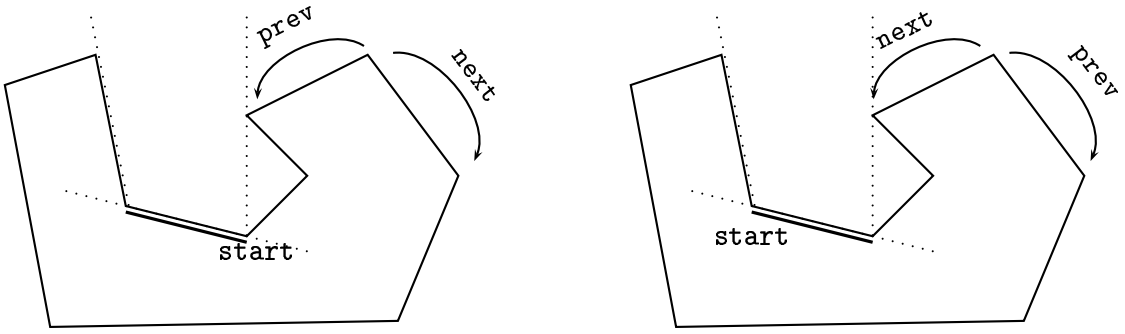
the symmetrical iterations around the model. These are the functions `next` and `prev`, which give the appropriate successor vertices to their arguments for the direction of iteration, and the terminal `start`, which refers to the proximal side of the model feature. This is illustrated in Figure 4.3.

The genetic program has no direct access to the features list; it accesses the current feature indirectly, by means of the terminal `start` and the start and end points of the iteration. The inverse of the feature halfplane is now added automatically to the answer, since adding it was shown in the previous system to be a trivial challenge for evolution.

The system would iterate around the model until the point specified by program branch 1 in Program 4.3; if this point had not been reached by the time iteration had proceeded right the way around the model, program execution would be terminated. This constitutes an explicit cap on iteration (cf. Section 3.4.2).

```
<genome> ::=
  MAIN:
    <indifferent>
    FOR SET/P[0] = NEXT (START) TO <point>
      IF <boolean> THEN <indifferent>
    END FOR
    IF <boolean> THEN <indifferent>
  ADF0: <boolean>
  ADF1: <angle>

<boolean> ::= and (<boolean>, <boolean>)
            | or (<boolean>, <boolean>)
            | not (<boolean>)
            | "<" (<angle>, <angle>)
            | ">" (<angle>, <angle>)
            | samesign (<angle>, <angle>)
            | adf0 (<integer>, <integer>, <angle>)
            | <generic>

<halfplane> ::= halfplane (<point>, <point>)
              | add (<halfplane>)
              | abort
              | <generic>


<point> ::= mem/p (<integer>)
          | set/p (<integer>, <point>)
          | start
          | arg0a
          | <generic>


<angle> ::= + (<angle>, <angle>)
          | - (<angle>, <angle>)
          | round (<angle>)
          | elevation (<point>, <point>)
          | adf1 (<integer>, <integer>, <angle>)
          | mem/a (<integer>)
          | set/a (<integer>, <angle>)
          | 0.0 | pi | 2pi | -pi
          | <generic>

<integer> ::= arg0i | arg1i | 0 | 1 | 2 | 3 | 4 | 5 | 6
            | <generic>

<generic> ::= &{<indifferent>; <indifferent>; <instantiation>}
            | if (<boolean>, <instantiation>)

<indifferent> ::= <boolean> | <halfplane> | <point> | <angle>
<instantiation> ::= <boolean> | <halfplane> | <point> | <angle>
```

Table 4.9: Definition of the first typed system's genetic machine in Backus Naur Form. (For simplicity, functions and terminals which were revised out of the system are here omitted.)



(a) (b)

Figure 4.3: Rebinding of `start`, `prev` and `next` in (a) the clockwise, (b) the anticlockwise iteration.

Syntax and semantics

Table 4.9 provides a definition of the new system in Backus Naur Form; Table 4.10 lists the functions and terminals according to functionality.

- Boolean operations:

  and (BOOLEAN, BOOLEAN)

  or (BOOLEAN, BOOLEAN)

  not (BOOLEAN)

  < (ANGLE, ANGLE)

  > (ANGLE, ANGLE)

  samesign (ANGLE, ANGLE)

  These perform their respective operations on their arguments, returning the BOOLEAN values *true* or *false*. For the purposes of **samesign**, zero is counted as negative.

- Model iteration operations: **next, prev** (POINT)

  These provide the points following and prior to the current model vertex (as stored in points memory cell 0). During the clockwise iteration, **next** returns the next point in a clockwise direction and **prev** the next in an anticlockwise direction; during the anticlockwise iteration these are the opposite way around.

Boolean operations:	
`and, or (BOOLEAN, BOOLEAN)`	→ BOOLEAN
`not (BOOLEAN)`	→ BOOLEAN
`<, > (ANGLE, ANGLE)`	→ BOOLEAN
`samesign (ANGLE, ANGLE)`	→ BOOLEAN
Model iteration operations:	
`[next (POINT)`	→ POINT]
`[prev (POINT)`	→ POINT]
Arithmetical operations:	
`+, - (ANGLE, ANGLE)`	→ ANGLE
`round (ANGLE)`	→ ANGLE
Conversion operations:	
`halfplane (POINT, POINT)`	→ HALFPLANE
`elevation (POINT, POINT)`	→ ANGLE
Programming operations:	
`& (INDIFFERENT, INDIFFERENT, GENERIC)`	→ GENERIC
`if (BOOLEAN, GENERIC)`	→ GENERIC
`add (HALFPLANE)`	→ HALFPLANE
`abort`	→ HALFPLANE
Memory operations:	
`mem/a (INTEGER)`	→ ANGLE
`set/a (INTEGER, ANGLE)`	→ ANGLE
`mem/p (INTEGER)`	→ POINT
`set/p (INTEGER, POINT)`	→ POINT
Automatically Defined Functions:	
`adf0 (INTEGER, INTEGER, ANGLE)`	→ BOOLEAN
`adf1 (INTEGER, INTEGER, ANGLE)`	→ ANGLE
Parameters:	
`start`	→ POINT
`[dirn`	→ DIRECTION]
`arg0i, arg1i`	→ INTEGER
`arg0a`	→ POINT
Constants:	
`0, 1, 2, 3, 4, 5, 6`	→ INTEGER
`0.0, π, 2π, -π`	→ ANGLE

Program Branch	Node																																								
	∧	∨	not	and	or	halfplane	elevation	add	&	ift	mem/a	set/a	mem/p	set/p	round	samesign	+	-	adf0	adf1	start	abort	0.0	pi	2pi	-pi	0	1	2	3	4	5	6	arg0i	arg1i	arg0a					
INITIAL	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
MAIN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
1st IF	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
1st THEN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
2nd IF	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
2nd THEN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓								
ADF 0	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓								✓	✓	✓					
ADF 1	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓								✓	✓	✓					

Table 4.10: Functions and terminals in the first typed system, listed by functionality.

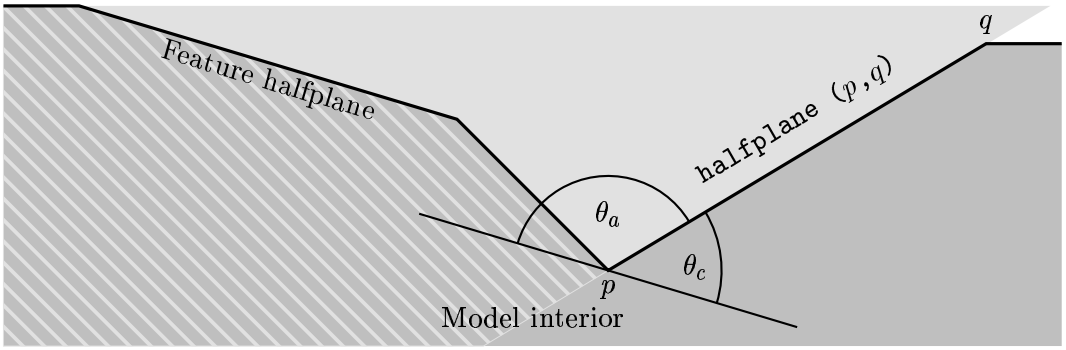


Figure 4.4: Illustration of the operation of `halfplane` and `elevation`. θ_a shows the elevation of q from p as measured during the anticlockwise iteration, θ_c shows the elevation measured during the clockwise iteration.

- Arithmetical operations:

  `+ (ANGLE, ANGLE)`

  `- (ANGLE, ANGLE)`

  `round (ANGLE)`

  `+` and `-` perform the appropriate arithmetic operations and return the `ANGLE` result. `round` returns its argument rounded into the range $-\pi \leq x \leq \pi$.

- Conversion operations:

  `halfplane (POINT, POINT)`

  `elevation (POINT, POINT)`

  `halfplane` returns the halfplane constructed from its arguments p and q,

  $$(y_p - y_q,\ x_q - x_p,\ y_p(x_p - x_q) + x_p(y_q - y_p))$$

  such that if p is on the viewer's left and q on the right, $ax + by + c > 0$ is above the line pq. If either argument is the null point $(0, 0)$, or the arguments are the same, the null halfplane $(0, 0, 0)$ is returned.

  `elevation` returns the elevation, in degrees, of its first argument q from its second argument p, with respect to a baseline passing through p parallel to the feature halfplane, in

the direction of model iteration.

These operations are illustrated in Figure 4.4.

- Programming operations:

  `& (INDIFFERENT, INDIFFERENT, GENERIC)`

  `if (BOOLEAN, GENERIC)`

  `add (HALFPLANE)`

  `abort`

  `&` evaluates its arguments in turn, discarding the values of all bar the last, which is then returned. It is equivalent to the `progn` operations in the previous system.

  `if` evaluates its second argument if the first argument evaluates to *true*. If not, it returns a null value of the second argument's type.

  `add` adds a halfplane to the answer list of halfplanes, which is not otherwise accessible to the genetic program. It returns this same halfplane.

  `abort` clears the answer list of halfplanes and terminates program execution.

- Memory operations:

  `mem/a (INTEGER)`

  `set/a (INTEGER, ANGLE)`

  `mem/p (INTEGER)`

  `set/p (INTEGER, POINT)`

  `mem/a` returns the contents of the cell of the angles memory indexed by its argument; `mem/p` does likewise with the points memory. `set/a` and `set/p` are used for writing to these memories; their first arguments indicate the memory cell to write to, and their second arguments hold the value to be written. Cell 0 of the points memory is defined as being read-only during the course of the main iteration.

- Parameters: `start`

  `start` returns the start point of iteration. On the clockwise iteration this is the clockwise vertex of the feature halfplane; on the anticlockwise iteration it is the anticlockwise one.

- Constants:

  `0, 1, 2, 3, 4, 5, 6`

  `0.0, π, 2π, -π`

  These have their expected values. `0–6` are integers, the others are angles.

- Automatically Defined Functions:

  `adf0 (INTEGER, INTEGER, ANGLE)`

  `adf1 (INTEGER, INTEGER, ANGLE)`

  `arg0i, arg1i`

  `arg0a`

  The ADFs take two integer arguments, and one angle argument. All arguments are evaluated before the ADF is entered. The ADF call returns the value returned by the ADF program branch.

Worked example

As mentioned in the previous chapter, the untyped system suffered from no hand-crafted program to completely solve the problem having been constructed in advance of running the system. The construction of such a program was therefore a priority for the new system; Program 4.4 shows the hand-crafted perfect solution that was devised. (Code in capital letters indicates the fixed framework of execution.) The operation of this program is as follows (see Figure 4.5):

Lines 2–37 are the main loop around the model. At the start of each iteration, the elevation of the current point P_j from *start* is measured, with respect to the feature halfplane (lines 5–7).

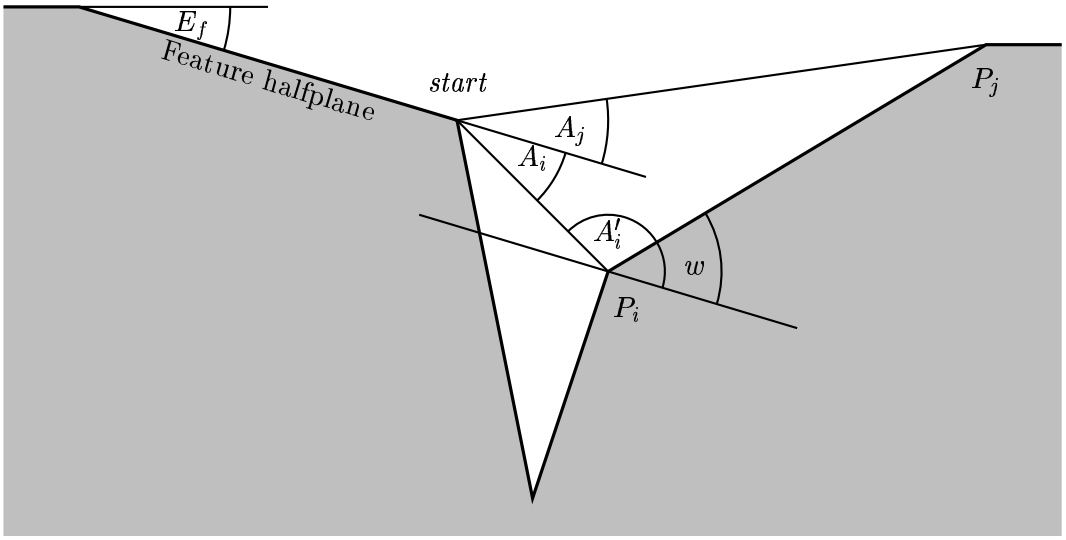
Lines 8–26 concern corrections to this value for cases where the measured value, which lies within the range $-\pi \ldots \pi$, is not the correct one, which therefore lies outwith this range. If the previous point, P_i, had an elevation of less than π but P_j's observed elevation is positive, then P_j is actually wrapped around in the negative direction, so 2π must be subtracted from the measured value (lines 8–9).

Lines 10–25 deal with the situation where the line of zero elevation has been crossed since the last iteration. If elevation of P_j from P_i with respect to the feature halfplane (w on the diagram,

```
 1.set/p[2] (start)†
 2.FOR SET/P[0] THROUGH MODEL POINTS‡
 3.   &{
 4.    &{
 5.      set/a[0]
 6.       (set/a[3]
 7.         (elevation (mem/p[0], start)));
 8.      if and (not (> (mem/a[6], -pi)), > (mem/a[3], 0.0))
 9.      then set/a[0] (- (mem/a[0], 2pi));
10.      if not (samesign (mem/a[3], mem/a[4]))
11.      then
12.         &{
13.           0;
14.           &{
15.             set/a[5] (elevation (mem/p[0], mem/p[2]));
16.             set/a[4] (round (+ (pi, mem/a[6])));
17.             if or (and (> (mem/a[6], 0.0),
18.                         < (mem/a[5], mem/a[4])),
19.                    and (< (mem/a[6], -pi),
20.                         > (mem/a[5], mem/a[4])))
21.             then abort
22.           };
23.           if and (> (mem/a[5], mem/a[4]), > (mem/a[3], 0.0))
24.           then set/a[0] (- (mem/a[0], 2pi))
25.        }
26.    };
27.    &{ set/a[6] (mem/a[0]);
28.       set/a[4] (mem/a[3]);
29.       set/p[2] (mem/p[0])‡ };
30.    0
31.  }
32.  IF > (mem/a[0], mem/a[1])
33.  THEN &{ 0;
34.          set/a[1] (mem/a[0]);
35.          set/p[1] (mem/p[0])
36.        }
37.END FOR
38.
39.IF > (mem/a[1], 0.0)
40.THEN add (halfplane (start, mem/p[1]))
```

‡ To replace the loss of `prev` in the function set—see Section 4.7. Until the point marked by the third dagger, `mem/p[2]` contains the equivalent of `prev (mem/p[0])`. appear on this page otherwise.

Program 4.4: Hand-crafted perfect solution for the first typed system.



Note that A_i is negative as illustrated. (A'_i is positive.)

If sign $(A_i^{obs}) \neq$ sign (A_j^{obs}), then:

	$w > A'_i$	$w = A'_i$	$w < A'_i$
$A_i > 0$	no action ($A_j > 0$)	(no action)	Feature occluded: abort
$-\pi \leq A_i \leq 0^*$	$A_j \leftarrow A_j^{obs} - 2\pi$	(no action)	no action ($A_j > 0$)
$A_i < -\pi$	Feature occluded: abort	(no action)	no action ($A_j < 0$)

* I.e. $A_j^{obs} > 0$.

Figure 4.5: Distinguishing $x°$ from $-(360 - x)°$.

calculated in line 15) is greater than the converse of A_i (line 16), then the crossing is wrapping around in a positive direction (i.e. clockwise on the clockwise iteration, and anticlockwise on the anticlockwise iteration), and vice versa.

If the model has wrapped around so far that the feature halfplane is completely occluded, then the program aborts: the feature is not visible from outside the model's convex hull (lines 17–21).

Otherwise, adjust the measured elevation accordingly (lines 23–24).

Lines 27–29 copy the current point and elevation into the memory cells used to hold the previous one, for the next time around the loop. If this point subtends the highest elevation measured yet, it and its elevation are stored for later (lines 32–36).

When the main loop is over, if the highest measured elevation is positive (line 39), then the feature halfplane lies within a concavity, so the halfplane constructed from *start* and the stored point subtending the highest elevation is added to the answer

A step-by-step dissection of this in pseudocode follows here. The meaning given to the variables is that shown in Figure 4.5. As before, the right-hand column shows the values returned by the evaluation of each subtree.

1.	$P_i \leftarrow$ *start*	$\Longrightarrow$ *start*
2.	**for** $P_j \leftarrow$ next(*start*) **to** prev (prev (*start*))	
3.	sequence:	
4.	sequence:	
6–7.	$A_j^{obs} \leftarrow$ elevation (P_j, *start*)	$\Longrightarrow$ elevation (P_j, *start*)
5–7.	$A_j \leftarrow A_j^{obs}$	$\Longrightarrow$ elevation (P_j, *start*)
8.	**if** $A_i \not> -\pi$ **and** $A_j^{obs} > 0.0$	
9.	**then** $A_j \leftarrow A_j - 2\pi$	$\Longrightarrow A_j$ or 0.0
10–11.	**if not** ($A_j^{obs} > 0$ **and** $A_i^{obs} > 0$) **then**	
12.	sequence:	
13.	0 (space-filler)	$\Longrightarrow 0$
14.	sequence:	
15.	$w \leftarrow$ elevation (P_j, P_i)	$\Longrightarrow$ elevation (P_j, P_i)
16.	$A_i' \leftarrow (A_i + \pi)$ rem 2π	$\Longrightarrow (A_i + \pi)$ rem 2π
17–18.	**if** $A_i > 0$ **and** $w < A_i'$	
19–20.	**or** $A_i < -pi$ **and** $w > A_i'$	
21.	**then** abort	**if** $\Longrightarrow (0, 0, 0)$
22.		sequence $\Longrightarrow (0, 0, 0)$
23.	**if** $w > A_i'$ **and** $A_j^{obs} > 0.0$	
24.	**then** $A_j \leftarrow A_j - 2\pi$	$\Longrightarrow A_j$ or 0.0
25.		sequence $\Longrightarrow A_j$ or 0.0
26.		**if** $\Longrightarrow A_j$ or 0.0

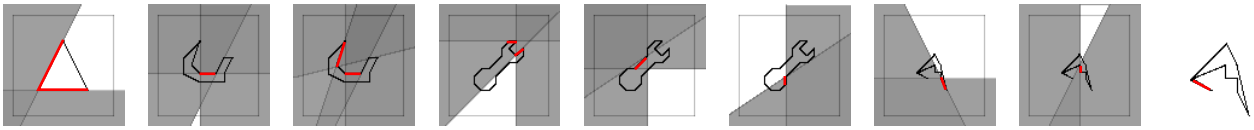
27. sequence:
28. $A_i \leftarrow A_j$ $\Longrightarrow A_j$
29. $P_i \leftarrow P_j$ $\Longrightarrow P_j$
29. $\Longrightarrow P_j$
30. 0 (space-filler) $\Longrightarrow 0$
31. sequence $\Longrightarrow 0$
32. **if** $A_j > A_{max}$ **then**
33. sequence:
33. 0 (space-filler) $\Longrightarrow 0$
34. $A_{max} \leftarrow A_j$ $\Longrightarrow A_j$
35. $P_{max} \leftarrow P_j$ $\Longrightarrow P_j$
36. sequence $\Longrightarrow P_j$
36. **if** $\Longrightarrow P_j$
37. **end for**
39. **if** $A_{max} > 0.0$ **then**
40. Add to answer the halfplane constructed from *start* and P_{max}.
$\Longrightarrow$ halfplane (*start*, P_{max})
if $\Longrightarrow$ halfplane (*start*, P_{max}) or $(0, 0, 0)$

Since the program consists of a number of disparate branches, there is no single value returned by program evaluation as a whole. The values returned by its constituent branches are those returned at the end of lines 1, 31, 32, 36, 39 and 40.

4.4.2 Fitness Function and Fitness Cases

The initial fitness measure used was that of sorting, aligning and comparing geometric descriptions of the answer halfplanes, as described in Section 3.6.2. The penalisations given in that section continued to apply.

The initial fitness cases for the first typed system are shown in Figure 4.6. To recap from the previous chapter, the diagrams show the halfplanes that make up the answer; the visibility



0 1 2 3 4 5 6 7 8

Figure 4.6: Original fitness cases for the first typed system.

space the answer describes is the intersection of these, and is shown on the figures in the deepest shading. In cases where the intersection of all the halfplanes is null, the halfplanes are left unshaded. As in the untyped system, progression from left to right across the diagram indicates increasing algorithmic complexity necessary to determine the answer:

Fitness case 0 is the trivial case of a convex polygon (no action necessary by the genetic program, the inverse of the feature halfplane being added automatically by the system). Fitness case 1 requires the calculation of a halfplane to add on either side of the feature.

Fitness case 2 has two features to be viewed; one requires the calculation of a halfplane to be added on either side of it; the other only requires a halfplane on one side. Fitness case 3 has two features requiring the construction of zero and one halfplanes.

Fitness cases 1, 3 and 7 introduce new models. Fitness case 7 tests that the system can discriminate between local maxima and global maxima of the elevation of candidate vertices for halfplane construction.

Fitness case 8 represents an instance in which no solution is possible: the model feature has no visibility area outside of the model's convex hull. This tests the programs' ability to use the `abort` instruction.

4.4.3 Seeding the Population

Unlike hill-climbing algorithms, evolutionary algorithms are capable of navigating past obstacles in the fitness landscape; but there is a limit to the size of obstacles evolutionary exploration can bypass. This is because new locations in the problem space are explored by building on pre-existent structures. A sufficiently byzantine fitness landscape will not be amenable to solution by evolutionary exploration at all.[98] It is therefore essential that the fitness function chosen is

one which evolution is capable of exploring efficiently.

If a fitness function is plotted against program space in an n-dimensional graph, where n is the number of different factors making up the fitness measure, the ideal fitness function would be one sloping smoothly from poor solutions to good ones. Large jumps in the fitness function cannot be navigated by evolutionary exploration; presented with obstacles taller than evolution can leap over, populations tend to premature convergence instead, in a local optimum in the fitness function. Interpolating between these extremes, we can see that a fitness function that slopes in one direction only is more likely to lead to success than one which has non-global optima; and the more sigmoid a fitness function, the less likely that evolution will be able to descend its slope.

It is impossible to predict in advance what shape fitness landscape would be produced by a given fitness function without a moderately thorough exploration of the problem space. Since this would defeat the purpose of searching this space efficiently by means of evolutionary computation, each of the fitness functions proposed in this chapter was instead analysed in advance by observing its performance on a small number of hand-crafted programs. These programs traced the route that evolution was expected to follow, and were ranked in order of complexity.

The hand-crafted programs could also be used for *seeding* the population, so that in situations where evolution was having difficulty finding the correct answer to a problem, runs could be done with evolution starting from an extant partial solution, termed a *seed*, or *seeded program*.[69,116] Should a correct solution be found in such runs, the parameters of the system could then be optimised (see Section 4.7.3) and experiments undertaken to see whether the system could now evolve the correct solution from a completely random start.

Table 4.11 gives a brief algorithmic description of each of the seeds (using p_i and a_i to refer to the ith cell of the points and angles memories respectively); the code for them is given in Appendix A.2.2.

The performance of these programs is compared in Figure 4.7. The numbering of the seeds reflects the order in which they were written: seeds 1–5 were devised in such a way as to present a gradual diminution of algorithmic complexity; however when the seeds were run it transpired that with the fitness measure used this led to a large gap in fitness between seeds 2 and 3. Seeds

Seed	Description
1	Complete solution (see p. 113)

2	**for** $p_0 :=$ `next(start)` **to** `prev(start)`
	  $a_3 :=$ **elevation** $(p_0, start)$
	  if $a_3 \not> 0.0$ **then** $a_1 := \pi$
	  if $a_3 > 0.0$ **and** $a_1 < \pi$ **then** $a_0 := a_3$
	  if $a_0 > a_1$ **then**
	  $a_1 := a_0$
	  $p_1 := p_0$
	add halfplane $(\texttt{start}, p_1)$ **to answer**

8	**for** $p_0 :=$ `next(start)` **to** `prev(start)`
	  if elevation $(p_0, \texttt{start}) <$ **elevation** $(p_1, \texttt{start})$ **and** $a_0 < \pi$ **then**
	  $a_0 := \pi$
	  add halfplane $(\texttt{start}, p_1)$ **to answer**
	  $p_1 := p_0$

7	**for** $p_0 :=$ `next(start)` **to** `prev(start)`
	  if elevation $(p_0, \texttt{start}) \not> 0.0$ **and** $a_0 < \pi$ **then**
	  $a_0 := \pi$
	  add halfplane $(\texttt{start}, p_1)$ **to answer**
	  $p_1 := p_0$

6	**for** $p_0 :=$ `next(start)` **to** `prev(start)`
	  if elevation $(p_0, \texttt{start}) > 0.0$
	  then $p_1 := p_0$
	  add halfplane $(\texttt{start}, p_1)$ **to answer**

3	**for** $p_0 :=$ `next(start)` **to** `prev(start)`
	  $a_3 :=$ **elevation** $(p_0, \texttt{start}))$
	  if $a_3 > 0.0$ **then** $a_0 := a_3$
	  if $a_0 > a_1$ **then**
	  $a_1 := a_0$
	  $p_1 := p_0$
	add halfplane $(\texttt{start}, p_1)$ **to answer**

4	Does nothing
5	Always aborts

Table 4.11: Description of the seeds for the first typed system.

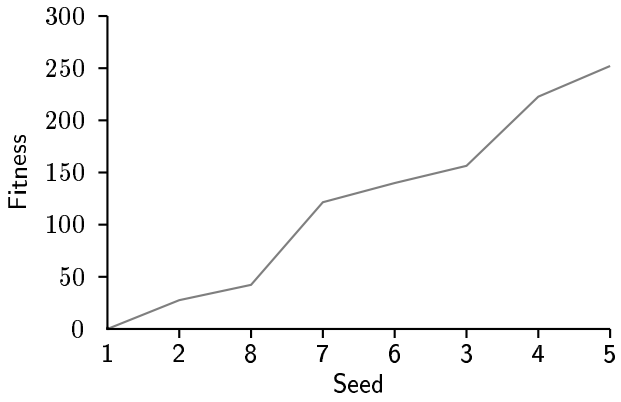


Figure 4.7: Fitnesses of the seeds with the original fitness measure.

```
FOR SET/P[0] FROM NEXT (START) TO prev (start)
 set/a [0] (elevation (mem/p[0], start) -
            elevation (start, prev (start)))
IF mem/a[0] > mem/a[1]
THEN &{ set/a[1] (mem/a[0]);
        set/p[1] (mem/p[0]);
        1
      }
IF > (mem/a[1], 0.0)
THEN add (halfplane (mem/p[1], start, dirn))
```

Program 4.5: Original conception of model solution before the implementation of the system.

6–8 were therefore written to fill this gap. All figures show the seeds in decreasing order of algorithmic complexity.

4.5 Complexity Analysis

The previous system suffered from the lack of a hand-crafted correct solution, for use as an index of the complexity of programs that should be expected. To verify that the proposed function and terminal set were sufficiently powerful to solve the problem, the program given in Program 4.5 was written.

This program iterates around the model to find the vertex subtending the highest elevation from the corner of the model feature. This program is short and simple, and this led me to believe that the visibility space problem would be readily amenable to solution by GP.

However, during the implementation of the system, it became apparent that the task of

discovering the vertex subtending the highest elevation masks a fairly complex subproblem, that of distinguishing a vertex subtending $x°$ from one subtending $-(360 - x)°$. In Figure 3.1, for instance (see p. 56), the highest elevation subtended from point L is that of point A (45°) and not point B (−250°, not 110°).

This is a task that humans carry out so easily that we are not even aware we do it. However, it would have to be explicitly carried out by any solutions evolved by the GP system.

In order to keep track of whether the angle subtended by the current vertex is positive or negative (or wrapped completely around, as exemplified in Figure 4.25 on p. 147), adjustments must be made to the currently observed angle, A_j^{obs}, depending on whether its sign is different from the previously observed angle, A_i; this was shown in Figure 4.5 on p. 115.

Revising the system to be able to cope with this involved the addition of several new instructions to the function set, along with the extension of the angles memory from just two memory cells to six, which was the minimum now necessary to solve the problem. This resulted in the complete hand-crafted solution (Program 4.4 on p. 114) becoming considerably more complex than had originally been expected.

Table 4.12 shows a complexity analysis of the problem; it can be seen that a GP run would only cover a small fraction of the search space. However, the very intention of GP is to avoid an exhaustive search. A comparison was undertaken with the size of search space of some other problems tackled by GP (Table 4.13).

The problems shown in this table are as follows: The Boolean N-multiplexer[84] has k Boolean data inputs and 2^k address input bits (where $N = k + 2^k$); the problem is to use the Boolean operations `and`, `or`, `not` and `if` to construct a function that will output the ith data input where i is specified in binary by the aggregated address inputs. The artificial ant problem[84] is to evolve a controller for an automaton in a simulated toroidal world grid such that it obtains the maximum number of food items from a non-contiguous food trail in a fixed run time. The trail used here is the Los Altos Hills trail.[87] The transmembrane domain prediction problem[88] was discussed on p. 38.

The table uses, where possible, known solutions as maximum tree depth. (This figure will be larger than the phenotype search space by a few orders of magnitude, because for every

Tree Height	Genotype Search Space	
1	13	
2	121	
3	3351	← short run (2.4×10^4)
4	67,527	← long run (1.6×10^5)
5	814,512	← seed 5
6	1.02421×10^7	← seed 6
7	1.30697×10^8	← seed 3
8	1.67864×10^9	← seeds 2, 4
9	2.16232×10^{10}	
10	2.78937×10^{11}	
11	3.60106×10^{12}	← seed 1 (perfect solution)
12	4.65122×10^{13}	
13	6.00976×10^{14}	
14	7.76737×10^{15}	
15	1.00416×10^{17}	
16	1.29847×10^{18}	
17	1.67942×10^{19}	

Table 4.12: Complexity analysis of programs of INDIFFERENT type root up to the maximum depth for crossover

Problem	Genotype Search Space (estimate)	Individuals to be processed
Visibility Space	10^{19}	10^{12} (est.)
Visibility Space (partial solution)	–	10^6 (est.)
Transmembrane Domain Prediction	6.36×10^{10}	1.02×10^6
Boolean 6-Multiplexer	10^{19}	1.6×10^5
Artificial Ant (ADFs included)	1.75×10^4	1.36×10^5

Table 4.13: Comparison of the complexities of different problems tackled by GP

genotype there will be a number of other programs that do exactly the same calculations, but with the variables permuted; and many more which differ slightly but produce the same ultimate phenotype.)

What the table shows is that the visibility space problem is significantly more complex than most of the other ones. Given this, it was not apparent when starting out whether GP would be able to solve it. GP has successfully been used to solve problems with very much larger search spaces, such as the Boolean 11-multiplexer, which has a search space of size 10^{616}; however Boolean problems are characterised by regular fitness landscapes, which is helpful for evolution toward a correct solution. Hence the figures given in the table cannot be taken as a guarantee that a problem will or will not be soluble.

As can be seen from Table 4.12, the size of the search space necessary to find the less complex seeds was much smaller than that necessary to find the complete solution. Consequently it was hoped that evolution would be able to evolve programs equivalent to these less complex seeds first. Functional equivalents of these would then disperse throughout the population, because they would be preferentially selected due to their good fitness. Once this had happened, they would become available as raw material that could be recombined to discover more complex solutions.

Should the system not be able to manage this by itself, there are techniques available that could force it to tackle subtasks in the complete solution first; these are described later in the chapter.

4.6 Implementational Issues

4.6.1 Redesign of the Function and Terminal Sets

During the implementation of the above architecture, it became apparent that the system was rather more complex than had been anticipated. The function and terminal set originally proposed did not contain the operations `abort`, `round`, `pi`, `-pi`, `2pi`, `samesign`, `<`, or `2–5`; instead it contained the operations `exceeds`, `*`, `%` and `dirn` and the type DIRECTION.

`exceeds` was to be dynamically bound, as were `next` and `prev`; it would take on the meaning of `<` or `>` depending on the value of `dirn`. During the implementation of the system, this was simplified by making the direction of iteration implicit. `exceeds` was replaced by the explicit `<` and `>`, and `halfplane` and `elevation` were made dynamically bound, constructing halfplanes and measuring elevations depending on which way around the model the program was iterating, rather than on the value of their DIRECTION parameter.

Now that this was done, there was no need for `dirn` or the DIRECTION type, and these were dropped from the system. The arithmetic operations `*` and `%` were also dropped from the system, since they were not required to construct a correct answer, and it has been shown that superfluous functions and terminals degrade the probability of discovery of correct solutions.[84]

4.6.2 Evaluation Caching

Program execution by evaluating every gene in turn can be highly inefficient, especially in the early, randomly generated programs. For example,

```
&{ mem/a[&{ &{ 0; 0; 0};
            mem/a[0];
            &{ 0; 0; 0}}];
   mem/a[&{ &{ 0; 0; 0};
            &{ 0; 0; 0};
            &{ 0; 0; 0}}];
   &{ mem/a[&{ 0; 0; 0}];
      &{ mem/a[0];
         &{ 0; 0; 0};
         &{ 0; 0; 0}};
      &{ &{ 0; 0; 0};
         &{ 0; 0; 0};
         &{ 0; 0; 0}}}}
```

(taken from a real evolved program) has no side-effects, so can be simplified to just `mem/a[0]`.

The reason for the great redundancy in this subprogram is because this structure comes from the first branch of the genetic program (see the program architecture on page 108). This branch was only of use in initialising variables, which only became necessary at a very late stage in program evolution; and it contained only the minimum set of functions and terminals necessary for satisfying tree construction according to the nodes possibilities table. Since this program was generated using the "full" creation method, all branches had to be of the maximum depth,

Parameter	Setting
Population size	1000
Number of generations	100
Selection type	Tournament, size 10
Demes	10
Creation Type	Ramped half and half
Crossover Probability	90%
Creation Probability	2%
Maximum Depth For Creation	6%
Maximum Depth For Crossover	17%
Demetic Migration Probability	10%
Swap Mutation Probability	5%
Shrink Mutation Probability	0%
Population replacement	Steady State

Table 4.14: Tableau for the first typed system

leading to a large and redundant tree.

This inefficiency can be reduced by caching results of previous evaluations.[94] In brief, the caching system used operates as follows (the algorithm is described in more detail in Appendix A.3):

Before the first evaluation of the program for each fitness case, every gene is labelled with the result-type UNKNOWN. After the first evaluation of each gene (starting at the terminals and working upwards), the label is emended to one of CONSTANT, VARIABLE or SIDE-EFFECTS as appropriate.

The result type is changed from CONSTANT to VARIABLE by use of `mem/a`, `mem/p` and `arg`x; and to SIDE-EFFECTS by use of `set/a`, `set/p`, `add` and `abort`.

A gene "inherits" the highest result-type of its children, except for `prog3`, for which VARIABLE only percolates upwards from its last argument. This is because the first two child subtrees of a `prog3` gene can only interact with the rest of the program via their side-effects.

In subsequent evaluations of the code, descendants of a CONSTANT node (including the node itself) need not be evaluated any more; instead the cached value is returned.

Calls to an ADF acquire the status of that ADF's root node.

4.7 Experiments and Results

The first round of experimentation consisted of as many runs of twenty generations as could be carried out overnight, using the parameters given in Table 4.14. Though the length of each run was rather short, it was not known in advance how fast the system would evolve, and this approach allowed the examination of a large number of separate evolutionary trails. Forty-seven runs were carried to completion, resulting in 9.4×10^5 non-unique individuals being processed; Figure 4.8 shows a sample evolution from these.

Program 4.6 shows the most interesting individual produced by these runs. This was the first individual observed to use the `FOR` loop counter sensibly; what it does is stop on the second iteration and check whether the last point considered is above the feature halfplane. If it is, then it constructs a halfplane from `start` and this point, and adds it to the answer.

The part outside the loop boils down to:

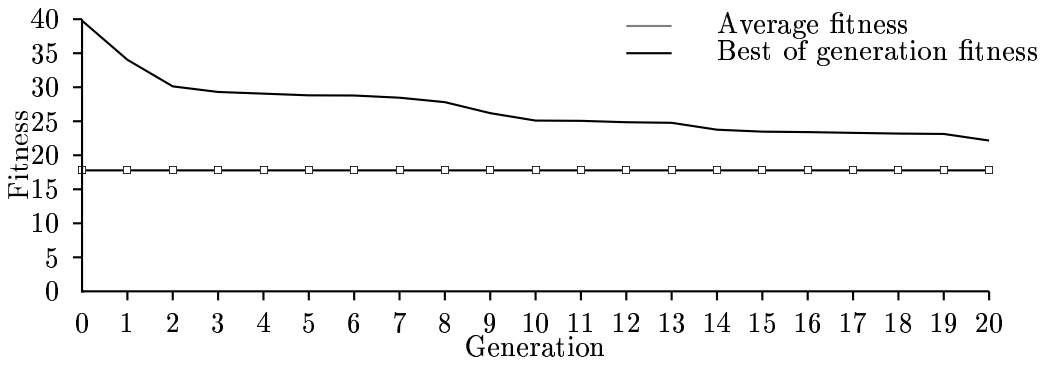
```
IF > (elevation (mem/p[0], start), 0) THEN add (halfplane (start, mem/p[0]))
```

which is an essential part of the correct answer. This program's execution is illustrated in Figure 4.9.

Most of the runs did not produce results so encouraging. Though conceptually superior to its predecessor, the architecture introduced in this chapter still retained a major loophole in its design, and a characteristic of GP systems is that evolution will exploit any unseen loopholes in the fitness function.[82]

In this case experimentation revealed that it was possible for programs to evolve that, though they completely ignored both the main program loop and the contents of the model, still delivered answers close enough to the correct solution to have a substantially good fitness. For example, consider the program in which the only active component is the line:

```
add (halfplane (start, next (next (start))))
```



Evolution style: steady state; random number seed: 880480402; run 1 out of 47.

Figure 4.8: Sample early run of the first typed system.

(a)

```
FOR SET/P[0], NEXT(START),
    mem/p[&{ 0.0;
            next (mem/p[&{ 0.0; next (next (start)); 5}];
            5}])
  add (if samesign (0.0, pi) then add (abort))
  IF > (- (elevation (mem/p[4], prev (start)),
           elevation (prev (start), prev (start))),
         + (- (- (pi, 0.0), round (2pi)),
            + (* (pi, 0.0),
               elevation (mem/p[4], prev (start)))))
  THEN mem/p[&{ -pi; 0; 3}]
END FOR

IF > (- (round (pi),
         - (round (pi),
            elevation (mem/p[0], set/p[1] (start)))),
      round (round (2pi)))
THEN add (if not (< (pi, 0.0))
          then if not (< (pi, 0.0))
               then halfplane (start, mem/p[0]))
```

(b)

```
FOR SET/P[0], NEXT(START), mem/p[5])
END FOR

IF > (elevation (mem/p[0], start),
      round (2pi))
THEN add (halfplane (start, mem/p[0]))
```

Program 4.6: Result of an early run. (a) Original code; (b) with redundant code stripped out.

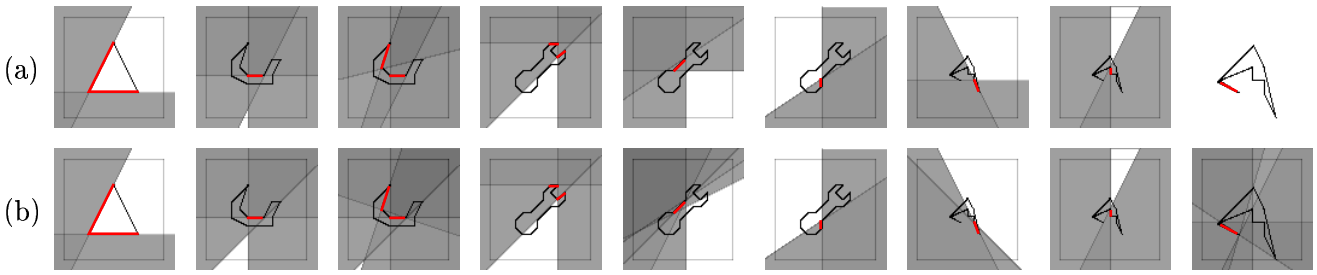


Figure 4.9: Execution of programs using the original fitness cases for the first typed system. (a) Correct solution, (b) Program 4.6.

Program	Fitness									
	Fitness case									Overall
	0	1	2	3	4	5	6	7	8	
seed 1 (perfect solution)	0	0	0	0	0	0	0	0	0	0
seed 4 (does nothing)	0	40	40	40	40	40	40	40	40	35.56
Program 4.6	20	20	20	0	20	0	20	0	20	13.33
`halfplane (start, next (next (start)))`	20	20	20	20	20	20	20	20	20	20
`halfplane (prev (start), start)`	0	20	20	20	20	20	20	20	20	17.78

Table 4.15: Fitnesses of hand-crafted and evolved programs.

The fitnesses of this program as against the perfect solution is shown in Table 4.15. Its overall fitness is substantially better than that of randomly generated programs.

I termed such programs *static solutions*, since they depended only on the programming language and not on the genetic program's input data (the contents of the model).

Such solutions have been observed in other instances of complex problems; for example in attempting to evolve a non-linear equation to describe chaotic data, Oakley suffered from static solutions in the form of constant values.[116] In this case, static solutions outperformed all bar the best three of the seeded programs, as illustrated in Figure 4.10, where the abscissa of the points on the lighter trace indicates the number of nested **`next`** statements in the second parameter to the call to **`halfplane`**.

Since these static programs had such good fitness, the population tends to converge on them, rather than following the expected route of evolution traced out by the hand-crafted programs. As a result of the smallness of their active components, this results in genetic information

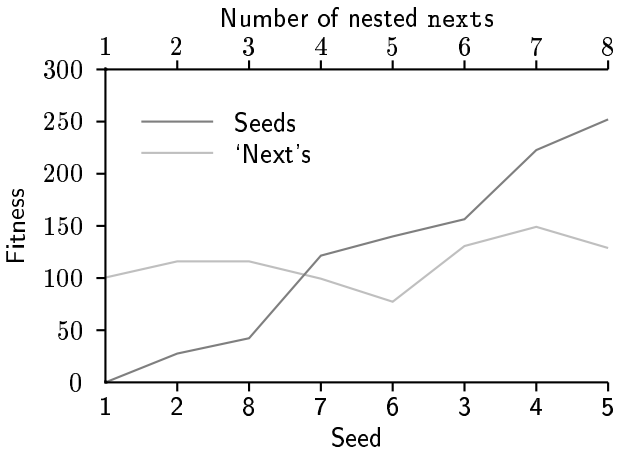


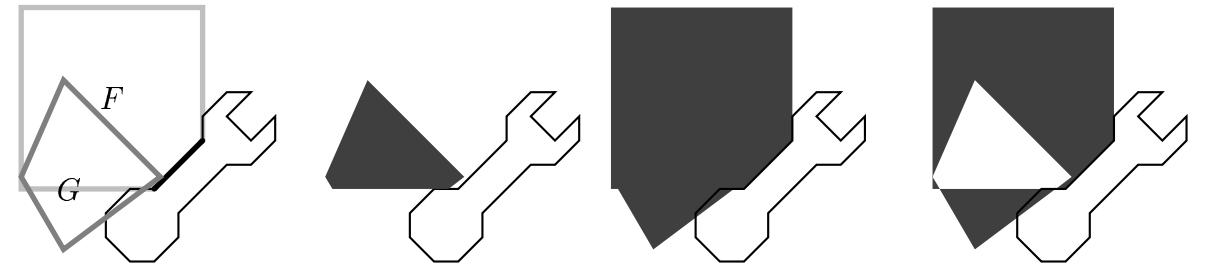
Figure 4.10: Fitness of the static solutions compared to that of the seeds.

becoming lost, eventually depriving the population of the ability to continue evolving towards better solutions. In short, the population converges and becomes trapped upon a local optimum in the fitness landscape.

An analysis (p. 167) of the prevalence of clones, or genetically identical individuals, in the population showed them to be few, indicating that this convergence was phenotypic and not genotypic.

Various strategies were undertaken to prevent the population from converging onto static solutions. These strategies, which are described in more detail below, included the following:

- Altering the fitness function to better reflect the actual goal of the system.
- Altering the fitness cases to diminish the measured fitness of static solutions.
- Parameter tuning to improve evolution.
- Altering the function set to make the codification of static solutions difficult.
- Dynamic specifications, both to counter static solutions and to encourage evolution of good ones.
- Use of program templates to guide evolution.



F = correct answer

G = genetic answer

$F \cap G = 56.555$ $F \cup G = 241$ $F \cup G - F \cap G = 184.445$

Fitness measure	Raw fitness
$100 \times \frac{F \cup G - F \cap G}{F \cap G}$	266.1
$100 \times \frac{F \cup G - F \cap G}{F \cup G}$	53.07

Figure 4.11: Fitness measures for the first typed system.

These strategies were carried out concurrently with one another, and are discussed in turn in the following sections.

4.7.1 Altering the Fitness Function

It was hypothesised that one reason why evolution was becoming trapped in local minima was because even using the geometric representation, the answers given by static programs were not that different to the correct answer geometrics. Consequently, a new fitness measure was introduced, that instead of comparing the halfplanes used to define the visibility area, compared visibility areas *directly*.

Since many such visibility areas are infinite in size, it was necessary to only consider a finite part of them. Consequently only the area within a boundary twice the model's maximum radius was considered. This seemed reasonable, as any realistic sensor planning setup would be similarly constrained by the walls of the room it was located in.

Fitness was then measured by comparing the visibility areas delivered by the genetic program, G, with the correct area, F:

$$100 \times \frac{F \cup G - F \cap G}{F \cap G}$$

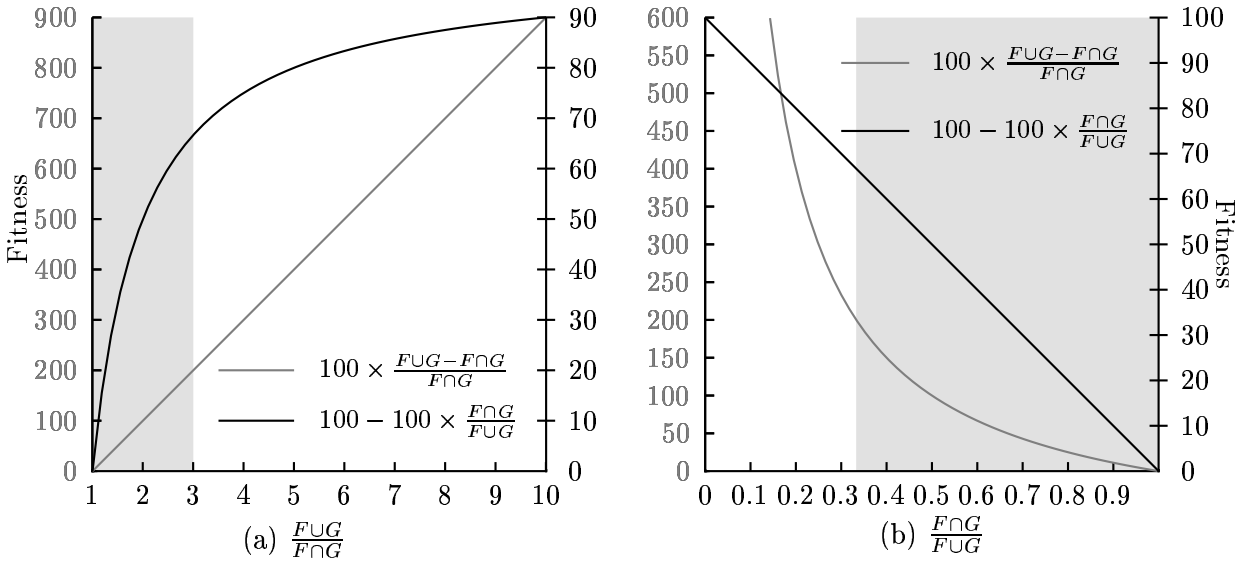


Figure 4.12: Comparison of fitness measures. Shaded regions mark the values for which most discrimination will be needed. $x = 1$ indicates a perfect solution.

(The multiplication by 100 is purely to make the resulting figures more intuitive for a human to understand; it makes the result into a percentage disjunction.)

The calculation of this fitness measure is illustrated in Figure 4.11. The denominator of the equation normalises the fitness with respect to the area involved; however, after some experimentation, it became apparent that this fitness measure was not very appropriate for the task in hand. The reason was that, though it performed well when the intersection between the genetic and correct answer areas was only a small fraction of the union of these (the grey line in Figure 4.12a), it did not distinguish well between the fitness of individuals with answers that had considerable overlap with the correct answer area, as indicated by the grey line in Figure 4.12b. Since most genetic programs of reasonable performance were expected to have considerable overlap with the correct solution, this meant that the closer two answers were to the correct solution, the worse the fitness measure discriminated between them.

Consequently, the fitness measure was altered to:

$$100 \times \frac{F \cup G - F \cap G}{F \cup G}$$

(that is, $100 - 100 \times \frac{F \cap G}{F \cup G}$). As can be seen from the black lines in Figure 4.12, this new measure discriminates fitnesses more finely in the useful region of the graphs (grey shading).

Moreover, Figure 4.15b on p. 135 shows that this change resulted in the fitnesses of static programs being rendered worse with respect to that of the hand-crafted programs, such that seed 7 was now more fit than the best static solution.

Both of these fitness measures included also various penalisations and rewards. These are summarised in Table 4.17 on p. 144, and shall be discussed in more detail in Section 4.7.3. The penalisation for length given in Section 3.6 continued to apply:

if $length > 150$ **then let** $fitness := fitness \times \frac{length}{150}$

The justification behind this was discussed on p. 81.

4.7.2 Altering the Fitness Cases

Initial attempts to discourage static solutions consisted of redesigning the fitness cases such that

```
add (halfplane (start, next (next (start))))
```

yielded solutions largely different from the correct one, and hence with a lower fitness. Figure 4.13 shows the revision of the fitness cases that was carried out (top line), and execution of the seeds and static solutions using these fitness cases.

When the system was rerun, however, a new optimum static solution was found:

```
add (halfplane (start, next (next (next (next (start))))))
```

On redistributing the fitness cases between these two data sets, it became apparent that static solutions, which reached medium fitnesses by finding the best compromise vertices for constructing answer halfplanes, still had better fitnesses than solutions evolving the computational constructs necessary for a correct answer. It was at this stage that the fitness function was completely overhauled, as described above. Figures 4.14b and 4.15b show the result of this revision, but as can be seen, static solutions retained better fitnesses than the low-end handcrafted programs, and so were still evolutionarily preferred.

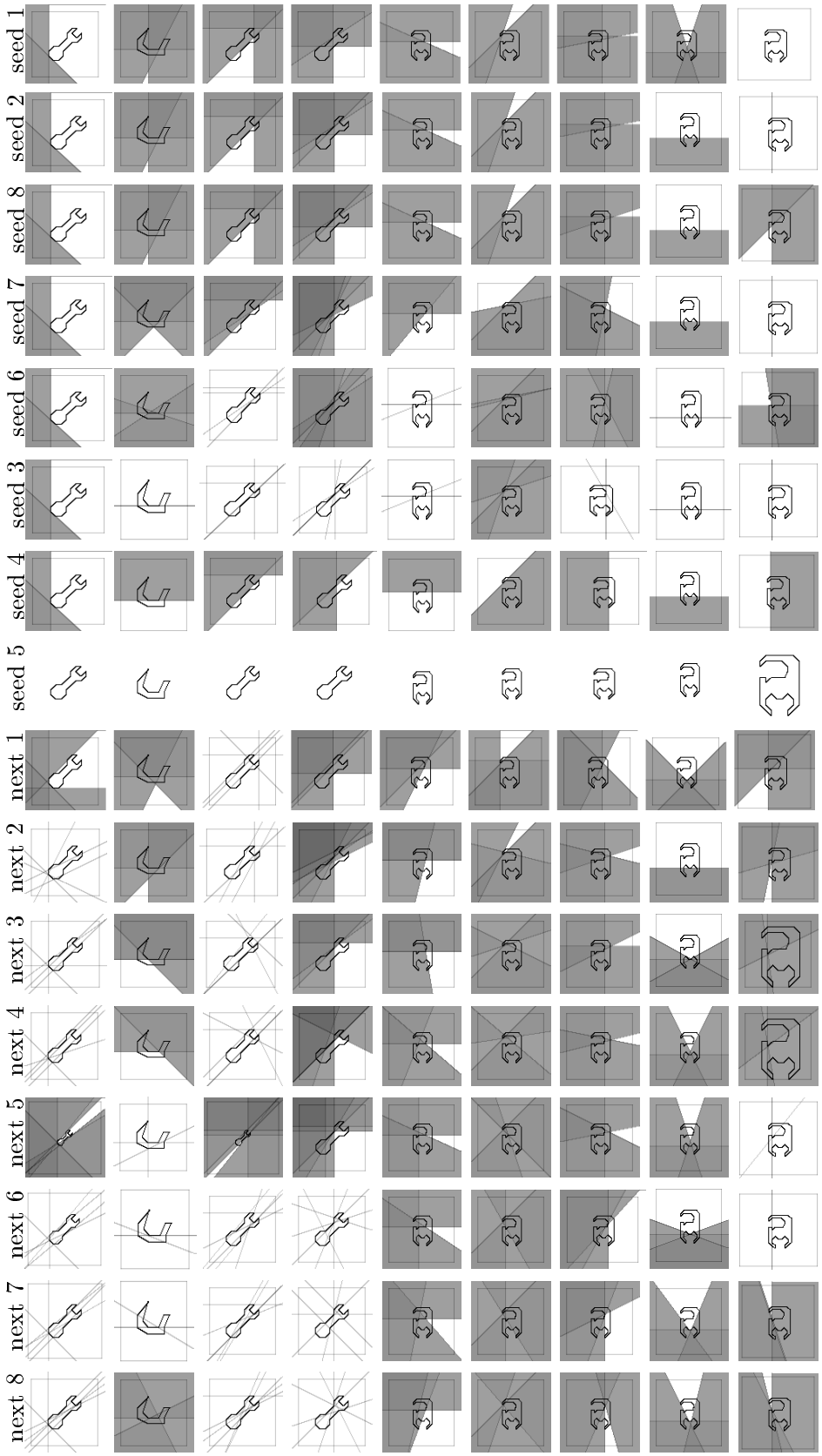


Figure 4.13: Execution of the programs in Figure 4.15b/4.14b.

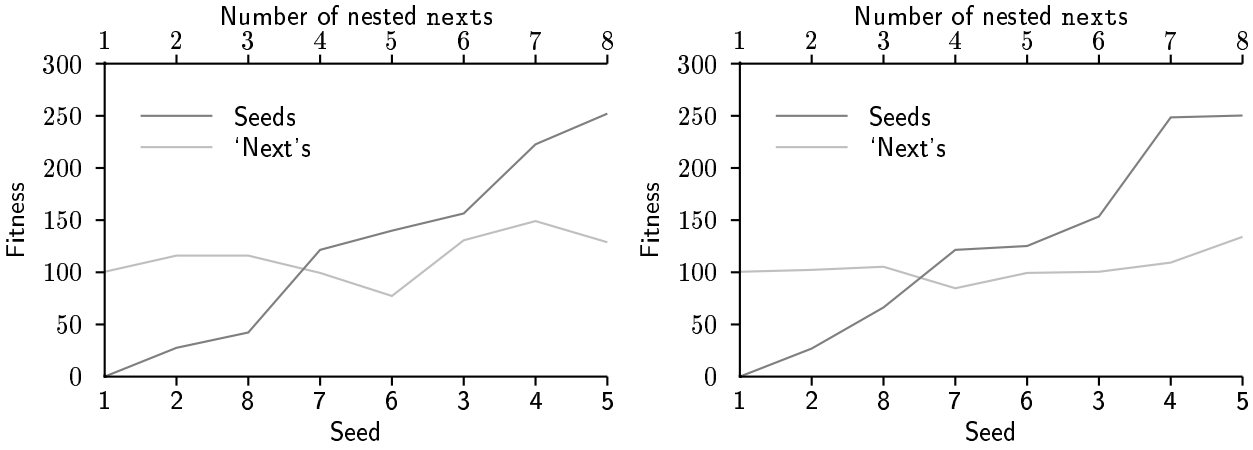


Figure 4.14: Fitness of the seeds and static solutions (a) with the original fitness cases, (b) with the revised fitness cases. Fitness base: $100 \times \frac{F \cup G - F \cap G}{F \cap G}$.

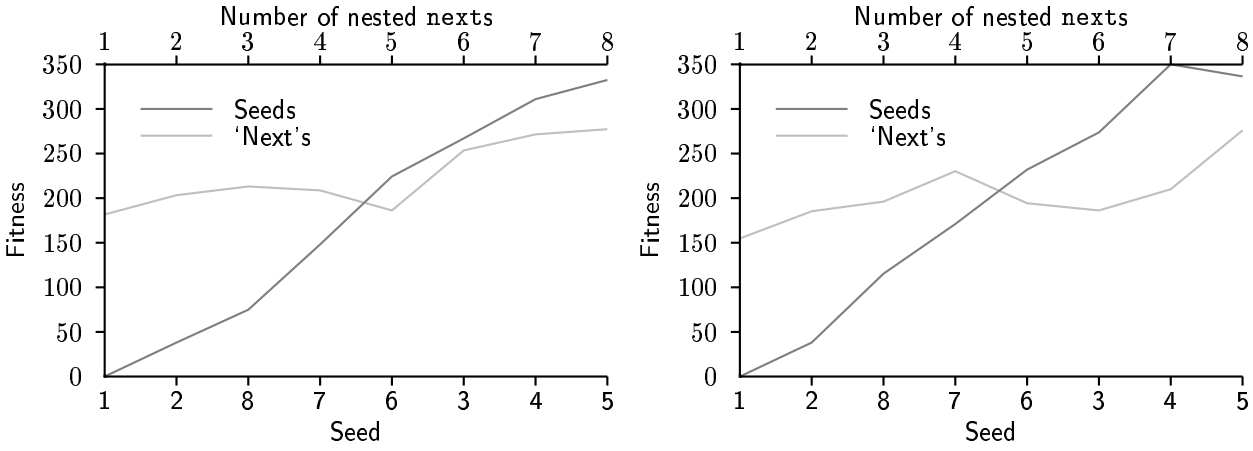


Figure 4.15: Fitness of the seeds and static solutions (a) with the original fitness cases, (b) with the revised fitness cases. Fitness base: $100 \times \frac{F \cup G - F \cap G}{F \cup G}$.

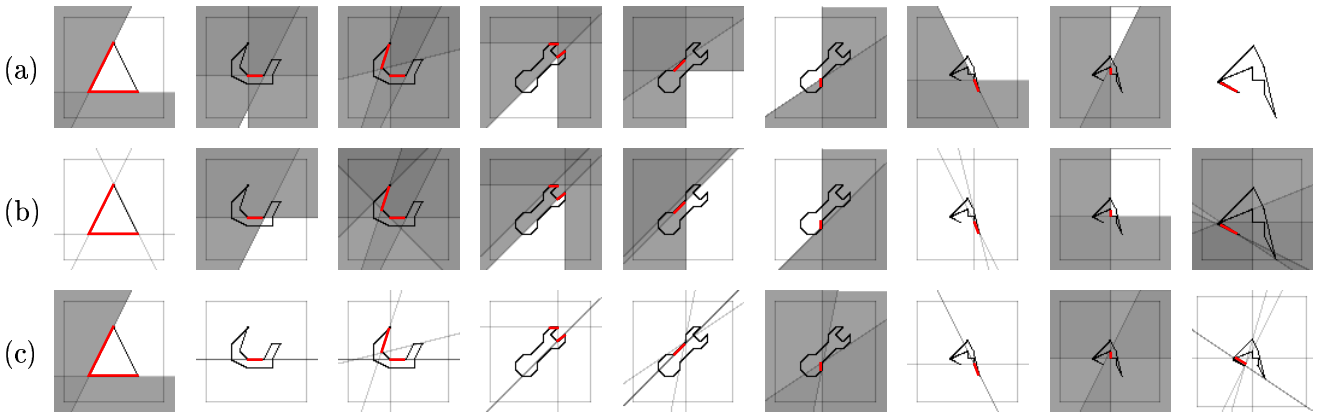


Figure 4.16: Execution of programs using the original fitness cases for the first typed system. (a) seed 1 (correct solution), (b) program on p. 136, (c) seed 3.

Moreover, one run produced an individual which, with the introns removed, boiled down to the following:

```
IF samesign (elevation (next (start), prev (prev (prev (start)))),
             - (0.0, 2pi))
THEN add (halfplane (start, next (start)))
```

Algorithmically speaking, this program ran on completely the wrong lines, but gave as good a fitness, and was closer to the correct answer, than seeded program 3. (See Figures 4.16 (b) and (c).)

Consequently a further revision of the fitness cases was undertaken (see Figure 4.17a), in which each static solution that gave a good result on any one fitness case would give a poor result on others. An example is shown in Figure 4.17b, where the static solution performs well on fitness cases 0 and 4, but extremely poorly on all the others.

Sample runs with these fitness cases are shown in Figure 4.18.

4.7.3 Parameter Tuning

GP parameters

There are various parameters to a genetic programming system (see Table 3.4) which can be altered to change the nature of the evolution.[48] For example, using a higher mutation rate can

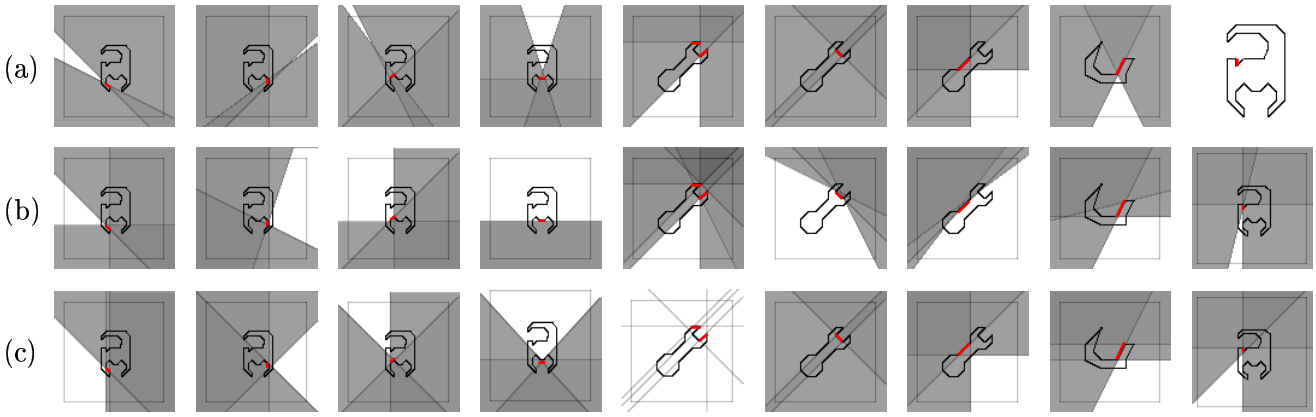
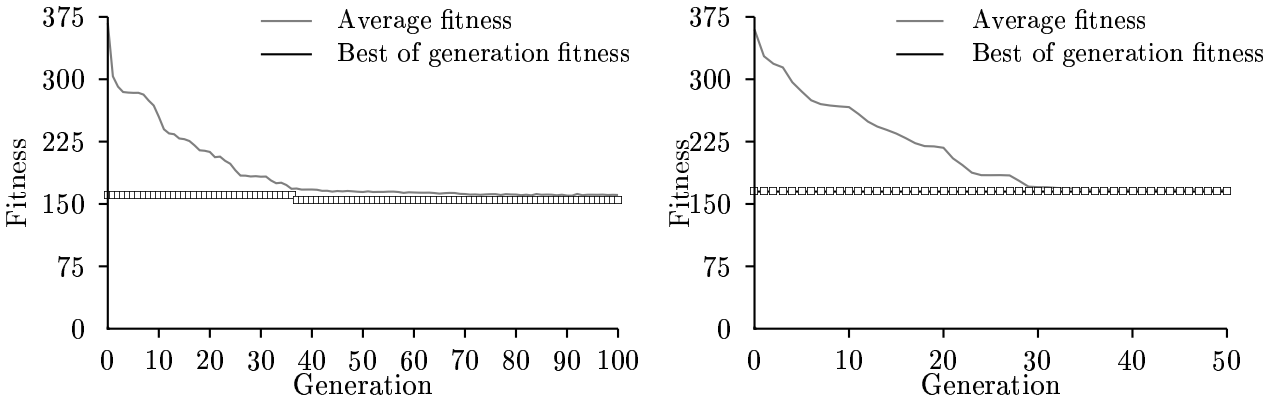


Figure 4.17: More difficult fitness cases for the first typed system. (a) Correct solutions; (b), (c) execution of static solutions.



Evolution style: steady state; Crossover: 90%, Creation: 2%; swap mutation: 5%; shrink mutation: 0%; random number seed: 887216647

(a)

Evolution style: steady state; Crossover: 75%, Creation: 2%; swap mutation: 15%; shrink mutation: 5%; random number seed: 886529557

(b)

Figure 4.18: Evolution with the fitness cases shown in Figure 4.17a.

Parameter	Settings			
Crossover	90	75	30	10
Creation	2	2	12	12
Swap Mutation	5	15	35	45
Shrink Mutation	0	5	20	30
Figure 4.19	(a), (b)	(c)	(d)	(e)

Table 4.16: Configurations of genetic parameters investigated.

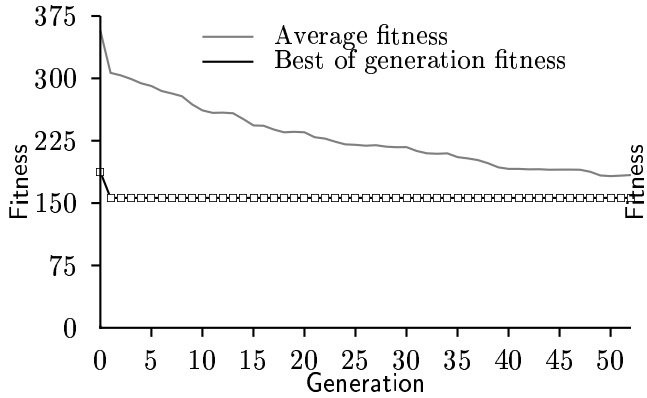
help programs escape from local optima; but also leads to the risk of useful program structures being lost through alteration. It has been remarked[48,82] that most Genetic Programming systems require a considerable amount of tuning of these parameters in order to optimise the evolution.

Alteration of these parameters would alter the ability of evolution to explore the fitness landscape. Table 4.16 shows the various configurations of genetic parameters used in experiments; and Figure 4.19 shows the results of runs carried out with these parameters. It can be seen that even varying the parameters over a wide range had little effect on the rate of convergence (as shown by the number of generations for the average fitness to converge on the best fitness). This fits the alternate view that in many problems, GP performs well under a broad range of crossover and mutation probabilities.[104]

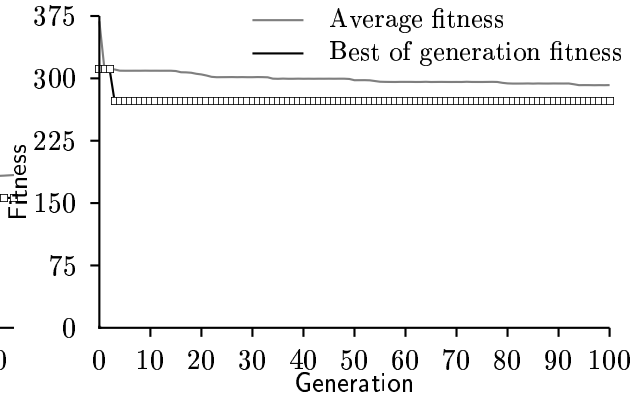
Fitness evaluation parameters

A second strategy was to alter the geometry of the fitness landscape itself. Comparing the fitnesses of the handcrafted partial solutions as against those of the static solutions (see Figures 4.14ff), it was hypothesised that if the line representing the static solutions could be pushed higher, that is above all bar the last two points on the other line (which represent effectively empty programs), static solutions would lose their selective advantage over non-static solutions. This could be attempted by altering the internal parameters of the system, such as the various penalties and rewards discussed in Section 3.6.

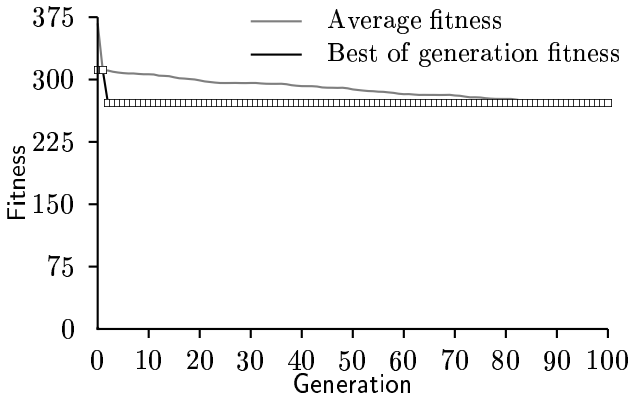
Whenever the fitness function was changed, the fitness was recalculated for the hand-crafted



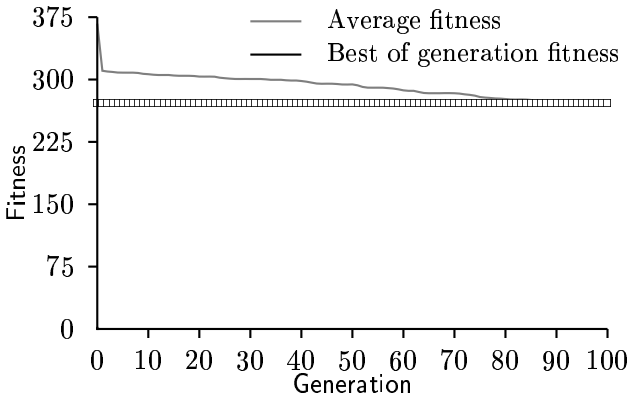
(a) Evolution style: steady state; Crossover: 90%, Creation: 2%; swap mutation: 5%; shrink mutation: 0%; random number seed: 887216641



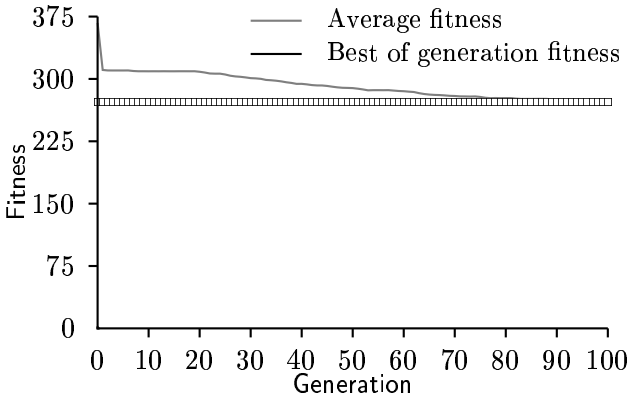
(b) Evolution style: steady state; Crossover: 90%, Creation: 2%; swap mutation: 5%; shrink mutation: 0%; random number seed: 887380395



(c) Evolution style: steady state; Crossover: 75%, Creation: 2%; swap mutation: 15%; shrink mutation: 5%; random number seed: 887385371



(d) Evolution style: steady state; Crossover: 30%, Creation: 12%; swap mutation: 35%; shrink mutation: 20%; random number seed:887650284



(e) Evolution style: steady state; Crossover: 10%, Creation: 12%; swap mutation: 45%; shrink mutation: 30%; random number seed:888417351

Figure 4.19: Evolution with the old fitness cases, (a) before and (b) after the removal of `prev` from the function set. (c), (d), (e) Runs with successively decreased crossover and increased mutation rates.

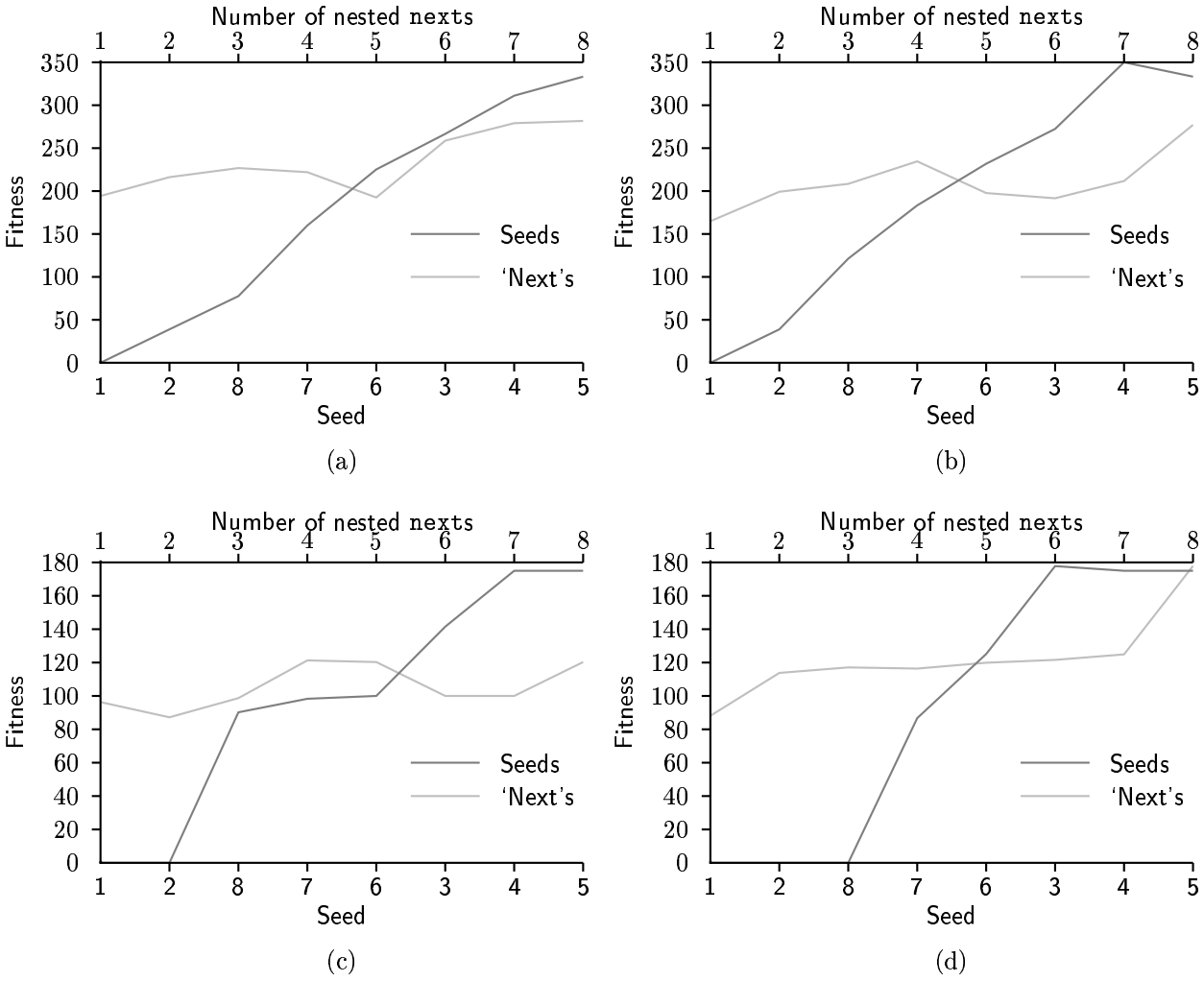


Figure 4.20: Fitness graphs with the raw fitness squared. (a) Original fitness cases, (b) fitness cases in Figure 4.17, (c) and (d) dynamically switching fitness cases.

programs (see Section 4.4.3) in order to verify that the fitness landscape had not acquired any undesirable local minima (see Figures 4.14ff).

Despite all the parameter tuning, so long as a single static solution retained a lower fitness than seed 8 (the third-last point on the line of handcrafted programs on the graphs in Figures 4.14ff), static solutions were still preferred; and furthermore parameter combinations which forced the static line upwards also tended to force the handcrafted line into a sigmoid shape, which for reasons discussed above (Section 4.4.3) was to be avoided. Eventually other measures were taken to prevent the formation of static solutions, as described in Section 4.7.4.

One change which was made early on was to alter the fitness function from the sum of

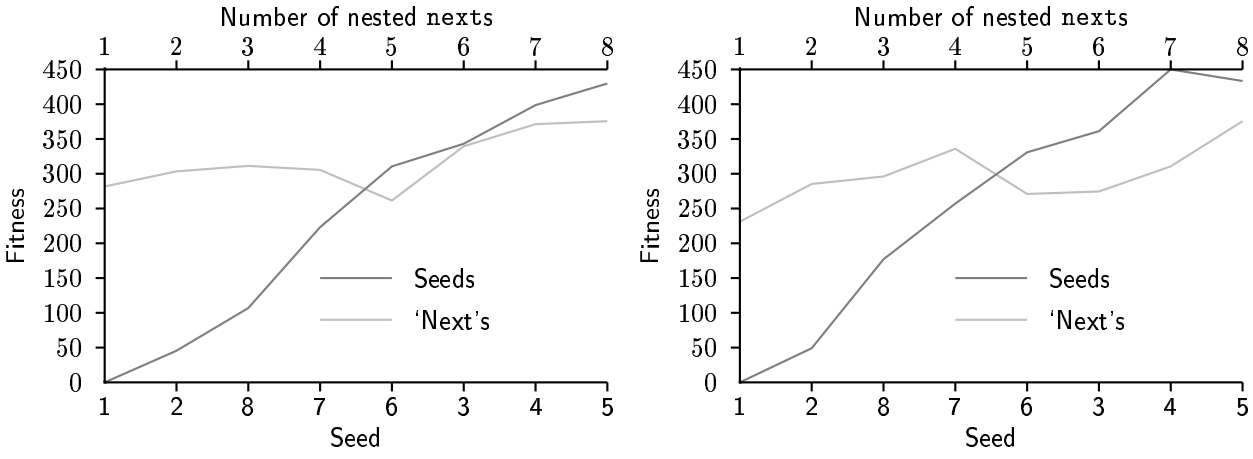


Figure 4.21: Fitness of the seeds and static solutions with an extra 2× *fitness-penalty* added for incorrect solutions; (a) with the original fitness cases, (b) with the revised fitness cases. (Cf. Figure 4.15.)

the individual raw fitnesses to the sum of their squares. The intention behind this was to discriminate better between programs with poor fitnesses. (Because tournament selection was being used, which is purely rank-based, this would have no effect on individuals' share of the roulette wheel. Instead, it would operate by lowering the ranking of programs with variable performance compared to those of the same average fitness with more uniform fitness on the fitness cases.) The fitnesses of the seeded programs with the new fitness measure are shown in Figure 4.20.

The penalty for an incorrect solution was increased from 2 × *fitness-penalty* up to 4 × *fitness-penalty* (see Figure 4.21), where *fitness-penalty* was a constant, set to 50, used as a basis for fitness penalties and rewards, as summarised in Table 4.17 on p. 144. The improvement, in terms of the performance of the static solutions compared to the hand-crafted partial solutions, was negligible, so the penalty was returned to its original value. Later, experiments were done with this penalty set to 10 × *fitness-penalty*. Evolution still led to static solutions under this configuration, however and, as can be seen from Figure 4.22, this configuration led to the fitness landscape becoming more sigmoid.

A second consequence of this parameter setting was that the less fit seeds did not get the proportional decrease in fitness that the more fit ones did. Because of this 100-point penalty,

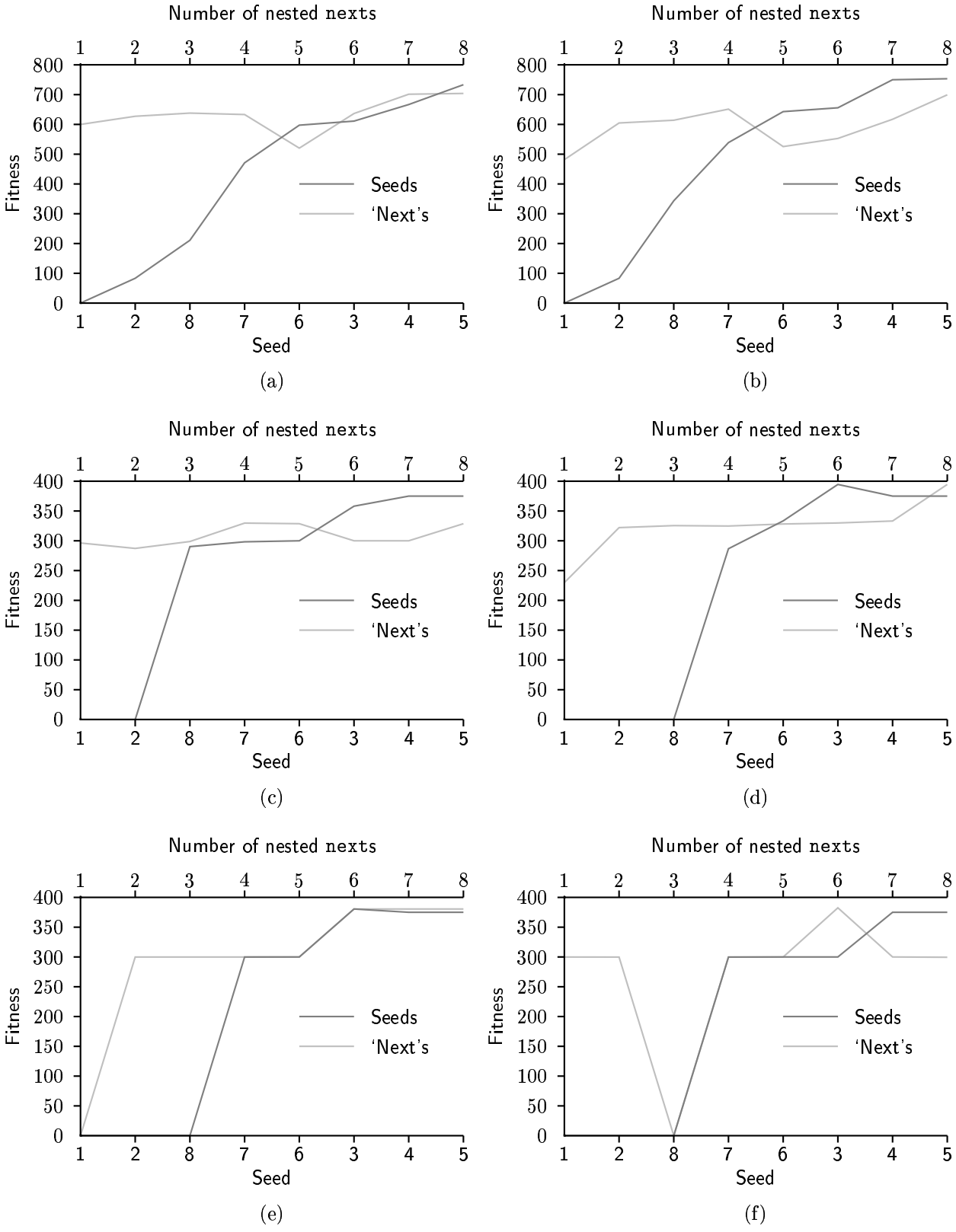


Figure 4.22: Fitness graphs with an extra reward for correct solutions. (a) Original fitness cases, (b) fitness cases in Figure 4.17, (c) and (d) dynamically switching fitness cases, (e) and (f) single dynamically switched fitness cases.

set against the 0–100 range of raw fitness for correct solutions, the difference in standardised fitness between a nearly correct program and a completely bad program was only 50%. If the penalisation were to be reduced, this proportional difference would be correspondingly increased, leading, it was thought, to the disjunction between the correct and genetic answer areas having an increased effect on the fitness of programs.

The penalisation was, accordingly, reduced. However, the expected effect transpired not to be the case, and the best results were found with a fairly high level of penalisation for incorrect solutions.

Table 4.17 shows the effect of varying the fitness calculation parameters on the fitness landscape. The first attempt to vary the parameters resulted in Program 4.8 (p. 153) possessing a better fitness than seed 7, which is algorithmically more complex. The parameters were therefore reconfigured to prevent this.

Figure 4.23 shows the fitnesses of the seeds with some of the fitness calculation parameters used, illustrating the fitnesses of seed 7 and Program 4.8, which is placed horizontally according to its relative complexity.

One set of parameters resulted in the first static solution with a low fitness seen since `prev` and `next` were removed from the function set (see Section 4.7.4). Two out of three runs carried out with these parameters were unable to evolve past this static program.

4.7.4 Altering the Function Set

Whilst other approaches were being used to discourage the evolution of static solutions (see Section 4.7.3), it was observed that the function `next` was not actually required for this problem. It had been included in the function set for such time as the complexity of the problem to be solved could be increased. This function was therefore now removed from the function set.

When this was done, a check was carried out that it would not be possible for static solutions to form with `prev` instead of `next`. The fitness of all static solutions of the form

`add (halfplane (prev`$_n$`(start), start))`

with up to twelve nested `prev`s was checked, and it was determined that they all had very poor

Parameter	**Revised values**				
fitness-limit	1000	1000	1000	1000	1000
fitness-penalty	50	20	80	80	70
incorrectitude-penalty	2	1	10	12	6
penalty-factor	5+2	20+1	10+10	8+12	4+6
Nodes changepoint 1*	242			880	420
Nodes changepoint 2*	60			177	80
Seed	**Fitness**				
seed 1	0	0	0	0	0
seed 2	38.9	46.7	177	177	77.8
seed 8	75.3	63.3	370	406	185.3
seed 7	148.7	103	754	861	400.1
seed 6	241.6	120	1047	1136	536.1
seed 3	299.9	338	1211	1229	568.8
seed 4	311.1	374	1422	1422	622
seed 5	311.1	374	1422	1422	622
Best static program		83			
Program 4.8*	147.3	76	749	911	432
Complement to Program 4.8†				917	
Best evolved program	147.3	76		917	432
Run results	Fig. 4.23a	Fig. 4.23b		Fig. 4.23c	

* See Section 4.7.5.

†
```
FOR SET/P[0] THROUGH MODEL POINTS
   IF > (0.0, elevation (mem/p[0], start))) THEN set/p [1] (mem/p[0])
END FOR
add (halfplane (mem/p[1], start))
```

Where:

if *aborted (genetic)* **and not** *aborted (correct)*
  or *genetic.size* = *correct.size* **and not** *convex (correct)* **then**
  rawfitness := *fitness-penalty* × *penalty-factor*
else
  rawfitness := $100 \times \frac{F \cup G - F \cap G}{F \cup G}$
  if *rawfitness* $\neq 0$ **then**
    rawfitness +:= *fitness-penalty* × *incorrectitude-penalty*

Table 4.17: Alterations made to the fitness evaluation parameters.

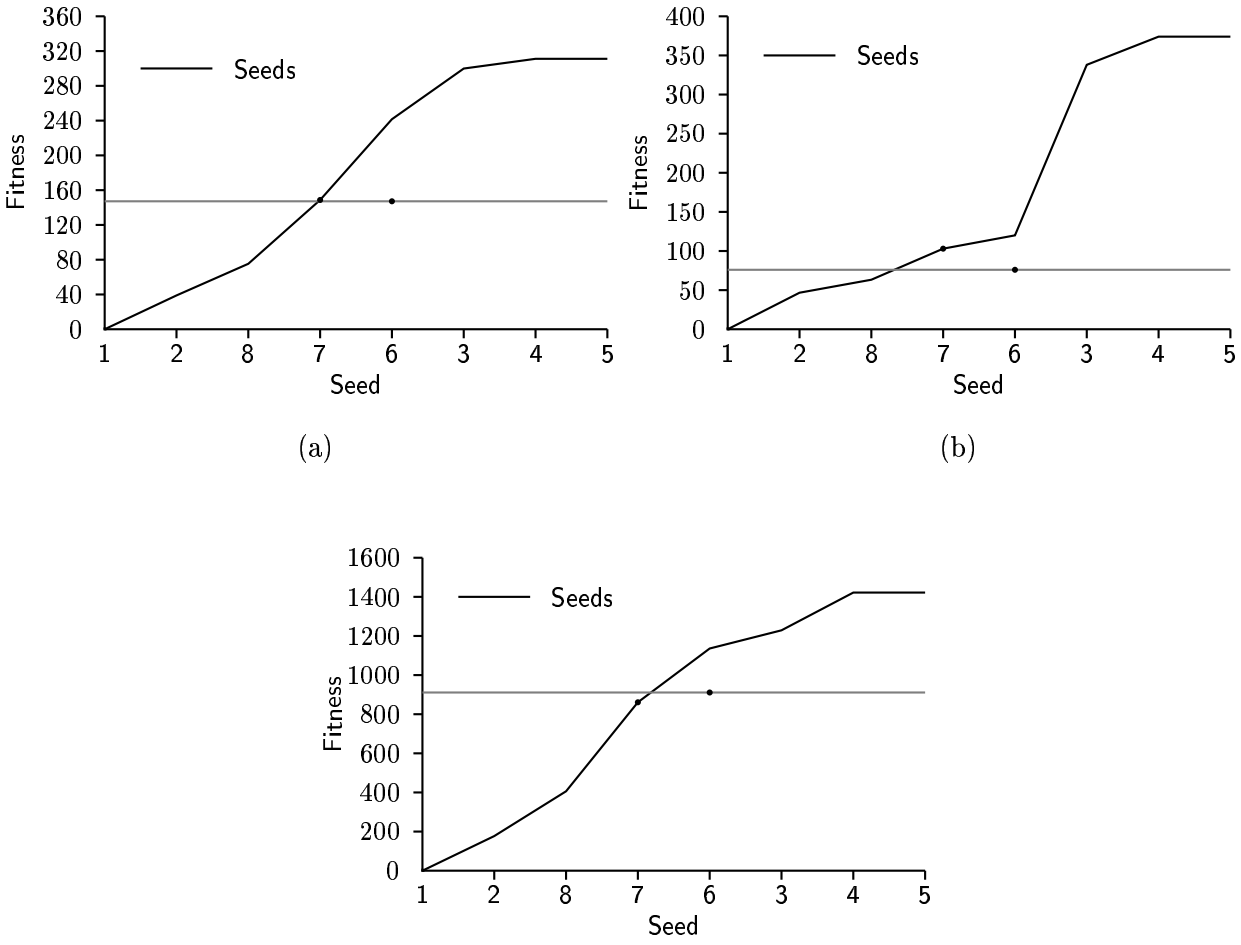


Figure 4.23: Seed fitness graphs for the different fitness evaluation parameters given in Table 4.17, showing the relative positions of seed 7 and the best static program.

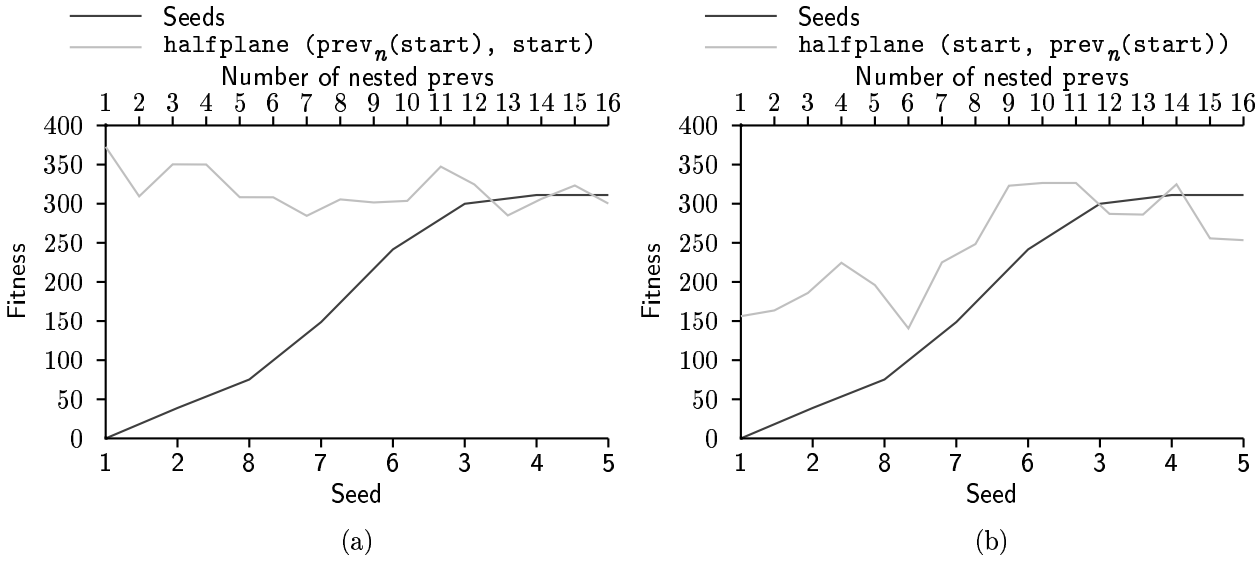


Figure 4.24: Effect of removing `next` from the function set.

fitness, as shown in Figure 4.24a: The line of static solutions has a fitness comparable to the worst of the handcrafted programs.

However, I had neglected to also check those with the arguments the other way around, viz.

`add (halfplane (start, prev`$_n$`(start)))`

Some of these had fitness better than all bar the best three seeds (Figure 4.24b). In the handcrafted programs, the function `prev` was only used once, and then only in the completely perfect solution. With a small alteration to the program architecture—the introduction of the program branch before the main `FOR` loop and the elimination of the one in the loop specification—this function too could be eliminated from the function set.

The elimination of this program branch would require the system to now automatically determined how far around the model to iterate. Since it is impossible to predict in advance how far around the model one must iterate before reaching the point subtending the highest elevation (see Figure 4.25), it was decided that the model iteration should not terminate until the iteration had proceeded right the way around the model (or execution was aborted).

This change to the program architecture would make the perfect solution slightly more complicated (see annotation to Program 4.4 on p. 114), but it was thought unlikely to prove a major deterrent in the discovery of a perfect solution. The reason for this was because the

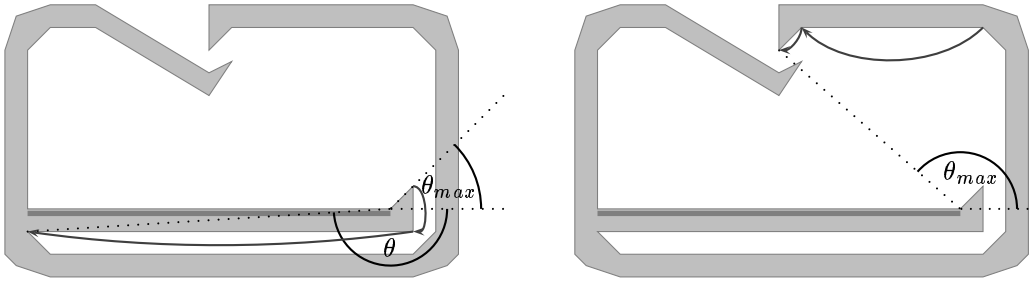


Figure 4.25: A (somewhat contrived) example to show the impossibility of predicting, whilst traversing around a model, whether one has traversed far enough to determine the highest elevation. On the left, clockwise traversal from the right-hand vertex of the feature has proceeded almost as far as $-180°$, but it is not far enough; the traversal has merely been tracing out an invagination, and vertices will subsequently be discovered subtending a higher elevation (right-hand figure).

perfect solution would not be discovered until evolution had found the previous stage, which provided almost all of the programmatic elements necessary for its construction. It was the first stage, constructing a useful program from random code, that would be the hard step.

However, as mentioned above, evolution will find any loophole in the fitness function, and a new static solution was found:

```
add (halfplane (mem/p[0], start))
```

The reason for this was that `mem/p[0]`, i.e. cell 0 of the points memory, was used as the loop index in the `FOR` loop; when the loop terminated the cell held the point corresponding to `prev (start)`. Hence the `FOR` loop was altered to clear its index variable after the loop terminated.

This would not prevent evolution from finding a new static solution; all it would have to do is copy `mem/p[0]` into another memory cell within the loop body—as seed 1 now did—and use that in constructing the answer halfplane instead. It seems that constraining the system to make it impossible to create a static solution would probably make it extremely difficult to find the correct solution too; but if the effort involved in finding a static solution could be rendered comparable to that in finding a partially-correct solution, then hopefully the local optimum in the fitness landscape represented by static solutions would cease to trap all paths of evolution.

Experiments confirmed this; Figures 4.19a and 4.19b (on p. 139) show the result of evolution before and after the removal of `prev` from the function set.

Use of ADFs

The initial implementation of the new system did not include the use of ADFs; these were added to the system as the experiments progressed. However, very few evolved programs made use of them. This tends to suggest that the ADF architecture as it stood was not a significant help to solving the problem. Since it has been shown[87] that many more complex problems cannot be solved without the use of ADFs, this may indicate that unless the ADF architecture were to be revised in such a manner that evolved programs utilised ADFs more, this system would continue to be incapable of solving the problem in hand.

One potential reason for the lack of use of ADFs is because their function and terminal sets were not powerful enough to be of practical use. However, this could not have been the case here, because they were provided with nearly complete function and terminal sets (see Table 4.10 on p. 110).

Another potential reason is because when static solutions were being evolved, evolution would converge on these before code complex enough to benefit from multiple subroutine calls could evolve. Since static solutions scored better fitness than more complex programs which used ADFs, they would be preferentially selected for, and only the static solutions would be seen in the best-of-generation individuals reported.

However, even when static solutions were finally eliminated, little use was made of ADFs. The reason for this is probably bound up with the reason why this system failed to evolve good solutions at all, in that programs complex enough to benefit from being able to call code multiple times were never able to evolve with this system.

4.7.5 Dynamic Specifications

Another way investigated to diminish the measured goodness of static solutions was to make the system specifications dynamic. There are various ways of going about this, which may be divided into two groups: The first, described by Koza[87] amongst others,[97] is altering the fitness

function, or fitness cases, from generation to generation to avoid overtraining the system on the one set of fitness cases or fitness measure. The alternative, used by Daida *et al.*[42] and Rosca and Ballard[133] amongst others,[102] is to alter the fitness function, etc, after a substantial number of generations have passed. That way the system may be trained on a simple sub-task necessary for the solution of the full problem; once this sub-task has been solved the specifications are then changed for solving the full problem.

Both of these approaches were used here.

Generational alternation of the fitness cases.

The first approach was used in this work by switching the fitness cases between two radically different sets on alternate generations. These are illustrated in Figure 4.26, and the fitnesses of the seeded programs on executing them is shown in Figure 4.27.

When this strategy had failed to deliver the required effect, a more drastic approach was adopted, in which generations alternated between just two fitness cases (Figure 4.28). For the first of these, the correct solution consisted of:

```
halfplane (start, next (start))
```

Anything other than this would fare very badly, since all the other vertices on the model were far away from the correct vertices from which to construct this halfplane. For the second fitness case, the correct solution consisted of:

```
halfplane (start, next (next (next (start))))
```

Likewise, anything other than this would do extremely badly. The models used would be the same size, so the system would be unable to evolve a static solution using the lowest common denominator of one and three `next`s.

The fitnesses of the seeds and static solutions for these two are shown in Figure 4.29.

The intention of constructing this system was that if static solutions did arise, they would fare so poorly on the following generation that they would not be selected for reproduction or recombination, and would be weeded out of the population.

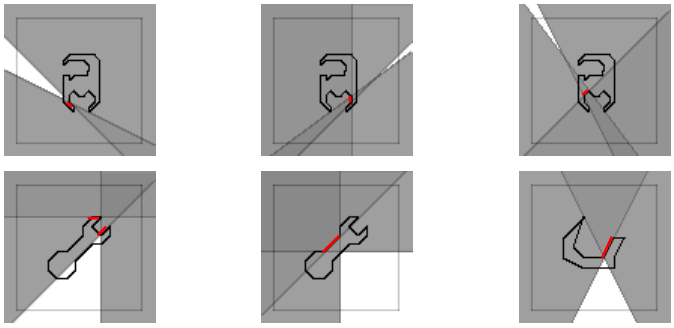


Figure 4.26: The two alternate sets of fitness cases used for dynamic fitness.

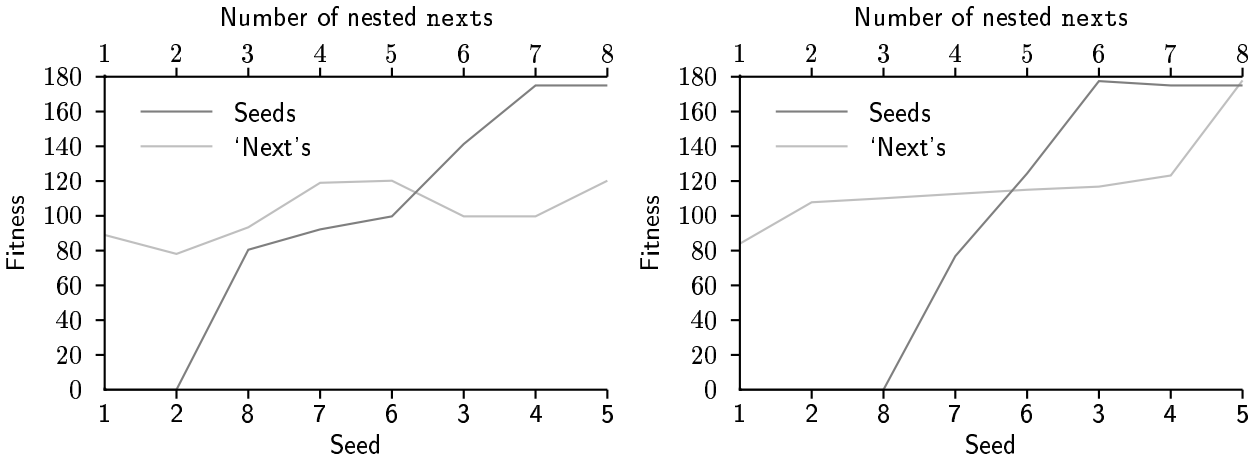


Figure 4.27: Fitness of the seeds and static solutions with the dynamically switching fitness cases.

Figure 4.28: Simple fitness cases for the single dynamically switched fitness case system.

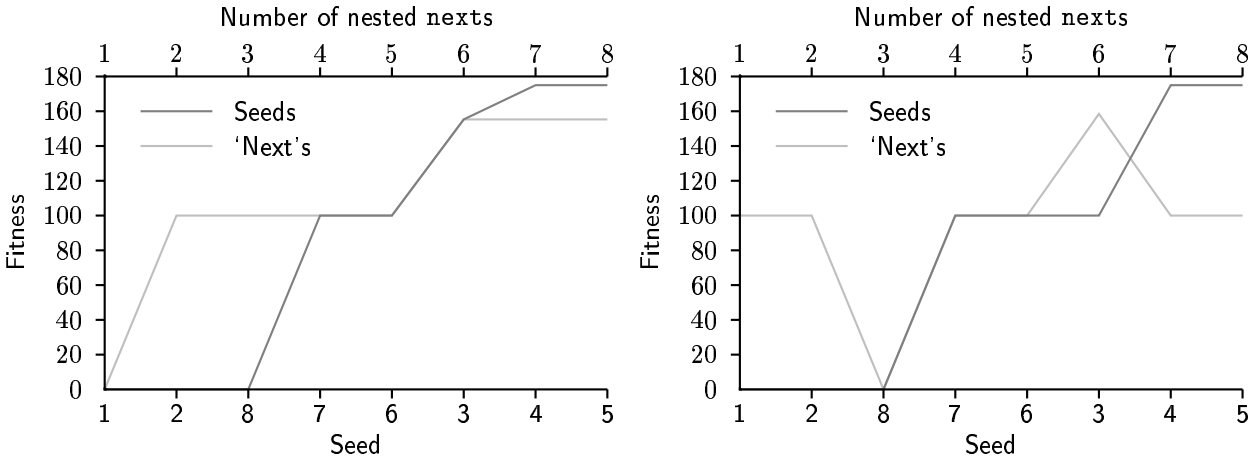


Figure 4.29: Fitness of the seeds and static solutions with two simple dynamically-switched fitness cases.

In practice this failed to happen: though static solutions fared badly on alternate generations, they did not fare sufficiently badly to be weeded out. One possible reason for this relates to the fact that evolution was steady state. In generational GP, every program is evaluated once every generation; but in steady state GP, though evaluations average out at the same rate, a program has a significant chance of not being evaluated throughout a generation-equivalent. Hence static solutions which perform well on one set of fitness cases can retain their good fitness throughout the period in which the other set is used. They would thus continue to have a good chance of being selected for recombination, and a poor one for being selected for replacement. Using steady-state GP thus effectively blurs the boundaries between the alternating fitness cases.

This hypothesis would explain the observation that on some runs a single individual persisted as the best of the population from generation to generation, even though it scored badly on every other generation. On other runs, however, two different static solutions alternated as best of generation. In that case, the above explanation does not apply, and static solutions must not have been weeded from the population because there was nothing better to displace them by being preferentially selected.

```
mem/a[0]
FOR SET/P[0] THROUGH MODEL POINTS
  &{ 0,
     if and (< (elevation (mem/p[0], start),
                elevation (mem/p[1], start)),
             < (mem/a[0], pi))
     then &{ 0,
             set/a[0] (pi),
             add (halfplane (start, mem/p[1])) },
     set/p[1] (mem/p[0]) }
  IF > (0.0, 0.0)
  THEN mem/a[0]
END FOR
IF > (pi, 0.0)
THEN halfplane (start, start)
```

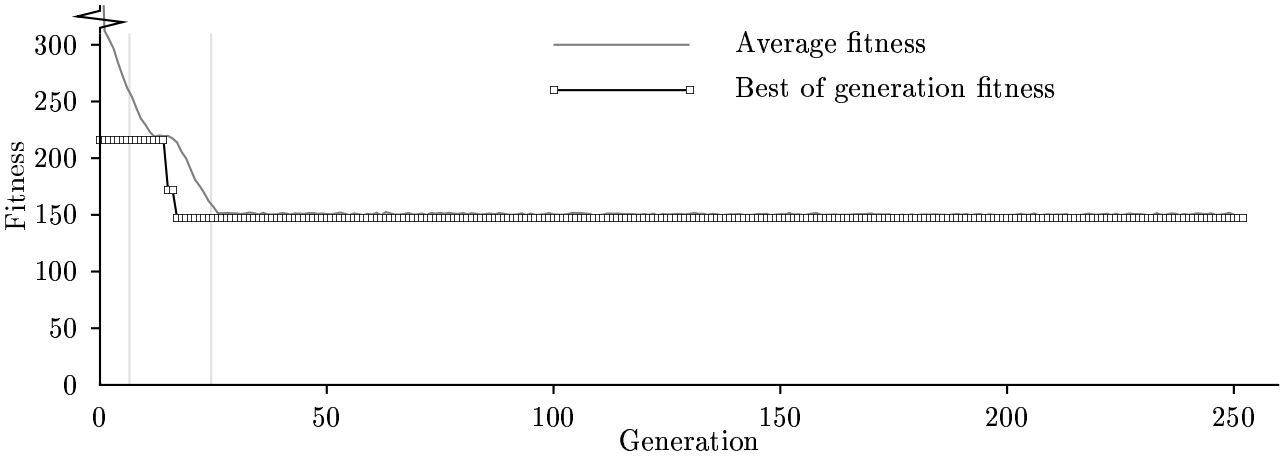
Program 4.7: Hand-crafted partial solution seed 8.

Fitness-dependent function and terminal sets.

The second approach was then adopted. It has been noted above (see p. 138) that superfluous functions and terminals degrade the probability of discovery of correct solutions in a linear fashion. Looking at the hand-crafted partial solutions, it became apparent that many of the functions and terminals in the programming language would only come in useful in the later stages of program development. For instance, seed 8 (the third hand-crafted data point in Figures 4.14ff), shown here as Program 4.7, requires only two of the seven angles memory cells—and hence only two of the seven integer terminals—necessary for a complete solution.

The function and terminal sets were therefore made dynamic in the following way.

The initial specification of function and terminal sets included only such program elements as were necessary to solve a certain subtask in the evolution of a correct program. Once the population had fulfilled certain criteria indicating that this subtask had been solved,[13] the function and terminal sets would be expanded to include the operation necessary to solve the next subtask, and the population regenerated to introduce these new genes into the gene pool.[133] The criteria used were that the best-of-population had reached a fitness only achievable by solving the current subtask, and that the average fitness was less than a certain threshold above this level, indicating that numerous programs had managed to solve this subtask but that the population



Evolution style: steady state; population size: 2000; crossover probability: 75%, swap mutation probability: 15%, shrink mutation probability: 5%; random number seed: 896959023

Figure 4.30: Evolution with dynamically altered node sets. Vertical lines indicate node set changepoints.

(a) Actual:

```
MAIN:
&{ 0; 0; 0}
FOR SET/P[0] THROUGH MODEL POINTS
  0
  IF not (< (set/a[1] (0.0),
             elevation (set/p[0] (start), start)))
  THEN set/p[3]
            (set/p [1] (mem/p[&{ adf1 (3, 1, 0.0); 0.0; 0}]))
END FOR
IF not (> (&{ 0.0; not (> (&{ 0.0; 0.0; 0.0}, 0.0)); 0.0}, 0.0))
THEN add (halfplane (mem/p[1], start))
ADF0: > (elevation (start, start), elevation (start, start))
ADF1: mem/a[arg0i]
```

(b) Interpreted:

```
FOR SET/P[0] THROUGH MODEL POINTS
  IF not (< (0.0, elevation (mem/p[0], start)))
  THEN set/p [1] (mem/p[0])
END FOR
add (halfplane (mem/p[1], start))
```

Program 4.8: Result of evolution with function and terminal set reduced to that necessary for finding seed 6.

was not yet converging upon a functionally equivalent set of programs (see Figure 4.30). The subsequent regeneration of the population was done by proceeding through it two programs at a time, and replacing the worse of them with a freshly created program.

This process could be repeated for as many subtasks as were judged necessary.[42] However, it was considered preferable to keep this number low, to minimise human intervention in the solution of the problem.

Program 4.8 and Figure 4.31 show the product of a 50-generation run of size 5000 with the function and terminal sets cut down to the minimum necessary to find seed 6. Seed 6 has fitness 241.6, this program has fitness 147.3, comparable to seed 7 (148.7). Figure 4.32 shows another run using dynamic node sets, which managed to evolve a program of similar functionality (Program 4.9, the execution of which is shown in Figure 4.33).

These both use the last model point with a negative elevation to construct a halfplane to add to the answer, using the points in the inverse order to the correct solution. This is contrasted with the correct approach in Figure 4.34. As can be seen, the answers that this approach gives have moderate overlap with the correct answer. More importantly, they work by making use of measured point elevations; hence they are not static in their means of operation.

This was as good as this approach was able to produce. One other result produced using this technique is worth mentioning. This is given in Program 4.10 and Figure 4.35. This program uses a point other than `start` for specifying the proximal end of the halfplane to construct; by doing this it achieves a better fitness (144.429) than seed 7. Furthermore it came up with reasonable solutions for some of the fitness cases that most programs did not do so well on, in particular the seventh fitness case.

4.7.6 Templates

Another technique used to guide evolution down the right track was to freeze part of the program in place. This was already partially the case with the framework for evolution shown on page 107; experimentation was now carried out with further freezing down of parts of the program, until only the terminals were allowed to evolve. The rationale for this was that since the dimensionality of the evolutionary parameter space (see Section 4.7.3) was so high, a thorough exploration of

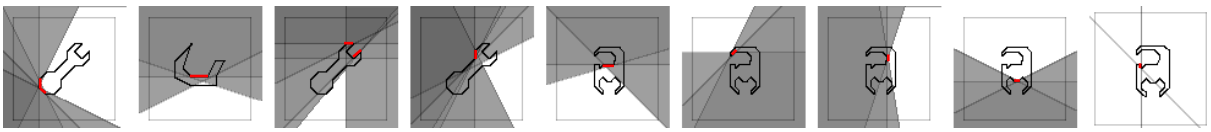
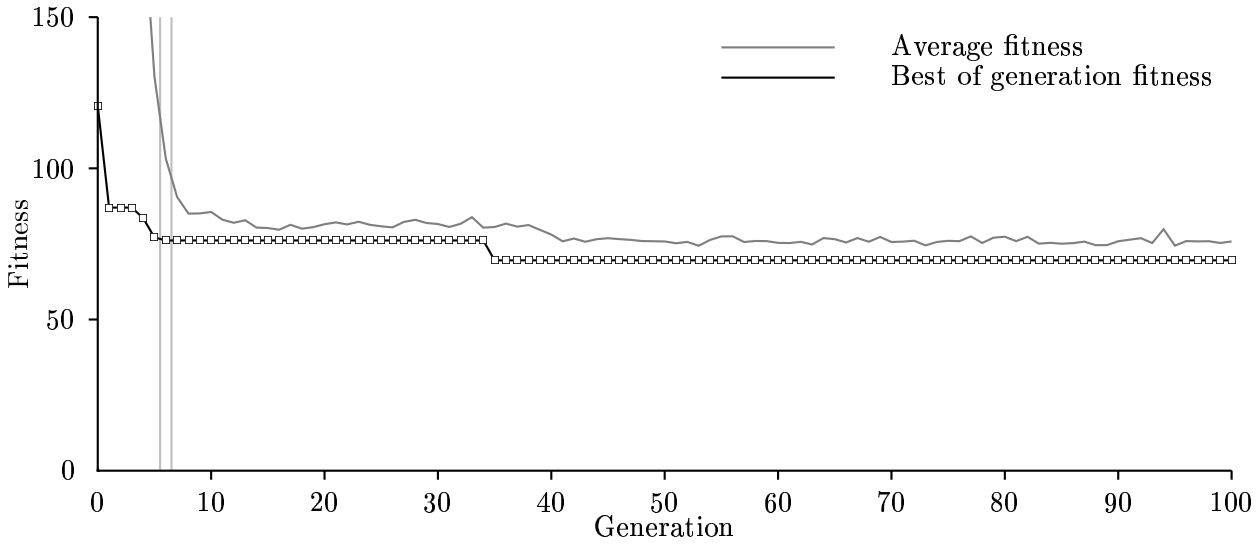


Figure 4.31: Execution of Program 4.8.



Evolution style: steady state; seeded programs: none; dynamic node sets; random number seed: 897042370

Figure 4.32: Evolution with dynamically altered node sets (100 out of 700 generations shown). Vertical lines indicate node set changepoints.

```
MAIN:
  0
  FOR SET/P[0] THROUGH MODEL POINTS
    1
    IF not (< (elevation (start, mem/p[3]),
               elevation (set/p[0] (start), start)))
    THEN set/p[3]  (set/p[&{ 0; 1; 0}] (mem/p[1]))
END FOR
  IF not (> (0.0, 0.0))
  THEN add (add (add (add (halfplane (mem/p[3], start)))))
ADF0:
  samesign (arg0a, arg0a)
ADF1:
  mem/a[arg0i]
```

Program 4.9: Best program of run in Figure 4.32.

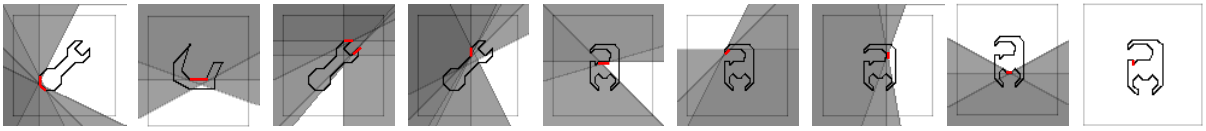


Figure 4.33: Execution of Program 4.9.

Figure 4.34: Comparison of the use of maximum and minimum elevations in constructing an answer halfplane.

```
MAIN:
 &{ mem/a[0]; 0; mem/a[0]}
 FOR SET/P[0] THROUGH MODEL POINTS
   &{ 0; 0; 0}
   IF < (elevation (set/p[1] (start), mem/p[0]),
         mem/a[3])
   THEN set/p[3]
              (&{ 0;
                  set/p[1] (mem/p[3]);
                  mem/p[0]})
 END FOR
 IF not (> (&{ 0.0; 0.0; 0.0},
            &{ 0.0; 0.0; 0.0}))
 THEN add (add (halfplane (mem/p[1], mem/p[3])))

ADF0: samesign (set/a[arg0i] (arg0a), &{ start; start; arg0a})
ADF1: mem/a[arg1i]
```

Program 4.10: An unusual evolved result.

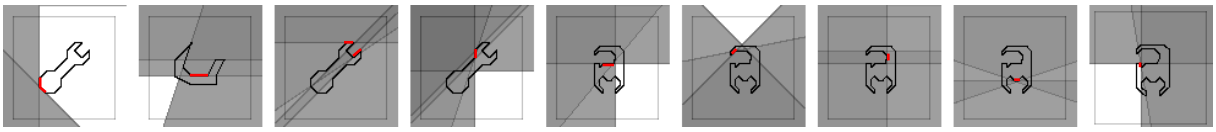


Figure 4.35: Result of executing Program 4.10.

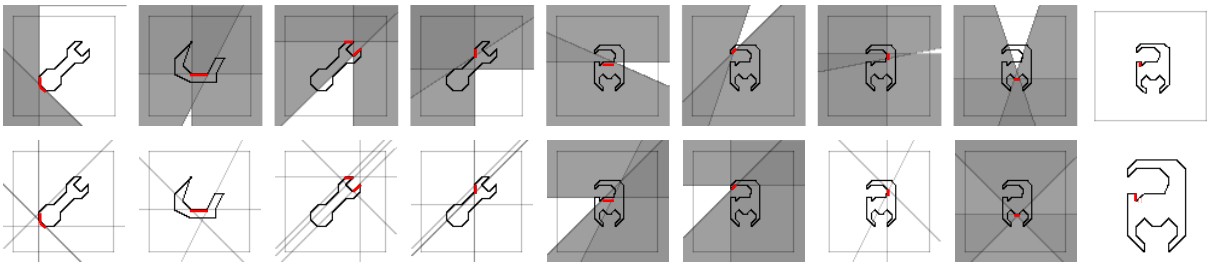


Figure 4.36: Execution of Program 4.11 (lower tier) compared to seed 1 (upper tier).

it, in order to find the optimum parameters for successful evolution of a correct program, would be infeasible. By reducing the size of the problem search space, it was reasoned, it would be possible to discover the optimal parameter configuration for evolution; once this had been found the frozen genes could be de-constrained to allow the entire program to evolve once more.

In actuality, frozen evolution did not proceed as easily as expected. Program 4.11 and Figure 4.36 show the results of a run in which the program had been frozen down almost completely. Capital letters indicate frozen code (or the fixed framework of evolution); italics indicate evolved code. Though there were only sixteen nodes to evolve correctly, evolution still failed to find the right answer. The only part of the frozen infrastructure of which use is made is the test for aborting the algorithm.

Out of six runs done to 1000 generations, three managed to correctly identify all eighteen terminals left evolvable in a seed 2 template, reaching completion of this task on generations 17, 38 and 936.

This requires identifying:

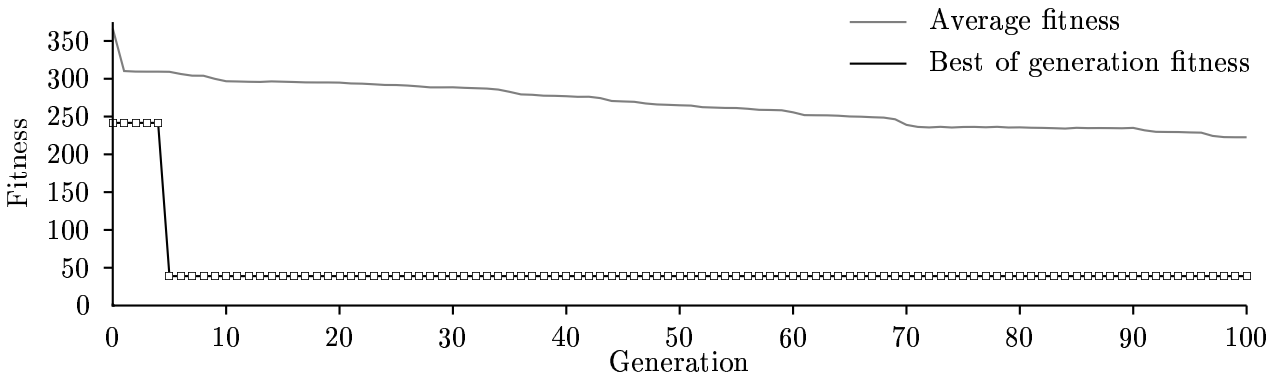
- 13 INTEGER terminals out of a terminal set of size 7
- 1 POINT terminal out of a terminal set of size 1
- 4 ANGLE terminals out of a terminal set of size 4

Only three variables are used by seed 2, of which one (the current point) is constrained by the system to be `mem/p[0]`. The other two may be freely chosen (i.e., have ${}^{6}P_{2} = 30$ permutations).

```
FOR SET/P[0], NEXT(START),  mem/p[5]*
   &{
    &{
      SET/A[0] (SET/A[3] (ELEVATION (MEM/P[0], START)));
      IF AND (NOT (> (MEM/A[6], -PI)), > (MEM/A[3], 0.0))
      THEN SET/A[0] (- (MEM/A[0], 2PI));
      IF NOT (SAMESIGN (MEM/A[3], MEM/A[4]))
      THEN &{
            2;†
           &{
             SET/A[5] (ELEVATION (MEM/P[0], PREV (MEM/P[0])));
             SET/A[4] (ROUND (+ (PI, MEM/A[6])));
             IF OR (AND (> (MEM/A[6], 0.0),
 < (MEM/A[5], MEM/A[4])),
                    AND (< (MEM/A[6], -PI),
 > (MEM/A[5], MEM/A[4])))
             THEN ABORT };
           IF AND (> (MEM/A[5], MEM/A[4]), > (MEM/A[3], 0.0))
           THEN SET/A[0] (- (MEM/A[0], 2PI))
        }
    };
    SET/A[6] (MEM/A[0]);
    SET/A[4] (MEM/A[3])}
  IF  > (0.0, pi)‡
  THEN &{  > (2pi, 2pi);†
          SET/A[1] (MEM/A[0]);
          SET/P[1] (MEM/P[0])
        }
END FOR

IF  < (mem/a[4], 2pi)**
THEN  add (halfplane (next (start), start))††
```

* Should be `prev(start)`.
† Value irrelevant.
‡ Should be `> (mem/a[0], mem/a[1])`.
** Should be `> (mem/a[1], 0.0)`.
†† Should be `add (halfplane (start, mem/p[1]))`.

Program 4.11: Attempt to evolve a correct solution from a mostly frozen template.



Evolution style: steady state; swap mutation rate: 45%, shrink mutation rate: 20%; random number seed: 888510883.

Figure 4.37: Discovery of a seed 2 hidden inside a seed 6.

Hence the size of the search space for this task, is

$$\frac{6^{13} \times 1 \times 4^4}{30} = 6.97^9$$

For runs of a thousand generations, with a population of one thousand, this should require 6966 runs to explore fully; GP did it in an average of 0.5 runs.

However, attempts to evolve less fully frozen templates were unsuccessful.

One problem with this approach was that programs algorithmically very close to the correct solution need not have good fitnesses, and would therefore be unlikely to be selected for recombination. This was a problem with adding seeded programs in general; if only one seeded program was added to the system, it could be recombined into a poorer version or lost from the population altogether before anything useful happened to it. This problem was solved in the following system by the addition of multiple seeds to the population.

In this system, however, experiments were undertaken to test the above hypothesis by concealing a subtree encoding seed 2 within an intron inside a program phenotypically identical to seed 6. To discover the subsumed program would require the alteration of just one terminal, or the changing of a `<` into a `>` in the predicate of the `if` that prevented the subsumed branch from executing.

In practice, the system showed itself able to do so within just a few generations, though it did not manage to do so on every run. However, the corollary was that the unmutated program

(seed 6-equivalent), though a poor performer, had been maintained as best-of-population for the half-dozen or so generations until the crucial mutation had been achieved. (See Figure 4.37.)

A second hypothesis was that the system was losing good programs. To test this, the system was seeded with programs of varying ability—seed 1 (the perfect solution), seed 6 and seed 8. Experimentation revealed that these seeds were not lost from the population.

Once the approach to freezing programs described above was shown not to work, another approach was adopted. Here, programs which had solved subtasks of the problem were frozen, and a newly created sub-program branch placed either side of the frozen one. The intention was to prevent the genes responsible for achieving these successful subtasks from being lost. This was done by assessing how many of the memory cells corresponded to their correct values on each round of the `FOR` loop; when a memory cell matched up for all the fitness cases the program would be frozen as described above.

This is similar to a technique described by Nordin and Banzhaf,[114] in which a few individuals were made "read-only" in each generation. This made their genetic information available for other individuals to utilise by means of crossover and reproduction, but also ensured that it was not destroyed in the process.

This strategy was not successful in my work, however, as the new program parts evolved on either side of the frozen parts would frequently undo the effects of the frozen part. Experimentation with seeded programs, in which the seeds would be frozen and new blocks added either side, failed to produce further progress either. Program 4.12 shows the result of such a run, containing several concentrically frozen blocks of code, with the code of seed 2 at their centre.

4.8 A Simpler System to Model Evolution

4.8.1 Rationale

The above system showed that evolving the correct answer to the visibility space problem was a much harder task than had been anticipated. Even once static solutions had been rendered difficult to evolve, the system failed to find more than the most elementary of partial solutions. The experiments that had been done offered little insight into the reasons why, or those why

```
MAIN:
MEM/A[5]
  FOR SET/P[0] THROUGH MODEL POINTS
    & add (add (abort));
       & ADF1 (& 1; 0.0; 2,
                & 2PI; PI; 4,
                - (-PI, -PI));
          & SET/P[& START; 2PI; 5] (MEM/P[0]);
             & & 0;
                   ADF0 (0, 5, -PI);
                   -PI;
                & & & 6; START; 2PI;
                      - (-PI, -PI);
                      MEM/P[2];
                   & 0;
                      IF NOT (> (SET/A[0] (ELEVATION (MEM/P[0], START)), 0.0))
                      THEN SET/A[1] (2PI);
                      IF AND (> (MEM/A[3], 0.0),
                              < (MEM/A[1], 2PI))
                      THEN SET/A[0] (MEM/A[3]);
                   HALFPLANE (SET/P[2] (START), SET/P[6] (START));
                & < (SET/A[4] (2PI), ADF1 (3, 4, PI));
             < (2PI, PI);
          < (MEM/A[0], - (0.0, -PI));
       halfplane (set/p[2] (start), mem/p[6])
    IF > (MEM/A[0], MEM/A[1])
    THEN & 0;
            SET/A[1] (MEM/A[2]);
            SET/P[1] (MEM/P[0])
END FOR
  IF > (2pi, 0.0) THEN ADD (HALFPLANE (start, MEM/P[1]))
ADF0:
  > (MEM/A[ARG0I], ARG0A)
ADF1:
  SET/A[ARG0I] (SET/A[ARG0I] (MEM/A[ARG1I]))
```

Program 4.12: Result of a run seeded with seed 2 in which correct use of memory locations led to incremental freezing of the program.

Problem				Node Set	
x_{max}	Σ	$\frac{\Sigma}{n}$	Π		
✓				`>` (FLOAT, FLOAT)	→ BOOLEAN
✓	✓	✓		`+` (FLOAT, FLOAT)	→ FLOAT
		✓		`%` (FLOAT, FLOAT)	→ FLOAT
			✓	`*` (FLOAT, FLOAT)	→ FLOAT
✓				`if` (BOOLEAN, GENERIC)	→ GENERIC
				`=` (FLOAT, FLOAT)	→ BOOLEAN
				`next` (INTEGER)	→ FLOAT
✓	✓	✓	✓	`mem` (INTEGER)	→ FLOAT
✓	✓	✓	✓	`set` (INTEGER, FLOAT)	→ FLOAT
		✓		`size`	→ FLOAT
✓	✓	✓	✓	`0...1/2/5`	→ INTEGER

Table 4.18: Genetic machine for the array manipulation problems.

evolution was not able to build up more complex solutions from them.

It was hypothesised that maybe there was a flaw in the system preventing further evolution from taking place. To test that this flaw was not in the library upgrading GPC++ to typed GP, the system was converted to run using untyped GP, but evolution managed to proceed no further using untyped GP.

It was then hypothesised that maybe the fault lay with GPC++ itself. To test this, the visibility space problem was converted into Lisp to run on lilgp, the original GP system devised in Lisp by John Koza.[84,87] This too failed to evolve any good solutions.

Consequently it was decided to implement a simpler system, to investigate the behaviour of evolution. Since visibility space calculation involved iterating through a list of data and searching for a maximum value related to these data, the model problem chosen was one for carrying out similar manipulations on arrays of numbers.

4.8.2 System specifications

The problems tackled in this study were finding the largest number in an array, finding the sum of the values in the array, finding the average of the values in the array and finding the product of the values in the array. Table 4.18 shows the function and terminal sets used in the genetic machine; an example correct program is shown in Program 4.13. A six-celled state memory was

```
set[1] (mem[0])
for loop through array
  if > (mem[0], mem[1]) then set[1] (mem[0])
end for
mem[1]
```

Program 4.13: Correct solution to the largest-in-an-array problem.

Parameter	**Setting**
Population size	1000
Number of generations	100, 200
Selection type	Tournament, size 10
Demes	10 when used
Creation Type	Ramped half and half
Crossover Probability	75%
Creation Probability	2%
Maximum Depth For Creation	6
Maximum Depth For Crossover	10
Demetic Migration Probability	100%
Swap Mutation Probability	15, 35%
Shrink Mutation Probability	5%
Population replacement	Steady State

Table 4.19: Tableau for the array problems.

provided. The fitness measure used was the difference between the correct answer and the value returned by the genetic program, expressed as a fraction of the latter (if this was not equal to zero). This allowed the concept of partial fitness, essential to having a sloping, and hence navigable, fitness landscape, to be implemented.

The fitness cases were designed such as to provide a broad coverage of possible answers, to discourage the evolution of static solutions.

4.8.3 Experiments and Results

Analysis of initial results

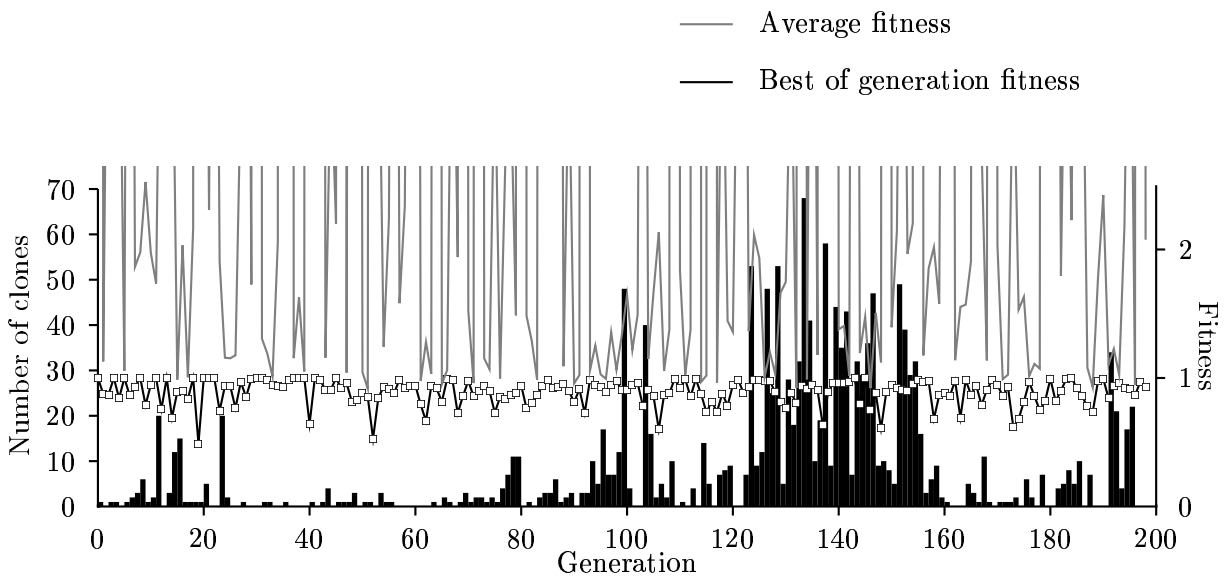
Initial results were unencouraging. The system appeared to be unable to solve the problems whether mutation rates were null, medium (15%) or high (35%), demes were used or not used, the population was the small value given in the tableau in Table 4.19 or large (100000 with demes of 10 000 and a migration probability of 50%), or the fitness function involved sum of absolute differences or sum of squares. Since this system was uncomplicated and the programs executed quickly, it was possible to carry out large numbers of runs for each of these experiments, up to 300 in some cases.

An initial hypothesis as to what was going wrong was that the system was converging too early. This was investigated by an analysis of the prevalence in the population of clones (genetically identical individuals). As can be seen in Figure 4.38, there were few clones in the population. Therefore premature convergence was not the cause of what was going wrong.

When a set of 279 runs with 35% swap mutation and 15% shrink mutation failed to find the answer, this raised questions about the coverage of the program space. Table 4.20 shows an analysis of the search space. For the minimum size tree (4 deep) needed to solve the problem in hand, the search space is 30 000 times larger than the number of individuals processed in these runs, but this ought to be within the capability of GP to explore.

This suggested something was wrong with the exploration of the program space. The nodes possibilities table was examined (Table 4.21) but found to have the form expected. (The nodes possibilities table of the visibility space system was also examined (Table 4.22); this too was in the form expected.)

Tables were then generated showing the actual usage of nodes in program construction (Table 4.23). The tables show biases in node usage; for example there were far more `mem`s than `set`s. This was because `mem`s could be constructed with a depth of just two, whereas `set`s required subtrees of depth 3. Re-counting the nodes usage for trees only of depth three or more evened things up (Table 4.24), though due to the composition of the node sets, it was not possible for all nodes to be used in equal proportions.



Evolution style: steady state; random number seed: 891880546

Figure 4.38: Clonal analysis of the array average problem.

Depth	Trees	3-branched programs
1	7	343
2	46	97 336
3	1097	1.3×10^9
4	20 404	8.4×10^{12}
5	387 548	5.8×10^{16}
6	7.3×10^6	4.0×10^{19}

Table 4.20: Table showing size of the program tree generation space.

Subtree Height	Node															
	`and, or`		`not`		`+, *, %`		`<, >`		`if`		`mem`		`set`		`0 - 5`	
	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow
1															✓	✓
2											✓	✓				✓
3					✓	✓	✓	✓				✓		✓		✓
4	✓	✓	✓	✓	✓	✓	✓	✓	2	3		✓		✓		✓
5	✓	✓	✓	✓	✓	✓	✓	✓	2	3		✓		✓		✓

Table 4.21: Nodes Possibilities Table for the array problems system. For reasons of space, type information has been omitted; numbers indicate number of occurrences of GENERIC and INDIFFERENT nodes.

Subtree Height	Node																											
	and not or		+ -		same-sign round		< >		half-plane		elev-ation		add		&		if		mem/a		set/a		mem/p		set/p		0 - 6		-pi pi 2pi		0.0		start		abort	
	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow	Full	Grow
1																											✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	4	4			✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓		✓		✓
3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	5	5	5	5	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓		✓		✓
4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	5	5	5	5	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓		✓		✓
5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	5	5	5	5	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓		✓		✓

Table 4.22: Nodes Possibilities Table for the Visibility Space Problem. For reasons of space, type information has been omitted; numbers indicate number of occurrences of GENERIC and INDIFFERENT nodes.

Maximum depth 3:

Type	Node															
	not	and	or	+	-	*	%	>	<	if	mem	set	0	1	2	3...
—	0	0	0	167	186	174	176	392	386	0	14 863	42	0	0	0	0
FLOAT	0	0	0	170	163	162	179	0	0	0	11 871	56	0	0	0	0
BOOLEAN	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
INTEGER	0	0	0	0	0	0	0	0	0	0	0	0	4527	4538	4477	4448

Maximum depth 6:

Type	Node													
	not	and	or	+	-	*	%	>	<	if	mem	set	0	1...
—	90	77	91	154	114	146	123	224	170	321	1052	44	0	0
FLOAT	0	0	0	1709	1718	1647	1765	0	0	607	14 448	517	0	0...
BOOLEAN	188	198	189	0	0	0	0	1038	1118	172	0	0	0	0...
INTEGER	0	0	0	0	0	0	0	0	0	155	0	0	2702	2792...

Maximum depth 10:

Type	Node												
	not	and	or	+	-	*	%	>	<	if	mem	set	0
—	89	117	107	128	125	145	140	147	162	517	350	53	0
FLOAT	0	0	0	17 137	17 302	17 329	17 362	0	0	7747	10 7964	3819	0
BOOLEAN	2425	2409	2531	0	0	0	0	8989	8969	2489	0	0	0
INTEGER	0	0	0	0	0	0	0	0	0	1577	0	0	18 617

Table 4.23: Nodes usage tables for the array problems system, in a population of 1000.

To determine whether the inability of the system to come up with a correct solution was an undiscovered effect of the typing, the array problem was reimplemented without types, in lilgp in Lisp. This too was unable to solve the problem.

Alterations to the system specifications

A better clue was provided by the fact that many programs consisted largely of `<oper> (0.0, 0.0)` with very few occurrences of `set` and `mem`. `0.0` was not actually necessary to solve these problems; it was a leftover from the visibility space problem. Removing it from the terminal set failed to allow the problems to be solved; but when the terminals `2` to `5`, which were also not necessary, were removed too, the system was now able to solve the sum-of-array problem. This agrees with the previously remarked observation (p. 138) that superfluous functions and terminals degrade performance in a linear manner.

The above suggests that the sensor planning system would be unable to evolve solutions with a full complement of functions and terminals given that partial solutions tend not to require the use of all of them; and that the dynamic node sets described in Section 4.7.5 may be necessary to the solution of the problem.

The average-of-array problem was finally solved by combining the use of these cut-down nodes sets with increasing the population size to 4000 (see Figure 4.39 and Program 4.14b). The fact that such a large population was necessary to solve this simple problem suggested that the size of population necessary to solve the considerably more complex visibility space problem would be larger than practicable.

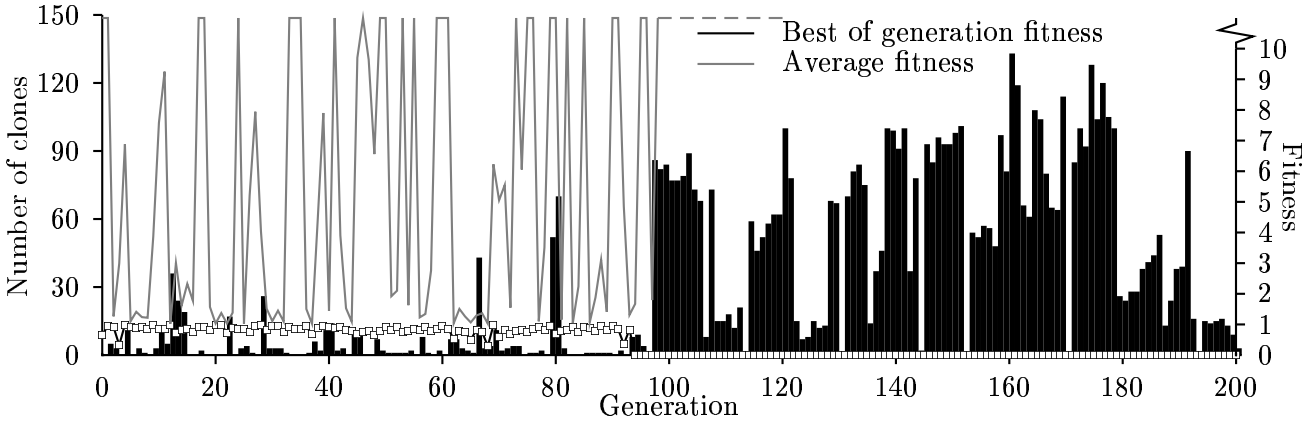
The figure also includes an analysis of clones for a run of the system solving the problem. As can be seen, the prevalence of clones was again low. Note, however, the reduction in genetic diversity after a correct solution had been found. Even so, compared to the population size (4000), the number of clones is still low. Clonal analyses were also carried out for the system with the superfluous terminals (see Figure 4.40).

A clonal analysis was carried out on the visibility space problem (Figure 4.41). With a larger set of functions and terminals, and therefore a larger search space, the prevalence of clones in this problem was even lower.

Maximum depth 6:

Type	Node														
	not	and	or	+	-	*	%	>	<	if	mem	set	0	1	2 ...
—	82	75	96	127	116	148	131	194	176	373	49	41	0	0	0
FLOAT	0	0	0	1792	1665	1665	1712	0	0	583	519	490	0	0	0
BOOLEAN	208	204	171	0	0	0	0	1080	1130	208	0	0	0	0	0
INTEGER	0	0	0	0	0	0	0	0	0	143	0	0	109	111	108

Table 4.24: Nodes usage table for the array problems system for trees only of depth three or greater.



Evolution style: steady state; population size: 4000; demetic grouping; random number seed: 892130198

Figure 4.39: Clonal analysis of a successful solution of the average-of-array problem. The best of generation 3 is a static solution.

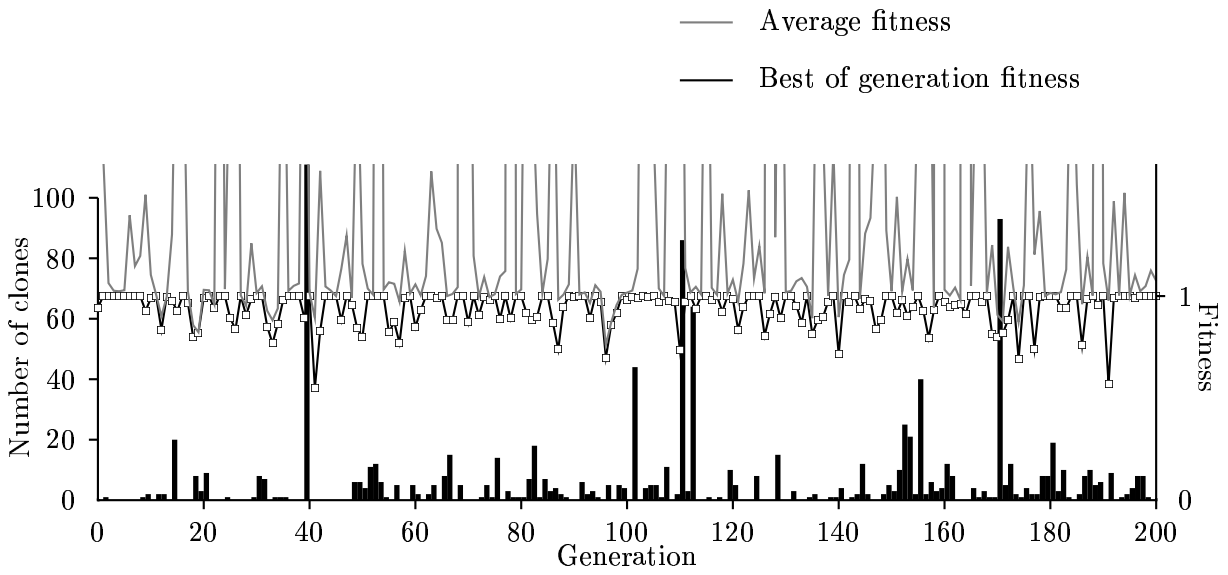
```
% (size, size)
for loop through array
  mem[1]
end for
% (set[0]
      (+ (+ (size, size),
          + (% (size, size),
               set[0] (size)))),
   size)
```

(a)

```
size
for loop through array
  set[1] (+ (% (mem[1],
                % (mem/[0], mem[0])),
             % (mem[0], size)))
end for
mem[1]
```

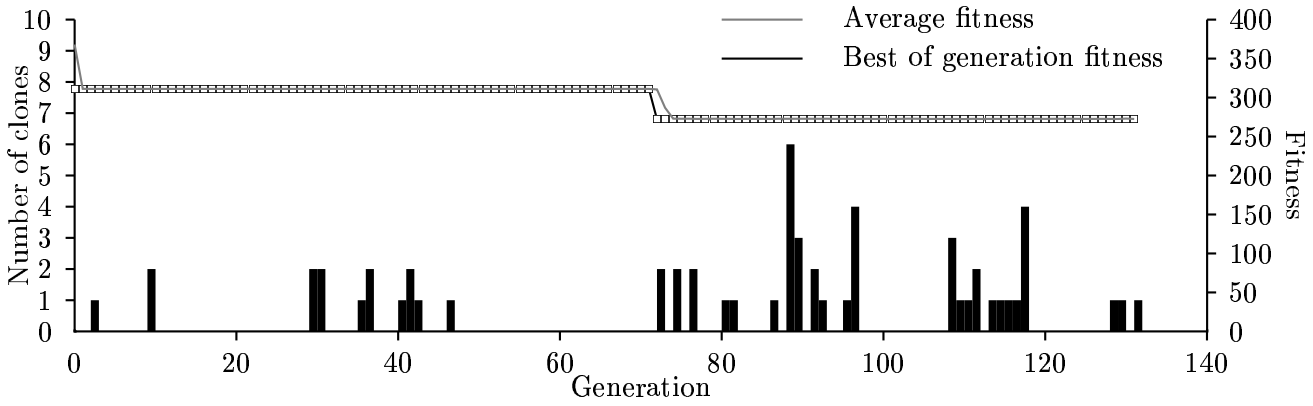
(b)

Program 4.14: Programs from the average-of-an-array run shown in Figure 4.39. (a) Static solution evolved on generation 3. (b) Correct solution to the average-of-an-array problem, evolved in generation 94 and whittled down to the minimum size it reached by generation 96.



Evolution style: steady state; no demes, random number seed: 952013159

Figure 4.40: Clonal analysis of the array average problem with extraneous terminals.



Evolution style: steady state; (population size: 100); random number seed: 89161648

Figure 4.41: Clonal analysis of the sensor planning problem.

4.9 Conclusions

This chapter has described how a strongly-typed GP system with a focused, minimalist function and terminal set was unable to solve the visibility space problem, and was instead liable to functionally converge onto static solutions, which achieved moderately good fitness without making intelligent use of the input data. The following conclusions can be drawn from the methods employed in this chapter:

- Seeding the population had little positive effect in this system. Though seeded programs were not lost from the population, they were not improved upon either, and seeds that were slightly altered were not easily rediscovered.
- Though altering the fitness function to make it based on area, rather than a blueprint for that area, resulted in a better mapping between the fitness function and the objectives of the system, in practice there was little evidence that its adoption led to an improvement in results.
- Altering the fitness cases to prevent the evolution of static solutions merely led to different static solutions being found, even when those static solutions performed poorly on up to half of the fitness cases. Dynamic alternation of the fitness cases was not sufficient to weed static solutions out, due to lack of better programs to replace them.
- No obvious improvement in evolutionary performance was seen by tuning the parameters of the GP system; this was probably, however, a symptom rather than a cause of the correct solution not being discovered. Tuning the parameters of the fitness function led to an improvement of the shape of the fitness landscape, as measured by the performance of the hand-crafted solutions tracing the expected line of evolution.
- Use of program templates failed to improve evolution, due to algorithmically close programs having greatly different fitnesses. Incremental freezing of programs did not work due to the code added outside the frozen material undoing its good effects.
- Alteration of the function and terminal sets led to the elimination of static solutions. In

the array problems system, minimisation of the function and terminal sets was critical for the evolution of correct answers; that this did not allow correct answers to evolve for the visibility space problem suggests that, while necessary, this is not sufficient. However, the use of fitness-dependent function and terminal sets was capable of evolving a program which acted as the first step on the road to a correct solution.

This system has failed to achieve more than evolving the first step along the road to solving the visibility space problem. What is it about this problem that has rendered it so intractable to solution by means of Genetic Programming?

One item which stands out is the complexity of the problem. Complex problems can only be solved by complex solutions (Ashby's Law of Requisite Variety[19]) and GP is best at coming up with solutions of moderate complexity, such as can be encoded by programs of the size seen in this chapter. As described in Section 4.6, the correct solution to the 2D visibility space problem, as originally designed, had not appeared too complex, and had thus been thought to be amenable to solution by typed GP. However, the simple solution masked a fairly complex subproblem, and in the redesign that this necessitated, the program encoding the correct solution became considerably more complex.

Another point that contributes towards the failure of the evolution is how genotype space maps poorly onto phenotype space.[151] The smallest of changes to a program's genotype—for example, '`<`' into '`>`'—can completely alter the program's phenotype, resulting in poor fitness. This has the knock-on effect of causing a program with good genetic material to be less likely to be selected for crossover, and thus liable to be lost from the population, because its good genes are not phenotypically evident. This was demonstrated by carrying out studies in which a correct solution was hidden within an intron by means of a construct such as `if < (0, pi) then [...]`. Because of its poor fitness the program was rarely selected for alteration; and when the correct solution was brought into expression, it was through the mechanism of mutating the '`<`' into a '`>`'.

Moreover, good solutions were not readily decomposed into small building blocks of good fitness; this therefore required evolution to be able to assemble large building blocks without

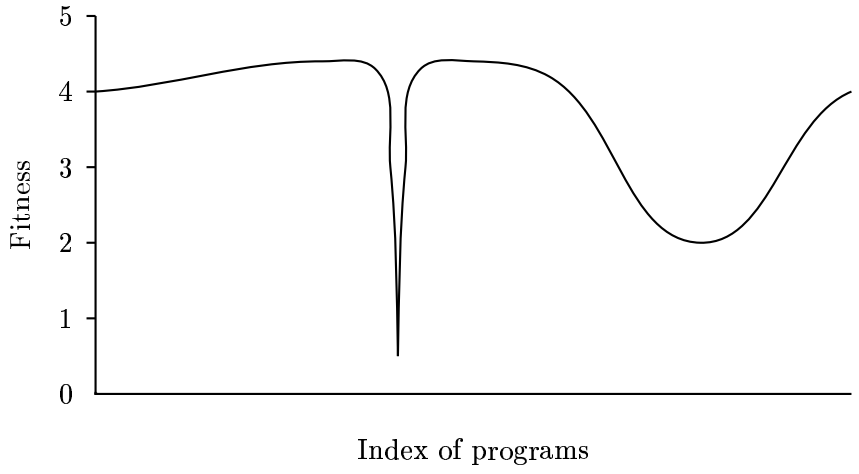


Figure 4.42:[113] A fitness landscape which GP is poorly equipped to explore.

any form of reward, before these blocks could be recombined into increasingly sophisticated solutions.

This suggests[98] that the synthesis approach to calculating visibility spaces is unsuitable for use in tackling this problem by GP. The No Free Lunch algorithm shows that every search algorithm is optimised to deal with certain kinds of problem,[139,160] and GP is no exception.[98] For example, on the fitness landscape shown in Figure 4.42, hill-climbing algorithms will outperform hill-descending ones, and random search will do better than either.[113]

However, it was felt that the main reason this system failed to solve its stated problem is because of the way in which extremely simple partial solutions, without any program elements useful for building good solutions, were able to achieve a fitness close to the good solutions; and the inability to recast the fitness function in such a way to smooth these local minima out of the fitness landscape.

If a solution to the visibility space problem is attainable by Genetic Programming, it is clear that this cannot be the case with a system such as the one described above.

Consequently some thought was put into how best it might be achieved, and the system was redesigned a second time.

Chapter 5

The Second Typed System

5.1 Introduction

Having shown the synthesis approach to be inappropriate for the evolution of programs to solve the visibility space problem, this chapter considers the use of the generate-and-test approach instead.

In this chapter, a new system is introduced which attempts to discover visibility spaces by growing contours of points, rather than by analysis of the data describing the model to be viewed.

Section 5.2 discusses the reasoning which led to this system, and Section 5.3 gives the specifications of this new system. Section 5.4 then describes the results of the experiments carried out with this system, and investigates to what extent the system was able to solve the problem with and without assistance in the form of seeded programs.

5.2 Rationale

5.2.1 Objectives of the New System

The previous chapter showed the synthesis approach to be poorly suited to solving the visibility space problem by means of Genetic Programming. In the light of the lessons learned from the previous system, any new one would have to prioritise the following:

- The fitness landscape would have to be smoother with respect to alterations in genotype.
- The system would have to be better at discouraging static solutions.
- Incremental improvements in programs would have to lead to incremental improvements in fitness at a finer level than in the previous system.

What is meant by this last point is that though successively better programs might be able to tackle successively more complex fitness cases, there was little concept of partial fitness *within* a fitness case in the previous system: If a program identified the wrong points from which to construct answer halfplanes; then if those points were spatially distant from the correct ones, it would score a poor fitness—but on the other hand, if they were spatially close to them, static solutions would be able to score well.

One possibility would be for the system to improve incrementally by means of *default hierarchies*,[84] in which programs are incrementally refined to fill in the deficiencies in their answers.

Another is for the genetic programs to be "Anytime algorithms",[151] in which the longer the program executed, the closer to the correct answer its output would be. Such a technique has already been used for planning problems in GP.[63] This would require a substantially different approach to the previous system, in which the output data could not be sampled during the running of the system, but only when it had ceased executing.

These points suggested switching from using a synthesis approach to solve the problem to using some form of generate-and-test approach. Various forms of how this might be done were then considered.

5.2.2 Approaches Considered

The initial idea was for programs to be evolved to manipulate a collection of points in two dimensions, such that when the program ceased executing they would delimit the problem feature's visibility space. Whether points fell in or out of the visibility space would be determined by ray-tracing. It would be desired to keep ray-tracing to a minimum, because it is computationally expensive, therefore some form of penalisation for excessive ray-tracing would be needed.

There are advantages to using a small number of points. It has been suggested[43, 58] that measuring the fitness of a 2D matrix or image by using only a small number of samples from it leads to better results than measuring the fitness of the whole thing. For example, in Daida's work on extracting curvilinear features from images,[43] the use of a subimage as data for Boolean classification produces inferior results to using single points. (The terminal set however includes the result of image processing operations carried out on 3×3 and 5×5 squares centred on the fitness case pixel.) The reason for this is because the use of subimages contains too much extraneous or redundant data.

Another possible reason is that the lower signal-to-noise ratio in a small sample allows the system to surmount small obstacles in the fitness landscape. In Reynolds' work on corridor-following behaviour,[129, 130] reactive controllers evolved in a noise-free environment were brittle—that is, overfitted to the environment they were trained in, and unable to perform in other environments from the same domain. When stochastic noise was added, this ceased to be a problem. An image analysis system has also been reported in which evolution suffered from programs becoming overfitted to the training data.[64]

The initial conception for the new system was to use a random distribution of points outside the model, which would be raytraced. Solutions would consist of a list of halfplanes, which would be evolved by a genetic algorithm. The halfplanes would be specified in the form (r, θ), as described in Section 3.6.2 and illustrated in Figure 3.10 on page 84*. Using this representation would ensure that small changes in genotype space (alterations to r and θ) would map onto small changes in phenotype space (the halfplanes). This would not necessarily have been the case were the halfplanes to have been specified instead using the parameters to the inequality $ax + by + c \geq 0$. A mutation operator could be adapted to perturb each of these numbers with a normal-shaped probability distribution; this would be facilitated by adapting Gray codes for representing the floating-point numbers. More radical changes to the floating-point values could also be effected using either a different mutation operator or crossover.

*The system described in that section also used a third descriptor, a flag indicating whether the origin fell in the positive or negative partition of the halfplane. This flag could be eliminated by removing the restriction that $r > 0$.

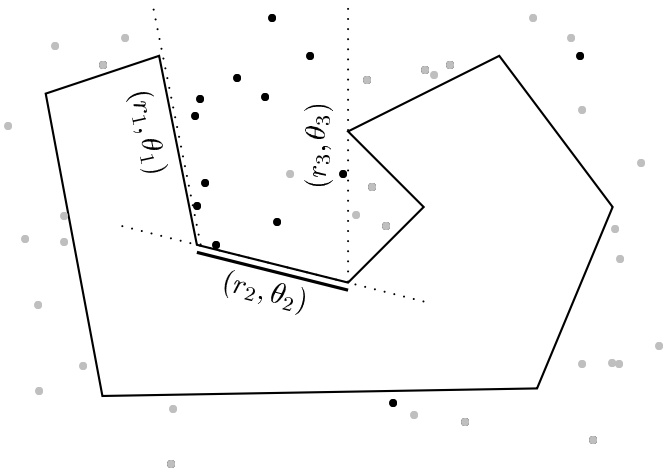


Figure 5.1: Example points distribution for the first fitness measure considered for the new system.

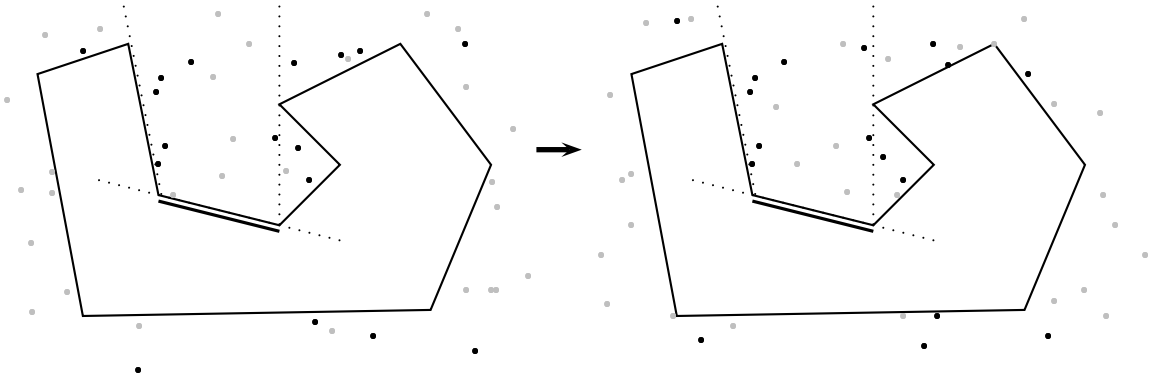


Figure 5.2: One iteration in execution in the second fitness measure considered for the new system.

Fitness would be measured by calculating which of the points comprising the fitness cases fell inside and outside the intersection of these halfplanes; the individual's fitness would consist of the number of misclassified points, either as an absolute number or as a proportion of the total number of points. Figure 5.1 shows an example genetic solution: black dots indicate points categorised as being within the visibility space, grey dots ones categorised as being outside it. Since there are three points misclassified (two black, one grey), the correct classification of the points being known *a priori*, this solution has a fitness of three.

An alternative system of representation was considered, in which the answer was delimited, rather than verified, by these points. In this system, program execution would alter the positions of the points until their locations matched their classification (see Figure 5.2). In this

case, raytracing would need to be done every time the points changed position. The fitness measurement in this system would consist of the number of point categorisation errors.

However, this arrangement would require a large amount of ray-tracing. How could the fitness be accurately estimated without having to ray-trace all the points? This could be done by only examining a subset of the points, but if the points are wholly independent then any subset will not reflect the entire point distribution.

The solution was to add connecting lines between the points. If points are maintained in an order, and evolution rewards programs that keep the points close together, then the closer in space two points adjacent to each other in this order are, the more likely the membership status of any examined point will be the same as that of its neighbours. (Note that such an evolutionary reward would have to be accompanied by a second one for keeping the points' distribution as large as possible, otherwise their coverage would shrink to a dot.)

This led to the idea of evolving a contour, rather than a random distribution of points: The convex hull of a random distribution automatically divides the points into those which define the boundary, and those which do not, those in the latter category serving no useful purpose. Fitness would then be measured, as in the previous GP system, by comparing areas.

Consequently, the aim of the new system was finalised as follows:

> To develop a GP system capable of evolving a program for discovering visibility spaces by manipulating a list of points defining that space. Fitness would be determined by comparing the delivered answer's area against the correct area.

A non-genetic system was implemented to carry this out, to determine that no unseen complications lurked within this seemingly simple problem (cf. Section 4.9).

5.3 System Specifications

Overview

The genetic virtual machine utilised the following types:

POINT or VECTOR (these being used interchangeably*)

REAL

BOOLEAN

INTEGER

GENERIC

INDIFFERENT

Models were represented by an ordered list of points, but were not directly accessible to the genetic programs. As before, the genetic program would be enclosed within a larger framework which called it once for every iteration around a loop; this is described in more detail below.

The genetic program manipulates a circular list of data of the form:

$$(P_i, \mathbf{v}_i)$$

where P_i specifies the coordinates of a point and $\mathbf{v}_i$ is a point or vector associated with it. In the handcrafted programs constructed for this system, these associated values are used as velocity vectors, indicating the magnitude and direction by which the genetic program displaces their associated point every time it is called for that point. Whilst it is anticipated that evolved programs should follow this behaviour, they are of course free to use these vectors in other ways.

The genetic program is called once for every point in this list in turn, until the termination criteria defined below are met. The work area for execution of genetic programs is an origin-centred square of size twice that of the maximum absolute value of any coordinate in the model.

*p_i will be used in the following pages to indicate a cell of this system's points memory being interpreted as a point, and $\mathbf{p}_i$ to denote the same cell interpreted as a vector.

Points memory	
0	P_i
1	$\mathbf{v}_i$
3	p_2
4	p_3
5	p_4
6	p_5

Reals memory	
0	*vector-threshold**
1	*distance-threshold**
3	r_2
4	r_3
5	r_4
6	r_5

* Read-only cell

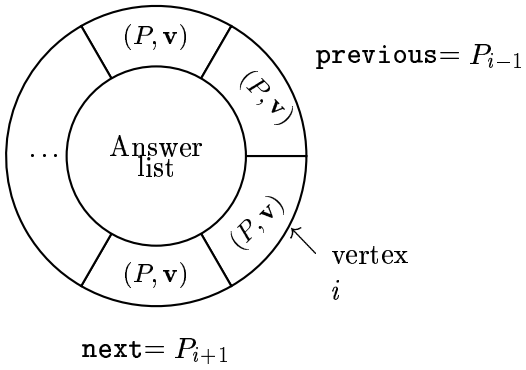


Figure 5.3: Architecture of the second typed system.

Architecture

Genetic programs consisted of a main program branch and two ADFs. Indexed memories of types POINT and REAL were provided, referred to below as $p_0 \ldots p_5$ and $r_0 \ldots r_5$. This is illustrated in Figure 5.3.

Program execution takes place in the following framework:

- Find start point:

  Execution of the genetic program does not take place until a start point within the visibility space has been found. This point is found by iterating around the sides of an origin-centred square, S, three-quarters of the size of the work area:

  let *interval* := 1/4

  foreach *direction d*

  for $i := 1$ **to** $\frac{1}{interval} - 1$

  let $p := \frac{i}{interval}$th the way along the dth side of S

  if *is-in-visibility-area* (p) **then break**

- Initialise answer list:

  Four points are added to the answer list, with location p and velocity vectors of magnitude one fifth of the length of the work area, facing in each of the compass directions.

- Initialise environment:

let $r_0 := \frac{\textit{work area}}{40}$

let $r_1 := 7r_0$

These two memory cells are set to thresholds used by the correct answer and available to all programs. They are read-only locations.

- Run genetic program:

while not *done* **and** *run-length* $\leq$ *maximum-run-length*

let *old-answer-list* := *answer-list*

foreach *vertex* P_i **in** *answer-list*

let p_0 **reference** P_i

let p_1 **reference** P_i's velocity vector $\mathbf{v}_i$

let `previous` := P_{i-1}

let `next` := P_{i+1}

run-genetic-program

if $\mathbf{v}_i = 0$ **for every** i **then let** *done* := *true*

if *answer-list* = *old-answer-list* **then let** *done* := *true*

As in the previous system, the main program iteration was part of the execution framework and not a function within the genetic programming language. However, there was no expectation now that the system might be able to evolve this loop functionality itself; moreover the loop did not iterate between specific points in a data structure, as in the previous system, but simply iterated continuously around the entire answer list. As a result, there was no need for the loop to be explicitly denoted in the genetic programs.

The run-length cap on iteration originally predicated on the number of points processed. Later, this criterion was changed to refer instead to gene evaluations, in order to penalise programs which spent a large amount of time doing nothing.

```
<genome> ::=
  main: <indifferent>
  adf0: <boolean>
  adf1: <vector>

<vector> ::= + (<vector>, <vector>)
           | - (<vector>, <vector>)
           | / (<vector>, <real>)
           | += (<integer>, <vector>)
           | /= (<integer>, <real>)
           | mem/p (<integer>)
           | set/p (<integer>, <vector>)
           | insert (<point>, <point>, <vector>)
           | rotate (<vector>, <integer>)
           | adf1 (<vector>)
           | next | prev | argvr | pNULL
           | <generic>

<real> ::= r/ (<real>, <real>)
         | mem/r (<integer>)
         | set/r (<integer>, <real>)
         | magnitude (<vector>)
         | 0.0 | 2.0 | 4.0 | 8.0
         | <generic>

<boolean> ::= "<" (<real>, <real>)
            | ">" (<real>, <real>)
            | adf0 (<integer>, <integer>)
            | checkpt (<integer>)
            | <generic>

<integer> ::= arg0i | arg1i | 0 | 1 | 2 | 3 | 4 | 5
            | <generic>

<generic> ::= "{" <indifferent>; <indifferent>; <indifferent>;
                  <indifferent>; <instantiation> "}"
           | if (<boolean>, <instantiation>, <instantiation>)

<indifferent> ::= <vector> | <real> | <boolean> | <integer>
<instantiation> ::= <vector> | <real> | <boolean> | <integer>
```

Table 5.1: Definition of the second typed system in Backus Naur Form.

Boolean and arithmetical operations:	
`+, - (VECTOR, VECTOR)`	→ VECTOR
`/ (VECTOR, REAL)`	→ VECTOR
`r/ (REAL, REAL)`	→ REAL
`+= (INTEGER, VECTOR)`	→ VECTOR
`/= (INTEGER, REAL)`	→ VECTOR
`<, > (REAL, REAL)`	→ BOOLEAN
Programming operations:	
`{...} (INDIFFERENT, INDIFFERENT, INDIFFERENT, INDIFFERENT, GENERIC)`	→ GENERIC
`if (BOOLEAN, GENERIC, GENERIC)`	→ GENERIC
`adf0 (INTEGER, INTEGER)`	→ BOOLEAN
`adf1 (VECTOR)`	→ VECTOR
Memory operations:	
`mem/r (INTEGER)`	→ REAL
`set/r (INTEGER, REAL)`	→ REAL
`mem/p (INTEGER)`	→ POINT
`set/p (INTEGER, POINT)`	→ POINT
`insert (VECTOR, VECTOR, VECTOR)`	→ VECTOR
`checkpt (INTEGER)`	→ BOOLEAN
`magnitude (VECTOR)`	→ REAL
`rotate (VECTOR, INTEGER)`	→ VECTOR
Parameters:	
`next`	→ POINT
`prev`	→ POINT
`arg0i, arg1i`	→ INTEGER
`argvr`	→ VECTOR
Constants:	
`0, 1, 2, 3, 4, 5`	→ INTEGER
`0.0, 2.0, 4.0, 8.0`	→ REAL
`pNULL`	→ POINT

Program Branch	Node																																							
	>	<	if	magnitude	mem/r	set/r	mem/r	set/r	prog5	/	-	+	/r	+=	/=	rotate	checkpoint	prev	next	0	1	2	3	4	5	0.0	2.0	4.0	8.0	pNULL	insert	adf0		adf1	arg0i	arg1i	argvr			
Main	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							
ADF 0	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓				
ADF 1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓				✓

Table 5.2: Definition of the second typed system's genetic machine.

Syntax and semantics

Table 5.1 gives a definition of the new system in Backus-Naur Form; the functions and terminals defining the system are listed according to functionality in Table 5.2.

This system does not possess closure of arithmetic in the same manner as the previous systems. Instead, any genetic program that suffers an arithmetic error, whether overflow (infinity) or division by zero, automatically aborts, and is assigned the same fitness as a program that does nothing.

A range check is provided for the POINT type, used by `set/v`, `+=`, `/=` and `insert`. In this, any point which has a coordinate with an absolute value in excess of 1000 is rounded down to 1000, at the intersection of that point's position vector with the origin-centred square of length 1000. This value was chosen as being large enough to preclude the possibility of points which rebounded from this boundary from having repercussions upon programs' fitness, whilst being small enough to make arithmetic overflow in the representation used (see Section 5.3.3) unlikely.

- Boolean and arithmetical operations:

  `+ (VECTOR, VECTOR)`

  `- (VECTOR, VECTOR)`

  `/ (VECTOR, REAL)`

  `r/ (REAL, REAL)`

  These perform arithmetic on points/vectors, integers and reals according to their arguments.

  `+= (INTEGER, VECTOR)`

  `/= (INTEGER, REAL)`

  These are shorthands for performing arithmetic on the indexed memories; that is, the subtree shown in Figure 5.4a is printed as in Figure 5.4b and is evaluated the same as that in Figure 5.4c.

  `< (REAL, REAL)`

  `> (REAL, REAL)`

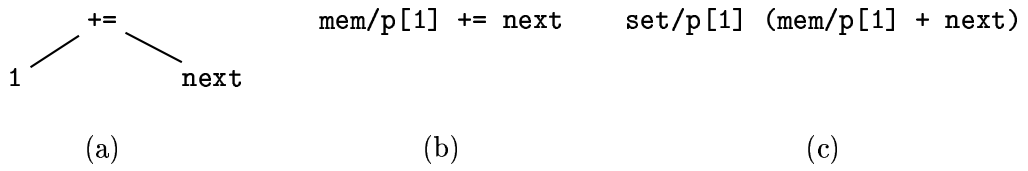


Figure 5.4: Illustration of the semantics of +=.

These perform comparison of real numbers, returning a boolean value.

- Programming operations:

  `{...} (INDIFFERENT, INDIFFERENT, INDIFFERENT, INDIFFERENT, GENERIC)`

  `if (BOOLEAN, GENERIC, GENERIC)`

  `{...}` evaluates its arguments in order, discarding the values of all bar the last, which is returned.

  `if` evaluates either its second argument or its third, depending on the Boolean value of the first argument.

- ADFs:

  `adf0 (INTEGER, INTEGER)`

  `arg0i`

  `arg1i`

  `adf1 (VECTOR)`

  `argvr`

  Two ADFs are provided; the former takes two integer arguments, the latter one point/vector argument. The arguments are evaluated a single time before the ADF is entered, and the results assigned to `arg0i` and `arg1i` for ADF 0, and `argvr` for ADF 1. Both ADFs return the result of executing their contents.

- Memory operations:

  `mem/r (INTEGER)`

  `set/r (INTEGER, REAL)`

  `mem/p (INTEGER)`

`set/p (INTEGER, POINT)`

`insert (VECTOR, VECTOR, VECTOR)`

`mem/r` returns the contents of the cell of the real number memory indexed by its argument; `mem/p` does likewise with the points memory. `set/r` and `set/p` are used for writing to these memories; their first arguments indicate the memory cell to write to, and their second the value to be written.

`insert` inserts a new vertex into the answer list, of which the coordinates are specified in its second argument and the velocity vector in its third. The first argument indicates where to insert the new vertex: If it is identical to the value of `next` (q.v.), the new vertex is inserted following the current one, otherwise it is inserted immediately before it.

- Geometric operations:

  `checkpt (INTEGER)`

  `magnitude (VECTOR)`

  `rotate (VECTOR, INTEGER)`

  `checkpt` returns *true* if the contents of the points cells specified by its argument is both outside the model's convex hull and inside the visibility area, as determined by ray-tracing to either end of the feature to be viewed; otherwise it returns *false*.

  `magnitude` returns the magnitude of the point/vector given in its argument.

  `rotate` returns its first argument rotated anticlockwise by the number of right angles given in its second.

- Parameters: `next`, `prev`

  `next` returns the coordinates of the vertex in the answer list following the current one, `prev` that of the vertex preceding the current one.

- Constants:

  `0, 1, 2, 3, 4, 5`

  `0.0, 2.0, 4.0, 8.0`

  `pNULL`

`0–5` evaluate to integers, `0.0–8.0` to real numbers, and `pNULL` evaluates to the null point $(0, 0)$.

Worked example

In the following, $p_0 \ldots p_5$ refer to the cells of the points memory, and $r_0 \ldots r_5$ to the cells of the real numbers memory. Some of these cells are assigned special values, and are here referred to by the following descriptive labels:

p_0	as P_i	: The answer vertex currently indexed by the main iteration.
p_1	as $\mathbf{v}_i$	: The velocity vector of P_i.
r_0	as *vector-threshold*	: One fortieth the length of the work area.
r_1	as *distance-threshold*	: 7× *vector-threshold*
`previous`	as P_{i-1}	: The vertex preceding P_i in the answer list.
`next`	as P_{i+1}	: The vertex following P_i in the answer list.

r_0 and r_1 are read-only locations.

A hand-crafted complete solution to the problem is given in Program 5.1. The operation of this program is as follows:

Lines 1–15 are the main conditional. If P_i's velocity vector $\mathbf{v}_i$ is positive, then lines 2–14 are executed, else the program returns, doing nothing (line 15).

If $\mathbf{v}_i$ is not less than the vector-threshold specified in r_0, then if $p_i + \mathbf{v}_i$ (lines 9–10) is inside the visibility area (line 11), P_i is overwritten with this new location (line 11). Otherwise P_i is left unaltered, and $\mathbf{v}_i$ is halved in preparation for when the program next operates on this vertex (line 12).

If $\mathbf{v}_i$ *is* less than r_0, then $\mathbf{v}_i$ is set to $(0, 0)$ (line 3). The program then calculates (calls to ADF 0) the vectors joining first P_{i-1} and P_i (lines 4–5), and then P_i and P_{i+1} (lines 6–7). If the magnitude of this vector, i.e. the distance joining the two points, is greater than the threshold specified in r_1, then a new vertex is created halfway between the two extant points (A2–A3), and the vector, rotated 270° to face outwards (line A4), is recycled for use as its velocity vector.

The rationale behind this is that the distance between points is likely to be proportional to the distance from the edge of the visibility space, once they start approaching it: the more

```
main:
 1. if mem/v[1].magnitude > 0.0 then
 2.   if mem/v[1].magnitude < mem/r[0] then {
 3.     mem/v[1] = pNULL;
 4.     mem/v[5] = previous;
 5.     if adf0 (0,5) then adf1 (previous) else pNULL;
 6.     mem/v[5] = next;
 7.     if adf0(5,0) then adf1 (next) else pNULL;
 8.   } else {
 9.     mem/v[2] = mem/v[0];
10.     mem/v[2] += mem/v[1];
11.     if checkpt(2) then mem/v[0] = mem/v[2]
12.     else mem/v[1] /= 2.0;
13.     0.0; pNULL;
14.   }
15. else pNULL

adf0:
    (mem/v[2] = (mem/v[arg0i] - mem/v[arg1i])).magnitude > mem/r[1]

adf1:
A1. {
A2.   mem/v[3] = (mem/v[0] + mem/v[5]);
A3.   mem/v[3] /= 2.0;
A4.   mem/v[4] = mem/v[2].rotate(3);
A5.   if mem/r[2] = (mem/v[4].magnitude r/ mem/r[0]) > 8.0 then
A6.     mem/v[4] /= (mem/r[2] r/ 4.0)
A7.   else pNULL;
A8.   insert (argvr, mem/v[3], mem/v[4]);
A9. }
```

Program 5.1: Hand-crafted solution for the second typed system.

points that have encountered the edge of the visibility space, the more new points will have been created between them, and the shorter the inter-point distance will be.

If the velocity vector calculated for the new vertex, $\mathbf{p_4}$, has a magnitude in excess of $8r_0$ (line A5), the above generalisation does not apply, so the velocity vector is scaled down to $4r_0\hat{\mathbf{p}}_\mathbf{4}$, i.e. $4r_0$ times the unit vector in the direction of $\mathbf{p_4}$ (line A6).

Finally, the new vertex is added to the model contour (line A8).

A step-by-step dissection of this in pseudocode follows.

1.	**if** $\|\mathbf{v}_i\| > 0.0$ **then**	
2.	**if** $\|\mathbf{v}_i\| <$ *vector-threshold* **then**	
2.	sequence:	
3.	$\|\mathbf{v}_i\| \leftarrow (0,0)$	$\Longrightarrow (0,0)$
4.	$p_5 \leftarrow P_{i-1}$	$\Longrightarrow P_{i-1}$
5.	**if** adf0 (0,5) returns *true*	
5.	**then** adf1 (P_{i-1}) **else** $(0,0)$	**if** $\Longrightarrow$ ADF return value or $(0,0)$
6.	$p_5 \leftarrow P_{i+1}$	$\Longrightarrow P_{i-1}$
7.	**if** adf0 (5,0) returns *true*	
7.	**then** adf1 (P_{i+1}) **else** $(0,0)$	**if** $\Longrightarrow$ ADF return value or $(0,0)$
8.		sequence $\Longrightarrow$ line 7 return value
8.	**else**	
8.	sequence:	
9.	$p_2 \leftarrow P_i$	$\Longrightarrow P_i$
10.	$p_2 \leftarrow p_2 + \mathbf{v}_i$	$\Longrightarrow p_2$
11.	**if** check-point (p_2) **then** $P_i \leftarrow p_2$	$\Longrightarrow p_2$
12.	**else** $\mathbf{v}_i \leftarrow \mathbf{v}_i \div 2.0$	$\Longrightarrow \mathbf{v}_i$
12.		**if** $\Longrightarrow p_2$ or $\mathbf{v}_i$.
13.	0.0 (space-filler)	$\Longrightarrow 0.0$
13.	$(0,0)$ (POINT-type space-filler)	$\Longrightarrow (0,0)$
14.		sequence $\Longrightarrow (0,0)$

14. **if** $\Longrightarrow$ line 8 return value or $(0,0)$

15. **else** (0,0) $\Longrightarrow (0,0)$

15. **if** $\Longrightarrow$ line 14 return value or $(0,0)$

ADF 0:

magnitude (

$p_2 \leftarrow p_{arg0} - p_{arg1}$ $\Longrightarrow p_{arg0} - p_{arg1}$

) $\Longrightarrow \|p_2\|$

$>$ *distance-threshold* $\Longrightarrow \|p_2\| >$ *distance-threshold*

ADF 1:

A1. sequence:

A2. $p_3 \leftarrow P_i + p_5$ $\Longrightarrow p_3$

A3. $p_3 \leftarrow p_3 \div 2.0$ $\Longrightarrow p_3$

A4. $p_4 \leftarrow p_2$ rotated three right-angles $\Longrightarrow p_4$

A5. **if**

A5. $r_2 \leftarrow \|p_4\| \div$ *vector-threshold* $\Longrightarrow r_2$

A5. > 8.0 **if** condition $\Longrightarrow r_2 > 8.0$

A6. **then** $p_4 \leftarrow p_4 \div \frac{r_2}{4.0}$ $\Longrightarrow p_4$

A7. **else** $(0,0)$ **if** $\Longrightarrow p_4$ or $(0,0)$

A8. $\Longrightarrow p_3$

A8. sequence $\Longrightarrow p_3$

ADF $\Longrightarrow p_3$

5.3.1 Fitness Function

Initially, the same fitness measure was used as in the first typed system, to wit comparison of the area delivered by the genetic program with the correct answer area, in the form:

$$100\left(1 - \frac{F \cap G}{F \cup G}\right)$$

where $F \cap G$ denotes the intersection of the correct answer area and that delivered by the genetic program, and $F \cup G$ denotes the union of these two areas.

The area produced by the genetic program was taken as the convex hull of the answer list of points (see further below). After seeing the results of running the system with such a fitness function, it was subsequently changed to:

$$\frac{(F - F \cap G) + k(G - F \cap G)}{F \cap G}$$

i.e. the difference between the area missing from the answer and a constant multiple of the superfluous area in the genetic program's answer, all expressed normalised by the intersection area.

The reason for this change was because of the regions of non-overlap with the correct answer that were given by partially fit evolved programs: It is better that a program which delivers a suboptimal answer fails to include part of the correct visibility area, than that it includes incorrect space in its answer. Therefore the fitness function was reworked to penalise inclusion of incorrect space to a greater degree than non-inclusion of correct space. The bias—the value of k—used was 2.

In both fitness measures, the raw fitness was then modified to penalise for excessive run length:

$$\mathit{fitness} = f(F, G)\left(3 + \frac{\mathit{genes\ evaluated}}{3000}\right)$$

where $f(F, G)$ denotes whichever of the two fitness measures given above. The inclusion of the 3 in the multiplicand is to prevent programs obtaining near-perfect fitness by aborting after a single gene execution. A factor was also included to penalise for excessive program length:

if $\mathit{length} > 150$ **then let** $\mathit{fitness} := \mathit{fitness} \times \frac{\mathit{length}}{150}$

Since both these factors are multiplicative, such penalisations can be counterbalanced by improvements in fitness, thus resulting in the selection pressure against long programs only being manifest when their length did not lead to improved fitness.

The fitness cases, shown in Figure 5.5, were the same as for the first typed system. A system of dynamic fitness cases was used (see Section 4.7.5), such that each generation would be tested

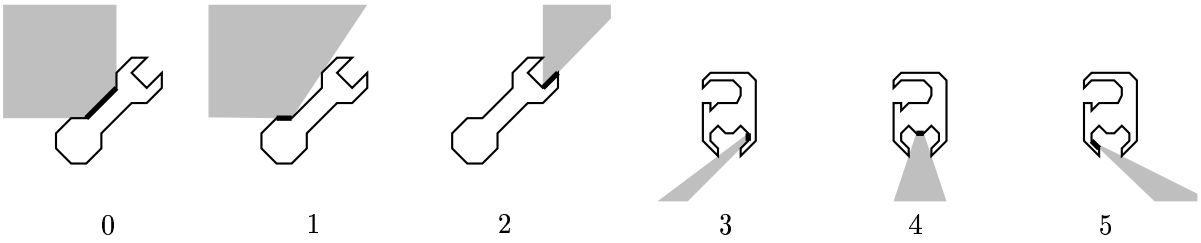


Figure 5.5: Fitness cases for the second typed system.

on one of the fitness cases, proceeding through them in a cyclical order. Thus fitness cases 0–2 test programs' ability to cope with wide visibility areas, and fitness cases 3–5 their ability to cope with narrow ones.

5.3.2 Seeded Programs

Hand-crafted programs for assessing the shape of fitness landscape were devised by incrementally removing lines from the perfect solution. The intention of this was to balance the objectives of maintaining an evolutionarily feasible pathway of increasing algorithmic complexity, and ensuring that this complexity correlated with declining fitness. Should the system again have difficulty evolving a complete solution *ex nihilo*, the hand-crafted programs could be used, as in the previous system, for seeding the population to kick-start evolution.

Table 5.3 gives a brief algorithmic description of each of the seeds that were devised. Table 5.4 shows their fitnesses for each of the fitness cases, and Figure 5.7 shows the results of executing them. Figure 5.6 summarises the format of these diagrams: the dark line shows the answer contour, and the grey line, often occluded by it, its convex hull. The light grey area denotes the correct solution, and the dark grey area the intersection of this with the genetic program's answer.

Success in the objective given above was limited. As in the previous systems, an unavoidable feature of the interaction of the representation used for programs, and the means of measuring fitness, was that it was possible to achieve results of moderate goodness with very simple programs. Fine-tuning of the fitness measure was carried out in order to minimise this; this was one of the reasons why the fitness measure was altered (see Section 5.3.1). However, as with

Seed	Description
1	Complete solution (see p. 186)
1a	**if** P_i is in visibility area **then** P_i $+:=$ $\mathbf{v}_i$ **else** P_i $+:=$ $rotate(\mathbf{v}_i, 180°)$ **insert** $\frac{P_i+P_{i-1}}{2}$, **with** $\mathbf{v} = \frac{\mathbf{v}_i+rotate(\mathbf{v}_i,270°)}{2}$ **before** P_i $\mathbf{v}_i := (0,0)$

2	**if** $\|\mathbf{v}_i\| >$ *vector-threshold* **then**
	P_i $+:=$ $\mathbf{v}_i$
	if P_i is not in visibility area **then**
	$\mathbf{p}_4 := \frac{\mathbf{v}_i+rotate(\mathbf{v}_i,270°)}{2}$
	if $\|\mathbf{p}_4\| >$ *vector-threshold* **then**
	insert $\frac{P_i+P_{i-1}}{2}$, **with** $\mathbf{v} = \mathbf{p}_4$, **before** P_i
	$\mathbf{v}_i := (0,0)$

3a	**if** $\|\mathbf{v}_i\| > 0$ **then**
	if P_i is in visibility area **then** P_i $+:=$ $\mathbf{v}_i$
	else
	P_i $-:=$ $\mathbf{v}_i \div 2$
	insert $\frac{P_i+P_{i+1}}{2}$, **with** $\mathbf{v} = \mathbf{v}_i$, **after** P_i
	$\mathbf{v}_i := (0,0)$

3	**if** $\|\mathbf{v}_i\| > 0$ **then**
	P_i $+:=$ $\mathbf{v}_i$
	if P_i is not in visibility area **then**
	insert $\frac{P_i+P_{i+1}}{2}$, **with** $\mathbf{v} = \mathbf{v}_i$, **after** P_i
	$\mathbf{v}_i := (0,0)$

4	**if** $\|\mathbf{v}_i\| > 0$ **then**
	if P_i is in visibility area **then** P_i $+:=$ $\mathbf{v}_i$
	else
	insert $\frac{P_i+P_{i+1}}{2}$, **with** $\mathbf{v} = \mathbf{v}_i$, **after** P_i
	$\mathbf{v}_i := (0,0)$

5	**if** $\|\mathbf{v}_i\| > 0$ **then**
	if P_i is in visibility area **then** P_i $+:=$ $\mathbf{v}_i$
	else
	insert P_{i+1}, **with** $\mathbf{v} = \mathbf{v}_i$, **after** P_i
	$\mathbf{v}_i := (0,0)$

6	**if** P_i is in visibility area **then** P_i $+:=$ $\mathbf{v}_i$
7	Does nothing
8	P_i $+:=$ $\mathbf{v}_i$

Table 5.3: Description of the seeds for the second typed system.

Seed	Fitness case						Average
	0	1	3	4	5	8	
1	0.6	0.6	1.2	14.2	2.1	1.9	3.454
1a	1.1	0.8	1.5	4.5	2.3	1.6	1.948
2	1.1	1.4	2.3	17.0	3.6	9.0	5.743
3a	1.4	1.3	2.9	16.0	6.1	7.5	5.853
3	2.7	2.8	5.1	32.8	6.3	13.1	10.477
4	1.8	2.4	2.5	22.4	4.1	14.7	7.983
5	1.7	1.9	1.8	9.8	2.8	6.2	4.016
6	4.3	5.1	1.8	9.6	2.1	6.4	4.880
7	3003.3	3003.3	3006.7	3005.0	3005.3	3005.7	3004.9
8	6334.0	6334.0	6333.3	6334.3	6333.3	6333.7	6333.8

Table 5.4: Fitnesses of the hand-crafted programs for the second typed system.

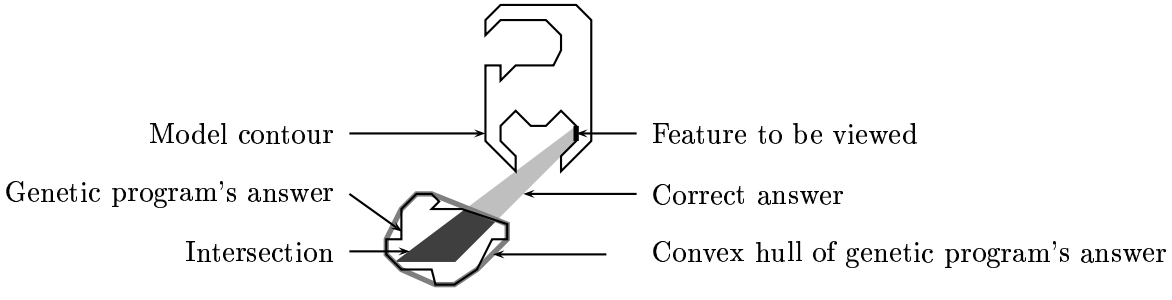
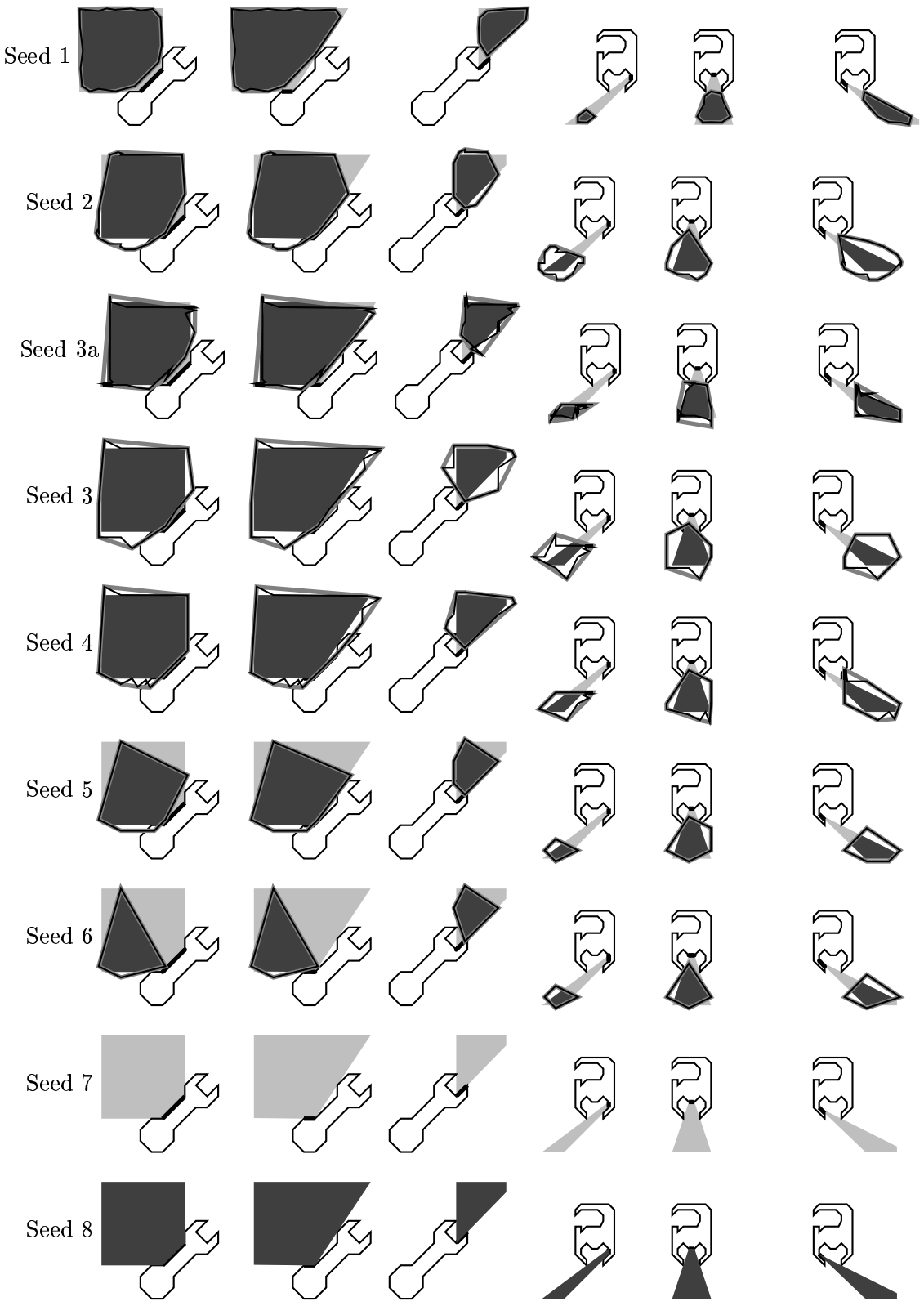


Figure 5.6: Format of diagrams used in this chapter.



(Genetic program's area for seed 8 on this scale is 9m wide.)

Figure 5.7: Execution of seeds.

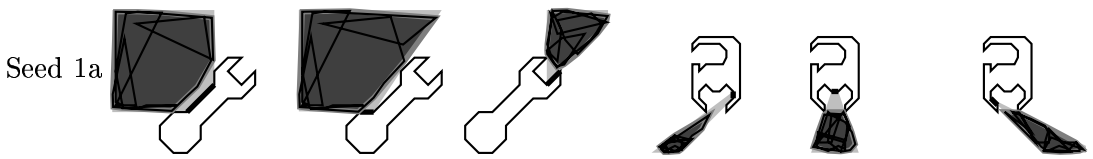


Figure 5.7: (continued) Execution of seed 1a

the previous systems, ensuring that all poor programs scored badly would as a consequence necessarily entail extreme difficulty in the evolution of good programs.

As a result, out of the eight seeded programs created, fitness declined only as far as seed 3; seeds 4, 6 and 5 were, in that order, incrementally more fit again. (Seed 8, which grew the answer area without limit, and seed 7, which did nothing, were both very unfit.)

To verify that a potential evolutionary path with continuously improving fitness (i.e. without local optima, which would lead to static solutions) did exist, two new seeds were devised. These new seeds did not lie on the evolution path staked out by the other seeds; for these seeds more emphasis was placed on good fitness than algorithmic compatibility. The first of these, seed 3a, is of similar complexity to seed 3, but its fitness is almost twice as good. The second new seed was also of similar complexity to seed 3, but as its fitnesses lay in between those of seed 1 and seed 2, it was dubbed seed 1a. Due to performing well on one fitness case on which almost all the other seeds performed poorly, this seed's average fitness was better than that of seed 1.

The complete code of the seeded programs is given in the Appendix.

5.3.3 Implementational Issues

Geometrical algorithms

As stated above, the expected result of the genetic programs was a contour delimiting the answer. However, because many evolved programs gave highly complex self-intersecting contours, it was determined to interpret the contour's convex hull as the genetic program's answer. This would allow the evolving programs the choice of taking advantage of the contour's ordering, or simply treating it as an unordered set of points for creating the convex hull.

A problem which became apparent once the system had been put in use was that, when the ratio of the distances of two points in the answer as measured from a third became particularly large, rounding errors led to the failure of the routines for constructing and intersecting convex hulls. In order to solve this, a protected real-number representation was devised, in which real numbers were represented by one hundred times their value, expressed as an integer, such that 1.23 was represented as 123. This put a cap on the resolution of the representation, but this was felt to be more than adequate for this system, since GP does not as a rule produce exact answers anyway—which was substantiated by the results obtained with this system (see figures throughout this chapter). Additional checking was then required for arithmetic operations, since the integer representation in C++ does not check for overflows and, unlike the floating-point representation, throws errors in the case of division by zero. The extra execution time this led to was partly offset by the faster speed of integer arithmetic; and the use of discrete arithmetic should have helped to make the system more portable between different systems, in terms of reproducibility of results.[20]

Unfortunately, though this change managed to eliminate most errors in the two convex hull routines, it failed to eliminate them all.

Genealogical audit trail

Code was implemented to maintain a genealogical audit trail for the population, so that the evolution of the best-of-population individual may be traced—which individuals it was descended from, and which genetic operators were used to produce it.

Since it cannot of course be known in advance which individual from any generation will contribute code to the next generation's best-of-population, audit trails thus had to be maintained for every individual in the population.

This is extremely expensive in terms of memory resources, involving the storage of up to $2^G M$ individuals, where G is the number of generations and M the size of the population. To reduce such exponential growth in storage requirements, an index of the population was maintained, so that individuals without descendants in the latest population could be pruned from this storage. Even so, maintaining audit trails consumed large amounts of storage and run-length time, and

Parameter	Setting
Population size	5000
Number of generations	300
Selection type	Tournament, size 10
Demes	10
Creation Type	Ramped half and half
Crossover Probability	75%
Creation Probability	10%
Maximum Depth For Creation	6
Maximum Depth For Crossover	17
Demetic Migration Probability	100%
Swap Mutation Probability	15%
Shrink Mutation Probability	0%
Population replacement	Generational / Steady State

Table 5.5: Tableau for the second typed system

was not practically viable for runs of the lengths described in this chapter.

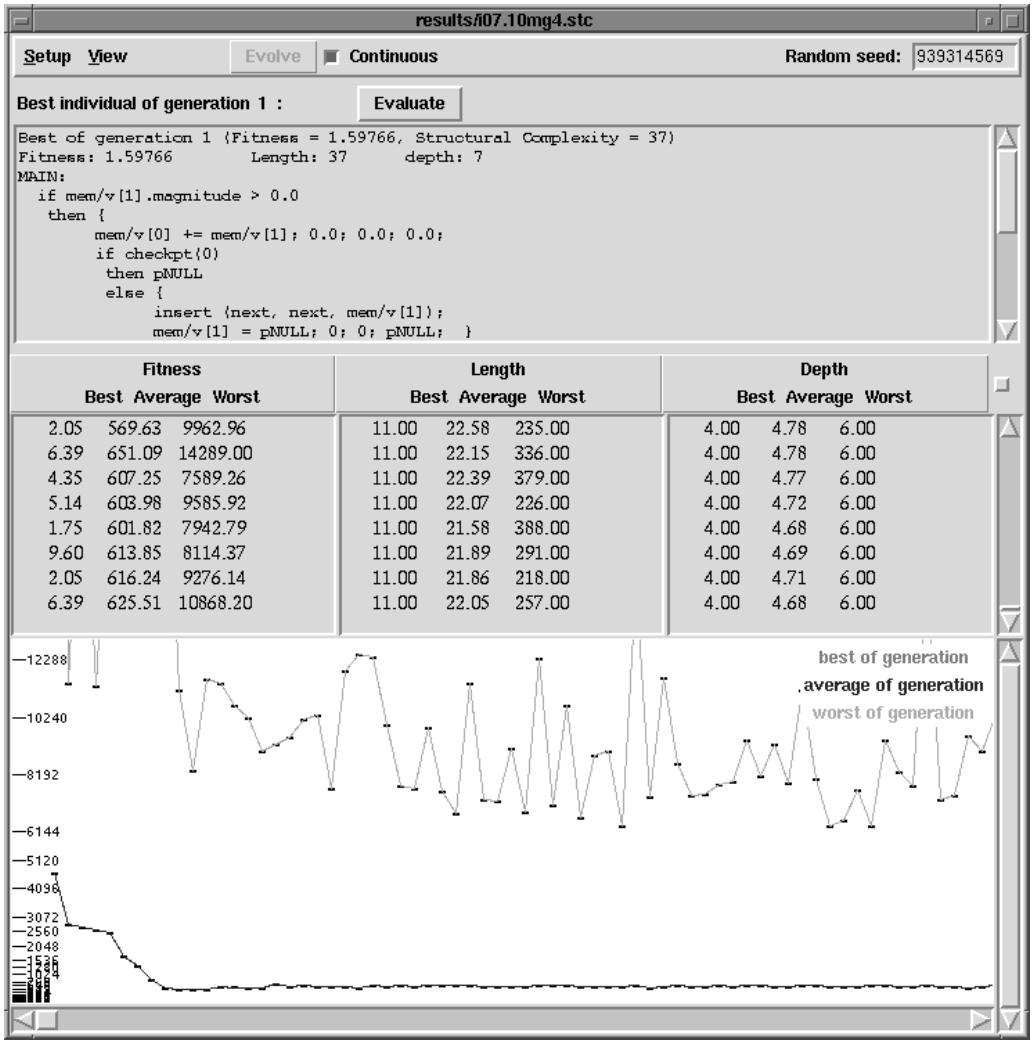
5.3.4 Graphical User Interface

A graphical user interface was developed for use with the second typed system, and is illustrated in Figure 5.8. This interface allowed the examination of run results, including the performance of programs on all fitness cases, and the performance of programs for model features which were not included in the fitness cases. It also allowed analysis of programs by graphical depiction of their step-by-step execution. This section provides a brief description of the parts of this graphical user interface.

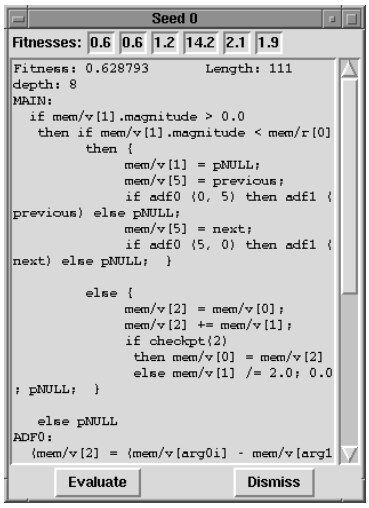
Figure 5.8a shows the main window, which is divided into three frames plus a menubar at the top. Each of the buttons and other manipulable objects has a balloon help*, which will pop up a second after the mouse pointer is brought to rest over the object.

The Setup menu allows for configuration of the genetic parameters (given in Table 5.5), loading of seeded programs, and starting of a new run. The View menu allows the user to

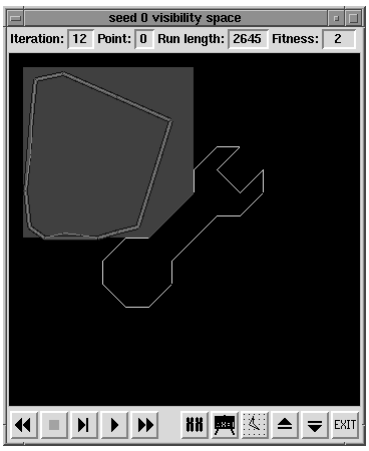
*Derived from code given in *Effective Tcl/Tk Programming*.[65]



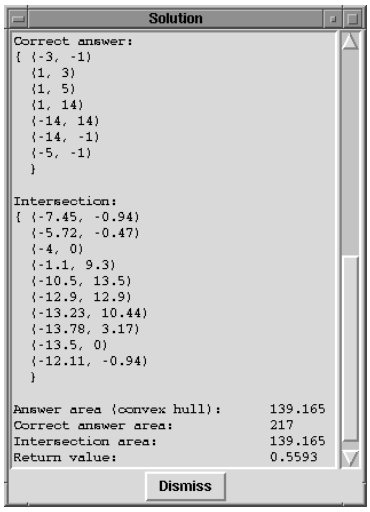
(a)



(b)

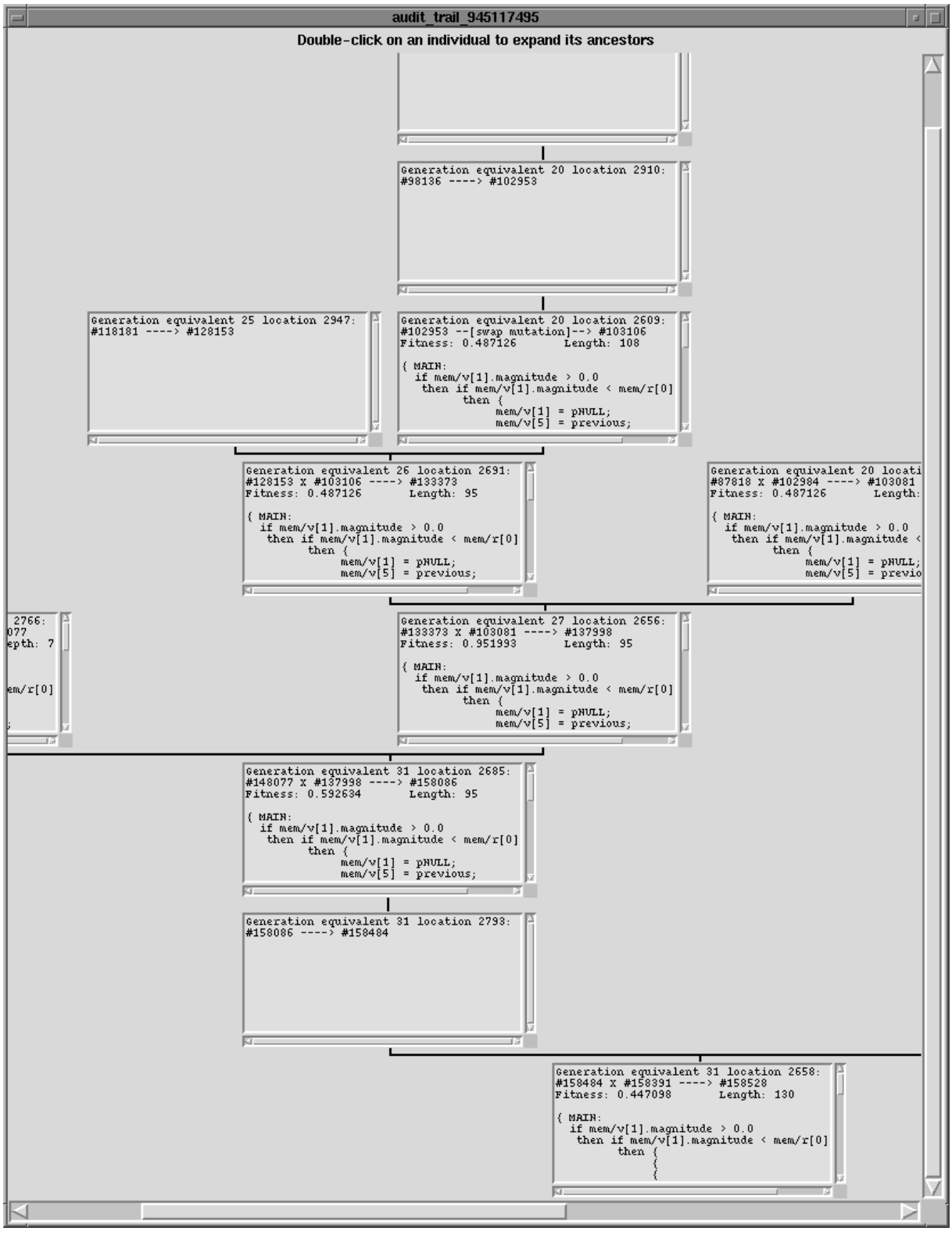


(c)



(d)

Figure 5.8: Graphical interface developed for the second typed system.



(e)

Figure 5.8: (continued)

view graphs of the seeded programs' fitness for each fitness case, to browse the code of all the individuals in the population, to view the current run's audit trail, and to load in the results of an old run for analysis. The menubar also includes the main "Evolve" button, a toggle for running the system continuously or one generation at a time, a timer indicating progress through the evolution of the current generation, and an indicator of the current run's random seed, which may be set by the user. Keyboard shortcuts are provided for all operations.

The top frame shows the code of the current best-of-generation individual. As a run proceeds, this is dynamically updated to show the best individual of the most recently completed generation. An "Evaluate" button allows evaluation of the currently selected individual, as discussed below.

The middle frame contains a table showing the fitness, length and depth of the best-of-generation individual, the worst-of-generation individual, and the average of these values across the population. At the right of the middle frame is a check button; toggling this switches the frame to display the fitnesses of the best-of-generation individual for all the fitness cases, and the average of these fitnesses.

The bottom frame depicts the values given in the middle frame graphically. The scrollbar allows scaling of the graph; the vertical scale is therefore logarithmic so as to be meaningful at all scales.

Clicking on a generation in the tables or graph switches the top window to contain that generation's best individual.

Clicking "Evaluate" for a best-of-generation individual or seeded program which has been manually loaded (see Figure 5.8b) brings up an evaluation window, Figure 5.8c. This shows a graphical representation of the execution of the program. The model, feature to be viewed and correct visibility area are indicated; and when the program is running, its answer, the convex hull thereof and the intersection with the correct solution are also indicated. A bar along the top indicates how many iterations the program has run through, how many points into the current iteration execution has reached, and the total number of genes evaluated in the current execution, along with the current fitness of the program. Buttons along the bottom control the execution of the program and selection of fitness cases. They also allow viewing of the program's

code, the model specification, and the working out of the current fitness (see Figure 5.8d).

The region viewed in the graph may be dragged around with the mouse, to see areas that are currently off-screen. Features for calculate the visibility space can be selected with the middle mouse button; these are not limited to those specified in the fitness cases.

Pressing the "Print Screen" key in both the evaluation window and the fitnesses graph will pop up a window of the form shown in Figure 5.8d, containing LaTeX code. This can be put into the clipboard with the mouse, and pasted into a LaTeX document.

Figure 5.8e shows an example audit trail. The best-of-run individual is shown at the bottom of the screen, with seven generations of antecedents displayed in the format of a family tree. Each program's window shows how it is derived from its parents. The tree may be extended by double-clicking on any individual.

GPC++ expects to load programs in a low-level, numeric representation. My problem-specific code implements a higher-level loader, triggered by the enclosing of the saved program within braces, which allows specification of seeded programs in the same form as that used for program output. Any error in program entry is caught by the graphical user interface, which throws up a window for editing the program and saving it back to disk, with the cursor positioned at the likely point of error.

For runs liable to be longer than the duration of the user's login, it is more convenient to run the system without the graphical user interface. This may be done by using the command-line option `-n`. The command-line option `-h` delivers a summary of the available command-line options.

The code for this system falls into discrete layers—the public-domain kernel, the library for typed GP developed for this system, and the problem-specific code. Due to the interdependence of the problem domain and its output it was not wholly possibly to separate the code for the graphical user interface from that of the previous three—the evaluation window (Figure 5.8c), for example, is highly problem-dependent. Nevertheless, much of the code for this GUI ought to be reusable for other problems. The GUI is implemented in Tcl/Tk.

5.4 Results

Experiments were carried out running the system with and without seeded programs, under various conditions. Run length varied with the average complexity of the programs; for the unseeded runs it was as little as three and a half hours. For the seeded runs, run length frequently exceeded the fifteen hour maximum that could be managed overnight, and would have to be halted so as not to monopolise machines during the day. This led to long runs taking from one to three and a half days to run to completion, in stages. Due to this extremely long time, only a few runs were carried out for each experiment. Except where indicated, each experiment had a minimum of two runs performed.

As before, the largest population resources permitted, 5000 individuals, was used for these experiments. Experiments ran for 300 generations. There was suggestive evidence that no evolutionary breakthroughs would be achieved after this time:

Over the course of generations, programs would become whittled down to their bare minima. Hence in the early part of a run, a good program might be produced by crossing two programs with large amounts of redundant or intronic code. Though the resulting program might suffer from fitness penalisation on account of its length, it would retain a large amount of redundant code, which could subsequently be used to evolve a better program. During the course of the run, programs which retained the functionality of this program but had lost parts of the redundant code would not suffer this penalisation, and would preferentially be selected for retention into subsequent generations. Thus by the end of the run, the descendants of the original good program would have little redundant code, and would thus be unable to serve as a repository of genetic material for recombination into better programs.

A second piece of evidence, that may be related to the first, is that over the course of a run the average fitness of the population, as measured across a full generational cycle of fitness cases,* often decreased for a period, then increased again as the amplitude of the fitness oscillations across fitness cases increased. This is illustrated in Figure 5.9. (Not all runs produced oscillations

*Though the same would probably hold true if the average fitness of the population across all fitness cases were to be measured for a single generation.

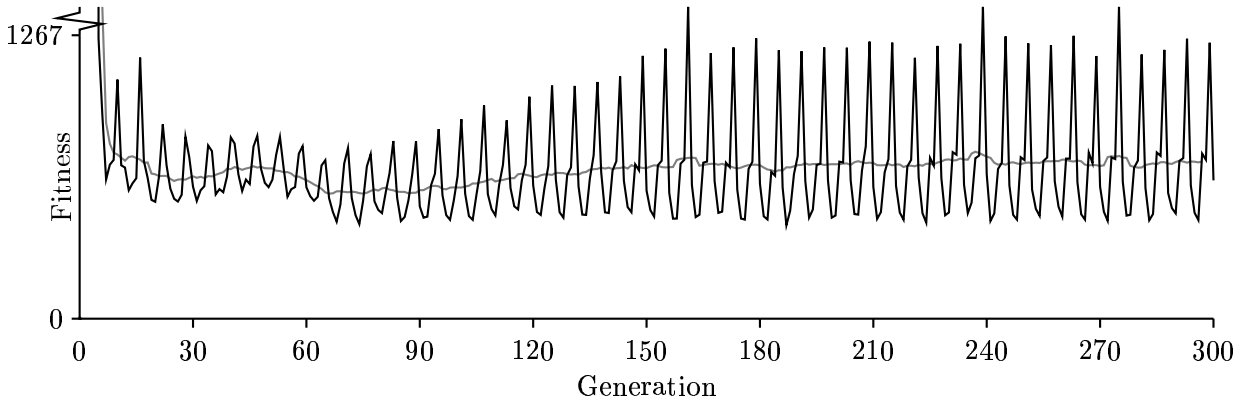


Figure 5.9: Increase in amplitude of oscillations of the average fitness of the population, with average across the cycle of fitness cases indicated. (Data taken from the run shown in Figure 5.15.)

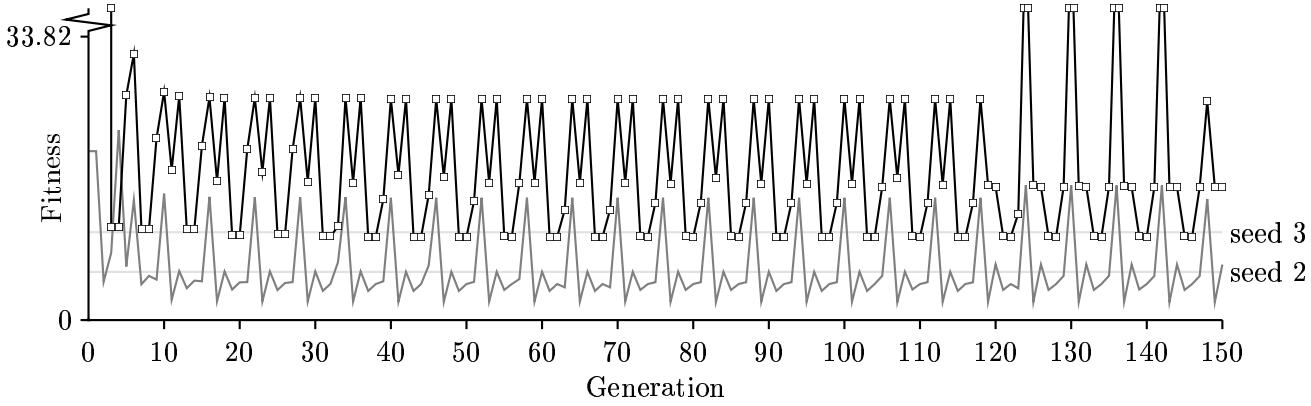
this marked.)

It should be noted that most of the results below are presented in the form of graphs of fitness averaged across all the fitness cases. Whilst this allows a presentation of data in which its prime characteristics can be easily identified, it also has a drawback: In making up an average fitness, a single bad result can make the fitness appear worse than it generally is, and an exceptionally good result will not be apparent when averaged with fitness cases of more moderate performance. The complete hand-crafted solution, for example, has five fitnesses in the range 0.6 to 2.1; on the one remaining fitness case it only scores 14.2, which takes its average fitness up to 3.5. This is why seed 1a scores a better average fitness than seed 1.

5.4.1 Unseeded experiments

Figure 5.10 shows the results of a run of the system without any seeds being used. In this and the following graphs, the main graph shows the average fitness of the best-of-generation program, whilst a grey trace in the background shows the best-of-generation's fitness according to the fitness measure used that generation.

It can be seen that the run quickly settled down to an oscillation between programs in successive generations; the identities of these programs depended on which fitness measure was currently being used.



Evolution style: generational; seeded programs: none; random number seed: 942859496

Figure 5.10: Unseeded evolution (only first 150 generations shown). Best-of-run found on generation 49.

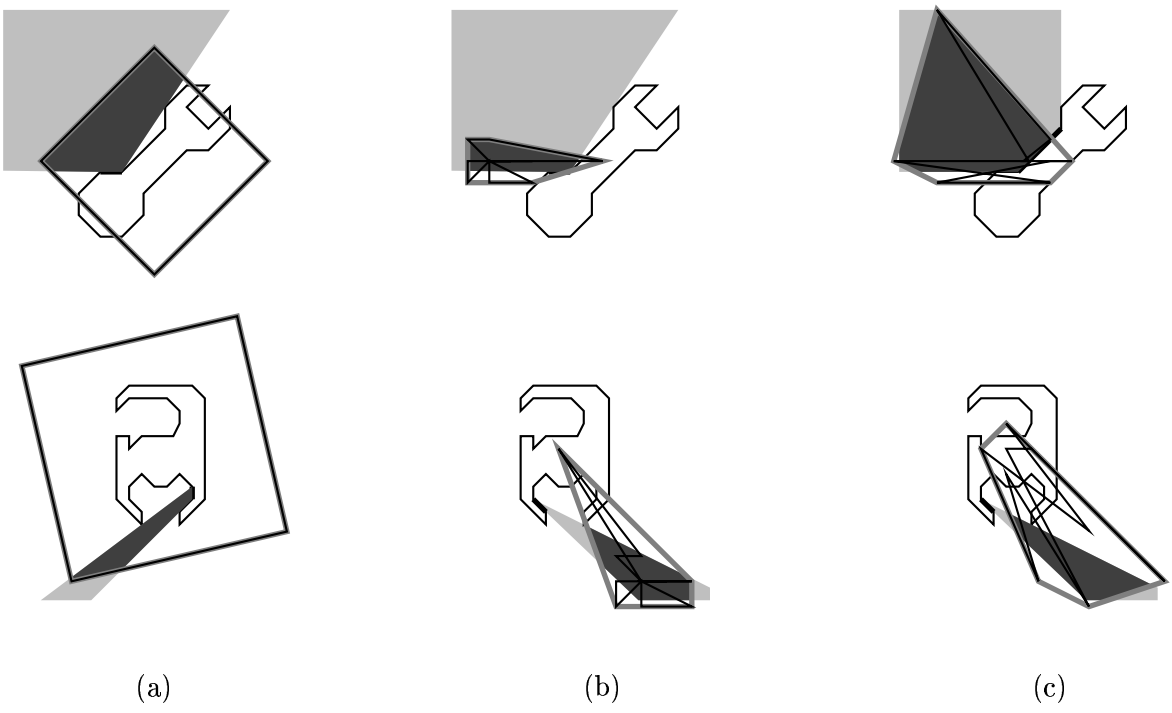
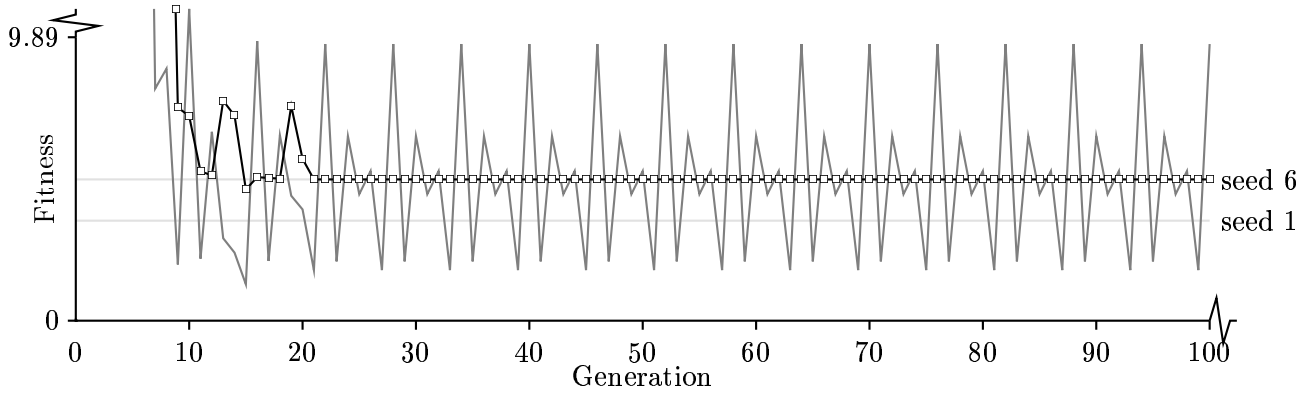


Figure 5.11: Execution of evolved programs from Figure 5.10. (a) A poor static solution. (b) A slightly better static solution. (c) A simple non-static solution.



Evolution style: steady state; seeded programs: none; random number seed: 942928988

Figure 5.12: Unseeded evolution (only first 100 generations shown).

```
MAIN: adf1 (mem/v[1])
ADF0: checkpt(0)
ADF1: mem/v[if checkpt(0) then 0 else 4] += mem/v[1]
```

Program 5.2: Best of generation 21 individual from Figure 5.12.

As in the previous system, most of the programs discovered by unseeded runs merely gave rise to static solutions. Figure 5.11 shows the results of running some of these programs.

Figure 5.11a shows a poor static solution evolved at the beginning of the run. As can be seen, it creates an area with the same geometry regardless of the fitness case used; in the upper figure, where the correct solution has a large area, it has a mediocre performance, but in the lower figure, where the correct solution has a much smaller area, its performance is much worse.

Figure 5.11b shows a static solution evolved later on in the run. This one creates a more complex area not centred on the origin; this area performs well on the fitness cases with narrow correct solutions, at the expense of those with broad ones. Since this approach involved missing out parts of the correct area, rather than wrongly including false visibility space, this program has a better average fitness, as explained in Section 5.3.1.

Unusually, this run managed to come up with a solution that was non-static. Stripped of its introns, the essence of this program is:

```
MAIN: adf1 (adf1 (adf1 (insert (pNULL, mem/v[1], mem/v[1]))))
ADF1: if checkpt(0) then mem/v[0] += previous else previous
```

```
MAIN: adf1 (insert (pNULL, insert (pNULL, pNULL + next, mem/v[1]), mem/v[1]))

ADF0:
{
  previous.rotate (1);
  mem/r[arg0i];
  checkpt(0);
  2.0 r/ 8.0;
  checkpt(2);
}

ADF1:
  mem/v[if checkpt( { 1; previous; 4.0; argvr; 0; } )
        then 0
        else 4]
    += mem/v[if checkpt( { 1; previous; 4.0; argvr; 0; } )
             then { 0; 4; 2.0; previous; 1; }
             else 3]
```

Program 5.3: Best evolved program from Figure 5.12—fitness average 1.227.

```
MAIN: adf1 (insert (pNULL, insert (pNULL, next, mem/v[1]), mem/v[1]))
ADF1:
  if checkpt(0) then mem/v[0] += mem/v[1]
  else mem/v[4] += mem/v[3]
```

Program 5.4: Best evolved program from Figure 5.12 edited for human readability.

and its performance is shown in Figure 5.11c. The program is non-static because it *does* raytrace each point to check whether it lies within the visibility space, before extending the answer area.

Figure 5.12 shows the results of another unseeded run. In this, the best-of-generation settles down after a few generations to a program (Program 5.2) with the functionality of seed 6 (see seed 6's entry in Figure 5.7).

However, in generation 15 a program did evolve that performed significantly better. The program's code is given in Program 5.3; Program 5.4 presents the functional essence of this program edited to aid human readability. The program's performance on the fitness cases is given in Figure 5.13.

As can be seen, the program manages to expand its answer area fairly well to fit the broader visibility spaces. Its performance is less good for the narrower ones, though the space it fails to include tends to be the part close to the model, which would not be suitable for placing a

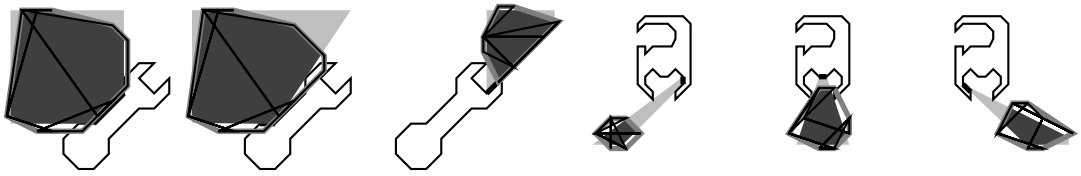
Fitness case:	0	1	2	3	4	5
Execution:						
Fitness:	1.5	2.4	1.3	11.4	3.2	7.7
Iterations:	14	14	7	6	7	7
Run length:	10031 (timed out)	10009 (timed out)	3057	1945	3065	3061

Figure 5.13: Execution of best evolved program from Figure 5.12.

Generation	Fitness case						Average
	0	1	3	4	5	8	
15	**1.5**	**2.4**	**1.3**	11.4	3.2	7.7	4.585
21	4.4	5.2	1.8	**9.7**	**2.1**	**6.4**	4.930

Table 5.6: Comparison of best-of-generation individuals from generations 15 and 21 (seed 6 equivalent) from Figure 5.12.

sensor in anyway. It generally overshoots in this expansion and includes false visibility space in its answer.

The means behind its good performance is its insertion of extra points into the answer contour every time it is called; this means that even such a simple strategy as is used in seed 6 (or ADF 1 of this program) can deliver more complex contours than seed 6 itself.

Figure 5.14 shows its performance on unseen input—model features it was not trained on—with that of the complete hand-crafted solution seed 1; it can be seen that the program performs similarly well to how it did on the fitness cases.

However, it will be observed from the fitness graph that this program was lost on subsequent generations: In three of the six fitness cases, this program was outcompeted by seed 6 (see Table 5.6); thus on those generations for which these fitness cases were used, Program 5.3 was

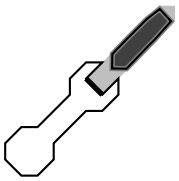
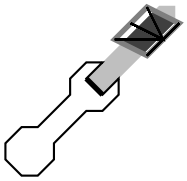
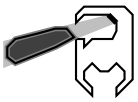
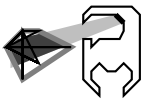
Program	seed 1	evolved	seed 1	evolved
				
Fitness*:	38	44	2	5
Iterations:	22	5	19	6
Run length:	3388	842	3634	1005

Figure 5.14: Performance comparison of the best evolved program from Figure 5.12 with seed 1, on unseen input.

at risk of being selected for replacement by the result of other programs' recombination.

5.4.2 Seeded experiments

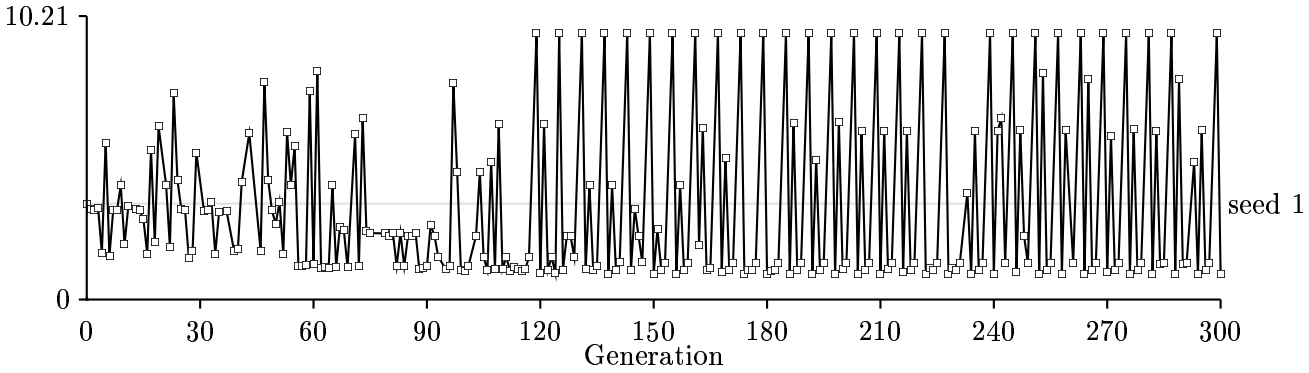
The following experiments, unless otherwise stated, were carried out using one hundred copies of the seeded program specified to seed the population. The seeded programs were added after initial program generation; the algorithm used for seeding the population ensured that, wherever possible, a seeded program would not replace a program with a fitness better than itself.

Evolution from seed 1

Figures 5.15 and 5.16 show the results of evolution when the system is seeded with the hand-crafted complete solution, seed 1. It was not expected that the genetic system would be able to evolve a program outperforming this; the experiment was performed in order to verify that programs as good as the hand-crafted solution would be retained in the population: If the population lost programs of this ability it would indicate there was a serious flaw somewhere in the system.

As it transpired, not only did the system not lose the perfect solution from the population,

*The fitnesses of evaluations for unseen data are not reported to as high a precision as for fitness case evaluations, because the fitnesses are only reported in the program evaluation window of the GUI, where to conserve space they are reported as integers.



Evolution style: generational; seeded programs: seed 1 × 100; random number seed: 939314568

Figure 5.15: Evolution from seed 1. Best-of-population average fitness (points with fitness calculation errors were removed from this graph.) Best of run reached on generation 174 (fitness average 0.924).

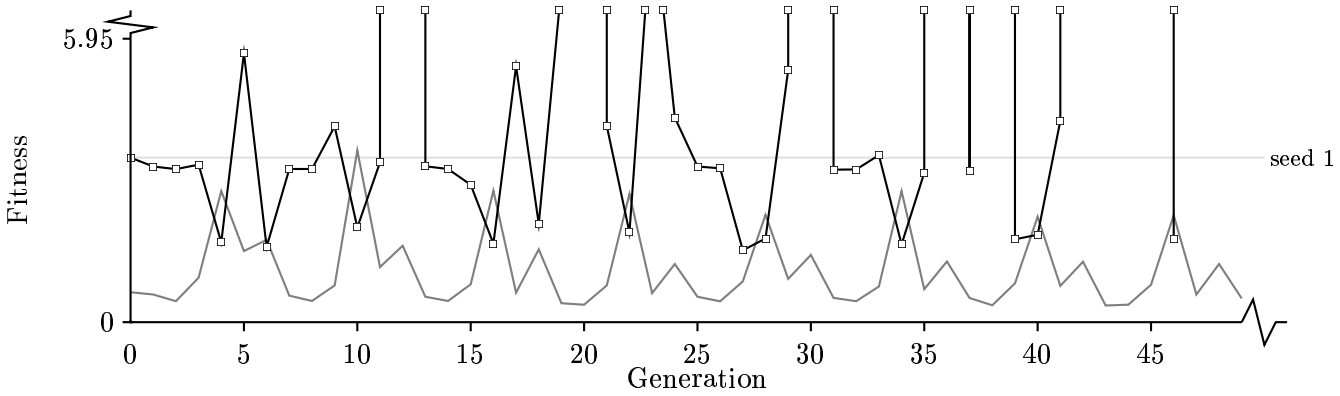


Figure 5.16: Close-up of the beginning of Figure 5.15 above, with seed 1's average fitness indicated.

but it even managed to better it, as measured by the fitness function in use. Detailed examination revealed that this program managed to outcompete the hand-crafted one on all six fitness cases, most notably on the most difficult fitness case, (number 3 in Figure 5.5), for which the hand-crafted program had a fitness of 14.2, but the evolved program just 1.9.

Figure 5.17 compares the performance of these two programs on four of the fitness cases, along with a model feature on which the system had not been trained. This comprises a test both of the choice of fitness cases, that they were distributed across the input data space, and of the evolved program, that it is robust enough to work well on data it had not been trained on. As can be seen, even on this unseen data the evolved program managed to outperform the

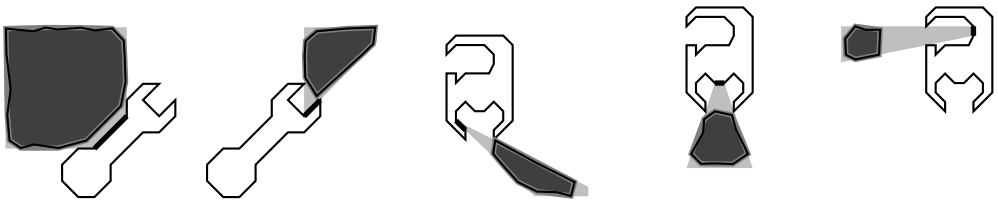
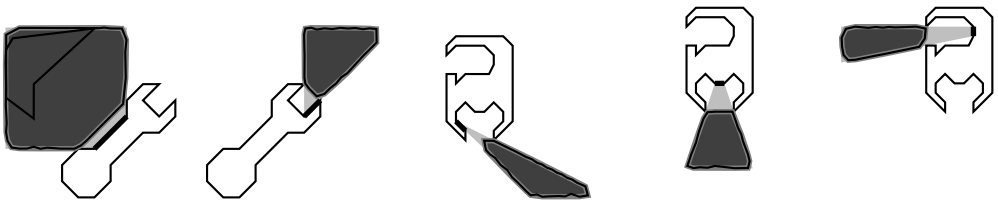
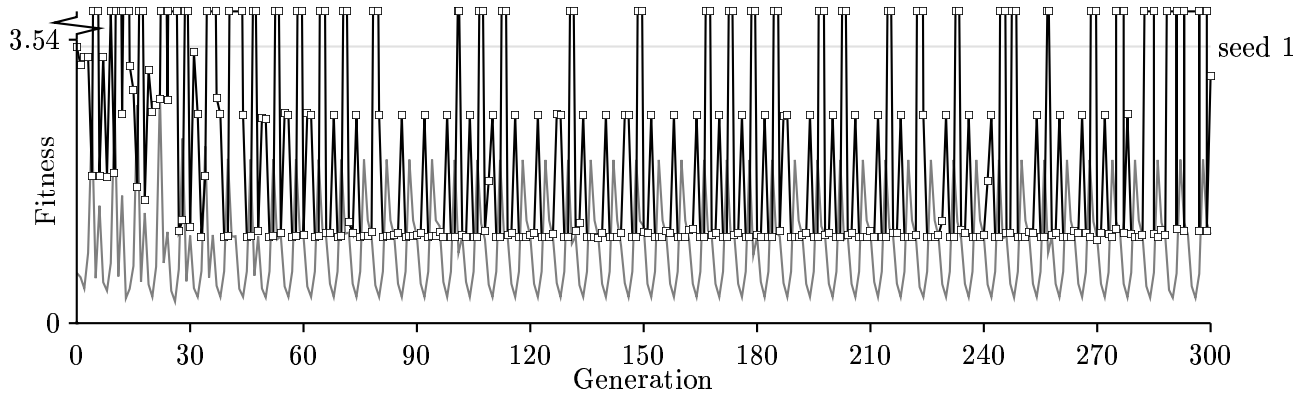
Fitness case:	0	2	5	4	n/a
Hand-crafted program:					
Fitness:	0.6	1.2	1.9	2.1	
Iterations:	28	17	19	13	10
Run length:	8770	3819	4246	2599	1677
Genetic program:					
Fitness:	0.5	0.6	1.0	1.1	1
Iterations:	12	12	13	10	16
Run length:	10049 (timed out)	7030	7975	5545	6650

Figure 5.17: Performance comparison from the run illustrated in Figure 5.15, of seed 1 (top), which was used to seed the run, and the best evolved program (bottom).



Evolution style: steady state; seeded programs: seed 1 × 100; random number seed: 942919136

Figure 5.18: Evolution from seed 1. Best of run reached on generation 270 (fitness average 1.041).

hand-crafted program.

The code of the best-of-run individual is given as Program 5.5, with the regions of code in which it differs from seed 1 indicated. Though most of the program is the same, there are several places of divergence, the largest of which is produced by repeated crossing with the same branch of seed 1 (probably with different individuals).

This is not necessarily a bad thing; it is common in biological evolution[99] and has been claimed as a driving force behind major evolutionary changes.[88] Many proteins are oligomeric, that is, consist of several subunits each comprised of a separate chain of amino acids. In many oligomeric proteins, the monomers are similar, but not identical, amino acid chains, derived by gene duplication and then diversification of the resulting duplicates. A classic example is haemoglobin, which consists of two alpha subunits and two beta subunits, which are similar both to each other and to the monomeric protein myoglobin*.

There are also examples of motif duplication within proteins, which may be more analogous to the situation seen here, as the biological gene is more directly analogous to the GP program than to the GP gene. (GP genes are also indivisible, unlike biological genes.)

Figure 5.18 shows the results of a second run seeded with seed 1, this one carried out with steady state GP.

Evolution from seed 1a

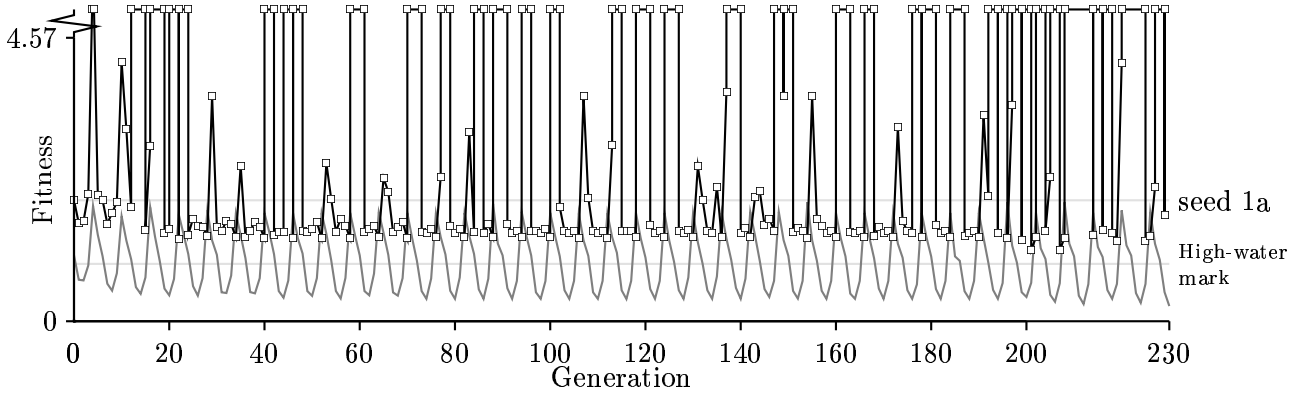
Runs with seed 1a were carried out at a late stage in this study, and a full set of results were not collected. As Figure 5.19 shows, these runs also managed to better the program with which they were seeded, but they did not evolve results as good as those from the runs seeded with seed 1, despite the fact that their initial seed possessed a better average fitness. This is because, as mentioned in Section 5.3.2, seed 1a's fitness average was beneath that of seed 1 as a result of its good performance in the single fitness case for which seed 1 performed badly; for the other fitness cases its performance was inferior.

The runs for seed 1a are of shorter duration than the other runs in this section; in order that a fair comparison may be made, the performance of runs seeded with seeds 1 and 1a are

*ibid. ch. 23

```
MAIN:
if mem/v[1].magnitude > 0.0 then
  if mem/v[1].magnitude < mem/r[0] then {
    mem/v[1] = pNULL;
    mem/v[5] = previous;
    if adf0 (0, 5) then adf1 (previous) else pNULL;
    mem/v[5] = next;
    if adf0 (5, 0) then adf1 (next) else pNULL;
  } else {
    {
      {
        mem/v[2] = mem/v[0];
        mem/v[2] += mem/v[1];
        if checkpt(2) then mem/v[0] = mem/v[2]
        else mem/v[1] /= 2.0;
        0.0; pNULL;
      };
      {
        mem/v[2] = mem/v[0];
        mem/v[2] += (mem/v[1] /= 2.0);
        if checkpt(2) then mem/v[0] = mem/v[2]
        else mem/v[1] /= 2.0;
        0.0; pNULL;
      };
      mem/v[2] = mem/v[0]; 0.0; pNULL;
    };
    mem/v[2] += mem/v[1];
    if checkpt(2) then mem/v[0] = mem/v[2]
    else next;
    0.0; pNULL;
  } else pNULL

ADF0:
 (mem/v[2] +=
     (mem/v[2] =
       (mem/v[arg0i] - mem/v[arg1i]))).magnitude
   > mem/r[1]

ADF1:
{
  mem/v[3] = (mem/v[0] + mem/v[5]);
  mem/v[3] /= 2.0;
  mem/v[4] = mem/v[2].rotate (3);
  4.0;
  insert (argvr, mem/v[3], mem/v[4] /= 4.0);
}
```

Program 5.5: Best evolved program from Figure 5.15, with differences from seed 1 indicated.



Evolution style: steady state; seeded programs: seed 1a × 100; random number seed: 954691694

Figure 5.19: Evolution from seed 1a (230-generation run). "High-water mark" indicates the best fitness evolved from runs seeded with seed 1. Best of run reached on generation 201 (fitness average 1.152).

Evolution	Seeds	Best running fitness average Generation:			
		0	159	230	300
Generational	seed 1 × 100	3.454	1.043		0.924
	seed 1a × 100	1.950	1.350	—	
Steady state	seed 1 × 100	3.454		1.076	1.041
	seed 1a × 100	1.950		1.152	—

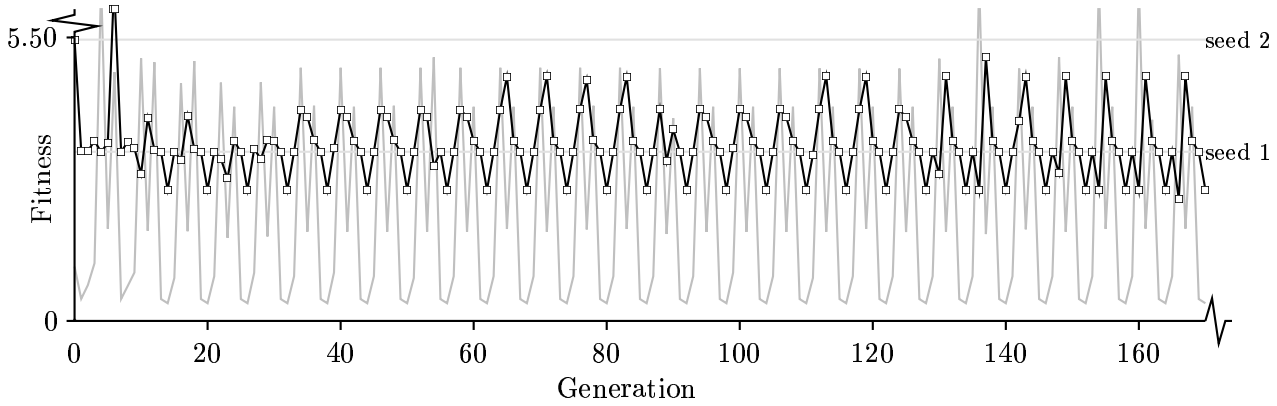
Table 5.7: Comparative performance of runs seeded with seeds 1 and 1a.

compared at the same generations in Table 5.7.

Evolution from seed 2

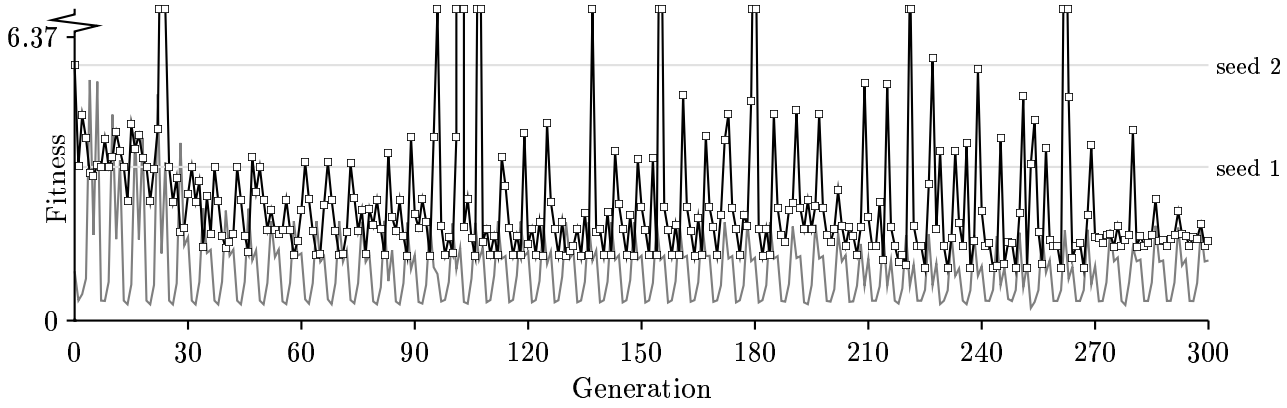
Figures 5.20 and 5.21 show the results of runs seeded with seed 2. These too managed to attain a fitness better than that of seed 1, but the best-of-runs were not as good as those produced by the runs seeded with seed 1. The run using steady-state GP performed better than that using generational GP.

Program 5.6 gives the code for the best individual evolved on this run. Like Program 5.5 in the previous section, this program evolved by duplicating a large block of code from its ancestral



Evolution style: generational; seeded programs: seed 2 × 100; random number seed: 939314573

Figure 5.20: Evolution from seed 2 (shown only to generation 170). Best of run reached in generation 166—fitness average 2.495.

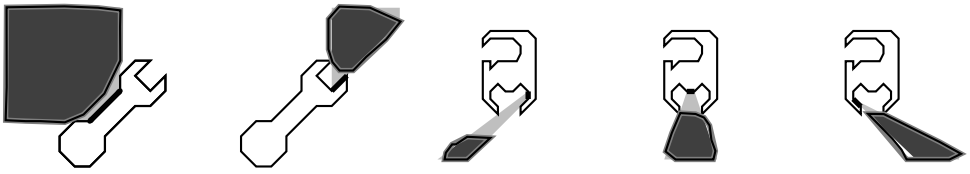


Evolution style: steady state; seeded programs: seed 2 × 100; random number seed: 939314564

Figure 5.21: Evolution from seed 2 (steady state). Best of run found in generation 231—fitness average 1.170.

```
MAIN:
  if checkpt(0) then {
    mem/v[0] += ({ mem/v[1];
                   mem/v[4] = (mem/v[1] - mem/v[1].rotate (3));
                   2.0;
                   mem/v[4] /= 2.0;
                   mem/v[4] /= 2.0;
                 });
    0.0; 0.0; 0.0;
    if checkpt(0) then pNULL
    else {
      mem/v[5]; mem/v[4];
      (mem/v[4] = mem/v[4].rotate (3)).magnitude;
      {
         mem/v[4] += (mem/v[1] + mem/v[4].rotate (3));
         mem/v[5] = (previous + mem/v[0]);
         mem/v[4] /= 2.0;
         insert (previous, mem/v[5] / 2.0, mem/v[4] /= mem/v[4].magnitude);
         mem/v[1];
       };
      mem/v[4];
    };
  } else pNULL
ADF0:
  8.0 > previous.magnitude
ADF1:
  next
```

Program 5.6: Best evolved program from Figure 5.21. Differences from seed 2 are indicated.

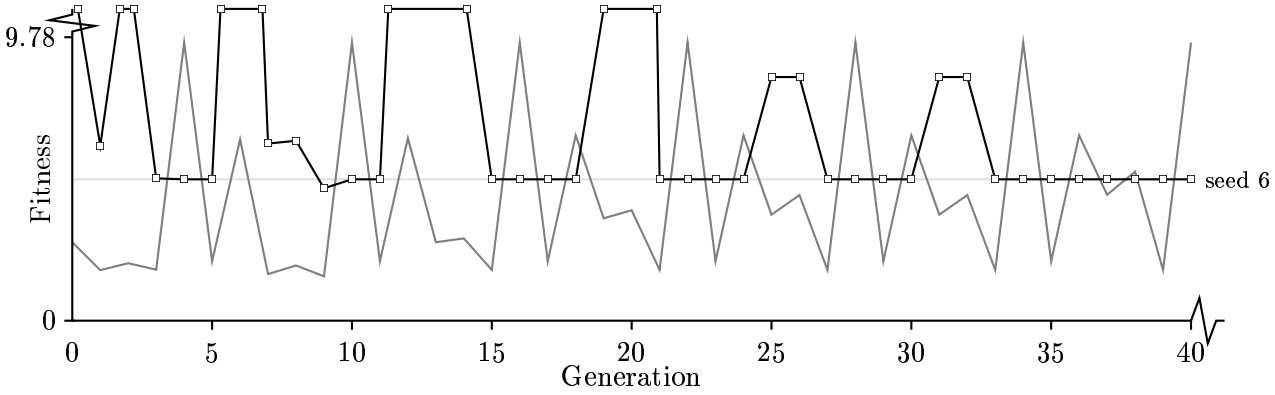
Fitness case:	0	2	3	4	5
seed 2:					
Fitness:	1.1	2.3	17.0	3.6	9.0
Iterations:	14	3	3	2	4
Run length:	2662	827	1212	627	1235
Evolved program:					
Fitness:	0.4	1.3	4.9	1.4	1.6
Iterations:	54	20	13	15	19
Run length:	6023	2630	1313	2231	2611

(a) Fitness cases.

seed 2:			
Fitness:	26	9	13
Iterations:	5	7	6
Run length:	1099	1898	1573
Evolved program:			
Fitness:	26	3	3
Iterations:	16	31	25
Run length:	2780	2975	3151

(b) Unseen data.

Figure 5.22: Performance comparison from the run illustrated in Figure 5.21, of the best evolved program and seed 2 (which was used to seed the run).



Evolution style: generational; seeded programs: seed 3 × 100; random number seed: 939314561

Figure 5.23: Evolution from seed 3, shown to generation 40 only: the best-of-generation individual stabilised after 33 generations.

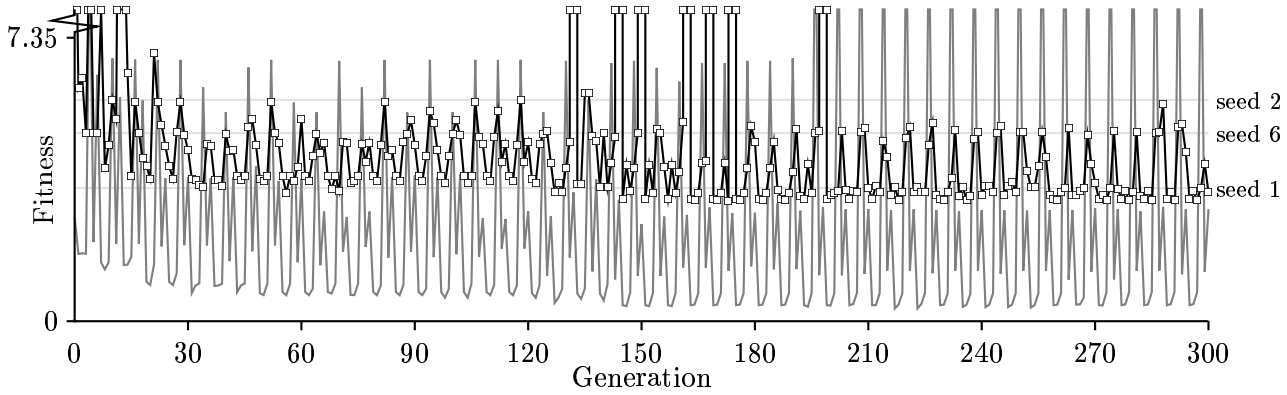
seed. Figure 5.22 shows the results of executing this program, compared to its originator, seed 2. As can be seen, the extra code included in this program resulted in a considerable improvement in performance, both on in-training-set and unseen input data. This diagram may be compared with Figure 5.17, to contrast its performance with those of seed 1 and the best evolved program from the run seeded with seed 1.

Evolution from seed 3

As mentioned earlier (see Section 5.3.2), it was an unavoidable feature of the system that results of moderate goodness could be achieved with very simple programs, such as seeds 5 and 6. The consequence of this is that runs with seed 3 tended to slip down, in terms of complexity if not fitness, into programs functionally equivalent to seed 6, rather than hauling themselves up towards seed 2 or similar programs.

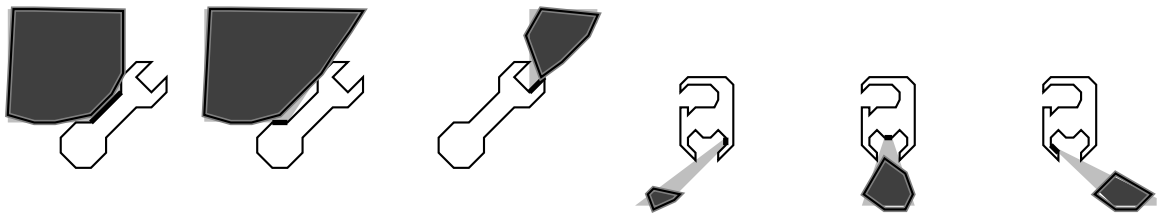
An example run is shown in Figure 5.23: a program functionally equivalent to seed 6 was discovered on generation 5, and the best-of-generation settled on this program's descendants from generation 33 onwards.

A run that managed to evolve programs of better fitness by adding algorithmic complexity rather than losing it is illustrated in Figure 5.24. The performance of one of the best programs from that run is given in Figure 5.25. As can be seen, it performed well on the fitness cases with



Evolution style: steady state; seeded programs: seed 3 × 100; random number seed: 943041115

Figure 5.24: Evolution from seed 3.



F. case:	0	1	2	3	4	5
Fitness:	0.4	0.4	1.1	11.0	2.0	4.7
Iter.s:	11	13	5	3	3	3
Run len.:	4132	4655	1192	394	771	602

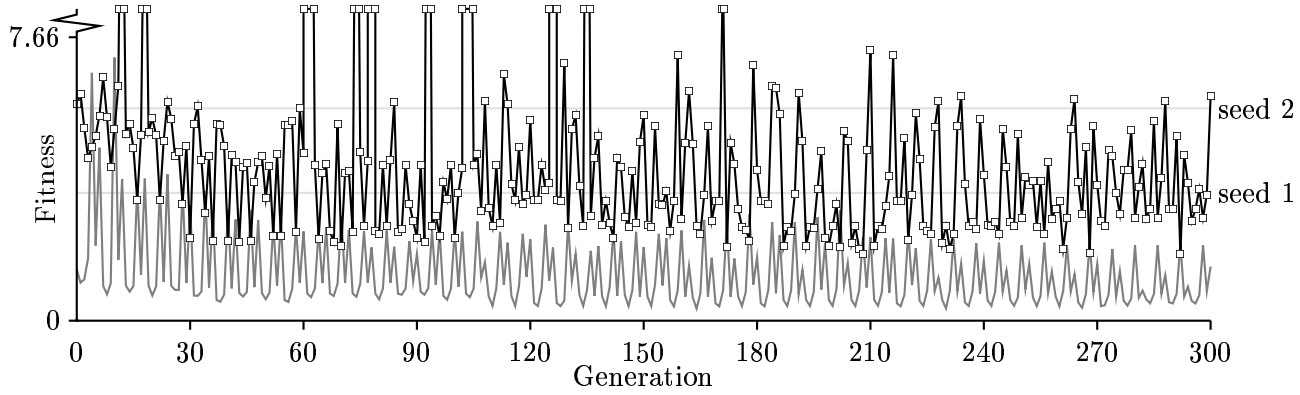
Figure 5.25: Performance of best individual of generation 272 from Figure 5.24—fitness average 3.276.

```
MAIN:
  if checkpt(0) then {
    {
      mem/v[0] += ( ( (mem/v[1] / 2.0)  / 2.0)  / 2.0);
      0.0; 0.0; 0.0; mem/v[1];
    };
    0.0; 0.0; 0.0;
    if checkpt(0) then {
      mem/v[0] += ( (mem/v[1] / 2.0)  / 2.0);
      0.0; 0.0; 0.0;
      if checkpt(0) then {
        {
          mem/v[0] += ( ( (mem/v[1] / 2.0)  / 2.0)  / 2.0);
          0.0; 0.0; 0.0;
          insert (next, next, mem/v[1]);
        };
        0.0; 0.0; 0.0;
        if checkpt(0) then {
          mem/v[0] += (mem/v[1] / 2.0);
          0.0; 0.0; 0.0;
          if checkpt(0) then {
            mem/v[0] += ( ( (mem/v[1] / 2.0)  / 2.0)  / 2.0);
            0.0; 0.0; 0.0;
            mem/v[0] += ( (mem/v[1] / 2.0)  / 2.0);
          } else pNULL;
        } else pNULL;
      } else pNULL;
    } else pNULL;
  } else pNULL

ADF0: checkpt({ 0; next + previous; 1; 2.0; 3; })
ADF1: mem/v[0] /= 2.0
```

Program 5.7: Best individual of generation 272 from Figure 5.24.

broad visibility spaces, but was unable to deal well with those with narrow visibility spaces. Despite this, it was able to attain an average fitness better than that of seed 1. As can be seen from the repeated motifs in its code (Program 5.7); it was derived from repeated crossovers with seed 3 and its derivatives (compare with seed 3, p. 270).

The fitness average of the best individual from this run was not as good as those from the runs seeded with seeds 1, 1a or 2.



Evolution style: steady state; seeded programs: seed 3a × 100; random number seed: 939314566

Figure 5.26: Evolution from seed 3a, with fitnesses of seed 2 and seed 1 indicated. Best of run reached on generation 208 (fitness average 1.808).

Evolution from seed 3a

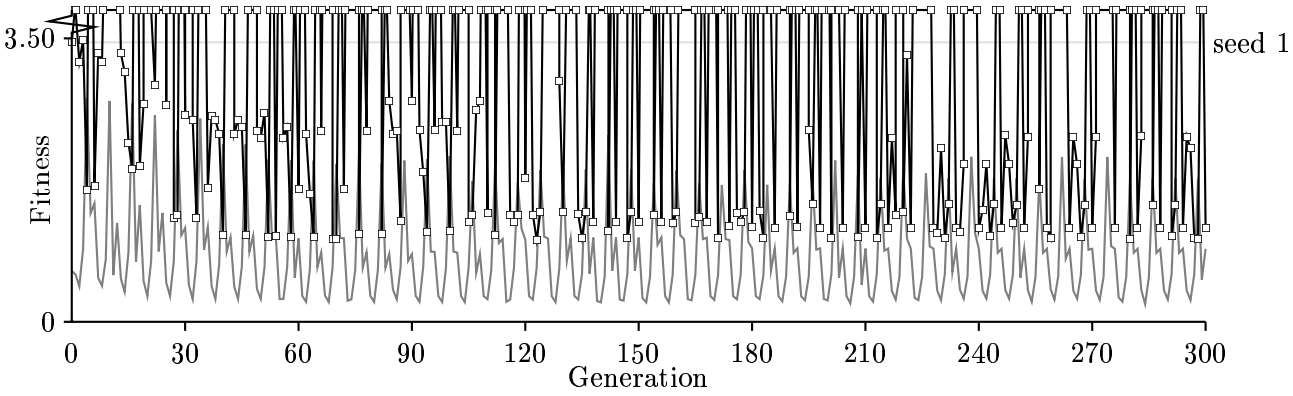
A run was carried out* seeded with seed 3a; this run evolved a seed 6-equivalent after three generations, and stabilised on it after 27.

A second run, however, managed to evolve programs that performed considerably better than seed 1 (Figure 5.26). These programs' fitness averages were better than those of the runs seeded with seed 3, but were not as good as those of the runs seeded with seeds 1, 1a and 2. This appears to indicate that a run's ability to improve on its initial population may be dependent on the complexity of the gene pool with which it is seeded. This is substantiated by the fact that the best individuals of seeded runs shown in this section are all formed by rearrangement and reduplication of material from seeded programs, with hardly any novel content.

Evolution from mixed seeds

In this experiment, 100 each of seed 1, seed 2 and seed 3a were used to seed the population. Evolution with this mixture did not produce as good a result as the above-stated hypothesis might have suggested, but it did still produce programs significantly better than seed 1, performing well on the narrower visibility areas. The best individual of generation 4 on the run shown in Figure 5.27 is particularly worthy of note; it differs from seed 1 in just a single subtree

*Evolution style: generational; seeded programs: seed 3a × 100; random number seed: 939314566



Evolution style: generational; seeded programs: seed 1 × 100, seed 2 × 100 & seed 3a × 100; random number seed: 939314569

Figure 5.27: Evolution from mixed seeds. Best of run reached on generation 123 (fitness average 1.014).

```
MAIN:
  if mem/v[1].magnitude > 0.0 then
    if mem/v[1].magnitude < mem/r[0] then {
      mem/v[1] = pNULL;
      mem/v[5] = previous;
      if adf0 (0,5) then adf1 (previous) else pNULL;
      mem/v[5] = next;
      if adf0(5,0) then adf1 (next) else pNULL;
    } else {
      mem/v[2] = mem/v[0];
      mem/v[2] += mem/v[1];
      if checkpt(2) then mem/v[0] = mem/v[2]
      else mem/v[1] /= 2.0;
      0.0; pNULL; }
  else pNULL

ADF0:
  (mem/v[2] = (mem/v[arg0i] - mem/v[arg1i])).magnitude > mem/r[1]

ADF1: {
  mem/v[3] = (mem/v[0] + mem/v[5]);
  mem/v[3] /= 2.0;
  mem/v[4] = mem/v[2].rotate (3);
  if  (mem/r[2] = (mem/r[2] = mem/r[2])) > 8.0
  then mem/v[4] /= (mem/r[2] r/ 4.0)
  else pNULL;
  insert (argvr, mem/v[3], mem/v[4]);
}
```

Program 5.8: Best individual of generation 4 from Figure 5.27.

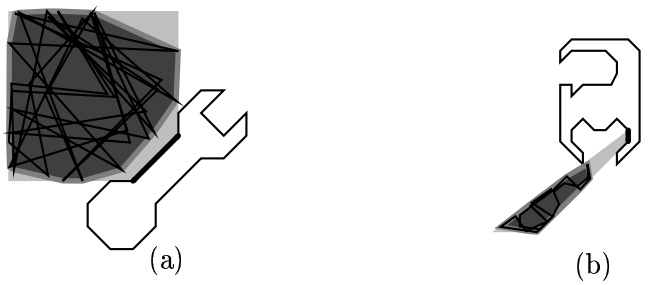


Figure 5.28: Execution of best individual of generation 4 from Figure 5.27.

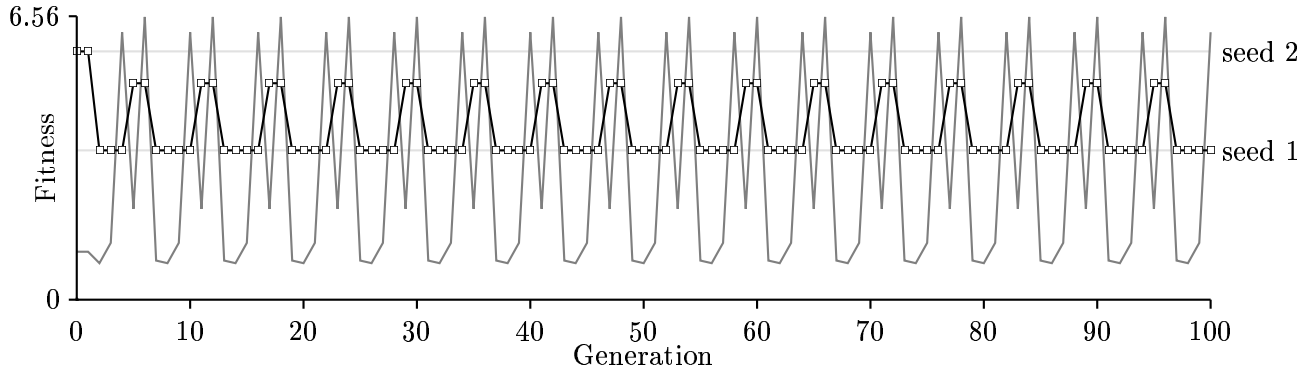
Figure 5.29: Stages in the execution of Program 5.8.

originally containing a mere five genes (highlighted in Program 5.8). The program's mode of operation is rather different from that of its progenitor, as shown in Figure 5.28a, but it is none the less effective for that—as is indicated by Figure 5.28b, where the program scores a fitness of just 2.8 against seed 1's fitness of 14.2. (Compare this with seed 1's performance on this fitness case in Figure 5.7 on p. 194.)

Programs of this form, which were not infrequently seen to evolve, illustrate well the difference between the way human and GP solutions work. In the hand-crafted solution, when an expanding point hits the visibility space boundary, new points are generated either side, half way to the next points, moving in related directions at half the speed. In these programs, the direction of newly-created points is, roughly, reversed, with the result that they end up extending the visibility area on the opposite side of the convex hull (see Figure 5.29). This is less efficient in terms of the run length of the program, but can allow less complex programs to fill in a greater portion of the correct visibility area.

5.4.3 Variations in GP

In the light of the controversy over the relative utility of crossover versus mutation (Section 2.2.11), experiments were undertaken to investigate which of these operators played a



Evolution style: steady state; seeded programs: seed 2 × 100; no crossover; random number seed: 939314576

Figure 5.30: Typical result of evolution without crossover (shown only to generation 100).

greater role in the evolution of successful solutions, using this system.

Evolution with crossover disabled

With the simple mutational operators provided in this system, it was not expected that any significant evolution might be able to occur with the use of the crossover operator disabled. Experimentation bore out this hypothesis (Table 5.8).

One unseeded run managed to produce a result of any worth, with functionality equivalent to seed 6. The program simplifies to:

```
mem/v[0] += mem/v[if checkpt(0) then 1 else 4]
```

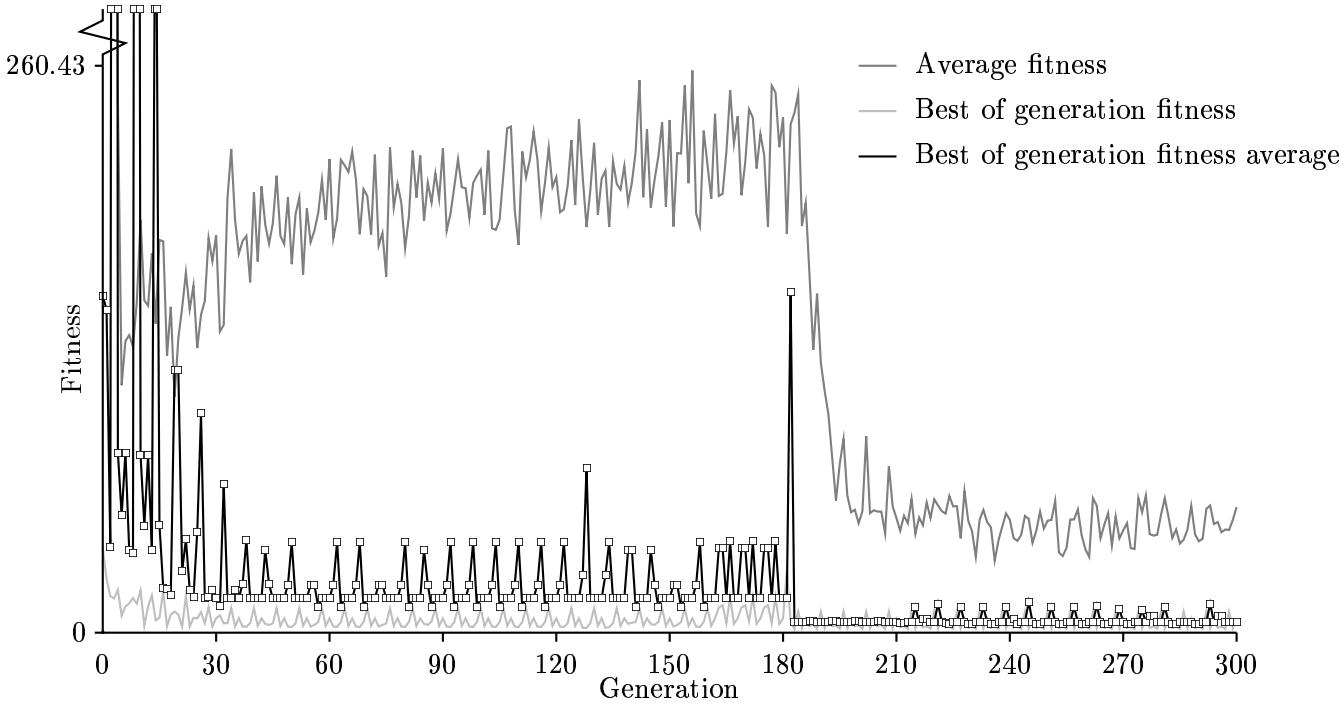
Because of the very restricted possibilities for creation of new genetic material, it is likely that something similar to this was produced in the initial random generation of programs, and that this program was subsequently found by swap mutation of just one or two genes.

Occasional point mutations might be able to produce programs that performed slightly better than the seeded programs, but that was all. Sometimes "scribblies" would be produced that, by virtue of their chaotic expansion of the answer area, would be able to cope better with the fitness cases with narrow visibility spaces. However, the gain in these fitness cases, though enough to lead to a net gain in the average fitness, would be generally counterbalanced by a loss of fitness for the other fitness cases, where the precision of the ordered expansion would be lost.

Evolution	Seeds	Rnd. Seed	Gen.	Fitness	Notes
Generational	seed 1×100	939314568	0	3.454	seed 1
			1	3.299	`>` → `<`
			46	2.926	"Scribblie" with worse fitness in every fitness case bar one.
			226	2.111	Three point mutations leadings to a slower, more methodical program
Steady State	seed 1×100	939314567	0	3.454	seed 1
			1	3.299	`>` → `<`
			61	3.266	`4` → `0`
			84	2.45	`previous` → `next`
Steady State	seed 1a×100	954691862	0	1.948	seed 1a
			2	1.813	`+` → `=`
Generational	seed 2×100	939314575	0	5.743	seed 2
			1	3.453	`+` → `-`
			5	5.006	`>` → `<` (better for some fitness cases)
Steady State	seed 2×100	939314576	0	5.743	seed 2
			2	3.453	`+` → `-` (as above)
			5	5.006	`>` → `<`
Generational	seed 3×100	939314576	0	10.477	seed 3
			5	10.354	`+` → `-`
			35	5.011	`mem/v[1]` → `mem/v[3]`. Capitalises on false negatives being better than false positives.
Steady State	seed 3×100	939314578	0	10.477	seed 3
			35	5.011	`mem/v[1]` → `mem/v[3]` as above.
			49	4.997	As gen. 35 plus `next` → `pNULL`.
Generational	seed 3a×100	943554501	0	5.853	seed 3a
			82	6.700	`next` → `previous`
			90	4.549	`next` → `pNULL`
			93	8.006	`mem/v[1]` → `mem/v[5]`
Steady State	seed 3a×100	939314576	0	5.853	seed 3a
			65	4.549	`next` → `previous`
			263	4.136	`mem[0] += [...]` → `mem[1] = [...]`.
Generational	seed 1×100	939314568	0	5.853	seed 1
	seed 2×100		2	3.299	As before
	seed 3a×100		46	2.926	
			150	2.457	Three point mutations
			226	2.111	`> 8.0` → `< 2.0`
Steady state	seed 1×100	939314573*	0	5.853	seed 1
	seed 2×100		1	3.299	As before
	seed 3a×100		112	6.260	`>` → `<`
			274	3.741	`+` → `+=`; `> 8.0` → `< 2.0`

* Run of 293 generations.

Table 5.8: Results of evolution without crossover.



Evolution style: steady state; seeded programs: none; no mutation; random number seed: 943465398

Figure 5.31: Run displaying an unusual collapse of genetic diversity.

Even so, these runs showed that it was possible for even handcrafted programs to be optimised by means of point mutation; Figure 5.30 shows seed 2 improving to the fitness of seed 1 by means of point mutation. The seed 6-equivalent program above illustrates this principle further. A useful lesson that could be drawn is to use the results of other runs for seeding runs with crossover disabled, in order to see whether single-gene mutations could be used to improve them further.

Evolution with mutation disabled

Experiments with mutation disabled tended to fall, though not always, slightly behind those with mutation enabled in the best-of-run's fitness average (see Table 5.9 on p. 232 for a comparison).

Two interesting results are presented here. Figure 5.31 shows an unseeded run in which, following the discovery of a program with the fitness (and structure) of seed 6, the genetic diversity of the population collapsed as the seed-6-equivalents flooded the population, due to being several times more fit than any other individual in the population.

Evolution style: steady state; seeded programs: seed 3a × 100; no mutation; random number seed: 943465055

Figure 5.32: An unusual style of execution.

Secondly, Figure 5.32 illustrates the execution of a program whose mode of operation appears to be as follows: When a vertex reaches the periphery of the visibility area, a new vertex is created at the same position as the next vertex anticlockwise, but with the same vector as the vertex which has reached the periphery. In visibility spaces which involve right-angles, this can result in the new vertex appearing to track along the visibility area's periphery (see figure).

The code for this program involves multiply-nested gene duplications, and is rather opaque to the human reader (Program 5.9).

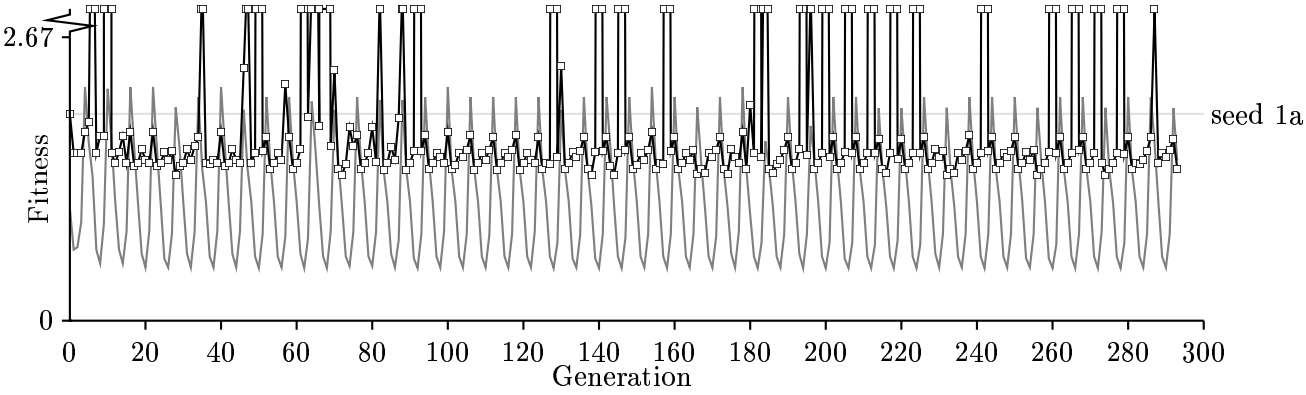
Generational versus steady-state GP

Kinnear has reported a higher success using steady state GP than generational GP.[80] In the experiments reported in this chapter, summarised in Table 5.9 on p. 232, just over half performed better with steady-state GP than generational. It is possible that improved performance with one particular method of turnover is problem-specific.

It should be noted that the reason given in Section 4.7.5 as to why programs might fare better under cycling fitness cases when steady state GP was used does not apply here. There the use of steady state evolution allowed individuals to survive through the use of fitness cases for which their performance is poor, without re-evaluation. Though this might allow programs which perform well on different fitness cases to recombine to produce new individuals scoring universally well, it also allows the persistence of static solutions, as noted on p. 151. Therefore in the current system all individuals were re-evaluated every time the fitness case was changed.

```
MAIN:
  if
    (if checkpt(0) then
       if (mem/v[0] += mem/v[1]).magnitude > 0.0 then
          if checkpt(0) then mem/v[0] += ( (mem/v[1] / 2.0)  / 2.0)
          else {
            mem/v[0] += (((mem/v[0] += ((mem/v[1].rotate (2) / 2.0) / 2.0))
                          .rotate (2)
                         / 2.0)
                        / 2.0);
            mem/v[1];
            if checkpt(0) then mem/v[0] += ( (mem/v[1] / 2.0)  / 2.0)
            else {
              mem/v[0] += ((mem/v[0] += ((mem/v[1].rotate (2) / 2.0) / 2.0))
                           / 2.0);
            mem/v[1];
            if checkpt(0) then mem/v[0] += ( (mem/v[1] / 2.0)  / 2.0)
            else pNULL;
            0;
            pNULL;
          };
          0;
          pNULL;
       }
     else pNULL
   else pNULL
      ).magnitude > 0.0 then
    if checkpt(0) then mem/v[0] += mem/v[1].rotate (2)
    else insert (next,
                 (next
                   +  ( (next
                      +  ( (next
                         +  ( (next
                            +  ( (next
                               +  ( (next + mem/v[0]) / 2.0))
                                  / 2.0))
                              / 2.0))
                           / 2.0))
                     / 2.0,
                 mem/v[1])
   else pNULL

ADF0: (2.0 r/ 4.0)  <  (2.0 r/ 4.0)
ADF1: mem/v[1] /= (0.0 r/ 8.0)
```

Program 5.9: A program with an unusual style of execution (See Figure 5.32.)



Evolution style: generational; seeded programs: seed 1a × 100; random number seed: 955129310

Figure 5.33: Untyped evolution seeded from seed 1a. Best of run reached on generation 138 (fitness average 1.375).

```
MAIN: adf1 (insert (0, mem/v[1], mem/v[1] = 2))
ADF0: 5
ADF1: mem/v[2.0] += previous
```

Program 5.10: Best evolved program of unseeded runs with no typing.

Evolution with untyped GP

Experiments were carried out with syntactic typing disabled, to investigate whether the system would be able to solve the problem without the help of typing to focus evolution. As with the experiments seeded with seed 1a, these were done at a late stage, and only a single run of each experiment was carried out.

Unseeded runs resulted in the evolution of static solutions only; as Table 5.9 on p. 232 shows, performance was inferior to that with typed GP. The best evolved solution is shown in Program 5.10.

When run with seeded programs, results were comparable in performance with those produced by typed GP; Figure 5.33 shows the result of one such run. However, these results do not necessarily indicate that typing was not needed for the evolution of good programs. Program 5.11 shows the best solution evolved by a seeded run; an examination of the code reveals that the only places where typing is ignored is where there are inconsistencies between the return types of the `then` and `else` branches of an `if` statement. The reason for this is because, as was

```
main:
  if mem/v[1].magnitude > 0.0 then
    if
      if mem/v[1].magnitude < mem/r[0] then {
        mem/v[1] = pNULL;
        mem/v[5] = previous;
        if adf0 (0, 5) then adf1 (previous) else pNULL;
        mem/v[5] = next;
        adf0 (5, 0);
      } else {
        {
          mem/v[2] = mem/v[0];
          mem/v[2] += (mem/v[1] /= 2.0);
          if checkpt(2) then mem/v[0] = mem/v[2] else mem/r[0];
          0.0; pNULL;
        };
        mem/v[2] += mem/v[1];
        if checkpt(2) then mem/v[0] = mem/v[2] else mem/v[1] /= 2.0;
        0.0; pNULL;
      }
      then adf1 (next)
      else {
        mem/v[2] = mem/v[0];
        mem/v[2] += mem/v[1];
        if checkpt(2) then mem/v[0] = mem/v[2] else mem/v[1];
        0.0; pNULL;
     }
  else pNULL
ADF0:
  (mem/v[2] += (mem/v[2] = (mem/v[arg0i] - mem/v[arg1i]))).magnitude
   > mem/r[1]
ADF1:
  {
    mem/v[3] =
      (mem/v[5 +
        ( {
            mem/v[3] = (mem/v[0] + mem/v[5]);
            mem/v[3] /= 2.0;
            mem/v[4];
            checkpt(4);
            mem/v[3];
           } ));
    mem/v[3] /= 2.0;
    mem/v[4] = mem/v[2].rotate (3);
    mem/v[5];
    insert (argvr, mem/v[3], mem/v[4]);
  }
```

Program 5.11: Best evolved program from an untyped run seeded with seed 1.

```
MAIN:
 if checkpt(0) then mem/v[0] += mem/v[1]
 else
   if mem/v[1] + previous then
     (mem/v[1]
        +  ( ( (mem/v[1] + mem/v[1].rotate (3))
               + (mem/v[1] / 2.0).rotate (3))
             / 2.0))
          / 2.0
   else
     mem/v[0] += mem/v[1].rotate (2); 0;
     insert (previous,
             (mem/v[0] + previous)  / 2.0,
              (mem/v[1]
               +  ((mem/v[1]
                    + ((mem/v[1]
                        +  ((mem/v[1]
                             + (mem/v[1] / 2.0).rotate (3))
                            / 2.0))
                       / 2.0).rotate (3))
                   / 2.0))
              / 2.0);
     mem/v[1] = pNULL;
     pNULL;


ADF0:  8.0
ADF1:  mem/r[0]
```

Program 5.12: Best evolved program from an untyped run seeded with seed 1a.

the case with the results of seeded runs with type-checking enabled, this solution evolved purely from gene recombinations and duplications within the initial seeds, with no genetic material being utilised from other sources. All the genetic material in the best-of-run program, therefore, derives from programs which utilise typing. Furthermore, apart from the `if` statements mentioned above, no crossovers resulted in the creation of function points with arguments of the wrong type.

Program 5.12 shows the best evolved solution of another untyped run, this one seeded with seed 1a. Typing is only violated in a single place in this program: the predicate of the inner `if`.

These results suggest, though they do not demonstrate clearly, that typing is necessary for the evolution of good solutions.

It could, however, be argued that the system presented here is not a good model for an

untyped system, since proper type-conversion was not implemented to compensate for the loss of type-checking. This led to wrongly-typed function arguments yielding values that were not very useful; for example 1 interpreted as a floating-point number yields 1.4013×10^{-45} rather than the expected 1.0. Moreover, even though type-checking had been removed, different types were still used in the input and output values of the functions and terminals.

To remedy these deficiencies, the system would have to be modified further, such that all operations used a universal type, as in the system presented in Chapter 3. One way to do this would be to have an information-poor universal type, with oligopartite information being stored elsewhere, either piecewise, as in Chapter 3, or wholesale. An alternative would be to make the universal type subsume all other types. This would require it to have the information-storage capacity of the most information-rich of the subsumed types, which is in this case POINT. Operations would then extract information of an appropriate type from their arguments by, for example, using the magnitude of the universal POINT type for real numbers, and so forth. Such a system has not been pursued in this study.

5.5 Conclusions

A system has been presented in this chapter that tackles the visibility space problem using the generate-and-test strategy rather than the synthesis one. The method of execution of the programs was designed to lead to better correlation between improvements in the program and improvements in fitness.

This system achieved the following results:

- Simple non-static partial solutions were produced with unseeded evolution.
- When evolution was seeded with hand-crafted programs, programs evolved that fared considerably better than the ones used to seed the runs. These were able to score better fitnesses than the hand-crafted complete solution.

This leads to the following conclusions:

Seeds	Evolution	Initial fitness	Best fitness average			
			Normal	No crossover	No mutation	No typing
—	Generational	—	9.926	4.922	13.454	43.55
	Steady state	—	11.224	44.24	3.975	16.380
seed 1 × 100	Generational	3.454	0.924	2.111	0.998	—
	Steady state	3.454	1.041	2.450	1.033	1.260
seed 1a × 100	Generational	1.950	1.350*	1.813	1.384	1.375‡
	Steady state	1.950	1.152†	—	—	—
seed 2 × 100	Generational	5.743	1.569	3.453	2.645	—
	Steady state	5.743	1.170	3.453	2.259	—
seed 3 × 100	Generational	10.477	4.571	5.011	4.571	—
	Steady state	10.477	3.1328	4.997	4.571	—
seed 3a × 100	Generational	5.853	4.880	4.549	3.058	—
	Steady state	5.853	1.808	4.136	2.542	—
seed 1 × 100 seed 2 × 100 seed 3a × 100	Generational	3.454	1.014	2.111	1.169	—
	Steady state	3.454	0.926	3.741	1.192	—

* 159-, † 230- and ‡ 293-generation runs.

Table 5.9: Summary of the best results of the second typed system.

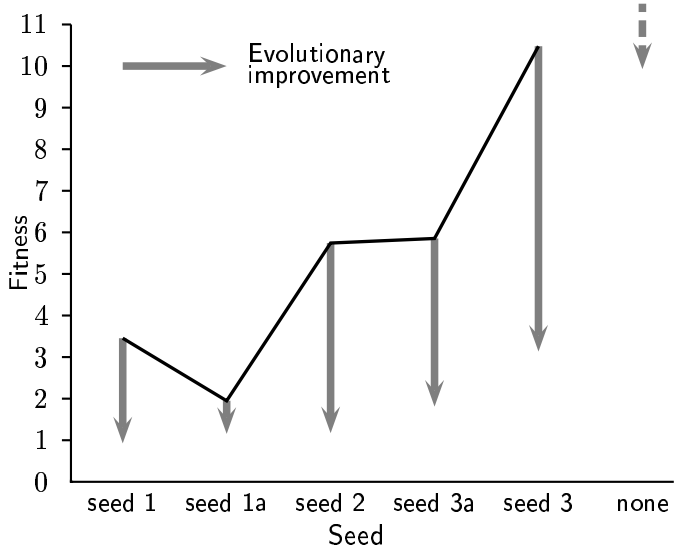


Figure 5.34: Summary of the best results of the second typed system, indicating fitness improvements between the commencement and termination of runs.

- The generate-and-test approach is far more appropriate when using GP to solve the visibility space problem than the synthesis approach.
- The system is better able to evolve good solutions to this problem with the use of *a priori* knowledge, in the form of seeded programs.

The objectives to be prioritised in this system, identified at the start of this chapter, can therefore be concluded to have had their desired effect.

The following aspects of the evolution were notable:

- Many of the programs evolved by means of nested gene duplications, thus paralleling biological evolution.
- Crossover was affirmed as being the most important operator in assembling new solutions, as expected; a role was also shown for mutation in optimising pre-evolved solutions.
- Evidence was obtained tentatively upholding the hypothesis that evolution of good results was dependent on the presence of typing in the genetic programs' language.

Chapter 6

Discussion and Conclusions

6.1 Review

This thesis has described a study carried out into the suitability of Genetic Programming for solving the visibility space problem in sensor planning for Machine Vision. This was identified in Chapter 1 as a problem the automation of which is to be desired, and Genetic Programming was identified as a potential technique for carrying this out. At the commencement of the work described in this study, the suitability of Genetic Programming for solving the visibility space problem was an unknown quantity: this was a field that had not hitherto been tackled by GP.

Chapter 2 provided a detailed introduction to the fields of both GP and sensor planning, and a literature survey of both. It mentioned the two main approaches for solving the visibility space problem, synthesis and generate-and-test. Both of these methods were used in this work in conjunction with GP, as described below.

Chapter 3 described the first system that was developed to tackle this problem. This used the synthesis approach, examining the model data to construct halfplanes for specifying solutions. It operated by untyped GP and low-level programming primitives. Halfplanes were represented as (a, b, c) triplets specifying the coefficients of the inequality $ax + by + c \geq 0$; however the system was not constrained to interpret the contents of its memories in any prespecified manner. Fitness was measured by the arithmetic difference between the triplets specifying the correct answer and the triplets, sorted, that were delivered by a program.

Evolution with this system proved capable of solving the visibility space problem only for the trivial case of convex polygons. Various techniques were put into practice to get around this problem: Runs were carried out with and without demes of different sizes, with different selection methods, with differing tournament sizes when tournament selection was used, with various mutation rates, and with dynamic fitness cases. None of these techniques was able to ameliorate the problem.

The following lessons were therefore derived:

- Visibility space computation for general two-dimensional polygons is a more complex problem than had been thought.
- The programming primitives used were too low-level to allow the expression of a parsimonious solution to the problem.
- The use of untyped GP may have resulted in evolution being insufficiently focused to solve the problem.

Chapter 4 describes the system developed to take these lessons into account. This system also applied the synthesis approach. It used typed GP, which made evolution more focused, by eliminating semantically meaningless operations. It also eased the burden on evolution, by freeing it from the need to evolve a semantic interpretation of the data structures used in the representation of the problem.

This system's programming operations were more high-level than in the previous system. This made the language of the genetic programs less general and more focused on the problem in hand. Control of iteration was taken out of the hands of the genetic program, with the intention of restoring it should the system have proved capable of solving the problem with iteration controlled by a fixed framework. Where the previous system had included large state memories, this one's memories were the smallest that could be used to solve the problem; due to the more focused programming operations, this was much less than in the previous system.

The results obtained with this system demonstrated an ability to evolve solutions of moderately good fitness but which nevertheless did not make informed use of the model data. Whilst

such static solutions were able to evolve easily, it would not be possible for more informed solutions to evolve. The following measures were therefore employed to discourage the evolution of static solutions:

- Seeding the population with hand-crafted partial solutions, in order to provide stepping-stones in the evolutionary pathway. This had little positive effect.
- Altering the fitness function to make it based on area. Though this improved the mapping between fitness function and evolutionary objectives, it did not in practice lead to improved results.
- Altering the fitness cases to prevent the evolution of static solutions. This merely led to different static solutions being found.
- Dynamic alternation of the fitness cases. This proved insufficient to weed static solutions out, due to lack of better programs to replace them.
- Tuning the parameters of the GP system. This also did not lead to improvement in evolution, indicating that the problem lay elsewhere.
- Tuning the parameters of the fitness function. This improved the shape of the fitness landscape.
- Use of program templates. This failed to improve evolution, but exposed weaknesses in the representation of the problem.
- Alteration of the function and terminal sets. This led to the elimination of static solutions, but not to the evolution of good solutions to replace them.
- Use of fitness-dependent function and terminal sets. This led to the evolution of a program which constituted the first step on the road to a correct solution.

Though these measures were in the end able to prevent the evolution of static solutions, the system was still unable to evolve programs capable of solving the visibility space problem

correctly. This was thought to be an effect of the complexity of the problem, and of the poor mapping between changes in genotype and changes in phenotype.

Modelling with a simpler system revealed that the use of superfluously large state memories impaired evolution of correct solutions. This added another reason why the previous system was not able to evolve a correct answer.

Chapter 5 describes the third system, that was consequently developed to incorporate the lessons learned from the second. This system used the generate-and-test approach rather than the synthesis one. It was also designed to incorporate better correlation between improvements in programs and improvements in fitness.

Like the second system, it used a typed genetic language that was focused on the problem to be solved and featured minimally small state memories. Like the first system, but unlike the second, it permitted manipulation of components of the answer after they were calculated. Unlike both the previous systems, the genetic programs evolved with it were "Anytime algorithms", in which answers were incrementally improved rather than calculated in one go.

This system achieved the following results:

- Simple non-static solutions were produced with unseeded evolution able to give fairly good answers to the problem.
- When evolution was seeded with hand-crafted programs, programs evolved, by recombination and reduplication of the genetic material in these, that fared considerably better than the programs used to seed the runs. These were able to score better fitnesses than the hand-crafted complete solution.
- Experiments carried out with no syntactic type-checking were suggestive of good results being dependent on the genetic programs' language being strongly typed.

It is, perhaps, worthy of note that the latter two systems both used a representation in which the answer assessed for fitness assignment was the best of the answer data produced by the genetic programs. In the first system, extraneous answer halfplanes would be penalised. In the second system, however, superfluous halfplanes would be ignored so long as they did not

affect the overall visibility area. The third system took this even further, in that the answer depended only on those points which were the furthest out from the centre of the cluster of answer points; calculated points which were not far enough from the cluster centre to contribute to the convex hull did not affect the program's fitness at all.

The following overall conclusions were drawn from the work reported in this thesis:

- The generate-and-test approach is far more appropriate than the synthesis one for use in genetically evolving programs for calculating visibility spaces.
- The system is better able to evolve good solutions to this problem with the use of *a priori* knowledge, in the form of seeded programs.

6.2 Future Work

The work described in this thesis was originally intended to be a preliminary study for extension to progressively more complex cases. Should GP have proved suitable for use in solving each of these, the ultimate goal might be the evolution of a closed-form algorithm solving the visibility space problem for generalised 3D shapes.

Since it has proved so difficult to evolve a solution even for the two-dimensional case, attempting to use GP to evolve complete solutions for the three-dimensional case would most likely be too hard a task to justify the computational effort.

However, given the third system's demonstrated ability to improve upon hand-coded solutions, a role may yet exist for GP in the more complex scenario, in optimising solutions written by humans. If this were done, it would have to be by a form of generate-and-test, since the systems in this study using the synthesis approach were shown to be incapable of improving upon seeded programs.

A putative three-dimensional system would need a change in representation. Three-dimensional visibility spaces have the potential to contain concavities and even disjoint topologies (see Figure 6.1, which contains both), and therefore cannot be represented as the intersection of halfspaces, nor as the convex hull of a cluster of points.

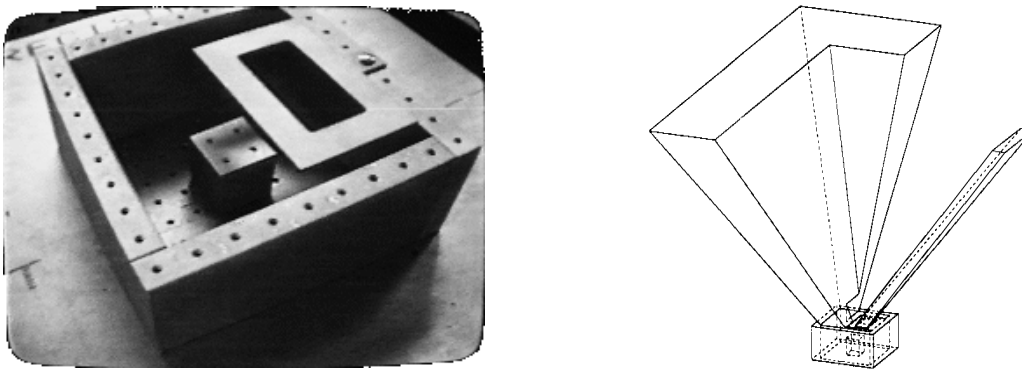


Figure 6.1:[148] Illustration of a disjoint visibility space in three dimensions. Left: Model as viewed from within visibility space. Right: Visibility space for the object inside the box.

One possibility would be to use a graph of points, specifying a polyhedral contour. To test whether GP would be able to handle this, the system presented in the previous chapter could be modified, to interpret as its answer the contour returned by the genetic program, rather than the convex hull thereof.

This had originally been intended as the answer specification for the third system, but it had been relaxed because of the many genetic programs producing self-intersecting contours: Due to the lack of success of the previous system, it had been decided to make the answer specification in the third system as lenient as possible.

6.3 Conclusions

A system has been developed capable of solving the visibility space problem for 2D polygons.

The difficulty in evolving an answer when using the synthesis approach, combined with the complexity of the problem and the poor mapping between genotype space and phenotype space, implies that even though GP has proved itself well able to carry out simple symbolic regression, the synthesis approach is not well-suited for solving the visibility space problem by Genetic Programming.

Conversely, when the generate-and-test approach was used, populations seeded with hand-crafted programs were able to evolve solutions greatly outperforming the ones they had been

seeded with. This serves to validate the generate-and-test approach as the most appropriate for use in evolving visibility space calculators.

The fact that the system fared best when seeded with partial solutions, but was not able to evolve solutions of comparable fitness *ex nihilo*, suggests that good solutions may be unable to evolve in this system without the use of *a priori* knowledge. This is further affirmed by the lesson learned in Chapter 3 regarding functions' and terminals' ability to allow a full solution to evolve: This was shown to be not necessarily guaranteed without the construction of a full solution by hand.

However, if a full solution can be constructed by hand, what is the point then of using GP to discover solutions?

Tying together these points with the demonstration in Chapter 5 that evolution by reproduction and mutation only was able to improve already evolved programs by a small but significant amount, we may conclude by postulating that for problems of this sort GP would better be used to optimise initial, approximate, solutions discovered by other means, rather than for *ab initio* construction of good solutions.

Bibliography

[1] S. Abrams and P. K. Allen. Sensor planning in an active robotic work cell. In *Proceedings of the DARPA Image Understanding Workshop*, pages 941–950, San Mateo, CA, USA, 26–29 Jan. 1992. Morgan Kaufmann.

[2] S. Abrams and P. K. Allen. Swept volumes and their use in viewpoint computation in robot work cells. In *Proceedings of the IEEE International Symposium on Assembly and Task Planning*, pages 188–193, Pittsburgh, PA, USA, 10–11 Aug. 1995. IEEE Press.

[3] S. Abrams, P. K. Allen, and K. Tarabanis. Dynamic sensor planning. In *Proceedings of the IEEE Conference on Robotics and Automation*, Atlanta, GA, USA, 2–6 May 1993. IEEE Press.

[4] S. Abrams, P. K. Allen, and K. Tarabanis. Computing camera viewpoints in an active robot work-cell. *International Journal of Robotics Research*, 18(3):267–285, March 1999.

[5] G. Adorni, F. Bergenti, and S. Cagnoni. A cellular-programming approach to pattern classification. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 142–150, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[6] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. D. Watson. *Molecular Biology of the Cell.* Garland, third edition, 1994.

[7] R. Aler. Immediate transference of global improvements to all individuals in a population in genetic programming compared to automatically defined functions for the EVEN-5 PARITY problem. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 60–70, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[8] D. P. Anderson. An orientation method for central projection programs. *Computers and Graphics*, 6(1):35–37, 1982.

[9] D. P. Anderson. Efficient algorithms for automatic viewer orientation. *Computers and Graphics*, 9(4):407–413, 1985.

[10] C. Andersson and M. G. Nordahl. Evolving coupled map lattices for computation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 151–162, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[11] D. Andre. Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 23, pages 477–494. MIT Press, 1994.

[12] D. Andre, F. H. Bennett III, and J. R. Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 3–11, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[13] M. Andrews and R. Prager. Genetic programming for the acquisition of double auction market strategies. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 16, pages 355–368. MIT Press, 1994.

[14] P. J. Angeline. Two self-adaptive crossover operators for genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA, 1996.

[15] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. Technical Report Technical report 92-PA-COEVOLVE, Laboratory for Artificial Intelligence, Ohio State University, Columbus, OH, USA, July 1993.

[16] P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, IL, USA, 17–21 July 1993. Morgan Kaufmann.

[17] D. Ballard and C. Brown. *Computer Vision.* Prentice Hall, 1982.

[18] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming—An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, Jan. 1998.

[19] S. M. Beer. *Designing Freedom.* Wiley, 1974.

[20] T. C. Belding. Numerical replication of computer simulations: Some pitfalls and how to avoid them. Unpublished paper; submitted to the Genetic and Evolutionary Computation Conference 2000.

[21] T. Belpaeme. Evolution of visual feature detectors. In R. Poli, S. Cagnoni, H.-M. Voigt, T. Fogarty, and P. Nordin, editors, *Late Breaking Papers at EvoIASP'99: the First European Workshop on Evolutionary Computation in Image Analysis and Signal Processing*, pages 1–10, Goteborg, Sweden, 28 May 1999.

[22] P. Besl. Active, optical imaging sensors. *Machine Vision and Applications*, 1(2):127–152, 1988.

[23] S. M. Bhandarkar and H. Zhang. Image segmentation using evolutionary computation. *IEEE Transactions on Evolutionary Computation*, 3(1):1–21, April 1999.

[24] T. Blickle. *Theory of Evolutionary Algorithms and Application to System Synthesis.* PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, Nov. 1996.

[25] T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik.

[26] K. W. Bowyer and C. R. Dyer. Aspect graphs: An introduction and survey of recent results. *International Journal of Imaging Systems and Technology*, 2:315–328, 1990.

[27] A. J. Briggs and B. R. Donald. Automatic sensor configuration for task-directed planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1345–1350, San Diego, CA, USA, 8–13 May 1994. IEEE Press.

[28] W. S. Bruce. Automatic generation of object-oriented programs using genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 267–272, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[29] S. Calderoni and P. Marcenac. Genetic programming for automatic design of self-adaptive robots. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 163–177, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[30] A. Cameron and H. Durrant-Whyte. A Bayesian approach to optimal sensor placement. *International Journal of Robotics Research*, 9(5):70–88, 1990.

[31] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):679–698, November 1986.

[32] G. M. Castore. Solid modelling by computers: From theory to applications. In M. Pickett and J. Boyse, editors, *Solid Modelling, Aspect Graphs and Robot Vision.* Plenum, 1984.

[33] K. Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, Sept. 1997.

[34] C.-H. Chen and P. G. Mulgaonkar. CAD-based feature-utility measures for automatic vision programming. In *Proceedings of the IEEE Workshop on Directions in Automated CAD-Based Vision*, pages 106–114, Maui, HI, USA, June 1991. IEEE Press.

[35] S. Chen and H. Freeman. On the characteristic views of quadric-surfaced solids. In *Proceedings of the IEEE Workshop on Directions in Automated CAD-Based Vision*, pages 34–43, Maui, HI, USA, June 1991. IEEE Press.

[36] J. Clark, E. Trucco, and H. Cheung. Polarization-based removal of spurious interreflections in active ranging. In *Proceedings of the IEEE Workshop on Physics-Based Modeling in Computer Vision*, pages 159–165, Cambridge, MA, USA, 18–19 June 1995. IEEE Press.

[37] J. Clark, E. Trucco, and L. Wolff. Using light polarization in laser scanning. *Image and Vision Computing Journal*, 15:107–117, 1997.

[38] C. I. Connolly. The determination of next best views. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 432–435, St. Louis, MO, USA, 25–28 Mar. 1985. IEEE Press.

[39] M. Conrads, P. Nordin, and W. Banzhaf. Speech sound discrimination with genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 113–129, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[40] D. K. Cook, P. Gmytrasiewicz, and L. B. Holder. Decision-theoretic cooperative sensor planning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(10):1013–1023, 1997.

[41] C. K. Cowan and P. D. Kovesi. Automatic sensor placement from vision task requirements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(407):407–416, 1988.

[42] J. M. Daida, T. F. Bersano-Begey, S. J. Ross, and J. F. Vesecky. Computer-assisted design of image classification algorithms: Dynamic and static fitness evaluations in a scaffolded genetic programming environment. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 279–284, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[43] J. M. Daida, J. D. Hommes, T. F. Bersano-Begey, S. J. Ross, and J. F. Vesecky. Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of arctic ice. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 21, pages 417–442. MIT Press, Cambridge, MA, USA, 1996.

[44] C. Darwin. *On the Origin of Species by Means of Natural Selection; or the Preservation of Favoured Races In the Struggle for Life.* Oxford University Press, sixth edition, 1872. 1996 reprinting.

[45] R. Dawkins. *The Selfish Gene*. Oxford University Press, 1976.

[46] P. D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261, Orlando, Florida, USA, 27–29 June 1994. IEEE Press.

[47] D. Eggert and K. Bowyer. Computing the orthographic projection aspect graph of solids of revolution. In *Proceedings of the IEEE Workshop on Interpretation of 3D scenes*, pages 102–108, Austin, TX, USA, 27–29 Nov. 1989. IEEE Press.

[48] A. E. Eiben, R. Hintering, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[49] A. E. Eiben, A. E. Koudijs, and F. Slisser. Genetic modelling of customer retention. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 178–186, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[50] H. Everett. Survey of collision avoidance and ranging sensors for mobile robots. *Robotics and Autonomous Systems*, 5:5–67, 1989.

[51] H. Everett, D. DeMuth, and E. Stitz. Survey of collision avoidance and ranging sensors for mobile robots. Technical report, U.S. Navy Naval Ocean Systems Centre, 1992.

[52] A. W. Fitzgibbon and R. B. Fisher. Practical aspect-graph derivation incorporating feature segmentation performance. In *Proceedings of the British Machine Vision Conference*, pages 580–589, Leeds, UK, 21–24 Sept. 1992.

[53] C. M. Fonseca and P. J. Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3(1):1–16, May 1995.

[54] M. Fuchs. Crossover versus mutation: An empirical and theoretical case study. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 78–85, University of Wisconsin, Madison, WI, USA, 22–25 July 1998. Morgan Kaufmann.

[55] C. Gathercole and P. Ross. Small populations over many generations can beat large populations over few generations in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 111–118, Stanford University, CA, USA, 13–16 July 1997. Morgan Kaufmann.

[56] Z. Gigus and J. Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Transactions on Robotics and Automation*, pages 1560–1566, 1988.

[57] Z. Gigus and J. Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):113–122, 1990.

[58] D. E. Goldberg. *Genetic Algorithms in Search Optimisation and Machine Learning.* Addison-Wesley, Redwood City, CA, USA, 1989.

[59] D. E. Goldberg and U.-M. O'Reilly. Where does the good stuff go, and why? how contextual semantics influence program structure in simple genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 16–36, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[60] M. S. Grant. An investigation into genetic programming. Master's thesis, Aston University, 1996. Currently available at `http://www.cee.hw.ac.uk/~msgrant/msc.zip`.

[61] K. D. Gremban and K. Ikeuchi. Planning multiple observations for object recognition. *International Journal of Computer Vision*, 12:137–172, 1994.

[62] F. Gruau. Genetic micro programming of neural networks. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 24, pages 495–518. MIT Press, 1994.

[63] S. G. Handley. The automatic generations of plans for a mobile robot via genetic programming with automatically defined functions. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 18, pages 391–407. MIT Press, 1994.

[64] C. Harris and B. Buxton. Evolving edge detectors with genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 309–315, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[65] M. Harrison and M. McLennan. *Effective Tcl/Tk Programming.* Addison-Wesley, Redwood City, CA, USA, 1997.

[66] T. D. Haynes, D. A. Schoenefeld, and R. L. Wainwright. Type inheritance in strongly typed genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[67] X. He, B. Benhabib, K. Smith, and R.Safaee-Rad. Optimal camera placement for an active vision system. *IEEE Transactions on Systems, Man. and Cybernetics*, pages 69–74, 1991.

[68] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, USA, 1975.

[69] P. Holmes. The Odin genetic programming system. Tech Report RR-95-3, Computer Studies, Napier University, Edinburgh, UK, 1995.

[70] B. Horn. *Robot Vision.* MIT Press, Cambridge, MA, USA, 1986.

[71] D. Howard, S. C. Roberts, and R. Brankin. Evolution of ship detectors for satellite SAR imagery. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 135–148, Goteborg, Sweden, 26–27 May 1999. Springer-Verlag.

[72] S. A. Hutchinson and A. C. Kak. Planning sensing strategies in a robot work cell with multi-sensor capabilities. *IEEE Transactions on Robotics and Automation*, 5(6):765–783, 1989.

[73] K. Ikeuchi and T. Kanade. Modelling sensors: Towards automatic generation of object recognition programs. *Computer Vision, Graphics and Programming*, 48(1):50–79, 1989.

[74] K. Ikeuchi and J.-C. Robert. Modeling sensor detectability with the VANTAGE geometric/sensor modeler. *IEEE Transactions on Robotics and Automation*, 7(6):771–784, 1991.

[75] T. Ito, H. Iba, and S. Sato. Non-destructive depth-dependent crossover for genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 71–82, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[76] R. Jain, R. Kasturi, and B. G. Schunk. *Machine Vision.* McGraw-Hill, New York, NY, USA, 1995.

[77] M. P. Johnson, P. Maes, and T. Darrell. Evolving visual routines. In R. A. Brooks and P. Maes, editors, ARTIFICIAL LIFE IV*, Proceedings of the fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 198–209, MIT, Cambridge, MA, USA, 6-8 July 1994. MIT Press.

[78] K. Kemmotsu and T. Kanade. Sensor placement design for object pose determination with three light-stripe range finders. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1357–1364, San Diego, CA, USA, 8–13 May 1994. IEEE Press.

[79] J. W. Kimball. *Biology.* Addison-Wesley, Redwood City, CA, USA, fifth edition, 1983.

[80] K. E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, San Francisco, CA, USA, 1993. IEEE Press.

[81] K. E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294, University of Illinois at Urbana-Champaign, IL, USA, 17–21 July 1993. Morgan Kaufmann.

[82] K. E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press, 1994.

[83] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774. Morgan Kaufmann, 20-25 Aug. 1989.

[84] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.

[85] J. R. Koza. The genetic programming paradigm: Genetically breeding populations of computer programs to solve problems. In B. Soucek and the IRIS Group, editors, *Dynamic, Genetic, and Chaotic Programming*, pages 203–321. John Wiley, New York, 1992.

[86] J. R. Koza. Hierarchical automatic function definition in genetic programming. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 297–318, Vail, CO, USA, 24–29 July 1992. Morgan Kaufmann.

[87] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.

[88] J. R. Koza and D. Andre. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 8, pages 155–176. MIT Press, Cambridge, MA, USA, 1996.

[89] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. Four problems for which a computer program evolved by genetic programming is competitive with human performance. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, volume 1, pages 1–10. IEEE Press, 1996.

[90] J. R. Koza, F. H. Bennett III, J. Lohn, F. Dunlap, M. A. Keane, and D. Andre. Automated synthesis of computational circuits using genetic programming. In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, pages 447–452, Indianapolis, IN, USA, 13–16 Apr. 1997. IEEE Press.

[91] J. R. Koza, David Andre, F. H. Bennett III, and M. Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufman, Apr. 1999.

[92] K. N. Kutulakos and C. R. Dyer. Recovering shape by purposive viewpoint adjustment. *International Journal of Computer Vision*, 12:113–146, 1994.

[93] S. Lacroix, P. Grandjean, and M. Ghallab. Perception planning for a multi-sensory interpretation machine. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1818–1824, Nice, France, 12–14 May 1992. IEEE Press.

[94] W. B. Langdon. Pareto, population partitioning, price and genetic programming. Research Note RN/95/29, University College London, UK, Apr. 1995.

[95] W. B. Langdon. Data structures and genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[96] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 37–48, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[97] W. B. Langdon and R. Poli. Genetic programming bloat with dynamic fitness. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 96–112, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[98] W. B. Langdon and R. Poli. Why ants are hard. Technical Report CSRP-98-4, University of Birmingham, School of Computer Science, Jan. 1998. Presented at GP-98.

[99] B. Lewin. *Genes VI.* Oxford University Press, 1997.

[100] H. Liu and X. Lin. Model-based next view planning by using rules—automatic features prediction and detection. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 773–776, Seattle, WA, USA, 21–23 June 1994. IEEE Press.

[101] G. F. Luger and W. A. Stubblefield. *Artificial intelligence: structures and strategies for complex problem solving.* Addison Wesley Longman, 1998.

[102] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler. Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[103] S. Luke and L. Spector. A comparison of crossover and mutation in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA, 13–16 July 1997. Morgan Kaufmann. Contains some flawed results superseded by a 1998 paper.[104]

[104] S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, University of Wisconsin, Madison, WI, USA, 22–25 July 1998. Morgan Kaufmann.

[105] J. Maver and R. Bajcsy. How to decide from the first view where to look first. In *Proceedings of the DARPA Image Understanding Workshop*, pages 482–496, Pittsburgh, PA, USA, 11–13 Sept. 1990.

[106] N. F. McPhee, N. J. Hopper, and M. L. Reierson. Impact of types on essentially typeless problems in GP. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 232–240, University of Wisconsin, Madison, WI, USA, 22–25 July 1998. Morgan Kaufmann.

[107] D. J. Montana. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., Cambridge, MA, USA, 7 May 1993.

[108] H. Murase and S. K. Nayar. Illumination planning for object recognition in structured environments. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 31–38, Seattle, WA, USA, 21–23 June 1994. IEEE Press.

[109] B. J. Nelson and P. K. Khosla. The resolvability ellipse for visual servoing. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 829–832, Seattle, WA, USA, 21–23 June 1994. IEEE Press.

[110] T. Nguyen and T. Huang. Evolvable 3D modeling for model-based object recognition systems. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 22, pages 459–475. MIT Press, 1994.

[111] N. Nikolaev and S. Slavov. Concepts of inductive genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 49–60, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[112] P. Nordin. A compiling genetic programming system that directly manipulates the machine code. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[113] P. Nordin. Speaking at a panel discussion at the First European Workshop on Genetic Programming, Paris, France, 14–15 Apr. 1998.

[114] P. Nordin and W. Banzhaf. Programmatic compression of images and sound. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 345–350, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[115] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

[116] H. Oakley. Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 17, pages 369–389. MIT Press, 1994.

[117] J. R. Olsson. How to invent functions. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 232–243, Goteborg, Sweden, 26–27 May 1999. Springer-Verlag.

[118] L. E. Orgel. Selection *in vitro*. *Proceedings of the Royal Society of London series B*, 205(1161):435–442, September 1979.

[119] N. Paterson and M. Livesey. Evolving caching algorithms in C by genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 262–267, Stanford University, CA, USA, 13–16 July 1997. Morgan Kaufmann.

[120] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, FL, USA, 27–29 June 1994. IEEE Press.

[121] R. Poli and W. B. Langdon. On the ability to search the space of programs of standard, one-point and uniform crossover in genetic programming. Technical Report CSRP-98-7, School of Computer Science, University of Birmingham, UK, Jan. 1998. Presented at GP-98.

[122] R. Poli and W. B. Langdon. A review of theoretical and experimental results on schemata in genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 1–15, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[123] R. Poli, W. B. Langdon, and U.-M. O'Reilly. Short term extinction probability of newly created schemata, and schema variance and signal-to-noise-ratio theorems in the presence of schema creation. Technical Report CSRP-98-6, School of Computer Science, University of Birmingham, UK, Jan. 1998. Presented at GP-98.

[124] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, second edition, 1992.

[125] J. C. F. Pujol and R. Poli. Efficient evolution of asymetric recurrent neural networks using a PDGP-inspired two-dimensional representation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C.

Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 130–141, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[126] S. E. Raik and D. G. Browne. Implicit versus explicit: A comparison of state in genetic programming. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996*, pages 151–159, Stanford University, CA, USA, 28–31 July 1996. Stanford Bookstore.

[127] T. S. Ray. How I created life in a virtual universe. *Le Temps Stratégique*, 47, 1992. Available in English at `http://www.hip.atr.co.jp/~ray/pubs/nathist/nathist.html`.

[128] C. W. Reynolds. Competition, coevolution and the game of tag. In R. A. Brooks and P. Maes, editors, *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 59–69, MIT, Cambridge, MA, USA, 6-8 July 1994. MIT Press.

[129] C. W. Reynolds. Evolution of corridor following behavior in a noisy world. In *Simulation of Adaptive Behaviour (SAB-94)*, 1994.

[130] C. W. Reynolds. Evolution of obstacle avoidance behaviour:using noise to promote robust solutions. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 10, pages 221–241. MIT Press, 1994.

[131] D. R. Roberts and A. D. Marshall. Viewpoint planning for complete three dimensional object surface coverage with a variety of vision system configurations. In R. Fisher, editor, *CVonline: On-Line Compendium of Computer Vision*, section 14.6. September 1998. Available at `http://www.dai.ed.ac.uk/CVonline/`.

[132] J. Rosca and D. H. Ballard. Evolution-based discovery of hierarchical behaviors. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*. AAAI / The MIT Press, 1996.

[133] J. P. Rosca and D. H. Ballard. Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994.

[134] C. Ryan. Pygmies and civil servants. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 11, pages 243–263. MIT Press, 1994.

[135] C. Ryan. Racial harmony in GAs. In *Proceedings of the German Annual Conference on Artificial Intelligence (KI-94) Workshop on GAs*, Saarbrücken, Germany, September 1994.

[136] C. Ryan, J. J. Collins, and M. O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 83–95, Paris, France, 14–15 Apr. 1998. Springer-Verlag.

[137] S. Sakane, M. Ishii, and M. Kakikura. Occlusion avoidance of visual sensors based on a hand-eye action simulator system: HEAVEN. *Advanced Robotics*, 2(2):149–165, 1987.

[138] S. Sakane and T. Sato. Automatic planning of light source and camera placement for an active photometric stereo system. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 1080–1087, Sacramento, CA, USA, 9–11 Apr. 1991. IEEE Press.

[139] O. Sharpe. Beyond NFL: A few tentative steps. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 593–600, University of Wisconsin, Madison, WI, USA, 22–25 July 1998. Morgan Kaufmann.

[140] S. Y. Shin and T. C. Woo. An optimal algorithm for finding all visible edges in a simple polygon. *IEEE Transactions on Robotics and Automation*, 5(2):202–207, 1989.

[141] E. V. Siegel. Competitively evolving decision trees against fixed training cases for natural language processing. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 19, pages 409–423. MIT Press, 1994.

[142] L. Spector and S. Luke. Cultural transmission of information in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 209–214, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[143] T. Sripradisvarakul and R. Jain. Generating aspect graphs for curved objects. In *Proceedings of the IEEE Workshop on Interpretation of 3D scenes*, pages 109–115, Austin, TX, USA, 27–29 Nov. 1989. IEEE Press.

[144] J. H. Stewman and K. W. Bowyer. Direct construction of the perspective projection aspect graph of convex polyhedra. *Computer Vision, Graphics and Image Processing*, 51:20–37, 1990.

[145] G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 94–101. Morgan Kaufmann, San Mateo, CA, USA, 1991.

[146] W. A. Tackett. Genetic programming for feature discovery and image discrimination. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 303–309, University of Illinois at Urbana-Champaign, IL, USA, 17–21 July 1993. Morgan Kaufmann.

[147] W. A. Tackett and A. Carmi. The donut problem: Scalability and generalization in genetic programming. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 7, pages 143–176. MIT Press, 1994.

[148] K. Tarabanis, P. K. Allen, and R. Y. Tsai. A survey of sensor planning in computer vision. *IEEE Transactions on Robotics and Automation*, 11(1), 1995.

[149] K. Tarabanis, R. Y. Tsai, and P. K. Allen. Analytical characterisation of the feature detectability constraints of resolution, focus and field-of-view for vision sensor planning. *Computer Vision, Graphics and Image Processing*, 59(3):340–358, 1994.

[150] A. Teller. The evolution of mental models. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[151] A. Teller. Genetic programming, indexed memory, the halting problem, and other curiosities. In *Proceedings of the 7th annual Florida Artificial Intelligence Research Symposium*, pages 270–274, Pensacola, FL, USA, May 1994. IEEE Press.

[152] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 3, pages 45–68. MIT Press, Cambridge, MA, USA, 1996.

[153] S. Teller and P. Hanrahan. Global visibility algorithms for illumination conditions. In *Proceedings of SIGGRAPH-94: Computer Graphics*, pages 443–450, Orlando, FL, USA, 24–29 July 1994.

[154] E. Trucco, M. Diprima, and V. Roberto. Visibility scripts for active feature-based inspection. *Pattern Recognition Letters*, 15:1151–1164, 1994.

[155] E. Trucco, M. Umasuthan, A. Wallace, and V. Roberto. Model-based planning of optimal sensor placements for inspection. *IEEE Transactions on Robotics and Automation*, 13(2):182–194, 1997.

[156] E. Trucco and A. Verri. *Introductory techniques for 3-D computer vision.* Prentice Hall, 1998.

[157] A. M. Wallace, B. Liang, E. Trucco, and J. Clark. Improving depth image acquisition using polarized light. *International Journal of Computer Vision*, 32(2):1–23, 1999.

[158] T. Weinbrenner and A. Fraser. GPC++ version 0.5.2. `http://thor.emk.e-technik.th-darmstadt.de/~thomasw/gpc++0.5.2.tar.gz`, 1994.

[159] E. Winfree, X. Yang, and N. C. Seeman. Universal computation via self-assembly of DNA: Some theory and experiments. In *Proceedings of the Second DIMACS Workshop on DNA-based computers*, pages 191–213, Massachusetts Institute of Technology, Cambridge, MA, USA, August 1996.

[160] D. Wolpert and W. G. MacReady. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[161] S. Yi, R. M. Haralick, and L. G. Shapiro. Automatic sensor and light source positioning for machine vision. In *Proceedings of the Tenth International Conference on Pattern Recognition*, pages 55–59, Atlantic City, NJ, USA, 16–21 June 1990.

Appendix A

Algorithms

A.1 Nodes Possibilities Table

A function or terminal can be used for program creation according to the regular routines for node selection in a typed GP system (not given here) so long as it is permitted according to the nodes possibilities table (see Section 4.3). This section gives the pseudocode for building up such a table from extant functions and terminals sets.

The table has the following dimensions:

- Node depth (or rather, height from the bottom of the tree).
- Creation type—Grow or Full.[84]
- Number of types
- Function or terminal
- Node index

class *nodespec*

  integer *height*

  creation-type *ctype*

```
  integer type


procedure TypedNodeSet :: closeNodeSet
  in integer c-depth_max
begin
  if numTypes = 0 then return                                          (1)
  nodespec n
  let fns-table [c-depth_max, 2, numTypes, fn-set.size]
      tls-table [numTypes, tl-set.size]
      fns-table-size [c-depth_max, 2, numTypes]
      tls-table-size [numTypes]                                        (2)

  foreach tl in tl-set                                                 (3)
    if tl.type ≠ GENERIC then
      tls-table [tl.type, tls-table-size [tl.type]] := tl.index
      tls-table-size [tl.type] +:= 1
    else                                                               (4)
      for type := 1 to numTypes
        tls-table [type, tls-table-size [type]] := tl.index
        tls-table-size [type] +:= 1

  for n.height := 2 to c-depth_max                                     (5)
    for n.ctype in {grow, full}
      let nodespec p := n                                              (6)
      p.height −:= 1
      if n.ctype = grow then                                           (7)
        for p.type := 1 to numTypes
          n.type := p.type
          for i := 1 to fns-table-size[n.height, n.ctype, n.type]
```

fns-table $[n.height, n.ctype, n.type, i]$:= *fns-table* $[p.height, p.ctype, p.type, i]$

for *fn* **in** *fn-set*
  let *okay* := **true**,
    generic-argument := **false**
  for *arg* := 1 **to** *fn.arguments*
    $p.type$:= $arg.type$
    if $p.type$ = GENERIC **then**
      generic-argument := **true**
    else if not *is-in-nodes-table* (p) **then**
      okay := **false**
      break

  if *okay* **then** (8)
    $n.type$:= $fn.type$
    if $n.type \neq$ GENERIC **then**
      add-to-fns-table (n, fn)

    else (9)
      if *generic-argument* **then**
        for $p.type$:= 1 **to** $numTypes$
          if *is-in-nodes-table* (p) **then**
            $n.type$:= $p.type$
            add-to-fns-table (n, fn)
      else (10)
        for $n.type$:= 1 **to** $numTypes$
          add-to-fns-table (n, fn)
end *TypedNodeSet* :: *closeNodeSet*

(1) An untyped node set needs no nodes possibilities tables.

(2) The dimension of these arrays given the value 2 is for selecting Full or Grow creation method.

(3) Create terminals possibilities table.

(4) Generic terminals are added to all types' possibilities tables.

(5) Create functions possibilities table.

(6) p for previous layer.

(7) In the Grow method of creation, the previous layer's function nodes are acceptable here too, since for specified depth d, the actual depth is $\leq d$.

(8) If the function's argument types are all found in the table at depth $d-1$ then the function can be added to the table at depth d.

(9) Special cases for generic functions. If any of the arguments of a function returning a generic value are generic themselves, then the return type is determined by the arguments' instantiation, so the function is added to the tables for such types that have non-empty tables at depth $d-1$.

(10) If this is not the case, then the return type is not shackled to child nodes' instantiation so it can be instantiated as anything. Accordingly, the function is added to all types' tables.

procedure *TypedNodeSet* :: *add-to-fns-table*
  in *nodespec* n,
  in *gpfunction fn*
begin
  let *integer* i
  for $i := 1$
    until *fns-table* $[n.height, n.ctype, n.type, i] > fn.index$
  if $i \leq$ *fns-table-size* $[n.height, n.ctype, n.type]$
    and *fns-table* $[n.height, n.ctype, n.type, i] = fn.index$

```
  then return                                                          (1)

  fns-table-size [n.height, n.ctype, n.type] +:= 1
  let oldval, newval := fn
  repeat
    oldval := fns-table [n.height, n.ctype, n.type, i]
    fns-table [n.height, n.ctype, n.type, i] := newval
    i +:= 1
    newval := oldval
  until oldval = 0                                                     (2)
end TypedNodeSet :: add-to-fns-table

function TypedNodeSet :: is-in-nodes-table
  in nodespec n
  out boolean
begin
  return
    n.type = INDIFFERENT                                               (3)
    ∨ fns-table-size [n.height, n.ctype, n.type] ≠ 0
    ∨ (n.ctype = grow                                                  (4)
        ∨ (n.ctype = full ∧ n.height = 1))
      ∧ tls-table-size [n.type] ≠ 0
end TypedNodeSet :: is-in-nodes-table
```

(1) The element is already present in the table, which comprises an ordered list in decreasing order. (The decreasing order means that no specific terminator is needed.)

(2) Inserted into ordered list.

(3) The INDIFFERENT specification effectively means any node.

(4) The Grow method of growth can produce a terminal at any depth; the Full method requires checking the terminals possibilities tables only for depth 1.

A.2 Seeded programs

A.2.1 The Untyped System

Below is given the code for the three "special ADFs" used in the untyped system.

intersect

This routine calculates the Cartesian coordinates of the intersection of two halfplanes. `arg0` and `arg1` point to the two halfplane data blocks; on exit `fsp` points to the result.

```
(defun adf2 (arg0 arg1)
  (prog2
    (prog3
      (- (* arg1.0 arg0.1 -1 1)
         (* arg1.1 arg0.0 -1 1) -1 1)
      (%
        (-
          (*
            (+
              (- fspc0 fspc0 1 2)
              arg1.1 1 2)
            arg0.2 1 1)
          (* fspc1 arg0.1 1 1)
          1 1)
        fspc2 1 1)
      (- fspc1 fspc1 1 2))
    (prog2
      (*
        (if (= arg0.1 fspc1)
            (% (+ (* (+ fspc1 arg1.0 1 1)
```

```
                fspc0 1 1)
              arg1.2 1 1)
            arg1.1 1 1)
        (% (+ (* (+ fspc1 arg0.0 1 1)
                 fspc0 1 1)
              arg0.2 1 1)
           arg0.1 1 1))
      -1 1)
    fspc0))
```

in_half_space

This routine calculates whether a point falls in a halfplane. `arg0` points to the halfplane data block, `arg1` points to the point data block; on exit `fsp` points to the result. The ADF returns the boolean value indicating whether the answer is greater than zero.

```
(defun adf2 (arg0 arg1)
  (prog3
    (prog2
      (copy arg0.0 fspc0 3)
      (* arg1.0 fspc0 1 2))
    (prog2
      (+ fspc1 fspc0 1 1)
      (+ fspc2 fspc0 1 1))
    (prog2
      (- fspc1 fspc1 1 1)
      (< fspc1 fspc0))))
```

evaluate

This routine calculates the result of feeding a point's coordinates into the formula $ax + by + c$. `arg0` points to the halfplane data block, `arg1` points to the point data block; on exit `fsp` points to the result. Since no hierarchical calling of special ADFs is permitted in this system, this entails

duplication of the code in `in_half_space`. The alternative would be to make `in_half_space` a type-2 ADF, which would complicate matters unnecessarily.

```
(defun adf2 (arg0 arg1)
  (prog2
    (prog2
      (copy arg0.0 fspc0 3)
      (* arg1.0 fspc0 1 2))
    (prog2
      (+ fspc1 fspc0 1 1)
      (+ fspc2 fspc0 1 1))))
```

A.2.2 The First Typed System

The below programs all had versions without ADFs too, in which the lines in the ADFs were inlined into the main branch.

In decreasing order of complexity and fitness:

Seed 1:

```
MAIN:
  set/p[2] (start)
  FOR SET/P[0] THROUGH MODEL POINTS
     &{
      &{
        set/a[0]
         (set/a[3]
           (elevation (mem/p[0], start)));
        if and (not (adf0 (6, 0, -pi)), adf0 (3, 0, 0.0))
        then set/a[0] (- (mem/a[0], 2pi));
        if not (samesign (mem/a[3], mem/a[4]))
        then
           &{
             0;
```

```
            &{
              set/a[5] (elevation (mem/p[0], mem/p[2]));
              set/a[4] (round (+ (pi, mem/a[6])));
              if or (and (adf0 (6, 0, 0.0),
                         < (mem/a[5], mem/a[4])),
                     and (< (mem/a[6], -pi),
                            adf0 (5, 0, mem/a[4])))
              then abort
            };
            if and (adf0 (5, 0, mem/a[4]), adf0 (3, 0, 0.0))
            then set/a[0] (- (mem/a[0], 2pi))
          }
      };
      &{ adf1 (6, 0, 0.0);
         adf1 (4, 3, 0.0);
         set/p[2] (mem/p[0]) };
      0
    }
    IF adf0 (0, 0, mem/a[1])
    THEN &{ 0;
            adf1 (1, 0, 0.0);
            set/p[1] (mem/p[0])
          }
  END FOR

  IF adf0 (1, 0, 0.0)
  THEN add (halfplane (start, mem/p[1]))
ADF0: > (mem/a[arg0i], arg0a)
ADF1: set/a[arg0i] (mem/a[arg1i])
```

Seed 2:

```
MAIN:
```

```
   mem/a[0]
   FOR SET/P[0] THROUGH MODEL POINTS
     &{
       0;
       if not (> (set/a[3] (elevation (mem/p[0], start)),
                   0.0))
       then set/a[1] (pi);
       if and (> (mem/a[3], 0.0),
               < (mem/a[1], pi))
       then set/a[0] (mem/a[3])
      }
     IF > (mem/a[0], mem/a[1])
     THEN &{
          0;
          set/a[1] (mem/a[0]);
          set/p[1] (mem/p[0])
      }
   END FOR

   IF > (pi, 0.0)
   THEN add (halfplane (start, mem/p[1]))
ADF0: > (mem/a[arg0i], arg0a)
ADF1: set/a[arg0i] (mem/a[arg1i])
```

Seed 7:

```
MAIN:
  mem/a[0]
  FOR SET/P[0] THROUGH MODEL POINTS
    &{ 0,
       if and (not (> (elevation (mem/p[0], start), 0.0)), < (mem/a[0], pi))
       then &{ 0, set/a[0] (pi), add (halfplane (start, mem/p[1])) },
       set/p[1] (mem/p[0]) }
```

```
    IF > (0.0, 0.0)
    THEN mem/a[0]
  END FOR

  IF > (pi, 0.0)
  THEN halfplane (start, start)
ADF0: > (mem/a[arg0i], arg0a)
ADF1: set/a[arg0i] (mem/a[arg1i])
```

Seed 8:

```
MAIN:
   mem/a[0]
   FOR SET/P[0] THROUGH MODEL POINTS
     &{ 0,
        if and (< (elevation (mem/p[0], start),
                   elevation (mem/p[1], start)),
                < (mem/a[0], pi))
        then &{ 0, set/a[0] (pi), add (halfplane (start, mem/p[1])) },
        set/p[1] (mem/p[0]) }
     IF > (0.0, 0.0)
     THEN mem/a[0]
   END FOR

   IF > (pi, 0.0)
   THEN halfplane (start, start)
ADF0: > (mem/a[arg0i], arg0a)
ADF1: set/a[arg0i] (mem/a[arg1i])
```

Seed 6:

```
MAIN:
   mem/a[0]
   FOR SET/P[0] THROUGH MODEL POINTS
```

```
     mem/a[0]
     IF > (set/a[3] (elevation (mem/p[0], start)), 0.0)
     THEN set/p[1] (mem/p[0])
   END FOR
   IF > (pi, 0.0)
   THEN add (halfplane (start, mem/p[1]))
ADF0: > (mem/a[arg0i], arg0a)
ADF1: set/a[arg0i] (mem/a[arg1i])
```

Seed 3:

```
MAIN:
   mem/a[0]
   FOR SET/P[0] THROUGH MODEL POINTS
     if > (set/a[3] (elevation (mem/p[0], start)), 0.0)
     then  set/a[0] (mem/a[3])
     IF > (mem/a[0], mem/a[1])
     THEN &{
           0,
           set/a[1] (mem/a[0]),
           set/p[1] (mem/p[0]) }
   END FOR

   IF > (pi, 0.0)
   THEN add (halfplane (start, mem/p[1]))
ADF0: > (mem/a[arg0i], arg0a)
ADF1: set/a[arg0i] (mem/a[arg1i])
```

Seed 4:

Does nothing.

```
MAIN:
  mem/a[0]
```

```
  FOR SET/P[0] THROUGH MODEL POINTS
     + (0.0, 0.0)
    IF > (0.0,0.0) THEN mem/a[0]
  END FOR
  IF > (0.0,0.0) THEN halfplane (start, start)
ADF0: > (arg0a, arg0a)
ADF1: mem/a[arg1i]
```

Seed 5:

Always aborts.

```
MAIN:
  mem/a[0]
  FOR SET/P[0] THROUGH MODEL POINTS
    abort
    IF > (0.0,0.0) THEN mem/a[0]
  END FOR
  IF > (pi, 0.0) THEN &{ 0;0; abort}
ADF0: > (arg0a, arg0a)
ADF1: mem/a[arg1i]
```

A.2.3 The Second Typed System

Seed 1:

```
main:
  if mem/v[1].magnitude > 0.0 then
    if mem/v[1].magnitude < mem/r[0] then {
      mem/v[1] = pNULL;
      mem/v[5] = previous;
      if adf0 (0,5) then adf1 (previous) else pNULL;
      mem/v[5] = next;
      if adf0(5,0) then adf1 (next) else pNULL;
    } else {
```

```
      mem/v[2] = mem/v[0];
      mem/v[2] += mem/v[1];
      if checkpt(2) then mem/v[0] = mem/v[2]
      else mem/v[1] /= 2.0;
      0.0; pNULL;
    }
  else pNULL

adf0: (mem/v[2] = (mem/v[arg0i] - mem/v[arg1i])).magnitude > mem/r[1]

adf1: {
  mem/v[3] = (mem/v[0] + mem/v[5]);
  mem/v[3] /= 2.0;
  mem/v[4] = mem/v[2].rotate(3);
  if mem/r[2] = (mem/v[4].magnitude r/ mem/r[0]) > 8.0 then
    mem/v[4] /= (mem/r[2] r/ 4.0)
  else pNULL;
  insert (argvr, mem/v[3], mem/v[4]);
}
```

Seed 1a:

```
main:
  if checkpt(0) then mem/v[0] += mem/v[1] else {
    mem/v[0] += mem/v[1].rotate(2); 0;
    insert (previous,
            (mem/v[0]+previous) / 2.0,
            (mem/v[1]+mem/v[1].rotate(3)) / 2.0);
    mem/v[1] = pNULL;
    pNULL;
  }

adf0: mem/r[2] > mem/r[1]
```

```
adf1: pNULL
```

Seed 2:

```
main:
  if mem/v[1].magnitude > mem/r[0] then {
    mem/v[0] += mem/v[1]; 0.0; 0.0; 0.0;
    if checkpt(0) then pNULL else {
      mem/v[4] = mem/v[1] + mem/v[1].rotate(3);
      mem/v[5] = (previous + mem/v[0]);
      mem/v[4] /= 2.0;
      if mem/v[4].magnitude > mem/r[0] then
         insert (previous, mem/v[5] / 2.0, mem/v[4]) else pNULL;
      mem/v[1] = pNULL;
    };
  }
  else pNULL

adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

Seed 3a:

```
main:
  if mem/v[1].magnitude > 0.0 then
    if checkpt(0) then mem/v[0] += mem/v[1]
    else {
      mem/v[0] += (mem/v[1].rotate(2))/2.0;
      insert (next, (next + mem/v[0]) / 2.0, mem/v[1]);
      mem/v[1] = pNULL;
      0; pNULL;
    }
  else pNULL
```

```
adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

Seed 3:

```
main:
  if mem/v[1].magnitude > 0.0 then {
    mem/v[0] += mem/v[1]; 0.0; 0.0; 0.0;
    if checkpt(0) then pNULL else {
      insert (next, (next + mem/v[0]) / 2.0, mem/v[1]);
      mem/v[1] = pNULL;
      0; 0; pNULL;
    };
  }
  else pNULL

adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

Seed 4:

```
main:
  if mem/v[1].magnitude > 0.0 then
    if checkpt(0) then mem/v[0] += mem/v[1] else {
      insert (next, (next + mem/v[0]) / 2.0, mem/v[1]);
      mem/v[1] = pNULL;
      0; 0; pNULL;
  }
  else pNULL

adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

Seed 5:

```
main:
  if mem/v[1].magnitude > 0.0 then
    if checkpt(0) then mem/v[0] += mem/v[1] else {
       insert (next, next, mem/v[1]);
       mem/v[1] = pNULL;
       0.0; 0.0; pNULL;
     }
  else pNULL

adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

Seed 6:

```
main: if checkpt(0) then mem/v[0] += mem/v[1] else pNULL
adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

Seed 7:

(Does nothing)

```
main: pNULL
adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

Seed 8:

(Grows unchecked)

```
main: mem/v[0] += mem/v[1]
adf0: mem/r[2] > mem/r[1]
adf1: pNULL
```

A.3 Evaluation Caching

The rationale behind caching the result of gene evaluation is described in Section 4.6.2; this section gives the algorithms used to carry out this caching in the form of pseudocode. In actuality many of the below functions would be coded inline for further speed.

```
class runstuff                                                    (1)

function Gene :: evaluate
  in out runstuff r,
  out GPReturnType
begin
  if result-type = CONSTANT then
    return result                                                 (2)

  result :=
    begin inner-evaluation-block
      case node
        [...]                                                     (1)
        when and:
          if child(1).eval(r).bl then return child(2).eval(r).bl
           else cache-forward-values(r)
          return false
        when or:
          if  not child(1).eval(r).bl then return child(2).eval(r).bl
           else cache-forward-values(r)
          return true
        when abort:
          if  not caching-forward-values then runstuff.abortflag = true
```

```
        return null (halfplane)
      when prog3:
        for i := 1 to 2
          if not (child(i).result-type = CONSTANT
                  or child.result-type = VARIABLE)
          then child.evaluate(r)
        return child(3).eval(r)
      when ift:
        if child(1).eval(r).bl then return child(2).eval(r)
         else cache-forward-values(r)
        return null (return-type)
      when set/a:
        let integer c1 := child(1).eval(r).intgr
        let angle c2 := child(2).eval(r).ang
        if c1 = null (integer) then return null (angle)
        else if caching-forward-values then return c2                (3)
        else [...]                                                   (1)
      when set/p:
        let integer c1 := child(1).eval(r).intgr
        let point c2 := child(2).eval(r).pt
        if c1 = null (integer) then return  null(point)
        else if caching-forward-values then return c2                (3)
        else [...]                                                   (1)
    end case
  end inner-evaluation-block

if result-type = UNKNOWN then
  result-type := CONSTANT
  case node
```

```
        when adf0:
          result-type := gp.Gene(adfstart + 0).result-type
        when adf1:
          result-type := gp.Gene(adfstart + 1).result-type
        when set/a or set/p or add or abort:
          result-type := SIDE-EFFECTS
        otherwise:
          if node in {read/a, read/p, arg0i, arg1i, arg0a} then
            result-type := VARIABLE
          for i := 1 to size
            if node = prog3 and i = 3 and result-type < SIDE-EFFECTS          (4)
            then result-type := CONSTANT
            if child(i).result-type > result-type then
              result-type := child(i).result-type                             (4)
      end case
    return result
end Gene :: evaluate
```

(1) The problem-specific internals of the genetic programs' virtual machine have been omitted here, being irrelevant to this algorithm. The only code given for executing genes is that which is closely bound up with the caching algorithm.

(2) This clause is to cache CONSTANT root nodes of top-level branches of the program.

(3) When caching forward values; the program should evaluate the gene's arguments for caching, but without actually executing the `set` instruction.

(4) Where UNKNOWN < CONSTANT < VARIABLE < SIDE-EFFECTS.

```
function Gene :: eval
  in out runstuff r,
```

```
  out GPReturnType
begin
  if result-type = CONSTANT then return Result
  else return evaluate(r)
end Gene :: eval

procedure Gene :: cache-forward-values
  in out runstuff r
begin                                                    (1)
  if child(2).result-type = UNKNOWN then
    let bool holding-var := caching-forward-values
    caching-forward-values := true
    child(2).eval(r).bl
    caching-forward-values := holding-var
end Gene :: cache-forward-values
```

(1) Caching of results is done on the first execution of a program for a given fitness case. Since some parts of the program will not be executed on this first execution, whenever one of these is encountered a look-ahead routine is used to check and store the values that would be produced by executing the other branches.

A.3.1 Sorting and Aligning Halfplanes

In the fitness function as described in Section 3.6.2, the genetic program returned a pointer to an answerblock within the state memory. From this, halfplanes (a, b, c) are constructed and converted into "geometrics" $(\theta, r,$*flipped*$)$ as follows:

$\theta := \arctan \frac{b}{a}$
$r := \frac{|c|}{\sqrt{b^2+a^2}}$ (or 0 if $\sqrt{b^2 + a^2} = 0$)
if $c < 0$ **then**

```
    if θ > 0 then θ −:= π else θ +:= π
    flipped := true
  else flipped := false
```

Invalid halfplanes of the form $(0, 0, c)$ are arbitrarily assigned the geometric value $(0, 0, true)$.

These geometrics are then aligned with the correct geometrics as specified in the fitness case. Anchor points are formed between identical geometrics; and starting from these and moving onwards in a manner described below in *calculate-raw-fitness*, the arithmetic difference between each pair of geometrics is calculated, as described below in *process-geometrics*. This whole process is illustrated in Figure 3.9 on p.83.

```
function evaluate
  in GP gp,
  in fitness-case fitness-cases [ ],
  out float stdFitness
begin
  let float rawfitness := 0, diff
  let integer-memory mem
  for f := 1 to fitness-cases.size
    diff := calculate-raw-fitness (gp.eval (mem, fitness-cases[f]), mem, fitness-cases[f])
    if diff > fitness-limit or  not finite(diff)) then
      diff := fitness-limit                                              (1)
    rawfitness +:= diff
  rawfitness := rawfitness/fitness-cases.size
  if gp.length > 100 then                                                (2)
    rawfitness ×:= gp.length/100
  if rawfitness > 500 then rawfitness := 500
  return rawfitness
end evaluate
```

(1) keep difference within range.

(2) Penalise excessive length/introns.

```
function calculate-raw-fitness
  in int genetic,
  in integer-memory mem,
  in fitness-case f-case,
  out float rawfitness
begin
  if genetic not in range 0 .. mem.state-size − 1 then return fitness-limit          (1)
  let boolean all-zeroes := true
  let float running-total := 0
  let tree<geometric> gtree                                                          (2)
  let queue<geometric> anchors
  let geometric g, f,
      next-anchor := null
  let integer index_f := 1

  for index_m := genetic step 3                                                      (3)
    if index_m = mem.rw-size then return fitness-limit                               (4)
    if not halfplane (mem[index_m]).is-valid then return fitness-limit               (1)
    g := geometric (mem[index_m])
    if not g.is-null then
      if g.is-in (gtree) then
        running-total +:= 1                                                          (5)
      else gtree.add (g)
    all-zeroes := all-zeroes ∧ g.is-null
    if g.is-null or index_m > mem.state-size
```

```
  then break
end for

if all-zeroes then return fitness-limit                                    (1)
foreach g in gtree                                                         (7)
  while g ≠ f-case[index_f] and
        g > f-case[index_f] and
        index_f ≤ f-case.answer.size
    do index_f +:= 1
  if g = f-case[index_f] then
    anchors.add (g)
    index_f +:= 1
index_f := 1
if not anchors.is-empty then
  next-anchor := anchors.read

foreach g in gtree                                                         (8)
  geometric f := f-case[index_f]
  if g = next-anchor then
    if anchors.is-empty then
      next-anchor := null
    else next-anchor = anchors.read
    while f ≠ g and index_f ≤ f-case.answer.size do
      process-geometrics (f, null)
      index_f +:= 1
      f := f-case[index_f]
    process-geometrics (f, g)
    index_f +:= 1
```

```
  else if g > f then
    while not (are-close (g, f)
          or index_f ≥ f-case.answer.size
          or f = next-anchor) do
       process-geometrics (f, null)
       index_f +:= 1
       f := f-case[index_f]
    if f = next-anchor then
      process-geometrics (null, g)
    else
      process-geometrics (f, g)
      index_f +:= 1

  else if g < f then
    if are-close(g, f) and f ≠ next-anchor then
      process-geometrics (f, g)
      index_f +:= 1
    else
      process-geometrics (null, g)

  else
    throw "Error in tree traversal."
end for

while not fcase.fcase-cell(index_f).is-null do                    (9)
    process-geometrics(fcase.fcase-cell(index_f), null)
    index_f +:= 1
if running-total ≠ 0 then                                         (10)
  if running-total = 2 × index_g then
```

running-total := 20 (11)

else *running-total* +:= 20

return *running-total*

end *calculate-raw-fitness*

(1) Discourage trivial solutions.

(2) The geometrics delivered by the genetic program are to be sorted into order. Since it is computationally easier to insert into a sorted list, this is done by means of a binary search tree. The tree traversal is done in such a way as to maintain alignment between the geometrics of the genetic and fitness case answers.

(3) Construct the binary search tree. $index_m$ indexes the numerical state memory, $index_f$ the list of geometrics making up the correct answer stored in the fitness case, and $index_g$ the answer as delivered by the genetic program. g is a geometric delivered by the genetic program; f the correct one associated with that fitness case.

(4) Discourage trivial solutions; the solution dealt with here simply returns the question.

(5) Penalise for repetition.

(6) Compensate for repetition.

(7) First traversal: locate the anchor points.

(8) Second traversal: compare the geometrics.

(9) Conclusion of traversal: "overhanging" correct answers.

(10) Penalisation for all incorrect answers, approximating to rewarding all correct ones.

(11) The program has simply copied the question to a new location. Whilst this is a good start, it does need to be distinguished from having near-perfect fitness.

function *are-close*

in *geometric f, g*,

out *boolean*

begin

return $g.\theta = f.\theta$ **or** $g.r = f.r$

end *are-close*

procedure *process-geometrics*

in out *float running-total*

in *geometric f, g*

begin

if *f.is-null* **then**

$running\text{-}total +:= \ | \ g.\theta(1 + g.r) \ |$

else if *g.is-null* **then**

$running\text{-}total +:= \ | \ f.\theta(1 + f.r) \ |$

else

let $\Delta\theta :=| \ g.\theta - f.\theta \ rem^* \ 2\pi \ |$,

$\Delta r :=| \ g.r - f.r \ |$,

$retval := (1 + \Delta r)\Delta\theta$ (1)

if $g.flipped \neq f.flipped$ **then** (2)

if $retval = 0$ **then** $retval := 2$

else $retval \times:= 2$

$running\text{-}total +:= retval$

end *process-geometrics*

(1) This prevents θ from becoming unmonitored should $r \to 0$.

(2) Right halfplane delineation, but the wrong half of the plane.

* "remainder" as distinct from "mod", as defined in ANSI/IEEE Std 754-1985.