Effects of Modifying the Expressive Power of the Language of Child Producing Programs in Autoconstructive Genetic Programming

A dissertation submitted in partial fulfilment of
the requirements for the Open University's
Master of Science Degree in Computing for
Commerce and Industry

Neil Singh
(Y4628661)

6th March, 2012

Word Count (Chapters): 14,958

Preface

This project has been a large part of my life during the past fourteen months and marks the final step of four years of study towards a MSc degree.

I would like to thank Dr Karl Jones, my project supervisor, for his invaluable advice and guidance; and Dr Rob Walker and my fellow students for their informative posts in the course forum.

Most of all, I would like to thank my wife, family and friends. I am indebted to them for their emotional and financial support.

Table of Contents

List of Figures

List of Tables

Abstract

This project focuses on a form of *genetic programming* called *autoconstructive genetic programming*. Genetic programming is an automatic programming technique that evolves an initial random population of programs until an adequate program is found or a fixed count of evolutions is reached. Potential solutions (programs) are known as *individuals* and are evolved by *genetic operators* that are chosen by a researcher after preliminary experiments.

The performance of a genetic programming system depends on its parameters, which include the choice of genetic operators, and the problem it is set to solve. Furthermore, the preliminary experiments can be laborious and are not guaranteed to result in optimal parameter values because of the large number of possible values and complex interactions between parameters. Consequently, researchers have investigated automating parameter setting, and autoconstructive genetic programming was developed as a result. In autoconstructive genetic programming, individuals are responsible for problem solving *and* child production. Instead of being predetermined, genetic operators are coded within individuals and are evolved.

This project investigates the effects of reducing the expressive power — the set of enabled instructions in a program's language — of the child producing parts of individuals in autoconstructive genetic programming. The project implements an autoconstructive genetic programming system that is executed in controlled laboratory experiments.

The project's results suggest that the expressive power of the language of the child producing parts of individuals does affect a system's reproductive competence — the percentage of individuals in a population which can successfully produce a child — and its problem solving performance. Indeed, the results suggest that a reduction of expressive power can be beneficial but only when the reduction is defined by the problem being tackled. Consequently, the implications of the results

are that the expressive power can be considered as another system parameter that can improve performance but which needs to be set on a problem by problem basis.

Chapter 1

Introduction

This chapter introduces the themes and concepts that are referenced throughout the project. The chapter presents the project's research question, and describes the project's expected outputs and how they will contribute to the existing body of knowledge. Finally, the chapter outlines five objectives which form the project's plan of action.

1.1 Problem Overview

The focus of this project is on a technique for automatically generating computer programs, known as *genetic programming*. Genetic programming is an extension of *genetic algorithms* (Holland, 1975) which is itself a branch of the *evolutionary computation* research area (Whitley, 2001).

1.1.1 Evolutionary Computation

Evolutionary computation has several branches including genetic algorithms (Holland, 1975), evolution strategies (Beyer & Schwefel, 2002) and evolutionary programming (Fogel & Fogel, 1996). All the branches use processes inspired by biological evolution to search for solutions to problems. Potential solutions are known as *individuals* and their adequacy at solving a problem is quantified by a *fitness function*. When a fitness function is applied to an individual it returns a number that is the individual's *fitness value*, and which ranks the individual among its peers. The algorithms used within the area of evolutionary computation are known collectively as evolutionary algorithms (Back *et al.*, 1997).

The generic evolutionary algorithm begins with the random creation of a population of individuals and then that population is repeatedly evolved until a *termination criterion* is met. An example termination criterion could be that a sufficiently fit

solution is found or a fixed number of generations is reached (Whitley, 2001).

Following the biological analogy, each individual in an evolutionary algorithm's population is represented by its *genome* which is made up of *chromosomes* which, in turn, contain *genes* (Affenzeller *et al.*, 2009). However, evolutionary algorithms typically place all the genes of an individual in one chromosome and hence the terms individual, genome, and chromosome are sometimes used interchangeably.

Evolutionary algorithms search the space of all possible solutions using only fitness values as a guide. The fitness space — the space of the fitness values of solutions in the solution space — has a topography, known as the *fitness landscape*. The fitness landscape depends on the problem being tackled and, typically, contains both local and global maxima and minima. It is the job of an evolutionary algorithm to find a global optimum — a global maxima or minima, depending on how the fitness value is defined. The complexity of a problem's fitness landscape is an indicator of how difficult the problem is to solve (Back *et al.*, 1997).

The main advantage of the evolutionary computation approach is that the problem domain does not have to be fully understood for the technique to be applied (Koza, 1992). The fitness function needs to distinguish good solutions from bad but it does not have to know what good solutions look like. However, a disadvantage of the technique is that it is not always possible to know when an optimal solution has been found, as the maximum possible fitness value may not be known. Furthermore, it may not be known *a priori* how much time a system would require to find an optimal solution or whether a system is capable of finding an optimal solution.

1.1.2 Genetic Algorithms

In 1975, John Holland published his book "Adaptation in Natural and Artificial Systems" and introduced the *canonical genetic algorithm* and the field of genetic algorithms (Holland, 1975; Whitley, 1994). The canonical genetic algorithm (herein referred to as simply the genetic algorithm) is an evolutionary algorithm that uses a fixed length character string chromosome. To evolve its population of individuals, the genetic algorithm applies a *selection operator* and *genetic operators* to chromosomes. The selection operator determines which individuals are selected to survive to the next generation, and the genetic operators modify and recombine the genes of

the selected individuals to produce new individuals. The field of genetic algorithms is considered to encompass all algorithms that are derived from the canonical genetic algorithm.

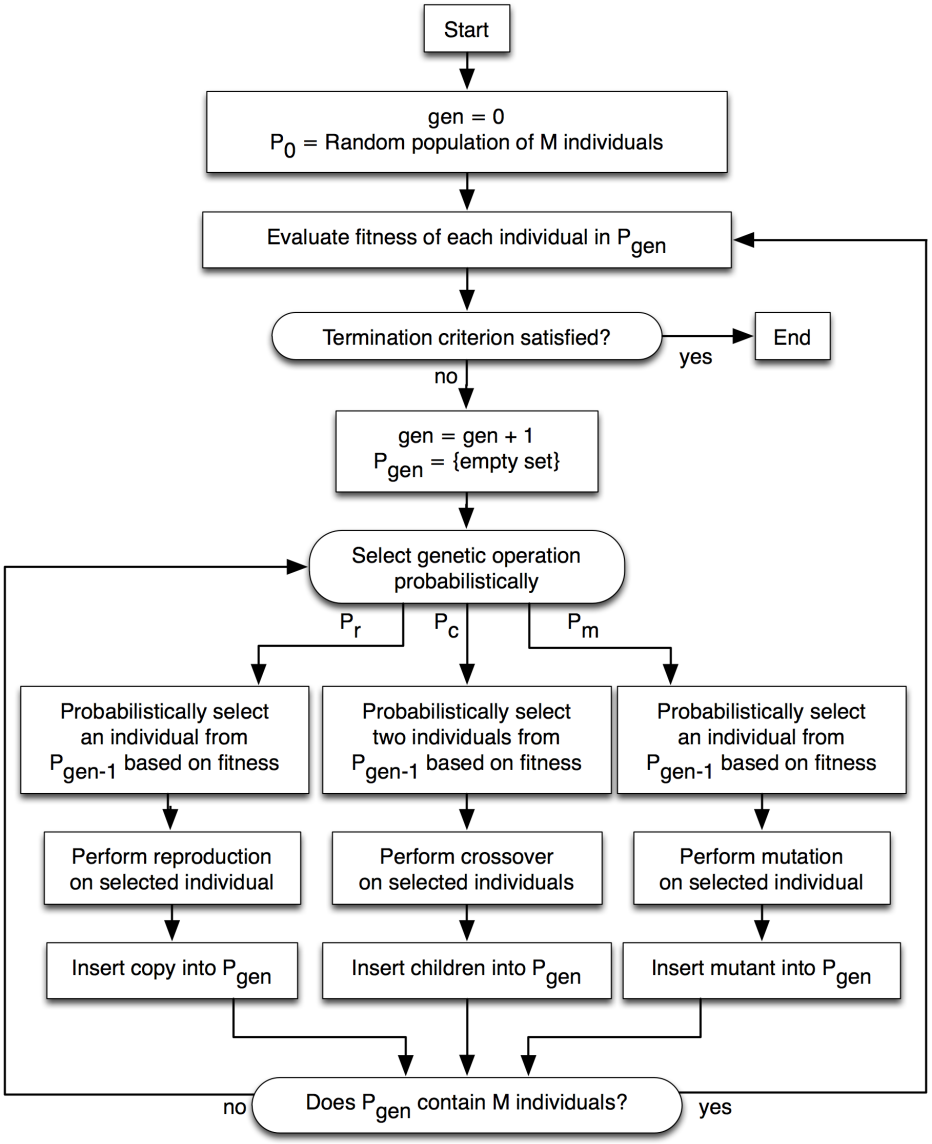


Figure 1.1: Flowchart of the genetic algorithm. Adapted from (Koza, 1992, p. 29).

The genetic algorithm is summarised in Figure 1.1. The algorithm is generational and creates a fixed-size population of individuals for each generation until the termination criterion is met (Whitley, 1994). The first generation is known as generation 0 and its population is filled with randomly created individuals. Then, for each subsequent generation, the generation's population is filled by repeatedly applying genetic operators to individuals selected from the previous generation's

population.

The genetic algorithm uses three genetic operators — *reproduction*, *crossover*, and *mutation* — and applies them probabilistically according to the pre-specified rates P_r, P_c, and P_m, respectively. The reproduction and mutation operators make copies of selected individuals, although the mutation operator goes one step further and randomly mutates copies. The crossover operator requires two individuals to be selected, known as *parents*, and creates two new *child* individuals. Each parent is split into parts which are recombined to create each child.

The effects of the genetic operators can be likened to processes found in nature. The reproduction operator allows fitter individuals to prevail in future populations; the crossover operator recombines parent individuals and sometimes produces a child that is fitter than both parents; and mutation allows the introduction of new genetic material and increases the variety of individuals. The three operators drive the search process in different ways. The crossover and reproduction operators *exploit* sampled parts of the fitness landscape to find and remember maxima. Whereas, the mutation operator *explores* unknown parts of the fitness landscape. As a result, the parameters P_r, P_c, and P_m configure the exploitative versus exploratory nature of a search process.

All genetic operators require individuals to be selected from the previous population and this process is known as an algorithm's *selection scheme*. Selection schemes follow the Darwinian principle of selection of the fittest, but also aim to maintain a level of population diversity.

Tournament selection is a scheme which selects the fittest individual of k randomly chosen individuals (Affenzeller *et al.*, 2009). The parameter k is known as the *tournament size* and controls the scheme's *selection pressure*. The selection pressure sets the scheme's bias for selecting the fittest individuals from a population. For example, if k is set to be the size of the population, then the fittest individual would always be chosen. Whereas, if k were set to 1, then fitness would not be a factor and individuals would be chosen randomly.

The selection pressure of a genetic algorithm controls how quickly the algorithm converges on a single solution (i.e., how quickly populations tend to contain only

similar individuals). If convergence is too quick, then a genetic algorithm will typically converge on a sub-optimal solution; it will have got caught in a local maxima or minima and will not have had the opportunity to escape. However, if convergence is too slow, then a genetic algorithm will tend to explore the fitness landscape *ad infinitum* and may never converge.

Over time researchers have devised a range of selection schemes and genetic operators; a survey of popular ones is given in (Poli *et al.*, 2008).

1.1.3 Genetic Programming

Genetic programming is an extension of the genetic algorithm technique. Genetic programming systems are the same as genetic algorithm systems except that individuals are represented by variable-length computer programs instead of fixed-length character strings (Affenzeller *et al.*, 2009). Genetic programming is an automatic programming technique that uses the heuristic search abilities of the genetic algorithm to find computer programs that solve a particular problem.

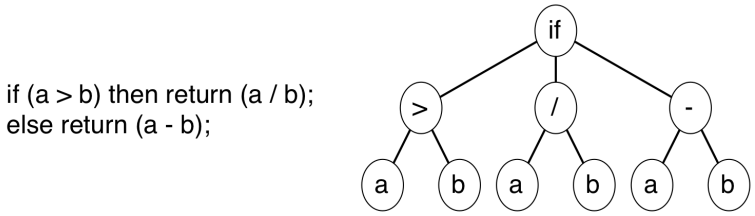


Figure 1.2: Two representations of the same program; a pseudo-code representation (left) and a tree representation (right).

The canonical genetic programming system presented by Koza (1992) uses tree structures to represent programs. An example program representation is given in Figure 1.2. The set of possible program tree nodes and the genetic operators must be designed to ensure that the application of an operator to a valid program never results in an invalid program — a program which disobeys the syntactic rules of its programming language and fails to execute from beginning to end. Consequently, program trees are restricted to take their leaf and non-leaf nodes from *terminal* and *function sets*, respectively. Terminal nodes must not require the presence of sub-trees, and function nodes must accept any number and kind of sub-tree.

Other representation schemes are also used in genetic programming. For example, *linear genetic programming* represents programs as linear sequences of as-

sembly code like instructions that access registers; *graph genetic programming* represents programs as graphs, of which trees are a sub-set; and Turing complete programming languages have also been used (Affenzeller *et al.*, 2009; Poli *et al.*, 2008; Spector, 2010).

1.1.4 Genetic Programming with the Push Language

Spector and Robinson (2002) present a stack-based Turing complete programming language, known as *Push*. Push was designed for representing programs in genetic programming systems. It features a uniform syntax and supports multiple data types, recursion, modularity and self-modification. The uniform syntax feature is important as it alleviates the need to curtail genetic operators to ensure they always produce valid programs. When using Push, almost any modification or recombination of a program will result in a valid program; the only syntactic restriction is that parenthesis be balanced.

1.1.5 Application of the Evolutionary Process to Itself

Even though evolutionary algorithm theory is problem independent, the implementation of an evolutionary algorithm is dependent on the problem. For example, the choice of representation used for individuals can affect the modality of the fitness landscape and make the search for a global optimum more difficult. Likewise, the setting of system parameters, such as the population size or the genetic operator rates P_r, P_c, and P_m, can determine the performance of an evolutionary algorithm since different problem domains have different optimal parameter settings (Eiben *et al.*, 1999; Nannen *et al.*, 2008). Back *et al.* give their opinion on parameter setting in the following quote taken from their 1997 review of evolutionary computation.

> *"In our opinion, evolutionary algorithms should not be considered as off-the-peg, ready-to-use algorithms but rather as a general concept which can be tailored to most of the real-world applications that often are beyond solution by means of traditional methods"* (Back *et al.*, 1997, p. 4).

In an attempt to automate the *tailoring* (parameter setting) required when implementing an evolutionary algorithm, some researchers have chosen to apply the evolutionary process not only to individuals but also to the algorithm's parameters.

The argument being that parameter setting is a complex task where parameters have interdependencies and trying all possible combinations is impractical. This is exactly the kind of optimisation problem on which evolutionary algorithms perform well, and consequentley it seems natural to apply evolution to parameter setting as well (Eiben *et al.*, 1999).

Two approaches exist for applying evolution to parameter setting: *meta-evolution* and *self-adaptation* (Kramer, 2010). In meta-evolution, a system is composed of multiple nested evolutionary algorithms. A 'meta' evolutionary algorithm evolves the parameters of a population of evolutionary algorithms that are tasked with problem solving. Whereas, a self-adaptive system is composed of a single evolutionary algorithm whose parameters are encoded within the chromosomes of individuals and, hence, are evolved along with the individuals.

Meta-evolution and self-adaptive systems are examples of systems that perform two searches in parallel. The first search is for a solution to the problem at hand and the second search is for parameters that tune the strategy of the first search (Eiben *et al.*, 1999; Kramer, 2010).

1.1.6 Autoconstructive Genetic Programming

Spector and Robinson (2002) introduced a self-adaptive approach called *autoconstructive evolution*. In autoconstructive evolution, instead of using pre-specified and static genetic operators, individuals are responsible for problem solving and for producing their own children. Effectively, each individual has its own genetic operator whose mechanism is evolved as the individual is evolved.

The Push programming language was designed for use in genetic programming in general, and for autoconstructive genetic programming in particular (Spector & Robinson, 2002). The fact that Push does not have syntax restrictions, like the character string chromosomes of genetic algorithms, permits the unrestricted evolution of genetic operators. A potentially unlimited variety of genetic operators could be evolved.

1.1.7 Reproductive Competence

To evaluate autoconstructive evolution, Spector and Robinson implemented two autoconstructive genetic programming systems: *Pushpop* and *AutoPush* (Spector &

Robinson, 2002; Spector, 2010). Both of these systems represent individuals as single Push programs that are responsible for problem solving and child production.

Spector and Robinson (2002) define the *reproductive competence* of an autoconstructive genetic programming system in terms of the ability of individuals to produce children. They consider a population to be reproductively competent when it produces a sufficient number of children so that no individuals need be randomly created to fill the next population.

When running Pushpop, Spector and Robinson found that the system first had to achieve reproductive competence and only then did selection begin to drive the search process.

1.2 Research Question

Having introduced autoconstructive genetic programming and reproductive competence, we can now present this project's research question. The question is:

> *In an autoconstructive genetic programming system, does the expressive power of the language of child producing programs affect the reproductive competence of individuals or the problem solving ability of the system?*

The hypothesis is that the expressive power of the language of child producing programs determines the space of all possible child producing programs, be they reproductively competent or not. Therefore, it should be possible to adjust the expressive power to increase the proportion of competent programs in the search space and ease a system's task of evolving competent child producing programs.

Implicit in the hypothesis is the issue of whether the space of competent child producing programs depends on the problem a system is trying to solve. If the space is *not* dependent on the problem then the expressive power of child producing programs could be defined and set statically for all systems. Alternatively, if the opposite is true then the expressive power of child producing programs would have to be considered as a new system parameter which would require setting for different problems.

This project will assume that the space of competent child producing programs is *not* dependent on the problem. However, there will be an opportunity in the final

analysis phase of the project to determine whether the collected data suggest the opposite might be true.

The investigation of the research question will contribute to the body of knowledge on the behaviour of autoconstructive evolution, and may suggest improvements for future autoconstructive genetic programming systems. Autoconstructive evolution is of value because it addresses some of the genetic programming open issues described by O'Neill *et al.* (2010); in particular, the complexity of system setup and the goal of producing an autonomous "one button" system.

An autoconstructive genetic programming system uses evolution to configure itself for a given problem. It reduces the complexity of system setup and leads to systems that are simpler to use than standard genetic programming systems. Also, autoconstructive systems have the potential, albeit in the distant future, to outperform hand configured genetic programming systems.

1.3 Contribution to Knowledge

It is expected that this project will be of general interest to researchers working in the field of genetic programming, and will be of particular interest to autoconstructive genetic programming researchers.

In order to answer the research question, the project will have to address the issue of whether the separation of an individual's single problem solving and child producing program into two separate programs has any side effects. The complete list of areas that this project will research is:

- the separation of the problem solving and child production concerns of an individual; and
- the modification of the expressive power of the language of child producing programs.

To my knowledge, neither of these areas has been researched directly, although there are works in the literature that are related to them. Hinterding (1997) and Cavill *et al.* (2005) investigated using multiple chromosomes to represent individuals in genetic algorithms and genetic programming, respectively. Also, Rahim *et al.* (2006) investigated how the movement of food sources in an artificial life environment affected the reproductive competence of agents that were evolved autoconstructively.

It is expected that the output of this project will be:

- numerical evidence that demonstrates if the reproductive competence of individuals or the overall problem solving performance of an autoconstructive genetic programming system differs when individuals are represented (1) as a single program (for problem solving and child production), and (2) as two programs (one for problem solving and the other for child production);
- numerical evidence that demonstrates if the reproductive competence of individuals or the overall problem solving performance of an autoconstructive genetic programming system differs when the child producing programs of individuals are (1) expressed in the Push language, and (2) expressed in a version of the Push language with restricted expressive power; and
- a Common Lisp implementation of an autoconstructive genetic programming system that can be used for non-commercial educational and research purposes, and which makes use of the Common Lisp implementation of the Push language interpreter.

1.4 Objectives

The previous sections described the research question and its contribution to current knowledge. This section introduces a set of project objectives which are designed to allow the research question to be answered. In all, there are five objectives and they form the project's plan of action.

1.4.1 Literature Review

The first objective is to review past and present academic literature that is related to the themes of this project. The purposes of this review are twofold. The first is to gain an overall appreciation of evolutionary computation research. The second is to focus on two areas: (1) the application of an evolutionary process to itself or another evolutionary process, within genetic algorithms and genetic programming; and (2) the identification of problems tackled by genetic programming which are candidates for use in this project's experiments.

1.4.2 Experiment 1: Establish Baseline

The second objective is to design, implement and execute an experiment which will provide baseline data. An autoconstructive genetic programming system will be implemented which will execute in one of several modes. In the first mode

individuals will be represented by one Push program, whereas in the second mode they will be represented by two. The purpose of this objective is to determine the baseline behaviour of the system in the first mode, and to detect any changes caused by running the system in the second mode.

A secondary purpose of this objective is to collect data on the program instructions most frequently used by reproductively competent individuals. This data will be required by the next objective.

1.4.3 Experiment 2: Reduce Expressive Power of Child Producing Programs

The third objective concerns the crux of the research question. In this objective, experiment 1's autoconstructive genetic programming system will be reused to investigate any changes in behaviour caused by reducing the expressive power — the set of enabled Push instructions — of child producing programs. The Push instructions to be disabled will be determined by analysis of experiment 1's data.

1.4.4 Experiment 3: Generalise Results

The fourth objective is to generalise on the results produced by experiment 2. In this objective, experiment 2 will be repeated, albeit with a different problem.

1.4.5 Analyse Results, Draw Conclusions and Make Recommendations for Future Research

Once the three experiments have been carried out, the fifth objective will be to analyse the data generated by the experiments. Conclusions should be drawn from the analysis and should lead to the making of recommendations for future autoconstructive genetic programming research.

1.5 Summary

This chapter provided an introduction to the project and gave an overview of the concepts which are central to the project: *autoconstructive genetic programming*, the *Push programming language* and *reproductive competence*.

The chapter presented the project's research question, which is, in brief, *whether the expressive power of the language of child producing programs affects an auto-*

constructive genetic programming system. The chapter then discussed the expected outputs of the project and their contribution to the body of knowledge on autoconstructive genetic programming.

Finally, the chapter outlined five objectives which form the project's plan of action. They are (1) to carry out a literature review; (2, 3 and 4) to execute three experiments; and (5) to analyse the generated data to draw conclusions concerning the research question.

Chapter 2

Literature Review

This chapter provides a description of the concepts central to this project. It describes the motivation of related research areas, and introduces some technical terminology. The chapter closes by describing works of research that lead into this project.

2.1 Introduction

Different evolutionary algorithms implement different search strategies. The design of an algorithm determines its overall search strategy, but the strategy can be tuned by setting the algorithm's control parameters. Even though parameters only tune a search strategy, researchers have found that they can strongly influence algorithm performance, and that different settings are appropriate for different classes of problems (De Jong, 2007; Eiben *et al.*, 1999).

Typically, researchers choose parameter values during preliminary algorithm runs and then leave the chosen values constant during the final runs (Eiben *et al.*, 1999). Parameters are set based on rules of thumb recommended by the literature (for example, (De Jong, 1975)), and on the performance of preliminary runs. Optimal parameter setting is a difficult task. Parameters often have complex interactions and an exhaustive search of all possible parameter values is impractical due to the large number of combinations.

In addition to the fact that parameter values affect algorithm performance, other factors have also led to the investigation of automated parameter setting. Firstly, manual parameter setting is a laborious and time consuming task. This is especially true in genetic programming where the computation of fitness values can require a significant amount of processing time (Eiben *et al.*, 1999). Secondly, an algorithm

that requires specialist tuning is less likely to be used as a problem solving tool by researchers in areas outside of evolutionary computation (O'Neill *et al.*, 2010). Thirdly, some researchers see automated parameter setting as a means of improving the performance of an evolutionary algorithm without sacrificing its generality (Edmonds, 1998) — its ability to solve problems which are like, but not the same as, the problem for which it was designed. Finally, researchers question whether evolutionary algorithms have different optimal parameter values for different stages of a run (Eiben *et al.*, 1999).

Meta-evolution and *self-adaptation* are two methods which have been developed to automate parameter setting in evolutionary algorithms. They both use an evolutionary process to evolve initial parameter values into values specific to the problem being tackled.

2.2 Terminology

In a meta-evolution system, evolutionary algorithms are nested within one another. A *meta-level* algorithm evolves the parameters of one or more *task-level* algorithms tasked with problem solving (Kramer, 2010). Furthermore, meta-evolution systems can implement multiple levels of nesting. The *isolation time* of a meta-evolution system describes the amount of time that a task-level algorithm is given to evolve its population for each generation of the meta-level algorithm. A high isolation time allows a task-level algorithm to evolve its population for many generations, or even several runs of many generations, for each generation of the meta-level algorithm. A low isolation time only allows a task-level algorithm time for a few generations of evolution for each meta-level generation.

Angeline (1995) defined self-adaptation as *"when an evolutionary computation evolves the values for its ... parameters"*, but this definition is somewhat general and could, conceivably, include meta-evolution systems. Indeed, many authors of evolutionary computation works use self-adaptation as a general term to describe any system that adapts aspects of itself (for example, Back *et al.* (1997), Clune *et al.* (2005) and Kim *et al.* (2011)). This project will use the more specific definition given by Eiben *et al.* (1999). Eiben *et al.* extend Angeline's definition of self-adaptation and specify that the parameters to be adapted must be encoded within the genomes

of individuals and, hence, are evolved as individuals evolve.

Alternative schemes are *deterministic* and *adaptive* (Eiben *et al.*, 1999). Instead of evolving parameter values, these schemes use static rules to modify parameters as an evolutionary algorithm runs. Deterministic and adaptive rules are distinguished by the fact that adaptive rules make use of the current state of the algorithm and its population to update parameters, whereas deterministic rules do not. Deterministic rules use, at most, the current generation number and are deterministic because they are not influenced by individuals in the population.

The parameters of an evolutionary algorithm include obvious items like population size and genetic operator rates. However, they can also include aspects such as the design of genetic operators or fitness functions (Eiben *et al.*, 1999; Tavares *et al.*, 2004).

2.2.1 Levels of Influence

In addition to specifying the scheme of automated parameter setting, it can be useful to also specify its level of influence. Three levels are defined: *population-level*, *individual-level*, and *component-level* (Angeline, 1995). Schemes working at these three levels influence parameters that are applied to the population as a whole, to individuals individually and to parts of the genome of individuals, respectively.

The nature of meta-evolution generally restricts it from working at anything but the population-level. Self-adaptation can be made to work at any level. Consider a self-adaptive scheme that encodes a mutation rate within the genome of each individual. The scheme would be working at the individual-level if, each time an individual were to be mutated, the mutation rate was taken from the individual's chromosome. However, if the scheme first calculated the average of all the individual mutation rates and used the average rate to mutate individuals then the scheme would be operating at the population-level.

2.3 Overheads of Meta-Evolution and Self-Adaptation

Compared to systems that do not employ automated parameter setting, meta-evolution and self-adaptive systems incur a computational overhead. In meta-evolution systems one or more additional evolutionary algorithms are executed, and

in self-adaptive systems extra computation is required by the evolution of extra genetic material. Meta-evolution systems multiply the existing computation cost for each new level, and self-adaptation systems add a constant cost for each individual in the population.

Any overhead incurred by automated parameter setting should be considered in the context of the benefits afforded. Potential benefits could be that an algorithm is able to find a solution quicker, that better quality solutions are found, or that the algorithm is able to solve a wider range of problems — an increase in generality.

In practice, automated parameter setting can be used in one of two ways. Firstly, it can be used as a laboratory tool to discover good parameter settings for an evolutionary system tasked with a specific problem. Afterwards, the discovered parameters can be set statically in a system which is deployed to solve the same problem time and again. Or secondly, a system could be deployed with automated parameter setting inbuilt to solve different problems, or to solve a problem whose fitness landscape varies with time.

It is the view of De Jong (2007) that the benefits of automated parameter setting will only outweigh the costs when used on problems with dynamic fitness landscapes. However, it is not clear whether he also discounts the case for using automated parameter setting as a laboratory tool for finding optimal parameter values for different classes of problems.

2.4 Related Works

The focus of this project is on autoconstructive genetic programming. In autoconstructive genetic programming, individuals are represented using a Turing complete programming language, and are evolved to solve the problem being addressed and to produce their own children. Autoconstructive genetic programming uses individual-level self-adaptation to evolve a reproduction operator for each individual.

This section reviews literature that deals with genetic algorithm (GA) or genetic programming (GP) systems that employ meta-evolution or self-adaptation. These areas have been chosen for multiple reasons. Firstly, the focus of this project is on a form of GP (autoconstructive GP) that uses self-adaptation. Secondly, self-adaptation and meta-evolution are closely related in that they both apply evolution

to the parameters of an evolutionary process; and finally, GP is an extension of the GA technique and, hence, their systems resemble one another and, typically, results in one area have validity in the other.

2.4.1 Meta-Evolution and Genetic Algorithms

One of the first studies involving meta-evolution was carried out by Grefenstette (1986). Grefenstette intended to discover parameter settings with a performance comparable to parameters recommended by the literature. A meta-level GA evolved the parameters of a population of task-level GAs that were set to solve five real-number function optimisation problems. Six parameters were evolved: population size, crossover rate, mutation rate, generation gap, scaling window size, and whether the selection scheme was elitist. The meta-level fitness function measured the performance of a task-level GA at solving the set problems.

The meta-level GA was configured to evolve a population of 50 task-level GAs for 20 generations in a single run. Each task-level GA evolved its population for the equivalent of evolving 100 individuals for 50 generations for each of the five problems. Furthermore, the task-level populations were re-initialised at the start of each meta-level generation (Clune *et al.*, 2005). The experiment consisted of running the meta-level only once, presumably due to the limited power of the computing equipment available at the time.

The best parameters evolved by the experiment were compared with a set of 'standard' parameter values, established by previous literature, using a GA tasked with image processing. Grefenstette found that his evolved parameters performed slightly better than the 'standard' parameters. He considered his experiment a success because it did evolve parameters that were distinct to, but which performed as well as, 'standard' parameters. However, he commented that the work was limited as it did not allow the meta-level GA to evolve the choice of recombination genetic operator used by task-level GAs.

Clune *et al.* (2005) built upon the work of Grefenstette (1986). They implemented experiments similar to Grefenstette's, but allowed the meta-level GA to also evolve the choice of selection scheme and crossover operator used by task-level GAs. In sum, the evolved parameters were crossover type and rate, mutation rate, the selection

scheme and the selection scheme's parameters. The choice of selection scheme was between *tournament*, *roulette* and *stochastic remainder sampling*. The choice of crossover operator was between *one-point*, *two-point* and *uniform*. The problems with which the task-level GAs were tasked were two boolean function optimisation problems: the 1000 bit *max-ones* problem, and the 1000 bit *4-bit deceptive trap* problem.

Notably, Clune *et al.*'s system differed from Grefenstette's in two major ways. Clune *et al.*'s system did not evolve the population size of task-level GAs, and their system did not reset the populations of task-level GAs for each meta-level generation. Clune *et al.*'s system had a low isolation time. Each task-level GA executed for only one generation per meta-level generation, and as a result their meta-level GA and task-level GAs all but ran in parallel. Clune *et al.*'s fitness function measured the *current* performance of a task-level GA at solving the set problem; Grefenstette's function measured *definitive* performance. Also, Clune *et al.*'s system only tackled one problem at a time, whereas Grefenstette's system tackled five problems at once.

Clune *et al.*'s experiments consisted of three stages. The first stage involved using their meta-evolution system to evolve *specialist* parameters for the two set problems. The second stage involved comparing their system with standard GA systems configured with the specialist parameters. The third stage involved running their system on a 3000 bit version of the 4-bit deceptive trap to see if it consistently chose different parameter values at different phases of the run.

In the first two stages, the meta-level GA ran 25 times for each of the two problems. In each run, the meta-level GA evolved a population of 36 task-level GAs for 100 generations. The best task-level individuals were allowed to migrate between task-level populations at the end of each meta-level generation. For each meta-level generation, each task-level GA evolved its population of 100 individuals for a single generation. In the third stage, Clune *et al.* allowed their meta-level GA to continue to evolve its population for 500 generations to achieve a better impression of the different phases of a run.

During the second stage experiments, Clune *et al.* found that the specialist GA found higher quality solutions than the meta-evolution system, and that the meta-

evolution system found higher quality solutions than the GA which was a specialist for the other problem. Across the two problems, the meta-evolution system found, on average, better quality solutions than the specialist GAs.

Clune *et al.*'s analysis of their third stage experiments concluded that the meta-evolution system did consistently choose different parameter values at different phases of a run, but they found that the system's choices did not seem to be beneficial in the long-term. They found that a standard GA system, configured with parameters chosen by the meta-evolution system early on in its run, discovered better quality solutions than the meta-evolution system.

Clune *et al.*'s work was limited to two somewhat similar boolean function optimisation tasks. Nevertheless, their work, taken together with Grefenstette's, does suggest that meta-evolution systems might be useful as general purpose solvers. Clune *et al.*'s work demonstrates that a meta-evolution system will consistently make different choices about parameter settings during a single run, although it is not clear how those choices affect the system's performance or generality.

Dioşan and Oltean (2009) used meta-evolution to evolve sequences of genetic operators for use in the cores of novel evolutionary algorithms. Their system was potentially capable of evolving sequences of operators that match established evolutionary algorithms, such as the genetic algorithm or those employed in evolutionary strategies or evolutionary programming.

In Dioşan and Oltean's system, a meta-level GA evolved individuals that specified sequences (quantity and order) of crossover and mutation operators which were used by task-level evolutionary algorithms (EAs) to evolve their populations of problem solvers.

Dioşan and Oltean tested their meta-evolution system on ten real-number function optimisation problems of varying difficulties. Two blocks of experiments were undertaken. In the first block, the task-level EAs represented individuals using 10 genes where each gene encoded a real-number. In the second block, the EAs represented individuals using 300 genes where each gene encoded a binary digit and consecutive sequences of 30 genes were read as a real-numbers.

Dioşan and Oltean chose not to evolve the crossover and mutation operator rates,

but instead opted to *parameter sweep* them. That is, they chose to run each task-level EA for each of 100 pairs of operator rates — the two rates were varied from 0.1 (10%) to 1.0 (100%) in steps of 0.1. They justified their decision by saying that they had previously tried to evolve the operator rates, but found that the system was unable to find global-optimal values.

Dioşan and Oltean placed a lot of importance on the selection of the operator rates. They argued that *"the structure of an evolved EA could be very good, but due to the inappropriate values for [crossover and mutation rates] we could get some very poor results"*. However, it is not clear whether the opposite may have been true in their experiments: that poor designs for task-level EAs were being compensated for by optimal operator rates.

Dioşan and Oltean configured their meta-level GA to evolve 100 task-level EAs for 100 generations. Isolation time was very high. For each meta-level generation, each task-level EA was run 100 times for each of the 100 parameter swept operator rate pairs. Each task-level run entailed evolving a population of 50 for 50 generations. Dioşan and Oltean reported that their meta-evolution system required a week of runtime for each problem, and that without operator rate parameter sweeping the runtime would have been a few hours per problem.

Dioşan and Oltean compared the results of their meta-evolution system with the results of several standard EAs: a steady-state GA, a generational GA and an evolutionary programming scheme. They found that the steady-state GA always outperformed the other standard systems, and that the meta-evolution system was comparable to, and sometimes beat, the steady-state GA. In addition, Dioşan and Oltean tested the generality of the evolved EAs by testing the best for each problem on the other 9 problems. In this case, they found that the evolved EAs continued to perform on a par with the steady-state GA for most problems. However, it is interesting to note that all 10 evolved EAs performed best on a problem which was not the problem for which the EA had been evolved to solve.

2.4.2 Meta-Evolution and Genetic Programming

Grefenstette (1986) and Clune *et al.* (2005) used meta-evolution to evolve scalar parameters for GAs. Dioşan and Oltean (2009) used meta-evolution to evolve evol-

utionary algorithms. Edmonds (1998), which will be discussed next, used meta-evolution to evolve the mechanisms of genetic operators.

Edmonds (1998) implemented a meta-evolution system that consisted of two GP systems. The meta-level GP system was set to evolve genetic operators which were used by a single task-level GP system to evolve problem solvers.

Initial trials of Edmonds' meta-evolution system failed because population diversity was destroyed too quickly. Edmonds discovered that the genetic operators being evolved were sensitive to the composition of the instruction set from which the operators were built. He found that he could help maintain population diversity by disabling some of the instructions from which the operators were built, and by deleting those operators which produced children that were the same as one of their parents. In addition, Edmonds discovered that the meta-level fitness function was also crucial for maintaing population diversity. He set the meta-level fitness function to measure how well the evolved genetic operators performed at increasing the fitness of the task-level population.

Having addressed the population diversity problem, Edmonds tasked his system with solving the *odd 4-parity* problem. The meta-level GP system was run 6 times to evolve a population of 200 genetic operators for 36 generations. For each meta-level generation, the 200 evolved operators were applied to the task-level GP system's population of 600. The task-level population was not reset for each meta-level generation and, hence, the system exhibited parallel running behaviour similar to Clune *et al.*'s (2005) system.

Edmonds compared his meta-evolution system with a standard GP system set to solve the same odd 4-parity problem. The meta-evolution and standard systems found perfect solutions after 36 and 93 generations, respectively. Edmonds calculated that, compared to a standard system, his meta-evolution system increased the computational cost by a factor of 1.33. However, the meta-evolution system found a perfect solution with a lower computational cost than was required by the standard system.

Kantschik *et al.* (1999) continued the work of Edmonds (1998) and researched meta-evolution systems that evolve genetic operators and which contain multiple levels of

nesting. Kantschik *et al.* define four GP system architectures: *task-random*, *smart-random*, *smart-self* and *meta-self* (Figure 2.1). Task-random is the architecture of a standard non-meta-evolution GP system; static genetic operators evolve a population of problem solvers. Smart-random is the same as the meta-evolution system implemented by Edmonds (1998); static operators evolve a population of operators that, in turn, evolves a population of problem solvers. Smart-self uses recursion; a population of operators evolves itself and a population of problem solvers. Meta-self adds an extra layer to the smart-self architecture. It has a population of genetic operators that evolves itself and another population of genetic operators which, in turn, evolves a population of problem solvers.

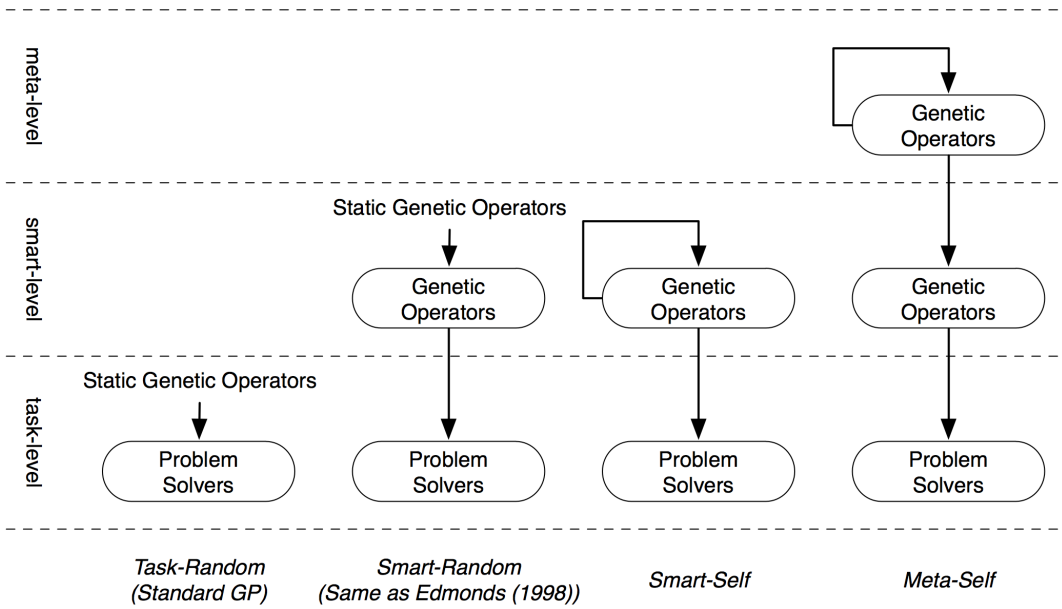


Figure 2.1: The four architectures defined by Kantschik *et al.* From left to right: task-random, smart-random, smart-self, and meta-self. Adapted from (Kantschik *et al.*, 1999, p. 18).

In Kantschik *et al.*'s terminology, what we have defined as the meta-level is renamed the smart-level, and the meta-level name is given to the level above the smart-level (Figure 2.1).

In their experiments, Kantschik *et al.* applied their four architectures to solving a *speaker identification* problem. Problem solving individuals were evolved to identify who, of a group of four speakers, spoke during a given sound sample. The task-level fitness function reflected the performance of the problem solving individuals at solving the given problem. Whereas, the smart-level and meta-level fitness

functions measured how well a genetic operator individual improved the fitness of the population being evolved — a population of problem-solvers in the smart-level case, and a population of operators in the meta-level case.

Kantschik *et al.* configured the population sizes to be 400 for the task-level, 100 for the smart-level, and 50 for the meta-level. Kantschik *et al.* do not detail the isolation time that they used, although as their work seems to extend that of Edmonds (1998) it is assumed that the isolation time was low. It is assumed that for each higher-level generation, a lower-level was executed for one generation. Kantschik *et al.* averaged their results over 30 runs of 50 generations of evolution.

Kantschik *et al.*'s results showed that their meta-self system consistently found solutions of high quality than their other systems. The ranking was meta-self, smart-random, task-random and then smart-self. Interestingly, the smart-self system performed worse than the simpler task-random (standard GP) system. Kantschik *et al.* concluded that it is possible to build a meta-evolution GP system that does not use static genetic operators, but that to do so an extra level of nesting is required (as compared to the 'standard' meta-evolution GP system of Edmonds (1998)).

Tavares *et al.* (2004) also implemented a system similar to Edmonds' (1998) meta-evolution GP system, although instead of setting their system to evolve genetic operators they set it to evolve genotype to phenotype mapping functions.

The concept of genotype to phenotype mapping is one that has been taken from biological evolution. A human's genotype is considered to be all the hereditary information that is encoded in his or her genome, whereas a human's phenotype is the description of his or her body. A human's genome may carry genes that code for blue and green eyes, but it is the genotype to phenotype mapping which decides what colour eyes the body has (King *et al.*, 2007). Many GA and GP systems do not distinguish between the genotype and phenotype, but, for those that do, it is an individual's genotype which is evolved by genetic operators and it is their phenotype which is fitness tested (Luke, 2009).

Tavares *et al.*'s system consisted of a meta-level GP system which evolved genotype to phenotype mapping functions for a population of GA systems that were tasked with solving a real-number function optimisation problem. Unlike the sys-

tems of Edmonds (1998) and Kantschik *et al.* (1999), Tavares *et al.*'s meta-level fitness function measured the performance of the task-level GA at solving the problem.

Tavares *et al.*'s experiment consisted of running their meta-level GP system for 30 runs. In each run, the meta-level evolved a population of 500 mapping functions for 500 generations. For each meta-level generation, the 500 mapping functions were used by 500 task-level GAs that were run 30 times. Each task-level run consisted of evolving a population of 100 problem solvers for 50 generations. The isolation time was high, and it is assumed that the task-level populations were reset for each task-level run.

Tavares *et al.* compared the results of the best evolved mapping function found by their meta-evolution system with an *identity* mapping function, that treated the genotype and phenotype as being identical, and a hand-designed *optimal* mapping function. They found that their evolved mapping function consistently found solutions of higher fitness than the identity function, and that the evolved function's performance approached that of the optimal function.

2.4.3 Self-Adaptation

It seems that, in the areas of GAs and GP, many more works have been published on meta-evolution than have been published on self-adaptation. For example, in two recent surveys of self-adaptation in evolutionary algorithms (Kramer, 2010; Meyer-Nieberg & Beyer, 2007) the majority of works described are in the evolutionary strategies and evolutionary programming areas. The surveys describe a couple of GA works but no GP works. Consequently, the following sub-sections describe only three works.

2.4.4 Self-Adaptation and Genetic Algorithms

Hinterding (1997) implemented an individual-level self-adaptation GA system that was tasked with solving a non-numeric problem, namely the *cutting stock* problem. Two chromosomes were used to represent each individual in the population. One chromosome encoded the individual's solution to the problem and the other encoded two self-adapted parameters. Each of the two chromosomes was evolved by a different pair of crossover and mutation operators. The self-adapted parameters

controlled, on an individual level, the operation of the mutation operator applied to the problem solving chromosome.

Hinterding's system evolved a population of 100 problem solvers for a varying amount of time depending on the difficulty of the problem. For the easiest problem, the system was run for the equivalent of 22 generations, and for the most difficult problem the system was run for the equivalent of 106 generations. Hinterding ran his system 20 times for each problem.

Like the canonical GA, the fitness function of Hinterding's system measured the performance of an individual at solving the set problem. Furthermore, to maintain population diversity the system discarded newly created individuals that were exact copies of individuals already in the population.

Hinterding tested his self-adaptation system on increasingly difficult variants of the cutting stock problem, and compared the results with the results of a fixed parameter GA system. The meta-evolution system found higher quality solutions than the fixed parameter system for seven of the ten problems, including the five most difficult problems. Hinterding concluded that self-adaptation incurs a computational cost but that the cost is outweighed by the better results found for the harder problems; that self-adaptation can help solve non-numeric problems; and that multi-chromosomes are useful for representing the different concerns of an individual within an evolutionary algorithm.

2.4.5 Self-Adaptation and Genetic Programming

Spector and Robinson (2002) researched using individual-level self-adaptation in a GP system which evolved genetic operators and problem solvers. They defined a scheme, known as *autoconstructive evolution*, in which the individuals of a population are not only responsible for solving the set problem but are also responsible for producing their own children. Spector and Robinson were motivated by autoconstructive evolution's potential to be useful in GP automated parameter setting research, as well as its potential to inform the *artificial life* field of research.

In order to implement an autoconstructive GP system, Spector and Robinson developed a stack-based Turing complete language, known as *Push*, and used that language to represent individuals in their GP system.

Push is further discussed in Sections 1.1.4 and 4.1, and has its own Internet home page: `http://hampshire.edu/lspector/push.html`.

Spector and Robinson state that their experiments were preliminary, but do describe aspects of their system. They state that they successfully applied their autoconstructive GP system, known as *Pushpop* (Spector & Robinson, 2002), to a range of symbolic regression problems. Like the work of Edmonds (1998) which also evolved genetic operators, Spector and Robinson found that they had to implement features in Pushpop that helped maintain population diversity. Specifically, they enforced that the population must not contain identical individuals. However, they found that sometimes population diversity would be destroyed by individuals which were identical but for a piece of program that was never executed.

In a later work, Spector (2010) describes a successor to Pushpop called *AutoPush*. Like Pushpop, AutoPush is an autoconstructive GP system that uses the Push programming language to represent individuals. However, AutoPush contains more advanced mechanisms for maintaining population diversity. The fitness function of AutoPush takes into account the problem solving performance of an individual and the history of relative fitness changes in that individual's ancestral lineage. Furthermore, Spector improved on Pushpop's boolean function for testing whether two individuals were identical, and substituted it for a function which measured how similar two individuals were. Using this similarity function, AutoPush discarded individuals that were too alike.

AutoPush is described further in Section 4.2.

2.5 Summary

Meta-evolution and self-adaptation are two techniques for implementing automated parameter setting in evolutionary algorithms. Meta-evolution and self-adaptation differ from other automated parameter setting methods in that they both use evolution to evolve initial parameter values into optimal ones. Meta-evolution involves using one evolutionary algorithm to evolve the parameters of another. Whereas, in self-adaptation parameters are encoded within the genomes of individuals and are evolved together with individuals.

The use of meta-evolution and self-adaptation incurs a computational overhead.

The overhead of meta-evolution is typically greater than that of self-adaptation. Some researchers (for example, De Jong (2007)) are of the opinion that automated parameter setting will only provide a net benefit when applied to problems with dynamic fitness landscapes.

Both meta-evolution and self-adaptation have been used in genetic algorithm and genetic programming research. However, more works have been published on meta-evolution than have been published on self-adaptation. Self-adaptation seems to be used more in evolutionary strategies and evolutionary programming research than in genetic algorithm or genetic programming research.

Typically the parameters of an evolutionary algorithm are considered to be scalar values, however some researchers have investigated using automated parameter setting to define the mechanisms of components such as the genetic operators. Edmonds (1998), Kantschik *et al.* (1999), Tavares *et al.* (2004) and Spector and Robinson (2002) investigated evolving the mechanisms of genetic operators and genotype to phenotype mapping functions in genetic programming. In particular, Edmonds discovered that maintaining population diversity was problematic when evolving genetic operators. He found that the problem was alleviated, although not solved, by modifying fitness functions to take into consideration the performance of a genetic operator's ability to improve fitness and by not allowing duplicate individuals in the population. Although Edmonds' work dealt with meta-evolution, Spector and Robinson re-confirmed Edmonds' findings on population diversity but with a self-adaptive genetic programming system.

Chapter 3

Research Methods

The research method for this project is to carry out a set of controlled laboratory experiments in the form of computer simulations. The experiments are designed to produce numerical data which can be analysed to shed light on the research question.

The use of computer simulations is ubiquitous in genetic programming research (for example, Koza (1992), Edmonds (1998), Kantschik *et al.* (1999), Spector and Robinson (2002), Tavares *et al.* (2004), and Dioşan and Oltean (2009)). Computer simulations are worthwhile because they are relatively cheap to implement and execute, they allow system variables to be controlled and recorded accurately, and they permit experiments to be reliably reproduced.

3.1 Experiments

The project's experimentation phase consists of three experiments. All the experiments involve executing essentially the same autoconstructive genetic programming system albeit with modifications for each experiment. The first experiment investigates the effects of representing individuals as two programs, one for problem solving and another for child production, as opposed to representing individuals with a single combined program. The second experiment investigates the effects of reducing the expressive power of child producing programs, and the third experiment attempts to generalise on the second experiment's results by repeating that experiment except with a different problem to solve.

The autoconstructive genetic programming system used in the experiments supports three execution modes. The first mode is like the Pushpop and AutoPush systems (Spector & Robinson, 2002; Spector, 2010) in that it evolves a population of individuals where each individual is represented by a program, expressed in the

Push programming language, which is responsible for both problem solving and child production. The second mode is the same as the first mode, except that individuals are represented by two Push programs; one program for problem solving and another for child production. The third mode is the same as the second mode except that child producing programs are expressed in a version of Push that has some instructions disabled — a version of Push with restricted expressive power. Table 3.1 summarises the execution modes.

Execution Mode	**Programs per Individual**	**Program Responsibility**	**Program Language**
1	1	Problem Solving and Child Production	Push
2	2	Problem Solving	Push
		Child Production	Push
3	2	Problem Solving	Push
		Child Production	Restricted version of Push

Table 3.1: Execution modes of the project's autoconstructive genetic programming system.

The first experiment compares the performance of the system in execution mode 1 with that of execution mode 2. The second experiment compares execution mode 2 with execution mode 3. The third experiment also compares modes 2 and 3, but for a different problem. Table 3.2 summarises the experiments. The two problems are described in the next section.

Experiment 1:	Compare execution modes 1 and 2 on problem 1
Experiment 2:	Compare execution modes 2 and 3 on problem 1
Experiment 3:	Compare execution modes 2 and 3 on problem 2

Table 3.2: Summaries of the project's three experiments.

3.2 Problems 1 and 2

As discussed in the previous section, experiments 1 and 2 will be set to solve one problem (*problem 1*), whereas experiment 3 will be set to solve another (*problem 2*). The two problems have been chosen to be *symbolic regression* (Koza, 1992)

problems. Symbolic regression tasks a genetic programming system with evolving a program that implements a function that fits a given sample of data points. That is, for a predefined set of input values, an evolved program should endeavour to output a predefined set of output values. For this project, problem 1's target function is $x^2 + x$ and problem 2's is *even 4-parity*. The even 4-parity function takes four boolean values as inputs, and outputs a 1 if the number of inputs with a value of 1 is even. Figure 3.1 presents plots of inputs versus outputs for the chosen functions.

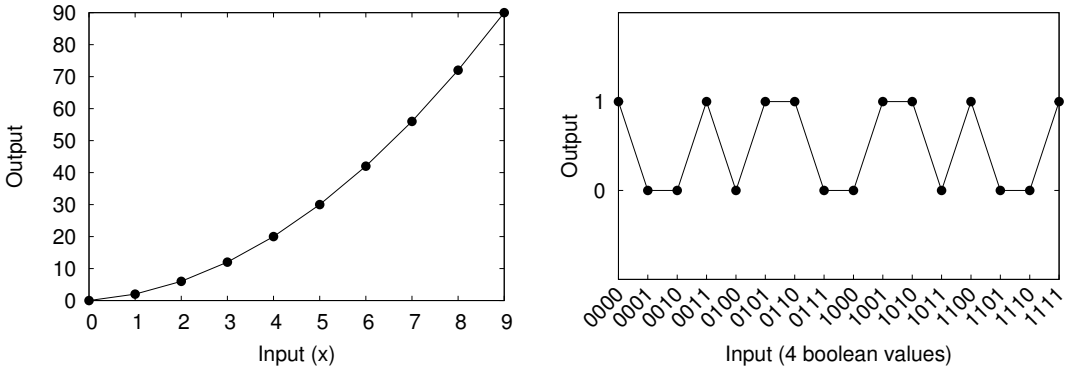


Figure 3.1: Plots of input versus output for the two chosen symbolic regression problems: $x^2 + x$ (problem 1, left) and even 4-parity (problem 2, right).

Both problems have ten fitness cases — input values used to assess the fitness of an individual. For problem 1, the fitness cases correspond to the values 0 to 9. For problem 2, the ten fitness cases have been chosen to span the range of sixteen boolean values from 0000 to 1111; they are the values 0000, 0001, 0010, 0100, 0110, 1000, 1010, 1100, 1110 and 1111.

Problem 1 has been chosen because Spector and Robinson (2002) and Spector (2010) state they successfully applied their Pushpop and AutoPush systems to similar symbolic regression problems. Problem 2 has been chosen to be slightly different to problem 1 in the hope that it will demonstrate the generality of the results found using problem 1.

3.2.1 Fitness Value Calculation

The calculation of an individual's fitness is the same for problems 1 and 2. An individual's fitness value is the sum, over all ten fitness cases, of the absolute difference between an individual's output and the predefined output. In this scheme, fitter

individuals have lower fitness values and perfect (global optimum) individuals have a fitness of zero.

3.3 Definition of Restricted Version of Push

Experiments 2 and 3 require the definition of a version of the Push language that has restricted expressive power. To define this version of Push, the individuals created by experiment 1 will be analysed to measure the frequency of Push instructions used by reproductively competent child producing programs. The least frequently used instructions will then be disabled for child producing programs in execution mode 3.

An alternative method for defining a restricted version of Push is suggested by Edmonds (1998) (Edmonds' work is discussed in Chapter 2). Edmonds wanted to analyse which mechanisms were most commonly used by genetic operators that significantly increased or decreased a chromosome's fitness. He applied 200 randomly created operators to 10,000 randomly created chromosomes, noted which operators consistently and significantly altered the fitness of chromosomes, and, finally, dissected the exceptionally good and bad operators.

Edmonds' random sample method could be used in this project to define a restricted version of Push without the need to implement a fully working autoconstructive genetic programming system. However, the random sample method has been discarded because other parts of this project do require an autoconstructive genetic programming system and the more that system is used, the more likely its defects will be discovered and fixed.

3.4 Summary

This chapter explained how the project's experiments involve the execution of the same autoconstructive genetic programming system albeit in different *execution modes*.

The chapter described that the execution modes determine two factors: (1) whether individuals are represented by one program for problem solving and child production, or by two programs; and (2) whether the expressive power of the language of child producing programs is restricted.

Finally, the chapter detailed the two symbolic regression problems ($x^2 + x$ and even 4-parity) which the experiments will be set to solve, and introduced how the expressive power of child producing programs will be restricted.

Chapter 4

Implementation

This chapter describes the implementation of the autoconstructive genetic programming system that was introduced in the previous chapter and which will be executed in the project's experiments. In addition, the chapter gives an overview of the Push programming language and describes the parameter values which will be used in the experiments.

4.1 Overview of the Push Programming Language

This project makes extensive use of the Push programming language and hence it is fitting that a practical introduction to it is given here. For a full description of version 3 of the Push language see (Spector *et al.*, 2004).

Push is a stack based language. Instructions pop their arguments from appropriate stacks, and push their result to an appropriate stack. Consider the instruction `INTEGER.+` which sums two integer values. It first pops its two arguments from the `INTEGER` stack, and then pushes its result to the `INTEGER` stack.

Push has six typed stacks: `EXEC`, `CODE`, `NAME`, `BOOLEAN`, `INTEGER` and `FLOAT`. The first two stacks are for executing and modifying code, the `NAME` stack is for assigning identifiers to literals and code, and the final three stacks support boolean, integer and float values. Like Lisp, Push allows code to be treated like data. Consequently, the `EXEC` and `CODE` stacks allow a program to modify its behaviour as it runs.

Push has a simple Lisp-like syntax. A Push program can consist of an instruction, a literal or a parenthesised list of zero or more Push programs. Formally, the syntax is `program ::= instruction | literal | ( program* )`. Like Lisp and other genetic programming languages, a Push program can consist of a tree structure of sub-programs and sub-sub-programs *ad infinitum*.

Instruction names typically begin with the name of the stack on which they operate. Hence, `INTEGER.+` sums integer values and `FLOAT.+` sums float values. The dot (.) is part of the instruction name and has no special significance.

When a literal is executed it is pushed to the relevant stack. For example, the program `( 1.141 )` pushes the value 1.141 to the `FLOAT` stack, and `( 23 )` pushes 23 to the `INTEGER` stack.

When an instruction finds there are insufficient items on relevant stacks to satisfy its arguments, it does nothing and execution continues. Consider the program `( 11 INTEGER.+ 22 INTEGER.+ )`. When the first sum instruction is encountered only one item is on the `INTEGER` stack and, consequently, it does nothing. When the second sum instruction is encountered, there are two values on the stack and the instruction pops those two values and pushes their sum — the value 33.

Push defines many instructions for each stack type. The Common Lisp implementation of Push version 3 that is used by this project has 142 instructions. Generally speaking, each stack type has instructions which manipulate the stack, convert stack values from one type to another, perform mathematical operations on stack values, assign identifiers to stack values and create random stack values. In addition, the `EXEC` and `CODE` stacks have instructions which support executing code in looping or recursive manners.

4.2 AutoPush

The model for this project's autoconstructive genetic programming system is Lee Spector's *AutoPush*. Spector describes aspects of AutoPush in (Spector, 2010) and has made the AutoPush source code and some log files available for download from `http://hampshire.edu/lspector/gptp10`.

Spector set AutoPush to solve the symbolic regression problem with target equation $x^3 - 2x^2 - x$, where x ranged from 0 to 9 (10 fitness cases). Spector provides two log files, 'success-12' and 'success-17', which document instances where the system found perfect solutions. In the cases of 'success-12' and 'success-17', AutoPush had a population size of 100,000, a tournament size of 100 and a maximum generation count of 10,000. It is unknown how many times Spector ran AutoPush to achieve the results of 'success-12' and 'success-17'. However, one could guess, from the names

of the log files, that it was at least 17 times.

The log file 'success-12' documents that AutoPush found a perfect solution in generation 31 and that the perfect individual had 31 ancestors. In 'success-17' a perfect individual was found in generation 56 with 6 ancestors.

4.3 Implementation of M801AGP

This project's autoconstructive genetic programming system was named *M801AGP*. M801AGP was developed using the Clozure CL implementation of Common Lisp (`http://ccl.clozure.com`), and uses the Common Lisp Push version 3 interpreter developed by Spector and others that is available from `http://hampshire.edu/lspector/push3`.

M801AGP is modelled on AutoPush (Spector, 2010). Specifically, M801AGP reuses AutoPush's definitions for *stagnancy*, *improvement value*, *code discrepancy*, and *point count*, but M801AGP implements slightly different *selection criteria* and *birth constraints*. The selection criteria and birth constraints determine (1) which individual wins during tournament selection and (2) when a selected individual's reproduction is prevented, respectively.

AutoPush is fully documented in (Spector, 2010), however, in the interest of producing a self-contained description of M801AGP, some of AutoPush's definitions are re-stated here.

M801AGP uses constant values to implement thresholds in its selection criteria and birth constraints. These constants have been taken directly from AutoPush where they were chosen *"more or less arbitrarily"* (Spector, 2010, p. 11). Spector comments that *"[o]ther values may perform better, and further study may provide guidance on setting these values or eliminating the parameters altogether"* (Spector, 2010, p. 11).

4.3.1 Selection Criteria

The selection criteria is used by tournament selection to decide which individuals are selected to have their children added to the next population. The selection criteria's description relies on a definition for stagnancy. An individual is considered to have a stagnant ancestry if it *has at least 6 ancestors* and *the ancestors in the most recent*

half of its ancestry have the same fitness value. The full description of the selection criteria is, in order of precedence, that:

- individuals with 2 or 1 ancestors beat individuals with 0 ancestors;
- individuals with 2 ancestors beat individuals with 1 ancestor;
- individuals with non-stagnant ancestries beat individuals with stagnant ancestries; and
- fitter individuals beat weaker individuals.

4.3.2 Birth Constraints

M801AGP's birth constraints restrict when an individual, chosen by tournament selection, is prevented from having its child enter the next population. Essentially, the birth constraints aim to prevent the continuation of lineages which are deemed to be in an evolutionary dead end. The birth constraints' definition relies on the definitions of improvement value, code discrepancy, point count and *invalidity.*

An improvement value is a scalar that represents how the fitness of a lineage of ancestors has improved. The calculation is weighted so more recent generations have more importance. First, a normalised vector of improvements ($\boldsymbol{i}$) is calculated where the change in fitness between parent and child for each of the lineage's generations is normalised to 1, 0, and -1 for improvements, non-changes and declines, respectively. Second, a vector of weights ($\boldsymbol{w}$) is generated. The weight for the current generation ($w_{current}$) is set to 1, and each preceding weight is set to 90% of its subsequent weight ($w_{g-1} = 0.9w_g$) — the most weight is given to the current generation and weighting declines for each preceding generation. Finally, the improvement value (iv) is taken to be the weighted average of the dot product of the vectors $\boldsymbol{i}$ and $\boldsymbol{w}$.

For example, consider a lineage where the most recent ancestor has a fitness of 300 and preceding ancestors have fitnesses of 200, 400, 500 and 500. The improvement value (iv) would be calculated as shown in Figure 4.1.

Improvement values range from -1.0 to 1.0 and represent the overall direction of change in problem solving ability of individuals' ancestries, with more weight given to recent changes. Values close to 1.0 denote consistent problem solving improvement, and values close to -1.0 signal consistent decline.

$$
\begin{aligned}
\boldsymbol{f} &= [500, 500, 400, 200, 300] \\
\boldsymbol{i} &= [0, 1, 1, -1] \\
\boldsymbol{w} &= [0.729, 0.81, 0.9, 1] \\
iv &= \frac{\boldsymbol{i} \cdot \boldsymbol{w}}{\sum_{j=1}^{n} w_j} = \frac{0 \times 0.729 + 1 \times 0.81 + 1 \times 0.9 + -1 \times 1}{0.729 + 0.81 + 0.9 + 1} = 0.206
\end{aligned}
$$

Figure 4.1: Example calculation of an improvement value (iv).

Code discrepancy is a measure of the difference between two Push programs. It is calculated as *"the sum, over all unique expressions and sub-expressions in either of the programs, of the difference between the numbers of occurrences of the expression in the two programs"* (Spector, 2010, p. 12). Two identical programs have a discrepancy of zero, and discrepancy increases the more different the programs are. In execution modes 2 and 3 (Section 3.1), where individuals are represented by two programs, the code discrepancy between two individuals is the sum of the discrepancy between the individuals' problem solving programs plus the discrepancy between the child producing programs.

Point count is a measure of the size of a Push program. It is the sum of the counts of all the program's literals, instructions and parenthesis pairs, and is used to limit the size of programs. In execution mode 1 (Section 3.1), individuals are restricted to have programs with point counts in the range 10 to 100, inclusive. In execution modes 2 and 3, individuals must have a minimum point count of 5 for each program, and a maximum combined count of 100.

When new individuals are created randomly in execution modes 2 and 3, problem solving programs are created with a count between 5 and 95, inclusive, and child producing programs are created with a minimum count of 5 and so that the individual's combined count does not exceed 100.

Invalidity concerns penalising individuals whose problem solving behaviour suggests they are undeserving of their fitness score. An individual is considered invalid if, during fitness assessment, it does not produce an output for a fitness case; outputs the input value for all fitness cases; or, outputs the same value for all fitness cases. Invalid individuals are tagged as such and are assigned an especially bad fitness

value of one thousand million (10^9).

Having described the pre-requisite definitions of improvement value, code discrepancy, point count and invalidity we can now define M801AGP's birth constraints. An individual that has been selected to have its child added to the next population is prevented from reproducing if it:

- is tagged as invalid;
- has 4 or more ancestors and an improvement value of 0.1 or less;
- has 3 or more ancestors and all the parent/child discrepancy values for the lineage are constant;
- has a child with a point count that is deemed too large or small;
- has a child which is a duplicate of any individual which has existed in any population since the start of the run; or,
- has a child with a parent/child discrepancy of less than 4.

When the birth constraints prevent an individual's child from being added to the next population, a new, randomly created individual is added instead.

4.3.3 Additional Features

In addition to the selection criteria and birth constraints described above, M801AGP has another feature which might not be expected in a genetic programming system. Any individual, be it a child or random individual, is discarded if it duplicates any individual that has been added to any population since the start of the run. This constraint is designed to improve population diversity.

4.3.4 Execution of Individuals

In M801AGP, the programs of individuals are executed many times; once for each fitness case when assessing problem solving ability and once to produce a child. This section describes how this is achieved for the case of execution mode 1 and the case of execution modes 2 and 3. Tables 4.1 and 4.2 present the execution schemes for the two cases. Both tables deal with the solving of problem 1 (Section 3.2). When solving problem 2 (Section 3.2), the fitness assessment input and output values are pushed to and popped from the `BOOLEAN` stack instead of the `INTEGER` stack.

The scheme for execution mode 1 matches that of AutoPush (Spector, 2010). An individual's program is responsible for both problem solving and child production,

1. Push input value to `INTEGER` stack. 2. Push program to `CODE` stack. 3. Execute program. 4. Take top item of `INTEGER` stack to be the output.

1. Push program to `CODE` stack. 2. Execute program. 3. Take top item of `CODE` stack to be the child's program.

Table 4.1: Steps for executing an individual, in execution mode 1 and for problem 1, for fitness assessment (top) and child production (bottom).

1. Push input value to `INTEGER` stack. 2. Push child producing program to `CODE` stack. 3. Push problem solving program to `CODE` stack. 4. Execute problem solving program. 5. Take top item of `INTEGER` stack to be the output.

1. Push child producing program to `CODE` stack. 2. Push problem solving program to `CODE` stack. 3. Execute child producing program. 4. Take 1st (top) and 2nd items of `CODE` stack to be the child's problem solving and child producing programs, respectively.

Table 4.2: Steps for executing an individual, in execution modes 2 and 3 and for problem 1, for fitness assessment (top) and child production (bottom).

and, when executed, the problem solving part of the program is able to make use of the child producing part of the program and vice versa.

The scheme for execution modes 2 and 3 has been designed to be as close as possible to execution mode 1's scheme while allowing an individual to have two programs. When one of an individual's programs is executed, the other program is made available to the executing program by being pushed to the `CODE` stack. Furthermore, and like execution mode 1's scheme, this scheme endeavours not to predefine the ratio of genetic material that is used for problem solving and child production. However, the scheme does impose a minimum of 5% of genetic material be allocated to each program.

4.3.5 Sanitisation of Child Production Programs in Execution Mode 3

In execution modes 2 and 3 when an individual is executed for child production, the executing program has access to the individual's other program. For execution mode 2, this is not a problem. However, for execution mode 3, where the expressive power of child producing programs is restricted, it is possible for disabled instructions to migrate from an individual's problem solving program to the individual's child's child producing program.

The scheme used by M801AGP to mitigate this problem is sanitisation. In execution mode 3 and just after a child is produced, disabled instructions in the child's child producing program are replaced by randomly chosen enabled instructions. This scheme was chosen because of its simplicity.

4.4 Parameters

Preliminary runs of M801AGP in execution mode 1 and with a population size of 20,000, a tournament size of 3 and 21 fitness cases, showed that the system took around 24 minutes to complete a single generation. This suggested that M801AGP with parameters comparable to AutoPush's — a population size of 100,000 — would take 9 days to complete a run of 100 generations. This was seen as impractical for this project, especially considering that, like all genetic programming systems, M801AGP implements a stochastic process and multiple independent runs must be carried out to determine the system's overall characteristic behaviour.

Consequently, it was decided that M801AGP would run with parameters one-tenth the size of AutoPush's parameters, that 10 fitness cases would be used and that a set of 20 independent runs, that vary only in their starting random seed, would be performed for each execution of M801AGP. Table 4.3 summarises the parameters.

Independent Runs:	20
Population Size:	10,000
Tournament Size:	10
Maximum Generation Count:	100
Fitness Cases:	10

Table 4.3: Parameters used for each execution of M801AGP.

M801AGP's termination criterion is to stop when the maximum generation count is reached. M801AGP does not stop if an individual with a perfect fitness of zero is found. This policy was chosen because preliminary runs of M801AGP showed that sometimes individuals would be randomly created with a fitness of zero. Those individuals were not a product of the autoconstructive nature of M801AGP that is this project's focus and, hence, M801AGP was allowed to continue running up to the maximum generation count.

4.5 Summary

The bulk of this chapter was dedicated to detailing the implementation of the project's autoconstructive genetic programming system (M801AGP). The chapter began with an example based introduction to the Push programming language. Then the chapter described the *AutoPush* (Spector, 2010) system on which M801AGP was modelled. Finally, the chapter gave an in-depth description of M801AGP's implementation, and described the parameter values that have been chosen for this project's experiments and how they were arrived at.

Chapter 5

Results

This chapter presents the results that were obtained during the project's experimentation phase. In addition to experiments 1, 2 and 3, studies of problems 1 and 2 were undertaken. Furthermore, a description is given of the procedure used to define execution mode 3's restricted version of Push.

In this chapter, comparisons between results are made qualitatively. A quantitative statistical analysis comparison of results was not undertaken because of two reasons. First, I have no experience of using statistical analysis, and I feared I might bias an analysis by not appreciating when different statistical analyses should and should not be used. Second, the project's resources only allowed 20 independent runs to be carried out for each execution in each experiment, and, as I understand it, this is less than ideal for a statistical analysis to be undertaken.

The chapter following this chapter discusses the results and draws conclusions.

5.1 A Study of Problem 1

A random sample study of problem 1 was undertaken to gauge how difficult it would be for M801AGP to solve the problem. The study's parameters were chosen to be comparable to an execution of M801AGP. 20 independent samples were made, and each sample evaluated 1,000,000 random and unique individuals. Ostensibly, 20,000,000 unique individuals were evaluated by the study. The actual value was probably lower because the study did not prevent individuals from being evaluated multiple times by different samples.

In the study, an individual was represented as a single Push program with an instruction set size of 142 and a length of between 10 and 100 code points, inclusive. In this regard, the study resembled M801AGP's execution mode 1. An approxim-

$$\sum_{n=10}^{100} 142^n = 1.7 \times 10^{215}$$

Figure 5.1: Approximation for the size of the space of all Push programs with an instruction set size of 142, a minimum point count of 10 and a maximum point count of 100.

ation for the size of the space of all possible individuals is given in Figure 5.1, and is 1.7×10^{215}. The sample size of 20,000,000 (2×10^7) represents a tiny fraction of the space of all individuals; it is 208 orders of magnitude smaller. However as the resources of any practical system are limited, the performance of the study is representative of a random search based system set to tackle problem 1.

The results of the 20 samples were averaged and are presented in Tables 5.1 and 5.2. Table 5.1 documents the performance of random search at solving the problem and the probability with which a random individual is (a) valid and (b) valid and a perfect solution. Table 5.2 presents the distribution of valid individuals with fitness values in the range 0 to 2000.

The study used the same fitness value calculation and classification of invalidity as does M801AGP (Sections 3.2 and 4.3). The performance measures *success rate* and *mean best fitness* are defined in (Eiben & Jelasity, 2002). Success rate is the percentage of samples which found a perfect individual, and mean best fitness is the mean of the fitnesses of the best individuals found in each sample.

Success Rate	50%
Mean Best Fitness	12.1
Percentage of all individuals that were valid	8.0%
Percentage of all individuals with a fitness of zero	0.00008%

Table 5.1: Results of random sample study on problem 1.

In summary, the study suggests that, for problem 1, 8 in 10 million (0.00008%) of all individuals have a perfect fitness of zero, 8% of all individuals are valid and the majority of valid individuals (94%) have a fitness in the range 201 to 400.

Fitness Value Range	Percentage of Valid Individuals within Range
0 to 200	1.24%
201 to 400	94.00%
401 to 600	2.49%
601 to 800	0.64%
801 to 1000	0.4%
1001 to 1200	0.51%
1201 to 1400	0.13%
1401 to 1600	0.04%
1601 to 1800	0.06%
1801 to 2000	0.03%

Table 5.2: Distribution of fitness values of valid individuals for random sample study on problem 1. Values averaged across 20 samples.

5.2 Experiments 1 and 2

This section presents the combined results of experiments 1 and 2. Experiment 1 compared execution modes 1 and 2, and experiment 2 compared modes 2 and 3. Both experiments tasked M801AGP with solving problem 1.

Table 5.3 gives the success rate and mean best fitness achieved by each execution mode. The table splits each performance measure into two categories: *whole population* and *reproductive population*. In any generation, the whole population is all of the individuals in the generation's population, and the reproductive population is the subset of the whole population that contains only individuals with one or more ancestors. The split highlights the difference in results between individuals that are a result of random creation and those that are a product of autoconstruction.

		Execution Mode 1	**Execution Mode 2**	**Execution Mode 3**
Success Rate	whole population	40%	70%	55%
	reproductive population	0%	0%	10%
Mean Best Fitness	whole population	16	10.25	12.25
	reproductive population	50.1	47.4	42.3

Table 5.3: Success rate and mean best fitness for execution modes 1, 2 and 3 on problem 1.

The success rates, given in Table 5.3, show that, in all execution modes, more perfect individuals were found by random creation than were found by autoconstruction. Execution modes 1 and 2 failed to find any perfect individuals by autoconstruction. However, execution mode 3 outperformed modes 1 and 2; it found some perfect individuals by autoconstruction.

The mean best fitness results show the change from execution mode 1 to 2 was beneficial for the whole and reproductive populations. However, the change from mode 2 to 3 was only beneficial for the reproductive population. In all execution modes, random creation outperformed autoconstruction.

Figure 5.2 gives a plot of reproductive population size versus generation for execution modes 1, 2 and 3 on problem 1. The reproductive population size is the number of individuals with at least one ancestor, and is a measure of the reproductive competence of the system. Ideally, we would like to see the reproductive population size start at 0%, rise quickly to 100% and stay at 100%.

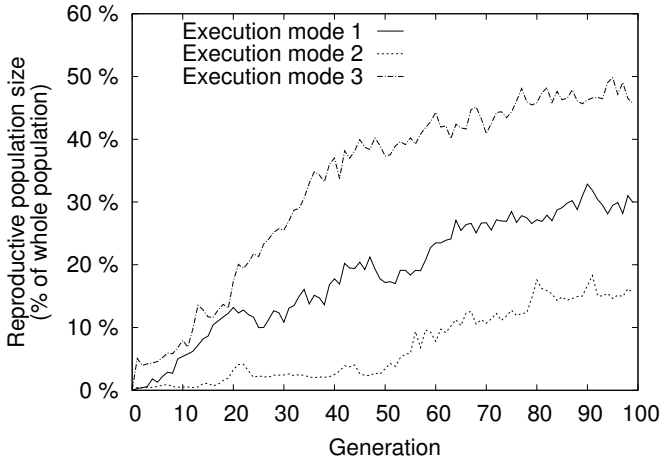


Figure 5.2: Plot of reproductive population size versus generation for execution modes 1, 2 and 3 on problem 1. Values averaged across 20 independent runs for each execution mode.

The plot shows that the switch from one to two programs per individual (execution mode 1 to 2) was detrimental to the reproductive population size. However, the loss was more than made up for by switching to a restricted version of Push for child producing programs (execution mode 2 to 3).

Figure 5.3 plots a characteristic of the reproductive population: the average number

of ancestors per individual. The measure describes the ability of individuals, in the reproductive population, to reproduce many times over and establish lineages. As we desire a reproductively competent system, higher values of this measure are better than lower values.

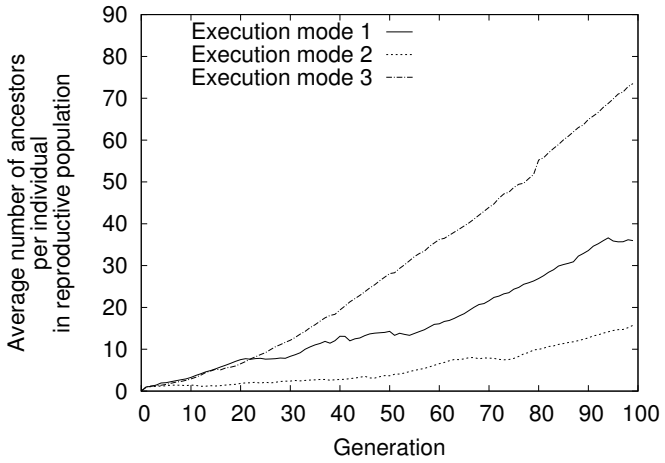


Figure 5.3: Plot of average number of ancestors per individual in the reproductive population versus generation for execution modes 1, 2 and 3 on problem 1. Values averaged across 20 independent runs for each execution mode.

The results of this plot mirror those of the previous plot. Execution mode 1 sets the baseline, mode 2 does worse than mode 1, and mode 3 outperforms both modes 1 and 2.

Figure 5.4 gives plots of the minimum fitness of the reproductive population versus generation. The bottom plot smooths the data using gnuplot's `smooth bezier` option (see `http://gnuplot.info`) to make it easier to see the trends within the top plot.

The plots depict the performance of the reproductive population at solving the problem at hand. If the system were able to converge on a solution, we would expect to see the minimum fitness start high but eventually drop to zero or an acceptably low value. However, these plots do not seem to show signs of convergence.

The top plot shows that all the execution modes began with high values that dropped rapidly. Execution mode 3's drop was deeper than that of the other modes. However, at around generation 40 all three execution modes stabilised on values around 250. The bottom plot shows that execution mode 3 tended to perform better than mode 2, and mode 2 tended to outperform mode 1.

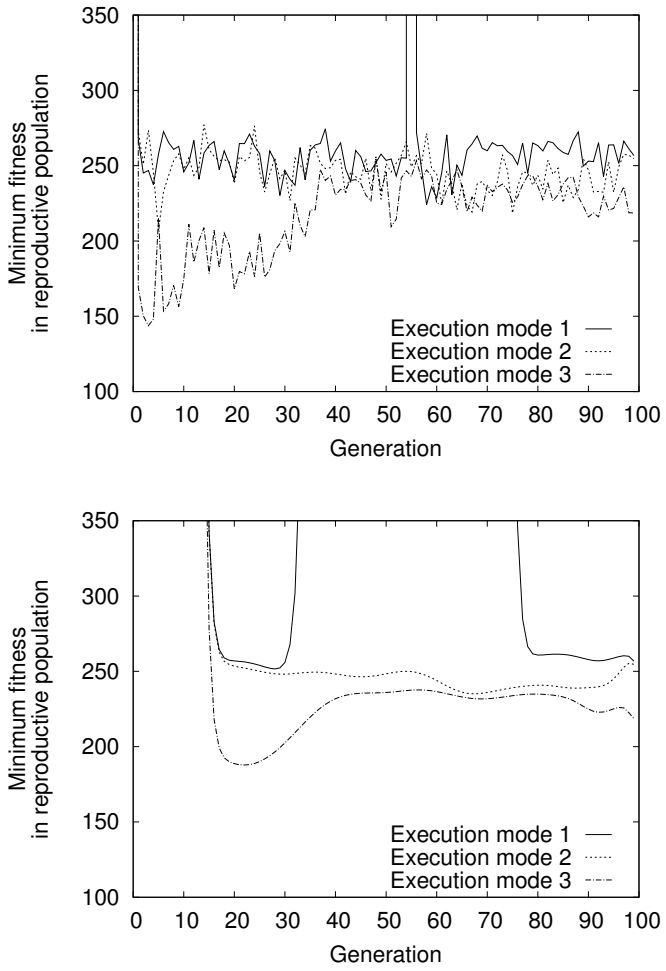


Figure 5.4: Plots of minimum fitness in the reproductive population versus generation for execution modes 1, 2 and 3 on problem 1. Values averaged across 20 independent runs for each execution mode. Bottom plot smooths the data using gnuplot's `smooth bezier` option. Note that the y-axis starts at 100.

5.3 Restricted Version of Push

The populations created by M801AGP in execution mode 2 on problem 1 were analysed to define the restricted version of Push required by execution mode 3. For each individual which successfully produced a child, a point was scored by each occurrence of each instruction in the child producing program. The scores were summed and ordered. The result was a list of instructions ordered by how frequently they were used by reproductively competent individuals in child producing programs.

The restricted version of Push was defined by disabling the least frequently used 25% of instructions. The value of 25% was chosen arbitrarily to be large enough to affect the space of all child producing programs, but not so large as to overly restrict

child producing programs. Appendix B lists the instructions that were disabled in execution mode 3's restricted version of Push.

As discussed in Section 1.2, this project assumes the space of competent child producing programs is *not* dependent on the problem a system is set to solve. Consequently, the same restricted version of Push was used for execution mode 3 on problems 1 *and* 2.

5.4 A Study of Problem 2

Like the random sample study of problem 1, a study was also made of problem 2. The study used the same parameters as the problem 1 study: 20 independent samples were made, and each sample evaluated 1,000,000 random and unique individuals. Tables 5.4 and 5.5 present the study's results.

Success Rate	100%
Mean Best Fitness	0
Percentage of all individuals that were valid	57.0%
Percentage of all individuals with a fitness of zero	0.02%

Table 5.4: Results of random sample study on problem 2.

Fitness Value(s)	Percentage of Valid Individuals with Value(s)
0, 1	0.46%
2	0.66%
3	3.07%
4	37.60%
5	48.00%
6	7.55%
7	1.87%
8	0.61%
9	0.16%
10	0.02%

Table 5.5: Distribution of fitness values of valid individuals for random sample study on problem 2. Values averaged across 20 samples.

The study suggests that, for problem 2, 2000 in 10 million (0.02%) of all individuals have a perfect fitness of zero, 57% of all individuals are valid and the majority of valid individuals (85.6%) have a fitness value of 4 or 5.

A comparison of these results with those of problem 1 indicates that, in the space of all Push programs, there is a higher proportion of programs that can solve problem 2 than there is for problem 1. The proportions are 8 in 10 million for problem 1, and 2000 in 10 million for problem 2. This indicates that problem 2 has a higher probability of being solved by random search and, by extension, genetic programming than problem 1.

5.5 Experiment 3

Experiment 3 compared execution modes 2 and 3 on problem 2, and was designed to test the generality of the results of experiment 2; to test whether the results of experiment 2 would hold for a different problem.

Table 5.6 gives the success rate and mean best fitness achieved by each execution mode. For each execution mode and category of population, the system consistently found a perfect individual. These results, when compared to those of problem 1 (Table 5.3), clearly support the results of the random study on problem 2 that suggest problem 2 is simpler to solve than problem 1.

		Execution Mode 2	**Execution Mode 3**
Success Rate	whole population	100%	100%
	reproductive population	100%	100%
Mean Best Fitness	whole population	0	0
	reproductive population	0	0

Table 5.6: Success rate and mean best fitness for execution modes 2 and 3 on problem 2.

Plots of experiment 3's results are given here in Figures 5.5, 5.6 and 5.7. These plots use the same measures as the plots for experiments 1 and 2 (Figures 5.2, 5.3 and 5.4). The first two figures are concerned with measures of reproductive competence, and the third is concerned with problem solving performance.

In some cases, the figures include plots that have been smoothed using gnuplot's `smooth bezier` option (see `http://gnuplot.info`) to make it easier to identify trends within the plots of the original data.

The plots concerning measures of reproductive competence (Figures 5.5 and 5.6) show that modes 2 and 3 began at 0 and steadily rose. Modes 2 and 3 behaved similarly; there was no notable difference between the behaviour of the two modes.

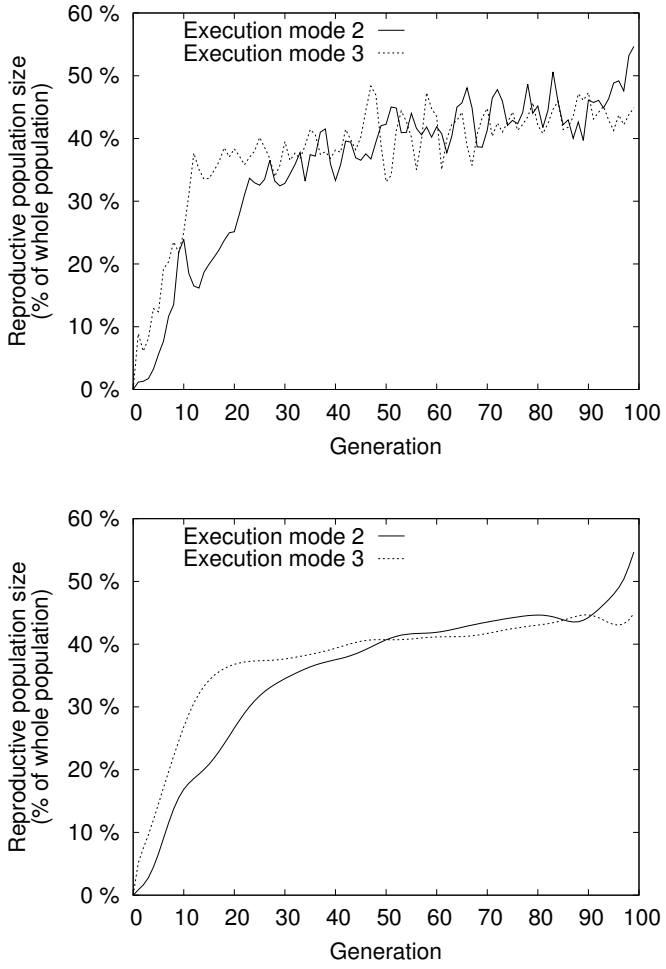


Figure 5.5: Plot of reproductive population size versus generation for execution modes 2 and 3 on problem 2. Values averaged across 20 independent runs for each execution mode. Bottom plot smooths the data using gnuplot's `smooth bezier` option.

The plots for problem solving performance (Figure 5.7) show that, initially, both modes rapidly found ever fitter individuals, but that the rate of improvement dropped drastically at around generation 10. Even though the rate of improvement dropped, it does not appear to have flatlined as it did in execution modes 1, 2 and 3 on problem 1 (Figure 5.4).

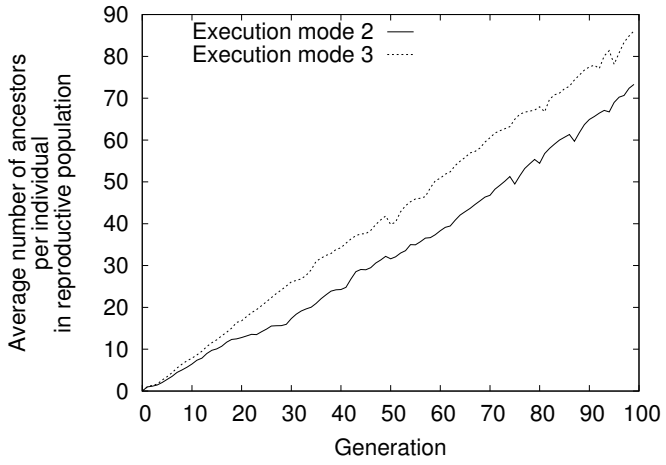


Figure 5.6: Plot of average number of ancestors per individual in the reproductive population versus generation for execution modes 2 and 3 on problem 2. Values averaged across 20 independent runs for each execution mode.

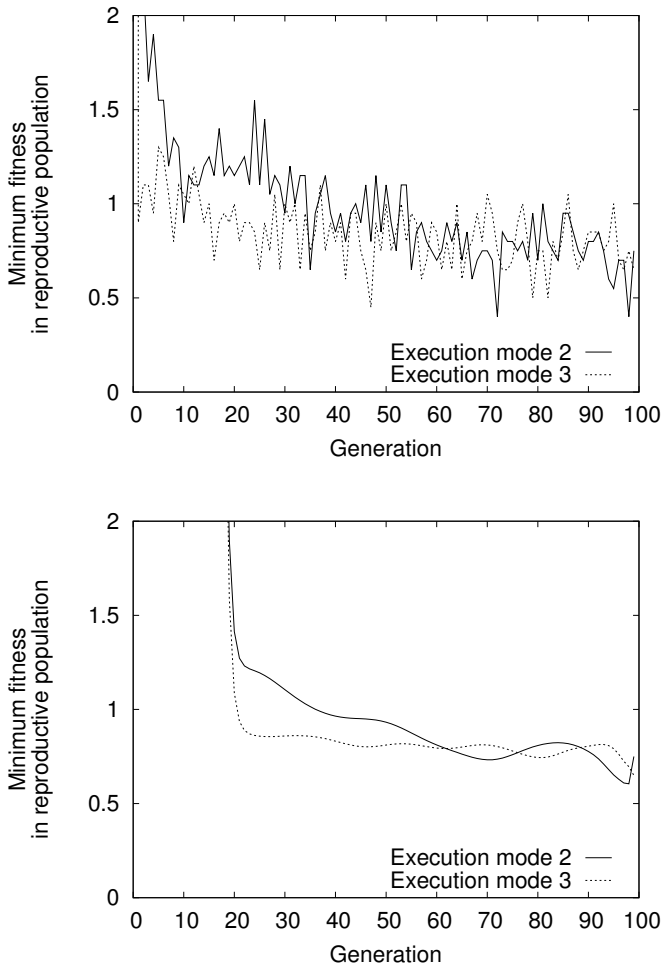


Figure 5.7: Plots of minimum fitness in the reproductive population versus generation for execution modes 2 and 3 on problem 2. Values averaged across 20 independent runs for each execution mode. Bottom plot smooths the data using gnuplot's `smooth bezier` option.

5.6 Summary

This chapter presented several sets of results: the results of random sample studies on problems 1 and 2, the results of the procedure for defining execution mode 3's restricted version of Push, the combined results of experiments 1 and 2, and the results of experiment 3.

The random studies described characteristics of problems 1 and 2. Namely, the percentage of all individuals that have perfect fitness, the percentage of all individuals that are valid and the distribution of fitness values of valid individuals. The results provide a benchmark for comparisons and indicate that, for genetic programming, problem 2 is simpler to solve than problem 1.

Experiments 1 and 2 compared execution modes 1, 2 and 3 on problem 1. The results for measures of reproductive competence showed that the modes performed differently. Mode 3 performed best followed by mode 1 and then mode 2. Concerning problem solving performance, the results showed that, for the success rate and mean best fitness of the reproductive population, mode 3 outperformed modes 1 and 2; and that for minimum fitness in the reproductive population, mode 3 initially performed better than the other modes, but that all the modes stabilised around a value of 250 at around generation 40.

Experiment 3 compared execution modes 2 and 3 on problem 2. The results showed, for measures of reproductive competence, that the modes performed similarly. For problem solving performance, the results showed that, initially, both modes rapidly found fitter individuals, but at around generation 10 the rate of improvement reduced considerably. However, the rate of improvement did not appear to flatline as it did in execution modes 1, 2 and 3 on problem 1.

Chapter 6

Discussion

This chapter discusses the results presented in the previous chapter considered in the context of the research question.

> *In an autoconstructive genetic programming system, does the expressive power of the language of child producing programs affect the reproductive competence of individuals or the problem solving ability of the system?*

Three issues are discussed: (1) the effect of switching from one program per individual to two; (2) the effect of reducing the expressive power of the language of child producing programs, and whether the space of competent child producing programs is dependent on the problem being tackled; and (3) the problem solving performance of the project's autoconstructive genetic programming system (M801AGP).

6.1 Effect of Switching from One Program per Individual to Two

Experiment 1 investigated the side effects of switching from one program per individual, for problem solving and child production, to two programs, one for each concern. In both cases, the same amount of genetic material was available to an individual, and the allocation of material to each concern was unconstrained.

The results of experiment 1 (Section 5.2) show that, for problem 1, the switch had a detrimental effect on reproductive competence and little effect on problem solving performance. For measures of reproductive competence, the switch resulted in a drop in performance of around 50%. For the problem solving ability of the reproductive population, the success rate was the same before and after the switch, and the mean best fitness dropped only slightly. Furthermore, the plot of the minimum fitness in

the reproductive population shows the system behaved the same before and after the switch.

These results suggest that, in an autoconstructive genetic programming system, it is possible to successfully separate the problem solving and child production concerns of an individual into two programs. Furthermore, they suggest such a separation would result in reduced reproductive competence performance and little change in problem solving performance.

These conclusions must be taken as tentative for the general case because the results were obtained from running an autoconstructive genetic programming system a limited number of times and only on one problem.

However, the results do concur with Hinterding (1997) who also found multi-chromosomes useful for representing the different concerns of an individual within an evolutionary algorithm (Section 2.4.4).

6.2 Effect of Reducing Expressive Power of Child Producing Programs

Experiments 2 and 3 investigated the effect of reducing the expressive power of the language of child producing programs. Experiment 2 investigated the effect with respect to problem 1, and experiment 3 investigated with respect to problem 2.

The results of experiment 2 (Section 5.2) show that, for problem 1, the effect of reducing the expressive power of the language of child producing programs was beneficial for reproductive competence and problem solving performance. However, the results of experiment 3 (Section 5.5) show that, for problem 2, the effect was negligible for reproductive competence and problem solving performance.

These results suggest that, in an autoconstructive genetic programming system where individuals have a dedicated program for child production, the expressive power of the language of child producing programs can be reduced to improve the reproductive competence and problem solving ability of the system, but only when the reduction is defined by the problem that is being tackled.

Furthermore, the results suggest the space of competent child producing programs *is* dependent on the problem being solved. If it were not dependent on the problem then we would have expected to see an improvement in reproductive com-

petence and problem solving ability in experiment 3 (Figures 5.5 and 5.6).

Again, these conclusions must be taken as tentative for the general case because they are based on the results of running an autoconstructive genetic programming system a limited number of times and only on two problems.

6.3 Problem Solving Performance

The problem solving performance of the project's autoconstructive genetic programming system (M801AGP) was weak. In all the experiments, the problem solving performance of the *whole population* was always the same as, or better than, the performance of the *reproductive population* (Tables 5.3 and 5.6). That is, M801AGP's autoconstruction part was outperformed by its random creation part.

This result is disappointing, but maybe explained by an observation made by Spector and Robinson (2002) on the performance of their autoconstructive genetic programming system named *Pushpop* (Section 1.1.7). They observed that, early on in a run, Pushpop first had to achieve a reproductive population size of 100% and only then did selection begin to drive the search process.

In this project's experiments, the size of the reproductive population never reached 100%. However, M801AGP's performance suggests that 100% would have been reached if it were allowed to continue running for more generations (Figures 5.2 and 5.5).

Consequently, we can conclude that the observed problem solving performance of M801AGP was poor, but that there is some evidence to suggest that if M801AGP were allowed to run for more generations and achieve a reproductive population size of 100% then problem solving performance would improve.

6.4 Summary

This chapter presented a discussion of the results presented by the preceding chapter. The results were discussed in the context of the research question, and the following tentative conclusions were drawn:

- that, in autoconstructive genetic programming, it appears possible to successfully separate an individual's concerns of problem solving and child production into two programs;

- that, in autoconstructive genetic programming where individuals have a dedicated program for child production, the reduction of the expressive power of the language of child producing programs appears to improve the reproductive competence and problem solving ability of the system, but only when the reduction is defined by the problem that is being tackled;
- that the space of competent child producing programs does appear to be dependent on the problem being solved; and
- that the problem solving performance of the project's autoconstructive genetic programming system (M801AGP) was weak, but that an improvement might be seen if it were allowed to run for more generations.

Chapter 7

Conclusions

This final chapter reviews the project and proposes some areas for future research.

7.1 Project Review

I believe this project has made headway towards answering the research question, and that its research method was appropriate and made good use of limited time and computing resources.

The project generated evidence, albeit limited, that the expressive power of the language of child producing programs does affect the reproductive competence of an autoconstructive genetic programming system. In addition, the evidence suggests that the expressive power can be modified to improve system performance.

However, the project's conclusions are tentative because, due to practical limitations, they are based on a small number of runs of an autoconstructive genetic programming system set to tackle just two problems. Nevertheless, the final executions of the project's experiments required around 19 days of computing time, and this is, I believe, not an insignificant amount for a student research project.

The problem solving ability of the project's autoconstructive genetic programming system was worse than expected. However, it was not known whether this was due to a limitation of the system's theory or a problem in its implementation. Consequently, random sample studies of the set problems were conducted to define performance benchmarks.

In hindsight, I would have preferred to have run *AutoPush* (Spector, 2010) — the autoconstructive genetic programming system on which this project's system was modelled — on the chosen problems to define benchmarks instead of running the random sample studies. AutoPush and this project's system were based on the

same theory. Therefore, benchmarks defined by AutoPush would have been fairer indicators for the expected performance of the project's system, and would, possibly, have highlighted any implementation problems.

AutoPush was not run for two reasons. Firstly, because the idea did not dawn on me until after I had invested effort in the random study of the project's first chosen problem; and secondly, because I was learning my first dialect of the Lisp programming language (*Common Lisp*) and AutoPush is based on another dialect (*Clojure*).

7.2 Future Research

This final section suggests research areas whose investigation would extend and complement this project's work.

7.2.1 Substantiate and Expand on this Project's Results

This project's conclusions are tentative because they are based on too few runs of an autoconstructive genetic programming system set to tackle too few problems (Chapter 6). Consequently, an obvious future research area is to repeat this project's experiments albeit for a higher number of runs (say 50 or 100, instead of 20) and, possibly, for a higher number of problems. Increasing the number of runs would also lend value to a statistical analysis of the results.

In addition, the parameters of the experiments could be made larger. This project's experiments used parameters which were one-tenth the size of those used by *AutoPush* (Spector, 2010) (Section 4.4) — the autoconstructive genetic programming system on which this project's system was based. Indeed, this project posits that the problem solving performance of the project's autoconstructive genetic programming system may improve when all individuals in a population are reproductively competent, and that more than 100 generations are necessary for this to happen (Section 6.3).

7.2.2 Dynamic Reduction of the Expressive Power of Child Producing Programs

In this project, the expressive power of the language of child producing programs was reduced in a static way (Section 5.3). However, there is no reason why this

could not be done dynamically. Indeed, the premise behind autoconstructive genetic programming is that it may better standard genetic programming by allowing reproductive mechanisms to evolve dynamically. Accordingly, it is only natural that this idea be extended to the reduction of expressive power.

An example dynamic population-level method for reducing expressive power is if all language instructions are enabled initially, but after a pre-specified number of generations a number of the least used instructions are disabled. In such a scheme, it is conceivable that the definition of the reduced expressive power may adapt to the problem being tackled.

7.2.3 Relation Between the Space of Competent Child Programs and the Set Problem

The evidence generated by this project suggests that the space of competent child producing programs *is* dependent on the problem being tackled (Section 6.2). Consequently, another possible research area would be to investigate *how* sensitive the space of competent child producing programs is to the set problem, and whether classes of problems exist with similar spaces.

7.2.4 Use Cloud Computing to Execute Experiments

Another possible research area is the use of *cloud computing* (Mell & Grance, 2011) to execute autoconstructive genetic programming experiments. This research area is important as it may permit more extensive experiments to be carried out by researchers without large investments of money and time in computing hardware.

Essentially, cloud computing provides a service where computing power can be rented on remote *cloud computers*. The renting is elastic in that a researcher can rent any amount of computing power for any amount of time. However, cloud computing is still relatively new and a researcher would have to adapt their system to run reliably on cloud computers and so that the bulk transfer of data to and from the computers is practical and inexpensive.

References

Affenzeller, M., Winkler, S., Wagner, S. & Beham, A. (2009). *Genetic algorithms and genetic programming: modern concepts and practical applications*. Chapman & Hall/CRC.

Angeline, P. (1995). Adaptive and Self-adaptive Evolutionary Computations. *Computational Intelligence: A Dynamic System Perspective*, 152.

Back, T., Hammel, U. & Schwefel, H.-P. (1997). Evolutionary computation: comments on the history and current state. *Evolutionary Computation, IEEE Transactions on*, *1*(1), 3–17.

Beyer, H.-G. & Schwefel, H.-P. (2002). Evolution strategies – a comprehensive introduction. *Natural Computing*, *1*, 3–52.

Cavill, R., Smith, S. & Tyrrell, A. (2005). Multi-chromosomal genetic programming. In *Proceedings of the 2005 conference on genetic and evolutionary computation* (pp. 1753–1759). GECCO '05. Washington DC, USA: ACM.

Clune, J., Goings, S., Punch, B. & Goodman, E. (2005). Investigations in meta-gas: panaceas or pipe dreams? In *Proceedings of the 2005 workshops on genetic and evolutionary computation* (pp. 235–241). GECCO '05. Washington, D.C.: ACM.

De Jong, K. (2007). Parameter setting in EAs: a 30 year perspective. In F. Lobo, C. Lima & Z. Michalewicz (Eds.), *Parameter setting in evolutionary algorithms* (Vol. 54, pp. 1–18). Studies in Computational Intelligence. Springer Berlin / Heidelberg.

De Jong, K. A. (1975). *Analysis of the Behavior of a Class of Genetic Adaptive Systems*. (PhD thesis, University of Michigan).

Dioşan, L. & Oltean, M. (2009, 3). Evolutionary design of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, *10*(3), 263–306.

Edmonds, B. (1998). Meta-Genetic Programming: Co-Evolving the Operators of Variation. Centre for Policy Modelling, Manchester Metropolitan University. CPM Report No.: 98-32.

Eiben, A. & Jelasity, M. (2002). A Critical Note on Experimental Research Methodology in EC. In *Proceedings of the 2002 ieee congress on evolutionary computation (cec 2002)* (pp. 582–587). IEEE Press.

Eiben, A., Hinterding, R. & Michalewicz, Z. (1999). Parameter control in evolutionary algorithms. *Evolutionary Computation, IEEE Transactions on*, *3*(2), 124 –141.

Fogel, D. & Fogel, L. (1996). An introduction to evolutionary programming. In *Artificial evolution* (pp. 21–33). Springer.

Grefenstette, J. (1986). Optimization of Control Parameters for Genetic Algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, *16*(1), 122–128.

Hinterding, R. (1997). Self-adaptation using multi-chromosomes. In *Evolutionary computation, 1997., ieee international conference on* (pp. 87 –91).

Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor.

Kantschik, W., Dittrich, P., Brameier, M. & Banzhaf, W. (1999). Meta-evolution in graph gp. In R. Poli, P. Nordin, W. Langdon & T. Fogarty (Eds.), *Genetic programming* (Vol. 1598, pp. 652–652). Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

Kim, M., McKay, R., Hoai, N. & Kim, K. (2011). Operator self-adaptation in genetic programming. In S. Silva, J. Foster, M. Nicolau, P. Machado & M. Giacobini (Eds.), *Genetic programming* (Vol. 6621, pp. 215–226). Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

King, R. C., Stansfield, W. D. & Mulligan, P. K. (2007). "phenotype" A Dictionary of Genetics. Oxford Reference Online, Oxford University Press. [Online; accessed 12 December 2011; http://www.oxfordreference.com/views/ENTRY.html?entry=t224.e4813].

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT press.

Kramer, O. (2010). Evolutionary self-adaptation: a survey of operators and strategy parameters. *Evolutionary Intelligence*, *3*(2), 51–65.

Luke, S. (2009). *Essentials of metaheuristics*. Lulu, available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

Mell, P. & Grance, T. (2011). *The NIST Definition of Cloud Computing*. Special Publication 800-145. Available at http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

Meyer-Nieberg, S. & Beyer, H.-G. (2007). Self-Adaptation in Evolutionary Algorithms. *Computational Intelligence (SCI)*, *54*, 47–75.

Nannen, V., Smit, S. & Eiben, A. (2008). Costs and benefits of tuning parameters of evolutionary algorithms. In G. Rudolph, T. Jansen, S. Lucas, C. Poloni & N. Beume (Eds.), *Parallel problem solving from nature – ppsn x* (Vol. 5199, pp. 528–538). Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

O'Neill, M., Vanneschi, L., Gustafson, S. & Banzhaf, W. (2010, 3). Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, *11*, 339–363.

Poli, R., Langdon, W. B. & McPhee, N. F. (2008). *A field guide to genetic programming*. (With contributions by J. R. Koza). Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk.

Rahim, A., Teo, J. & Saudi, A. (2006). The effect of dynamic parameter stability on reproductive competence. In *International conference on computing informatics (icoci)* (pp. 1–6).

Spector, L. (2010). Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems. In *Genetic programming theory and practice viii*. Springer.

Spector, L. & Robinson, A. (2002). Genetic Programming and Autoconstructive Evolution with the Push Programming Language. *Genetic Programming and Evolvable Machines*, *3*(1), 7–40.

Spector, L., Perry, C., Klein, J. & Keijzer, M. (2004). *Push 3.0 programming language description* (tech. rep. No. HC-CSTR-2004-02). School of Cognitive Science, Hampshire College. Available at http://hampshire.edu/lspector/push3-description.html.

Tavares, J., Machado, P., Cardoso, A., Pereira, F. & Costa, E. (2004). On the evolution of evolutionary algorithms. In M. Keijzer, U.-M. O'Reilly, S. Lucas, E. Costa & T. Soule (Eds.), *Genetic programming* (Vol. 3003, pp. 389–398). Lecture Notes in Computer Science. Springer Berlin / Heidelberg.

Whitley, D. (1994). A Genetic Algorithm Tutorial. *Statistics and computing*, *4*(2), 65–85.

Whitley, D. (2001). An Overview of Evolutionary Algorithms: Practical Issues and Common Pitfalls. *Information and software technology*, *43*(14), 817–831.

Index

T

U

Appendix A

Extended Abstract

Effects of Modifying the Expressive Power of the Language of Child Producing Programs in Autoconstructive Genetic Programming

Neil Singh (Y4628661)

Extended Abstract of Open University MSc Dissertation Submitted 6th March, 2012

Introduction

Genetic programming (GP) is a computer science technique for automatically generating programs that solve a given problem. This project focuses on a form of GP called *autoconstructive GP* (AGP).

GP is inspired by biological evolution and potential solutions (programs) are known as *individuals*. It begins by creating a population of random individuals. Then the initial population is evolved by *genetic operators* that reproduce and mutate individuals, and some of the *fittest* individuals are *selected* to survive to the next population. This evolutionary cycle continues until an adequately fit individual is found or a fixed number of cycles passes.

The choice of genetic operators is set by a GP system's parameters which are typically predetermined by a GP researcher based on the results of preliminary experiments. This is done because a GP system's performance depends on the combination of its parameters and the problem being tackled. However, due to the large number of possible parameter values and complex interactions between parameters, the preliminary experiments can be laborious and are not guaranteed to result in optimal parameters. Consequently, some researchers have investigated automating parameter setting, and AGP was developed as a result.

In AGP, instead of using pre-determined genetic operators, individuals are responsible for problem solving and for producing their own children. Each individual has an in-built genetic operator whose mechanism is evolved along with the individual, the idea being that the genetic operators will evolve to suit the problem.

Implicit in AGP is the concept of *reproductive competence*. Early on in an AGP run, not all individuals will successfully reproduce and random individuals have to be created to fill the next population. Accordingly, an AGP system is said to be reproductively competent when its individuals can fill the next population with children and no random individuals need to be created. Preliminary work, carried out by other researchers, suggests that AGP systems first have to achieve reproductive competence and only then do they begin to evolve individuals to solve the problem.

This project investigates the effects of modifying the expressive power — the set of enabled instructions in a program's programming language — of the child

producing programs of individuals in an AGP system. The hypothesis is that the expressive power determines the space of all possible child producing programs, and, consequently, it should be possible to reduce the expressive power to change the space to contain a higher proportion of competent child producing programs.

Method

The project's research method involves carrying out a set of controlled laboratory experiments in the form of computer simulations. An AGP system is implemented that runs in one of three *execution modes* and which is set to solve one of two problems (*problem 1* and *problem 2*). Both of the problems task a system with evolving programs that output specific values for predefined inputs.

In execution mode 1, individuals are represented by a single program responsible for problem solving and child production. It is this mode which most closely resembles the operation of AGP systems of research works that inform this project. In modes 2 and 3, individuals are represented by two programs; one for problem solving and another for child production. Furthermore, mode 3 reduces the expressive power of the language of child producing programs by disabling instructions identified to be least frequently used by competent child producing programs in an execution of the AGP system in mode 2 and on problem 1.

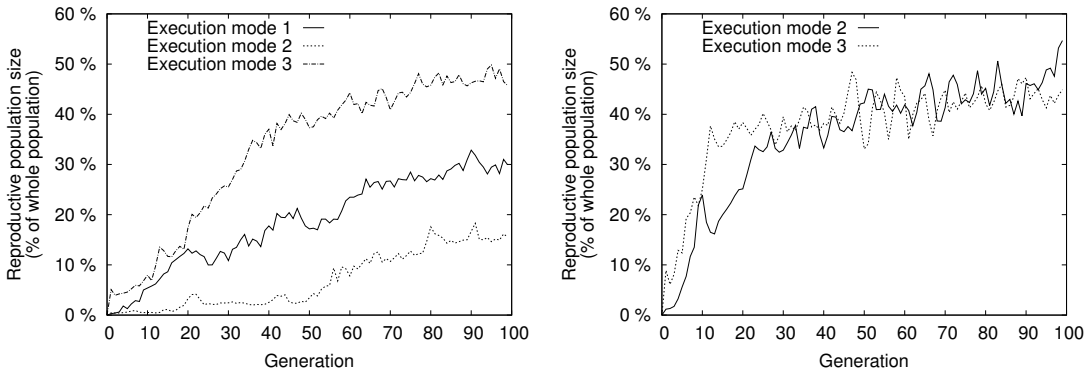
Three experiments are carried out. The first investigates the side effects of representing individuals with two programs instead of one. The second investigates the effects on reproductive competence caused by reducing the expressive power of child producing programs. The third attempts to generalise on the results of the second experiment by repeating that experiment except with another problem; the first and second experiments tackle problem 1, the third tackles problem 2.

Results

The following figures plot a measure of reproductive competence — the percentage of individuals in a generation's population that successfully produce children — against the generation number for experiments 1 & 2 (left) and experiment 3 (right).

The results of experiments 1 and 2 show that the switch from one program per individual (execution mode 1) to two (execution mode 2) harmed reproductive competence, but that the loss was more than made up for by reducing the expressive power of child producing programs (execution mode 3).

The results of experiment 3 show little difference between execution modes 2 and 3.



Plots of reproductive population size versus generation for experiments 1 & 2 (execution modes 1, 2 & 3 on problem 1, left) and experiment 3 (execution modes 2 & 3 on problem 2, right). Values averaged across 20 independent runs for each execution mode.

Analysis

The primary conclusion drawn from the project's results is that the expressive power of the language of child producing programs does affect the reproductive competence of an AGP system. Furthermore, the results hint that the space of reproductively competent child producing programs may be dependent on the problem being tackled, and this is why no difference was seen between execution modes 2 and 3 of experiment 3.

However, the project's conclusions are tentative because, due to practical limitations, too few runs of the AGP system were carried out for the results to be statistically significant.

Discussion

The implication of the project's tentative conclusions is that the expressive power of the language of child producing programs in an AGP system can be considered as a system parameter which can be tuned to improve performance, but that the parameter needs to be tuned on a problem by problem basis.

The project's outcome is interesting but more work is needed to substantiate and expand on it. Future research areas could involve investigating the relationship between the space of reproductively competent programs and the problem being solved; whether classes of problems exist for which the space of reproductively competent programs is similar; and whether the expressive power could be reduced in a dynamic and self-adaptive way.

Appendix B

Restricted Version of Push

This section lists the results of the analysis of M801AGP in execution mode 2 on problem 1 that determined the restricted version of Push which was used for execution mode 3 (see Section 5.3).

Tables B.1 and B.2 list all of the Push instructions and their associated scores in descending order. Table B.1 lists the top 75% (107 of 142) of instructions that were left enabled in the restricted version of Push, and Table B.2 lists the bottom 25% (35 of 142) of instructions that were disabled.

Instruction	Score	Instruction	Score
INTEGER.RAND	1568732	CODE.DEFINITION	166722
EXEC.IF	987926	CODE.FROMFLOAT	157780
CODE.CONS	852829	EXEC.S	147867
CODE.RAND	812313	FLOAT.=	118685
NAME.RAND	769966	BOOLEAN.RAND	112662
CODE.FROMNAME	736888	CODE.DO*RANGE	103066
NAME.POP	717024	FLOAT.YANKDUP	101015
NAME.ROT	714937	NAME.STACKDEPTH	92997
CODE.YANKDUP	690150	CODE.SWAP	88035
EXEC.DO*RANGE	635649	INTEGER.SHOVE	87617
CODE.QUOTE	623027	CODE.NOOP	85875
INTEGER.DUP	579754	EXEC.=	85370
CODE.YANK	539041	EXEC.STACKDEPTH	85291
FLOAT.SWAP	528533	NAME.FLUSH	83974
FLOAT.<	507948	FLOAT.ROT	80822
CODE.ATOM	448773	EXEC.DEFINE	80464
FLOAT.-	440698	BOOLEAN.AND	78561
FLOAT.DUP	437517	CODE.FROMINTEGER	76225
CODE.INSERT	433293	CODE.EXTRACT	75388
INTEGER.FLUSH	395595	INTEGER.=	72630
CODE.DUP	384801	BOOLEAN.OR	72328
EXEC.POP	384325	INTEGER.*	72082
EXEC.YANKDUP	378030	NAME.SHOVE	71801
INTEGER.FROMBOOLEAN	374924	FLOAT.%	71209
NAME.SWAP	374666	BOOLEAN.NOT	67794
CODE.STACKDEPTH	374448	CODE.POP	63405
NAME.YANK	365857	NAME.QUOTE	60948
BOOLEAN.YANKDUP	361975	CODE.ROT	58686
CODE.NULL	360997	FLOAT.SHOVE	57544
CODE.INSTRUCTIONS	353276	BOOLEAN.FROMINTEGER	55420
FLOAT./	320571	EXEC.DUP	55056
BOOLEAN.=	307942	CODE.CDR	53392
CODE.DO*TIMES	305776	CODE.NTH	51885
FLOAT.FLUSH	303071	INTEGER.>	51716
CODE.DO*COUNT	300543	EXEC.SWAP	51225
INTEGER.MAX	289787	BOOLEAN.POP	49809
INTEGER.%	289494	BOOLEAN.YANK	49123
CODE.SIZE	287693	FLOAT.MIN	48740
FLOAT.COS	286821	CODE.FROMBOOLEAN	48638
FLOAT.STACKDEPTH	286625	FLOAT.*	48622
BOOLEAN.DEFINE	285159	INTEGER.DEFINE	48217
CODE.FLUSH	281550	CODE.LENGTH	48088
NAME.=	278394	BOOLEAN.SHOVE	46840
CODE.IF	275383	EXEC.ROT	44010
CODE.DISCREPANCY	272946	BOOLEAN.ROT	43943
CODE.POSITION	264781	FLOAT.POP	43460
EXEC.K	262077	INTEGER.YANK	41760
CODE.CAR	246351	INTEGER.+	41463
EXEC.FLUSH	238723	FLOAT.MAX	40752
INTEGER.ROT	238203	INTEGER.SWAP	40486
NAME.RANDBOUNDNAME	234985	FLOAT.FROMBOOLEAN	40382
FLOAT.FROMINTEGER	228260	BOOLEAN.FLUSH	38803
NAME.YANKDUP	203347	BOOLEAN.SWAP	37003
INTEGER.POP	198887		

Table B.1: Top 75% (107 of 142) of instructions that were left enabled in the restricted version of Push. The scores were derived from an analysis of the populations generated by M801AGP in execution mode 2 on problem 1. The scores indicate how frequently instructions were used by reproductively competent individuals in their child producing programs.

Instruction	Score	Instruction	Score
FLOAT.TAN	36642	CODE.NTHCDR	18717
FLOAT.YANK	36471	FLOAT.+	18235
EXEC.Y	36253	INTEGER.-	17078
BOOLEAN.FROMFLOAT	34340	EXEC.DO*COUNT	15853
CODE.APPEND	33751	FLOAT.SIN	15809
CODE.LIST	33640	CODE.SHOVE	15608
EXEC.DO*TIMES	32180	BOOLEAN.DUP	11643
INTEGER.STACKDEPTH	30921	INTEGER./	11344
CODE.DEFINE	28989	EXEC.YANK	10119
INTEGER.YANKDUP	28855	INTEGER.<	9496
INTEGER.MIN	28225	BOOLEAN.STACKDEPTH	9025
FLOAT.DEFINE	27607	CODE.DO	5148
EXEC.SHOVE	26860	CODE.CONTAINS	4368
CODE.SUBST	25844	CODE.DO*	3343
NAME.DUP	25832	CODE.MEMBER	3223
INTEGER.FROMFLOAT	25522	CODE.CONTAINER	2703
FLOAT.>	24092	CODE.=	2419
FLOAT.RAND	23928		

Table B.2: Bottom 25% (35 of 142) of instructions that were disabled in the restricted version of Push. The scores were derived from an analysis of the populations generated by M801AGP in execution mode 2 on problem 1. The scores indicate how frequently instructions were used by reproductively competent individuals in their child producing programs.

Appendix C

Online Publication of Source Code and Data

The source code for M801AGP — this project's Common Lisp implementation of an autoconstructive genetic programming system — and the data generated by the project's experiments are (from the 13th March, 2012) available for download from `http://sites.google.com/site/aboutneilsingh/m801`. The download can be used for non-commercial educational and research purposes.

The download is provided for two reasons. Firstly, to allow independent verification of the project's methods and results, and secondly to allow researchers to reuse and build upon the project's implementation.