Empirical Analysis of Schemata in Genetic Programming

by

Will Smart

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2011

Abstract

Schemata and buiding blocks have been used in Genetic Programming (GP) in several contexts including subroutines, theoretical analysis and for empirical analysis. Of these three the least explored is empirical analysis. This thesis presents a powerful GP empirical analysis technique for analysis of all schemata of a given form occurring in any program of a given population at scales not previously possible for the kinds of global analysis performed.

There are many competing GP forms of schema and, rather than choosing one for analysis, the thesis defines the *match-tree meta-form of schema* as a general language expressing forms of schema for use by the analysis system. This language can express most forms of schema previously used in tree-based GP.

The new method can perform wide-ranging analyses on the prohibitively large set of all schemata in the programs by introducing the concepts of *maximal schema*, *maximal program subset*, *representative set of schemata*, and *representative program subset*. These structures are used to optimize the analysis, shrinking its complexity to a manageable size without sacrificing the result.

Characterization experiments analyze GP populations of up to 501 60-node programs, using 11 forms of schema including rooted-hyperschemata and non-rooted fragments. The new method has close to quadratic complexity on population size, and quartic complexity on program size. Efficacy experiments present example analyses using the new method. The experiments offer interesting insights into the dynamics of GP runs including fine-grained analysis of convergence and the visualization of schemata during a GP evolution.

Future work will apply the many possible extensions of this new method to understanding how GP operates, including studies of convergence, building blocks and schema fitness. This method provides a much finer-resolution microscope into the inner workings of GP and will be used to provide accessable visualizations of the evolutionary process.

Acknowledgments

Thank you to my supportive family (yes Frida, new members too) and chiefly my wonderful Jen. Many thanks to my father Robin for much appreciated help preparing the final document. And many thanks to my mother Janet for help preparing of the final student.

Very special thanks to my supervisors Mengjie Zhang and Peter Andreae whose wise councel and unflagging support has been crucial to the completion of this thesis.

Contents

Chapter 1

Introduction

Genetic programming (GP) [41] is an automatic learning method used to obtain computer programs which approximately solve a given task and which are the progeny of successive populations of ancestors using random programs as a starting point. GP is an evolutionary algorithm derived from *Genetic Algorithms* (GAs) and *automatic programming*. It is the focus of this thesis.

1.1 Motivators of this thesis

A range of motivators inspired this particular topic. Firstly, I chose the topic of genetic programming for its flexibility and power.

GP shows promise

Since its inception GP has been used with considerable success on a wide range of machine learning problems due to its powerful evolutionary search, flexibility of configuration and expressive representation of solutions. GP now has several devoted international conferences and, from designing robots [13, 122] to schedules [72, 7], GP has met considerable success in attempts to solve a great many machine learning tasks.

GP works as an evolutionary algorithm applying *genetic operators* successively to populations of *genetic programs*, directed by a *fitness function* which measures how good each program is. These three italicized main components produce many parameters, both numeric and algorithmic, which grossly affect the performance of GP for any given task. For GP researchers, this process of setting many parameters is both a blessing and a curse giving both flexibility and at times excessive dimensionality.

Thus one thesis motivator is the promise of GP and another is that we don't know very well how to improve it.

We would like to better understand genetic programming

Genetic programming, like many other machine learning paradigms, increases in power from year to year by incremental improvements to the way it works. Often the relevant researcher bases these improvements on instinct alone. Sometimes, however, it is the results of analysis of GP that motivates improvements to GP. In the former, improvements share similarity with blind luck. In the latter, improvements could be considered to follow more sound science. Analysis of GP is therefore very desirable in the process of improving GP.

In order to analyze GP a researcher might look at the fitnesses of programs in evolution and their lineage. Alternately given a set of programs the researcher may want to determine *why* the best program is the best and the worst is the worst. In doing so the researcher would look to a finer grain than the program itself by looking for subprograms or patterns within each program. In GP, these patterns are *schemata* (note that the words "schemata" and "schemas" are interchangeable).

There are a number of methods to analyze schemata in theoretical evolutions but far fewer for empirical analysis. This thesis will do the latter by creating a powerful method for the empirical analysis of GP schemata. Thus three major thesis motivators are the desirability, difficulty and scarcity of this kind of analysis tool in GP.

There are few empirical GP schema analysis methods

Much past research analyzes GP by looking at the propagation of schemata through GP evolution and, of this past research, most has focused on schema theory. Through research in schema theory the research community has a variety of complicated formulae predicting the multiplicity of a particular schema or the value of a particular population measurement, based on information from the given current population.

But GP is stochastic. Evolution forms an initial population randomly, then repeatedly processes it by partly-random operators, before producing the final output. Analysis of such a system theoretically is complex. Schema theory has typically overcome this complexity by producing equations predicting one generation in advance, which must be repeatedly evaluated to predict further in advance, with the complexity of the evaluation increasing markedly with each generation. Often the analyses have been forced onto toy problems or systems of limited practical use for instance infinite populations [67] or linear trees [65].

Empirical analysis bypasses many of these problems by reporting on the propagation of schemata as seen in GP evolution. However empirical analysis of GP schemata comes with one large problem of its own.

Numbers of schemata in GP populations make global empirical analyses impractical

With previous methods, if a researcher wishes to perform an analysis on the set of all schemata that occur in any program or programs of a given set then they need to keep the form of schema simple or the size of the set of programs small. There have been a number of tools developed to analyze GP schemata empirically. But for what may be considered “interesting” schemata, for instance fragments or hyperschemata, the number of distinct schemata in a population of programs makes this kind of global analysis of schemata impractical.

For example, there are around $O(2^{2^d})$ distinct schemata of the commonly used form *fragment* occurring in a typical single full binary program tree of depth d. Such a tree of depth eight may contain about 2×10^{45} distinct schemata of that form and a tree of depth nine would contain orders of magnitude more. Previous empirical research along these lines has employed two techniques to avoid this complexity problem:

- Use a restricted form of schemata and/or scale of GP system, then perform this kind of global schema analysis.

  Such research is by its nature suited to very small scale GP systems or forms of schema that are possibly of limited interest. An example is study of numeric terminals [130, 131].

- Analyze a small subset of the set of all schemata. An example is the analysis of only the best-of-run program [48].

  Such research relies on sampling, random or otherwise, and the researcher's prior knowledge. When dealing with an understudied research area such as GP schemata the prior knowledge may be hard to come by.

  Research falling into this category also depends on a large enough sample size. The space of GP schemata, however, is heavily biased toward large schemata, making even very large sample sizes too small. For instance if we were to randomly sample fragments from a uniform distribution in our depth eight binary tree then the number of samples would need to be very large to for us to sample even one fragment with less than five nodes.

This thesis presents a method for global empirical analysis of complex schemata in non-toy sized GP populations.

There are many possibilities for what a GP schema may look like. Thus the final thesis motivator is the unfortunate multiplicity of GP forms of schema.

There is no language to represent a form of schema

GP schemata take many forms; from time to time GP schemata have been defined as: numeric constants [18, 130, 131], subtrees [105], clipped subtrees [6], sets of subtrees [41], fragments [61], unordered-fragments [114], multi-sets of fragments [70], hyperschemata [85] and others. A researcher about to embark on schema research and eager to properly cover the space of schemata faces the daunting prospect of writing algorithms and code for all these forms of schema and more as they come up.

Therefore this thesis defines the *match-tree form of schema language*, a language able to express forms of schema precisely. Most previously used forms may be expressed in this language.

1.2 Goals of this thesis

The first goal of this thesis is to:

> *Provide a method to perform efficient and useful analysis on all the schemata occurring in an input set of genetic programs.*

Achieving this goal is of great interest to the GP community. Such an analysis method would be used to test systems in practice and complement the theoretical knowledge of schema theory.

This goal has some further qualifications:

1. Usefulness: Some of the analyses enabled by the method are useful to researchers and the research community.

2. Globality: The method provides analysis on all schemata of the given form in the given programs. The result is not dependent on random numbers or selective sampling.

3. Complex forms: The method provides analysis on complex and interesting forms of schemata.

4. Larger-than-toy scale: Though no attempt is made to analyze large populations, the method should be able to analyze all schemata in 100 seven-deep programs. The analysis should be available on real world, non-toy problems.

Achieving this higher goal will involve achieving subsidiary goals:

- Unfortunately, there are many forms of GP schema in the literature and most are incompatible with each other. Thus in order to provide coverage to analyze a wide range of forms of schema, a subsidiary goal defines our use of "schema" by defining a language to describe "forms of schema":

  > *Provide a language for forms of GP schema unifying common forms of GP schema.*

  and

  > *Build a new analysis method to be compatible with this form of schema language.*

- A subsidiary goal to test the efficacy of the analysis method:

  > *Characterize the method by using it for GP analysis in a range of situations.*

The second goal is to develop an implementation of the new method in C++.

1.3 Contributions

This thesis presents a range of new methods, analyses and structures which are important contributions to the GP community by being newly introduced in this thesis, and being potentially very useful to the researchers and users of GP:

- The thesis defines a very general form of schema (the *Match-tree Form of Schema*), and language for specifications of this form of schema (the *Match-tree form of schema language*). These allow both far greater precision, to the point of machine-readability, in definition of forms of schema and more easy comparison of systems conforming to the schema form language.

- The thesis introduces the concept of *maximal schemata* and algorithms using them for the empirical analysis of the relationship between genetic programs of any given population and the schemata they match. The empirical analysis is exhaustive, providing a deterministic method to represent each schema in the population with the set of programs it occurs in. Together the algorithms provide such exhaustive empirical analysis at a much larger scale than previously possible, allowing GP researchers to analyze larger scales of evolution without the selection of schemata that was previously required.

- An implementation of these algorithms in C++ has been made available to the GP community.

This thesis has shown how to perform empirical analysis on schemata in GP populations at a scale previously prohibitive for many types of analysis. Previous approaches were typically hard-coded to one or two forms. By using the match-tree form of schema language, analysis by the new method is applicable across a wide range of forms of schema, allowing experiments showing that different forms have widely different characteristics.

1.4 Structure of this thesis

After this introduction this thesis presents an overview of relevant background to machine learning, GP and GP schemata. Chapter 3 defines the match-tree form of schema and the match-tree form of schema language. Chapter 4 introduces the core idea of the analysis method: representative pairs. It also presents algorithms which use these representative pairs to answer analysis questions about a GP population. Chapter 5 presents algorithms to find the representative pairs for a large number of the forms possible using the match-tree form of schema language.

Chapter 6 presents the results of verification experiments. Chapter 7 presents the results of experiments showing the usefulness of the method in practice. Chapter 8 presents the conclusions of this thesis, and gives the research directions I would like to see explored in future research.

Chapter 2

Background

This chapter presents the current state of the art in this thesis' problem domain.

The chapter starts with a very broad overview of machine learning and evolutionary algorithms. It then presents summaries of relevant current work in genetic programming (GP), schemata in genetic algorithms and GP and with most specificity empirical work on schemata in GP.

2.1 Overview of machine learning

This thesis defines a tool for the analysis of genetic programming, which is an evolutionary algorithm and a machine learning method.

At a high level, an intelligent machine could be expected to modify its behaviour given new information about its environment or information about a change in its environment. Machine learning is the computing discipline to allow this intelligent modification of machine behaviour, based on environmental input. Machine learning may be seen to derive high-level "knowledge" about a given set of data and use that knowledge to make decisions, although it must be noted that this high-level knowledge may not be in any way human-readable.

2.1.1 Information given to the algorithm

While the problem space for machine learning is vast, the overall data flow is most typically one of the following:

- Supervised learning.

  The input to a supervised machine learning algorithm is a set of examples of both input and corresponding desired output. Using these examples it must be able to produce output of a similar quality given new input.

  For example, a supervised machine learning algorithm may be given one hundred waveforms of Bob speaking and one hundred of Sally speaking. For each it is told who was behind the microphone. Given a new waveform it must then recognise who is speaking.

- Unsupervised learning.

  Unsupervised machine learning algorithms are only given input data and produce output by exposing a particular structure in the data. The input data instances are not labelled or predefined.

  For example, the trainer gives an unsupervised machine learning algorithm one hundred waveforms of Bob speaking and one hundred of Sally speaking. By identifying that half of the input waveforms featuring slower speaking tone and speed (those of Bob), the algorithm may associate a new (low toned) waveform with Bob's previous waveforms.

- Reinforcement learning.

  Reinforcement algorithms depend on feedback about their performance. The algorithm must accumulate the feedback over time, analogous to learning through doing.

  For example, a reinforcement machine learning system controls an elevator. Though the elevator algorithm is not told directly which floor it should have chosen to go to last, the system provides the elevator algorithm with feedback on the quality of its choices.

Hybrid

Naturally there are a great many combinations of the above strategies.

2.1.2 Data partitions for supervised learning

Typically supervised machine learning systems train using a dataset partitioned into a *training set* and a *test set*. Sometimes a *validation set* is also used. The machine learning algorithm receives three types of input.

- Training set.

  The training samples are patterns with both the input data to be recognised and the respective desired output data. The training set provides the learning algorithm with a measure of how good any prospective solution is. The final solution should perform well on the training set. A supervised learning system will typically tend to try to exploit the previously tested prospective solutions which performed well on the training set.landscape which - making up the training set.

- Validation set.

  The validation set, if used, is logically part of the training set and aims to avoid *over-fitting* which is analogous to a student rereading past exams and solving them with precision then proceeding to fail on the new material of the exam. This part of the training set is kept at arm's length to the training process, although not as strictly as the test set. Performance on the validation set is a useful indicator of how well the algorithm will do on the test set rather than the specific items it must learn.

- Test set.

  The test samples also have both input data and desired output defined but are used differently to the training samples. The learning algorithm may only use the desired output for each pattern to test the quality of a prospective solution. The measure of quality is never used as part of the search for better solutions but rather as a test measuring the quality of the solutions.

- Eventual use.

  The eventual end of the system is patterns provided by the real world for which there is input data but no known desired output.

2.2 Main learning paradigms

The following are four of the many paradigms of machine learning:

- Case-based [1].

  Case-based machine learning algorithms compare training data directly with test data, using flexible matching methods. Popular case-based machine learning methods are *nearest neighbour* [16] and *nearest centroid*.

- Connectionist [23] – a form of hill-climbing search.

  Connectionist methods represent their solution as a network of nodes and edges. Starting from an unrefined network, the learning algorithm refines the weights on the edges or parameters associated with the nodes. Input samples, when passed through the refined network, will produce close to the desired output.

  The most common connectionist learning method is *neural network* [10].

- Evolutionary [115] – a form of hill-climbing beam search based on the Darwinian principle of natural selection.

A number of *individuals* are kept in one or more *populations* with the names belying the biological inspiration for this paradigm. A *fitness function* obtains the *fitness* of a given individual, which is a measure of how "good" the individual is at producing the desired output for each input sample in the training set.

The algorithm aims to improve the quality of successive populations by applying *genetic operators* to the individuals of previous populations and keeping those new individuals which have improved fitness.

This thesis' topic, *genetic programming*, descends from *genetic algorithms* [34] which is an evolutionary algorithm using evolutionary search.

2.3 Overview of evolutionary computation

Evolutionary Computation (EC) has its origins in the 1950s [22]. This section describes features of EC relevant to its specialization GP.

2.3.1 General concepts

EC passes a number of concepts down to GP:

- Individuals: a candidate solution of some form which may transform input samples into something like the desired output.
- Fitness: a measure of how good a given individual is at transforming the training set samples into their desired outputs.
- The population: a set of individuals.
- Genetic operator: an operator which produces one or more individuals, given one or more individuals. Genetic operators create new in-

dividuals semantically close to previous individuals that were found to perform well on the training data.

- Selection mechanism: a mechanism which selects individuals based on fitness. The selected individuals are passed as input to the genetic operators. Selection mechanisms typically give greater numbers of offspring to the fitter individuals which is basic to Darwinian natural selection.
- Generation: the period over which the system produces a new population by applying genetic operators to the previous population.
- Evolution: the period of the entire process until the final generation.

2.3.2 General algorithm

The general algorithm of EC is as follows:

1. Initialization: a population is made from scratch. Typically the individuals in the population are randomly created, although they may also incorporate material of previous evolutions or prior knowledge on the part of the user.
2. Refinement: successive populations are created, each from its predecessor.

   Each step of the refinement has several steps:

   - Evaluation: the fitnesses of individuals in the population is assessed.
   - Selection: the better individuals are passed to the genetic operators for mating.
   - Application of the genetic operators: a new population is formed from the output of the genetic operators given the selected individuals as input.

3. Final solution individual: the evolution will be brought to an end according to some *early-stopping criteria*. An example is to end the evolution after some number of generations show no improvement in best individual fitness. Typically the very best fitness individual found during the run or in the final population will be used as the output of the EC algorithm as a whole.

2.3.3 Aspects of EC

EC provides a very basic algorithm and is typically specialized for actual use. This section lists some parameters common to most implementations of EC variants and common settings.

Representation of individuals

Each individual instantiates an entire prospective solution to a particular problem. Therefore the representation used to express individuals, which are the candidate solutions, has often been a focus of research aiming to improve EC as a whole. A change in the representation of individuals has flow on effects to the EC algorithm's genetic operators and its method to derive fitness from individuals.

Variously, individuals may be represented as vectors of real numbers as in evolutionary programming [22] and evolutionary strategies [22], vectors of bits as in genetic algorithms [34], graphs of nodes [55, 21], trees of nodes as in tree-based GP [41], or lists of instructions as in linear GP [11].

Fitness function

The EC fitness function derives a fitness from an individual by comparing the evaluated output of the individual for each pattern in the training set and the respective pattern's desired output. Often the fitness function returns the fraction of correctly identified samples or a representation of distance from the ideal output (in which case it is referred to as an *evaluation function*).

The design of a fitness function is crucial to the performance of any EC system using it. The combination of problem and fitness function forms a *fitness landscape* as the space of fitnesses for all possible programs. Some such landscapes are smooth with a clear local indication of how to improve any proposed solution.

Parent selection mechanism

The following are often used to select parents in EC systems:

- Roulette wheel selection/Proportional selection [41] - each individual has a probability of being selected in proportion to its fitness.
- Rank selection [41] - each individual has a probability of being selected in proportion to its rank. That is for N individuals, the program ranks count from one for the worst to N from the best.
- Tournament selection [41] - selection is a two step process: first it samples a set of individuals from the population randomly with no reference to fitness into the tournament. Next the individual of the tournament with best fitness is chosen.

While these mechanisms are for *parent-selection* some systems also have *child-selection mechanisms* which select which generated children are kept for the next generation.

Genetic operators on individuals

Though intrinsically linked to the representation of individuals used, typically there are three genetic operators:

- Reproduction or *elitism* - reproduction copies individuals unchanged to the next population. With use of reproduction of the best individual it may be ensured that the best individual in the next generation is not worse than that of the current generation.

- Mutation - one parent is partially altered in a random way to produce one child for the next population. Mutation introduces new material and is often used to increase population *diversity*.
- Crossover - two or more parents mix to produce offspring. No new random material is introduced as with mutation. Crossover may combine the good elements of existing individuals to produce better child individuals.

2.3.4 Common forms of EC

There are several main offshoots of EC, each being a discipline in its own right.

- Genetic algorithms (GAs) [34] - GAs is an EC method originating in the 1960's in which the representation of individuals, *chromosomes*, is as vectors. Typically chromosomes are fixed-length vectors of bits or *bit-strings*.

  In GAs, mutation and crossover typically preserve the position in the chromosome of parts of the chromosome passed from parent to child, thus crossover or *recombination* involves choosing the source parent for each element of the chromosome and mutation involves choosing whether to randomly assign each element.

  A key feature of GAs is the high emphasis placed on crossover.

- Evolutionary Strategies (ES) [22], Evolutionary Programming (EP) [22] - both originated around 1964 in parallel from different communities.

  The representation of individuals used in both ES and EP is as vectors of real numbers. ES uses these vectors directly as solutions, for instance as parameters to a model. EP interprets the solutions as finite-state-machines.

Both ES and EP have mutation as the key genetic operator with crossover used less or not at all.

- Genetic Programming [41] - described in the next section.
- EC also has other forms, including learning classifier systems [112], swarm intelligence (particle swarm optimization [15], ant colony optimization [20], artificial immune systems [19]), and others.

2.4 Overview of genetic programming

Genetic Programming (GP) is similar to a specialization of GAs with the representation of individuals as computer programs or *genetic programs*. But, through a very active research community, GP has become very differentiated from GAs. The precise early roots of GP are hard to distinguish. In 1954, Nils Aall Barricelli used programs as the representation for evolutionary algorithms and during the 1960s, 1970s and 1980s the core ideas and mathematics behind GP were developed by various researchers including: Turing [118], Price [95] and Holland [34]. Researchers to apply early versions of tree-based GP include: Forsyth [25], Cramer [17], De Jong [35] and Bickel and Bickel [9]. Despite this early work, Koza [40, 41] originated many of the standard constructs still found in today's canonical GP.

The broader method of GP has the key feature of complex representations for individuals. Historically these representations described simplified runnable computer programs but the term "numerical expression" is often more apt than "program" since the programs often involve few of the typical constructs found in computer programs; typically these *genetic programs* use no loops or subroutines. However voluminous research has focussed on the best way to add such constructs [69, 14, 54].

2.4.1 Representation of individuals

GP features an ever growing set of representations for individuals. While GAs typically use bit-strings, GP researchers are perhaps freer to invent more complex representations. Despite the name, some modern representations of genetic programs bear no resemblance to computer programs of any sort. Some commonly used representations for genetic programs include:

- Tree [41] - the structure for the original genetic programs was as Lisp S-expressions or trees of functions and terminals, used in *tree-based GP*, and this form is still most popular.

- Grammar tree [123] - distinct from tree-based GP is grammar-based GP. Each individual is a partial derivation tree, which may be used to derive a tree-based GP genetic program.

- Linear [11] - each individual in Linear GP (LGP) is a sequence of instructions, similar to those of assembly language. LGP uses a *register set* as both input and output.

- Graph [55, 21, 66] - each individual is a graph of functions and terminals, more generally than the trees of tree-based GP. LGP genetic programs may also be interpreted as graphs.

- Final representation - recently, GP researchers have been foregoing the step from genetic program to final output and have been evolving genetic programs of the final representation directly. An example is the evolution of electronic circuits [43].

This thesis focuses on tree-based GP only. The sections after this one and the remaining thesis, exclusively reference tree-based GP.

2.4.2 Tree-based GP individuals

A genetic program in tree-based GP is a tree of *functions* and *terminals*. Functions are the internal nodes of the tree and terminals are the leaf nodes of the tree. The tree structure of a genetic program means that any number of inputs may be arbitrarily processed by component functions and one output produced. The GP system may evaluate the tree in the same way as a Lisp program is evaluated.

The functions and terminals of a program tree may be evaluated to produce an output. Terminals have no children and their output values depend on stored value, as in the case of numeric terminals, or information from the "world", as in the case of feature terminals. Functions have children and use the outputs of those children to determine their own output. An algorithm may evaluate each node by ensuring children evaluate before parents. The evaluated output of the root node is the output of the genetic program as a whole.

2.4.3 Generation of an individual

There are a variety of ways to generate the tree for a genetic program, including:

- Full - randomly assign the nodes of each level, starting at the root. If the desired depth is d, then all nodes at depths less than d are functions and all nodes at d are terminals.
- Grow - similar to full, but nodes at depths less than d may also be assigned as terminals with a given probability.

2.4.4 Generation of the initial population

There are a variety of ways to generate the initial population of genetic programs, including:

- Half-and-half [41] - this method generates half of the genetic programs of the population using full and the other half by grow.
- Ramped [41] - this method varies the desired depth of the genetic programs linearly from some minimum for the first genetic program to some maximum for the last genetic program.
- Ramped half-and-half [41] - a commonly used combination of "ramped" and "half-and-half"

2.4.5 Genetic operators on individuals

Publications have focussed on improving the genetic operators in GP [47, 133] but the original set of operators is still most common:

- Subtree mutation - produces a single child from the single parent with one randomly selected subtree replaced by a randomly generated subtree.
- Subtree crossover - produces two children from the two parents, each sourced from alternate parents. Crossover first selects a location in the tree of each parent and produces two children each with the main body of one parent with its selected subtree replaced by the selected subtree of the other parent.

2.4.6 Current developments in genetic programming

GP is being actively developed in several ways:

- Co-evolution [127, 27] - uses two or more concurrent evolutions, in which each evolution depends on each other for its fitness function. That is, each population learns off the other.
- Multi-objective [73, 24] - in which the fitness function produces a vector of numbers, rather than a single real-valued fitness.

- Parallelization [26, 44] - GP need not simply repeatedly produce one population from the last. Recent research has focussed on parallelizing GP over multiple populations.

- Variation of parameters during evolution [119, 33] - The optimal setting for each of the many parameters to GP need not be constant through each evolution or between evolutions. Research has focussed on adjusting parameters during evolution.

- Bloat control and removal of underused code [120, 125] - bloat or the tendency of genetic programs in later generations to be larger, has long been a problem in GP.

- Utilizing GPUs (GPGPU) [50, 68] - a new topic in GP is the use of highly parallel architectures such as GPUs which are most often associated with graphics.

- Analysis of GP - to direct the improvements to GP a large amount of recent research focuses on the analysis of GP:

  - Building blocks - hypothesized as a loose entity by Goldberg in 1985 [29], the term building-block has come to mean any one of many concrete structures found in genetic programs. Much recent research uses building-blocks to analyze improvements in performance.
  - Research of schemata and building-blocks in GP [39, 117, 45, 36] - research looks to schema theory to explain the power of GP.

2.5 Overview of schemata in GAs

In the 1970s, researchers started looking to explain the power of Genetic Algorithms. John Holland presented a persuasive argument in his book [34], stating that GAs do not simply sample a set of chromosomes in search-space but perform a far greater parallel search of the patterns common to chromosomes. He called the patterns *schemas*. A common definition for a schema is as a specification of a set of points in search-space that share some syntactic characteristics.

2.5.1 GAs form of schema

For bit-string chromosomes a schema or "similarity template", is a string of symbols taken from the alphabet 0,1,# [34]. The schema represents the set of bit-strings that match the schema symbol by symbol for every symbol in the schema that is not #. So the schema 0101#1# represents the set of bit-strings 0101010, 0101011, 0101110, 0101111.

The number of non-# symbols in schema H is its *order* or $\mathcal{O}(H)$. The distance between the furthest two non-# symbols in schema H is its *defining-length* or $\mathcal{L}(H)$.

2.5.2 GAs schema theory

Typically research on schemata in GAs has been largely theoretical making up *schema theory*. Less numerous has been research identifying or using schemata from GAs chromosomes in runs of evolution.

GAs exhibit inherent parallelism

Holland proposed that GAs derive a great amount of their power by performing a search on not only chromosomes but also schemas. Each schema has an *expected payoff*, similar to a chromosome's fitness, which is defined as the average fitness of all possible chromosomes containing the schema. When assessing a chromosome for fitness, the fitness adds new information about the expected payoff of all schemas it contains. The probability of each schema propagating to the next generation is thus affected by the assessment of the chromosome's fitness, much like the probability of the chromosome contributing to the next generation. Since each chromosome of length N contains 2^N schemas, the assessment of one chromosome's fitness alters 2^N distinct, if not independent, variables, each of which affects the make-up of the next population. Holland termed the simultaneous processing of this large number of schemas *inherent parallelism* [34].

However in a population of chromosomes some schemas are certain to occur more than once. The number of schemas sampled by a population of M chromosomes is not $O(M \cdot 2^N)$ as might be supposed but is in fact only $O(M^3 N)$ [29].

Holland's schema theorem

Holland presented in [34] an equation specifying a lower bound on the expected number of chromosomes containing a schema in the next generation in terms of the number of chromosomes containing the schema in the current generation, the fitnesses of the chromosomes in the population, and other pieces of information obtainable from the population:

$$E[m(H,t+1)] \geq \\ m(H,t) \cdot \frac{f(H,t)}{\bar{f}(t)} \cdot (1-p_m)^{\mathcal{O}(H)} \cdot \left[1 - p_c \frac{\mathcal{L}(H)}{N-1}\left(1 - \frac{m(H,t)f(H,t)}{M\bar{f}(t)}\right)\right]$$

where $m(H,t)$ is the number of chromosomes matching schema H at time t, $f(H,t)$ is the mean fitness of chromosomes matching schema H at time t, $\bar{f}(t)$ is the mean fitness of chromosomes in the population, p_m and p_c are the probabilities of mutation and crossover per bit respectively, N is the length of the chromosomes and M is the number of chromosomes in the population.

In order from left to right the terms of the right hand side deal with: the frequency of the schema in the population, the effects of selection, the effects of mutation and the effects of crossover.

The formula given by Holland gives only an lower bound. Practically, this causes the formula to be of limited use in reliably predicting the expected number of instances of a schema in some cases when the value produced is far below the actual expected value, due to the simple model of evolution used in the formula. This prediction error increases exponentially as predictions are made further into the future. In addition, Holland's schema theorem counts only the genes it disrupts, not the genes it creates, making it less reliable for predicting the actual behaviour of a GAs population.

2.5.3 The building block hypothesis

Building on the Holland's schema results, Goldberg proposed an ambitious theory on how GAs work [29], calling it the *Building Block Hypothesis* (BBH). The hypothesis states that GAs work by combining short, highly-fit schemas of low order called *building blocks* into larger schemas potentially of higher fitness. These larger schemas combine into higher-order schemas and eventually a solution.

Goldberg inferred from schema theory that these high-performance, short-defining length, low-order building blocks would be sampled in the population in numbers increasing at least exponentially.

2.6 Overview of schema theory in genetic programming

Research applying the GA schema theorem to GP started in about 1992. But the variable-length representation of programs in GP gave difficulties not present when dealing with the fixed-length bit-strings typically found in GAs. Early work includes that of Altenberg and Ballard [4, 3], O'Reilly and Oppacher [70], Whigham [124], Poli and Langdon [85, 86] and Rosca [106]. Subsequently, the basic GP schema theorem has been extended in many directions and the following subsections describe some of these variants.

2.6.1 Variants of GP schema theorems

Innovations in schema theory for GP varied along a few basic lines as follows:

Different forms of GP schema

Koza presented GP's first form of schema in his 1992 founding work [42]. He described the form of schema as follows:

> "A *schema* in genetic programming is the set of all individual trees from the population that contain as subtrees one or more specified trees. A schema is a set of LISP S-expressions sharing common features." [42]

An example schema by this definition is shown in figure 2.1(a). A key feature of Koza's definition is that schemas are *complete* subtrees. By way of example for a program to be a member of the schema defined by the single S-expression (IF 1 2 3), it must contain a subtree exactly matching (IF 1 2 3). Incomplete trees such as (IF # 2 3) where # is a wild card and can match any valid subtree, are not allowed.

O'Reilly and Oppacher [70] proposed another more general form of schema that allowed these incomplete subtrees. These schemata were defined as follows:

> "A GP *schema H* is a set of pairs. Each pair is a unique S-Expression tree or fragment and a corresponding integer that specifies how many instances of the S-Expression tree or fragment [a program instantiating *H* must contain]."

where a *fragment* is a program tree with # or *don't care* added to the set of possible terminals. An example of this form of schema is shown in figure 2.1(b).

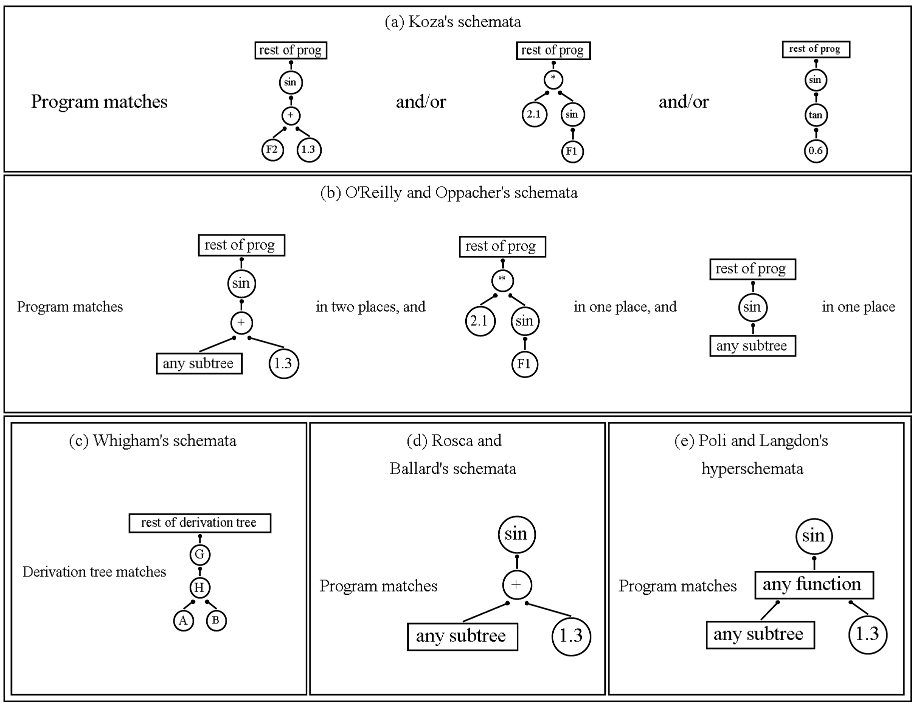

Figure 2.1: Previously defined forms of schemata in GP

While this is a powerful definition of GP schema and allows for a great number of program patterns and potential building blocks to be defined in terms of a set of fragments, producing a schema theorem using this definition of schema proved difficult with only a basic theorem presented in [70]. Note that O'Reilly and Oppacher did not consider program trees to be fragments but defined fragments as requiring at least one node that is a #. In this document this requirement is removed and the set of program trees is a subset of the set of fragments.

Whigham produced a form of schema for his Context-Free-Grammar based Genetic Programming (CFG-GP) in [124]. Each schema is a partial derivation tree for the grammar used to create individuals. An example of Whigham's schema is shown in figure 2.1(c).

Rosca and Ballard in [108] presented a definition of schema for ordinary tree-based GP, based on that of O'Reilly and Oppacher:

> "A *rooted-tree schema* or *tree-schema* of order k is a rooted and contiguous tree fragment specified by k function and terminal labels" [108].

An example schema by this definition is shown in figure 2.1(d). Poli and Langdon in [85] further developed the schema of Rosca and Ballard: where rooted-tree schemas make use of the # node, which can replace any valid subtree, the *hyperschemata* of Poli and Langdon also make use of the = node, which can replace any single node, be it a function or terminal. An example hyperschema is shown in figure 2.1(e).

Different values being measured

Early on in the study of schema theories in GP, the difficulty in finding exact equations describing transmission probabilities of schema led to research into finding other related quantities.

In 1998, Poli, Langdon, and O'Reilly [88] investigated the variance of schema transmission or the certainty from run to run of a particular schema's transmission probability in lieu of the actual probability itself. The same paper investigated the extinction probability of schemas, which was found to be higher than expected for even high-fitness schema. For example, a high-fitness schema has a 50% probability of being broken-up after two generations.

Exactness

Most schema theorems provide only a lower bound on the expected number of instances of a schema in the next generation. But recently exact schema theorems have been developed for various configurations of GP: Most notably, in [77, 80, 90, 89, 91, 83, 92, 79], Poli described exact schema theorems for his *hyperschema* form of schema and various configurations of GP operators.

2.6.2 Current research of schema theory in GP

Over the past few years, several researchers have been active in improving and applying GP schema theory.

In 1997, Haynes [31, 32] attempted to bridge the gap between the relatively sound schema theory of GAs and the more tenuous schema theory of GP, by proposing *phenotypical building blocks* for a graph based GP. The system evolved phenotypical building blocks as an intermediate representation able to produce the actual solution graph. The phenotypical building blocks were closer to the phenotype or behaviour of the program than the genotype or structure of the program which is most often evolved in GP. Haynes concluded that this work reconciled previous GP observations of recombination with the theoretical work of O'Reilly and Oppacher.

Riccardo Poli has co-authored several papers on the subject. In the 1998 paper [87], Poli and Langdon presented the first of a line of schema theorems. The paper developed a schema theorem for hyperschemata using point crossover and point mutation. Poli, Langdon, et al. extended this work considerably, developing new schema theorems for variations of the GP algorithm [76, 75, 82, 90, 80, 78, 81, 92, 82, 91]. Recent extensions include the formulation of exact schema theorems for GP involving subtree crossover, using Poli's hyperschema.

Li et al. [53] present a method for evolving schemas directly. The method encodes each population as an *instruction* matrix from which individuals can be extracted. If an extracted individual has good fitness, it passes its fitness back into this instruction matrix, making extractions of similar individuals more likely in the future. In similar work Shan et al. [111] used a grammar model to guide the construction of population individuals, by using fitnesses of previous individuals. Both methods showed significantly improved performance over canonical GP on the problems tested.

McPhee et al. [65, 63, 64] used recent results in schema theory to find the size bias of subtree crossover and mutation operators on linear program representations. The paper constructed the simple "one-then-zeros" task in which all programs are variable length bit-strings and a program gains high fitness only if it consists of a one followed by all zeros. Subtree crossover was found to increase the size of programs on average and subtree mutation was found to increase or decrease the size of programs toward a point of equilibrium. This research presented a practical way to use schema theory in designing operators but the relevance of results was limited by the simplicity of the representation and task used.

Mitavskiy [67] extended previous work of Poli, Stephens, Wright and Rowe [94], developing schema theory for GP based on Geiringer's Theorem. Geiringer's theorem provides a limit reached by repeated crossover in an infinite population. Mitavskiy provided a version of this theorem for nonlinear GP with homologous crossover in a finite population but no selection. The paper reached no firm conclusions but suggested further work would extend to infinite populations with selection.

2.6.3 The building block hypothesis in GP

Since GP was conceived, researchers in GP have analyzed the predictions of the BBH when applied to the new representation of individuals in GP. There exist several arguments for and against the GP BBH, many of which also apply to the GAs BBH.

Arguments against the BBH in GAs and GP

In [70], O'Reilly formulates a schema theorem for GP and summarizes some previously recognized basic failures of the GAs BBH and some assumptions present in deriving the GP BBH from the GAs BBH:

- As stated in [30, 126] and others, schema theory in GAs fails to take into account the interactions of schema in evolution, something that is core to the BBH, thus the BBH is not supported by schema theory. This applies equally to GP.

- All individuals in a population of GA chromosomes share the same set of features, differing only in the expression of each feature. However GP genetic programs not only differ in the exact set of features used but generate arbitrary high-level features from this set of base features. This means the behaviour of schemas in GP may be very different from the behaviour of schemas in GA as the two may describe quite different structures. This indicates translation of the GAs BBH which is founded on GA schemas, to the GP BBH which is founded on GP schemas, may be difficult.

- The BBH relies on schemas having a stable fitness, but there may be high variance in the fitness of individuals sampling a schema, making it hard to identify the true fitness of the schema from only the few individuals sampled during evolution. In addition, convergence in the population makes it more difficult to find the true fitness of a schema since individuals sampling the schema become more similar in fitness.

- GA schema theory only predicts the next population's make-up, yet is the basis for the BBH which looks forward many generations, predicting exponential growth of some schema. The BBH assumes that the conditions for a schema to grow in one generation will be present for future generations, which is not ensured by the schema theory formulae.

- GP building blocks may only occur at certain times during the run as building blocks, by definition, must be resistant to disruption. For example, a building block in one generation may cease to be a building block in later generations if the programs containing it have decreased in size, increasing the building block's chance of disruption.

- The BBH prediction that schemas may be combined to form schemas of higher fitness relies on independence of subcomponents of the individual. Especially in GP, there is little basis for belief in this independence of sub-solutions in creating a solution.

To make progress in determining the validity of the BBH in GP, empirical evidence must be compiled by measuring schema statistics in actual GP evolutions. This could verify the validity of the schema theory in predicting schema frequencies past the next generation or could show that such predictions are subject to large errors. Such measurements could also measure transition of schemas from parents to children in genetic operators in the context of an evolution.

Arguments for the BBH in GAs and GP

In GAs, Goldberg has proven [29] that the BBH is a natural consequence of schema theory, on the assumption that it can be used iteratively to make long term predictions. The formulae of schema theory simply state that schemas of small order and high fitness increase in the population exponentially as evolution progresses. While schema theorems have typically given only lower bounds on the number of instances of a schema in the next population, exact formulae are available for some GP setups [77, 81, 90, 89, 91, 83, 92, 79], making predictions for the number of instances of a schema in future generations more accurate.

Arguments against the transposition of the GAs BBH to a GP platform typically centre on the different nature of schemas in GP, compared to schemas in GAs. In [85, 84] Poli and Langdon developed a schema theory for GP using one-point crossover and the hyperschema form of schema. They concluded that their schema theorem was a more natural counterpart to Holland's schema theorem for GAs [34] than previous GP schema theorems. They also concluded that some difficulties found applying the BBH to GP with standard crossover may be alleviated when using one-point crossover.

2.6.4 Theoretical research in building blocks in GP

Ryan et al. [109] proposed the existence of competitive rooted building blocks. In contrast to the building blocks of the BBH which do not generally compete with each other, these blocks are of high fitness only when occupying the root node of a program and so must compete with each other for this position.

Sastry et al. [110] developed a theory on the supply of building blocks in GP. They developed formulae giving lower limits on the minimum program depth and population size in order to guarantee expression of all building blocks, using formulae giving the *expression* of building blocks in the simple "ORDER" task. An expressed building block in an individual was defined as a block placed in such a way as to contribute positively to the fitness of the individual.

Diada et al. [18] reduced the complexity of building block analysis in GP by considering only the very simplest of potential building blocks: Ephemeral Random Constant terminals (ERCs). They concluded that despite their simplicity, the ERCs did exhibit characteristics of building blocks. But they found that the utility of each ERC varied considerably during evolution and concluded that the nature of GP building blocks is quite different from their DNA analogue with GP building blocks far more transitory in their use.

Other research has used building blocks as a model in determining why some problems are difficult to solve in GP [52, 51].

Poli and Stephens [93] presented a thorough generalization of the *Building Block Basis* (BBB) of GAs to GP. The tensor mathematical construct was used as a more elegant way to deal with the highly complicated formula involved in the schema theory, leading to a formulation of building blocks as pairs of conjugate schemata.

There remain reasons to believe building blocks propagate in GP populations in a way similar to predictions of the BBH. But there are also several persuasive arguments why GP could not support building blocks that act in this way. There continues to be a lack of solid evidence for or against the existence and nature of building blocks in GP.

2.6.5 Current issues with schema theory in GP

Schema theory is provably correct in its predictions: given information about the current population, it can predict properties of schema frequencies in the next population, after the effect of genetic operators. But there are several major issues with relying on schema theory to predict the make-up of future populations:

- Even exact schema theorems can only predict the expected frequency of a schema as it progresses from generation to generation. But the pressures on the schema, for instance its fitness in relation to other schemata in the population, may change from generation to generation in unexpected ways and many of these factors are terms of the schema theorem formulae. Thus even exact schema theorems cannot reliably make exact predictions of the frequency of a schema past the next generation.

- There is a trade-off between complexity of the model of GP used and ease of deriving and using the schema theorems for the model of GP. Simple schema theorems tend to use models of GP simplified by for example removing mutation [74, 80, 75], using restricted representations [65, 63, 90] or only finding a lower bound on the number of instances of the schema [34, 75]. More interesting GP models produce complex schema theorems that are difficult to produce and use.

- Schema theory is hard to apply to new variants of the GP algorithm; each change to the algorithm must be matched with careful analysis of the mathematical basis for the change. Unusual variants of the algorithm may have no easily derived schema theorem at all.

2.7 Empirically identifying or using schemata in GP

In a general sense, all GP systems actively use schemata as an essential part of their algorithm; crossover can be seen as combining GP schemata more naturally than combining genetic programs since some parts of the genetic programs are thrown away. Similarly mutation may be seen as adding to a GP schema. But beyond this ubiquitous use of GP schemata, little GP research has involved their explicit identification and/or use.

There may be many reasons to identify schemata in GP:

- To measure things about the evolution, population or genetic program in order to analyze GP
- To measure things about the evolution, population or genetic program in order to improve GP
- To use the schema directly as a kind of program subroutine

2.7.1 Using Schemata to improve or analyze GP

As early as 1994, O'Reilly et al. [71] were defining *building block functions* as explicit schemata in GP evolutions. The goal was to answer questions surrounding the BBH in GP. While this working report defines the structure of a building block function, it reaches no conclusions of the BBH by their use.

Some other early work in the area includes Rosca's extraction of the values of important terms in his schema theorem from actual populations [101] and analysis of several run statistics by looking at rooted-tree-fragments during evolution [108]. While Rosca used a large population size, each experiment either analyzed the best-in-run individual in detail or presented a more basic statistic, such as average fragment size, over the whole population. Rosca concluded that the rooted-tree schema was a powerful analysis technique and presents insight into the mechanisms of GP at work.

In 1997, Poli and Langdon [84] performed empirical experiments, tracking creation and transmission of all hyperschemata in populations of Boolean programs. The paper put constraints on the scale of the GP system: programs could not exceed three or four nodes deep, the population was set at fifty programs and only crossover and reproduction were used. As expected, crossover was found to be innovative and destructive by counteracting reproduction and maintaining diversity in the population.

Angeline [5] performed analysis comparing various forms of GP crossover, concentrating on their effect on the schemata in the programs of the population. The paper concluded that crossover could be described more as a population-limited macro-mutation operator than as an engine for composing building blocks.

Veraria et al. [121] presented empirical analysis of the subtree schemata introduced by Koza [41]. The research analyzed the effects of *selective crossover* on the schemata in the population, using the royal road task. The paper concluded, through the use of schemata in the analysis, that the new form of crossover has no positional bias.

Langdon and Banzhaf [48] presented an analysis of schema repetition in best-of-run genetic programs from evolutions on two benchmark problems. They found that the solution programs contained large repeated patterns and suggested that these patterns were both larger and fitter on the whole problem than the classic concept of GA building-blocks.

Wilson and Heywood [128] built on work by Langdon and Banzhaf in [49], analyzing repeated blocks of instructions in linear genetic programs. Experimental results suggested the existence of re-usable modules or building blocks in the population. While this research was directly looking for schemata in the population, it constrained the form of those schemata to contiguous blocks of nodes.

Majeed [56, 58] defined a schema as a subtree of a set maximum depth. All such schema that occurred in at least half of the population in the last generation of evolution were found. The paper analyzed the schema for one hundred runs of a evolution on a symbolic regression problem. While no firm conclusions were reached, the paper noted that the fitness of a schema in one run seemed independent of the fitness of the same schema in other runs.

Wong and Zhang [130, 131] empirically analyzed the schemata in GP evolutions for a very simple form of schema: numerical nodes. The research analyzed the effects of program simplification on schema disruption by tracking each schema as it is passed from parent to child during evolution. It concluded that the simplification both disrupted existing schemata and created new schemata. Wong and Zhang also concluded that further research needed to verify the results using a more general and more complex form of schema than the one used. Building on this research, Kinzett, Johnston and Zhang [37, 38] analyzed the effect of program simplification on the survival of schemata in GP evolutions. The schemata analyzed were subtrees clipped to depth two or three and at each generation, the inclusion of every possible such building block in the population was presented graphically. Kinzett and Zhang concluded that this research confirmed the earlier conclusions of Wong and Zhang, and that while program simplification disrupts building blocks, it also constructs new building blocks. Further work [39] on a similar vein concluded that while program simplification reduced the diversity of the population this did not lead to worse accuracy at the classification task.

McPhee et al. [61, 62] empirically analyzed rooted-tree-fragments in GP, by looking at their *semantic contexts*. A semantic context for a fragment was termed *fixed* if all parameters to the fragment were introns (that is, have no effect on the evaluated result of the program) and *compatible* if the fragment and a target fragment could match by setting their respective parameters correctly. Two run statistics were measured: the percentage of fixed contexts and the percentage of compatible contexts. Each value was an average over every fragment in any program. A Boolean task with a limit of one *don't-care* node per context, was used in order to limit the complexity of the analysis. The research concluded that for this task the majority of crossover events performed no useful search since the context they affected was fixed.

McKay et al. [60] analyzed building blocks in GP using compression on each generation's population. Further work by McKay et al. [59] also looked at the flow of building blocks from one generation to the next. While neither paper reached firm conclusions to do with GP, both found the compression technique to be powerful in the analysis of building blocks.

Tanji and Iba [117] presented *Program Optimization by Random Tree Sampling* (PORTS) as a system for preserving tree-fragments in evolution, based on the fragments' *differential fitnesses*. The paper concluded that while competitive in performance with standard GP, PORTS is more able to preserve tree-fragments.

We have also contributed to the literature with papers leading to this thesis [113, 114].

2.7.2 Schemata as subroutines in GP

The use of subroutine or function calls is basic to most widely-used computer languages. Subroutines extend the power of the language considerably, by allowing the solution of a given task to delegate its work to lower-level solutions for subtasks of the main task. This allows the solution to the main task to be simpler than if it had attempted to solve the task without relying on other modules.

Schemata, essentially defined as "parts of genetic programs" are a closely related concept to program subroutines. There have been several attempts to incorporate subroutines into the GP representation. Some of these methods have gained reasonable performance increases, while others have been less successful. The following sections describe the current existing methods at evolving programs with subroutines. While this thesis does not develop subroutines in GP, their use in the literature is relevant as background.

Automatically defined functions

The first method of using subroutines in GP was the *GP with Automatically Defined Functions* (GP-ADF) technique introduced by John Koza in [41].

When using GP-ADF, a genetic program takes the form of a n-tuple with program trees as entries. The first entry is the *Result Producing Branch* or RPB. The other entries in the tuple are *Automatically Defined Functions* or ADFs.

Variable nodes augment the terminal sets of the ADFs and *ADF* nodes augment the function sets of the RPB and ADFs such that the RPB or an ADF may use an ADF node to *call* any ADF that occurs later in the tuple. This means that the RPB may *call* an ADF using an ADF node, effectively inserting its program tree into the RPBs program. The arguments to the ADF node replace the variable nodes to the new subtree. In a similar way, an ADF can call another ADF, though ensuring ADFs can only call other ADFs that are later in the program tuple ensures no recursion.

In [41], Koza introduced ADFs and tested them on a range of tasks such as the n-parity problems, regression and the Santa-Fe ant-trail. Since then, ADFs have become a highly popular method and variations of ADFs have been used in a variety of contexts.

- Brock [12] used the ADFs evolved in a run of evolution as a starting point for the ADFs of future runs. As may be expected this practice improved performance, but the performance increase was significant only when the ADFs selected from the initial evolution were of sufficient usefulness to the programs in further evolutions. The task used was the n-parity problem and the ADF resulting in a performance increase was a solution to the 2-parity problem. This ADF is easily used by programs to solve higher order parity problems. The method had weaknesses: the hand-selection of *good* ADFs from the previous evolution and the need to run this previous evolution in full before the method may be engaged.

- Wong et al. [129] emulated the action of ADFs in a more general system using logic grammars. The paper used the logic grammar system to evolve a program for calculating the dot product of vectors. Compared to a standard ADF system, the grammar based system found a solution program very quickly and repeatably. But the grammar was hand designed for the problem.

- Spector [116] defined Automatically Defined Macros (ADMs). An ADM is essentially the same as an ADF but expands as a macro in the caller program before it is evaluated. The paper compared two methods on a range of problems and ADMs were found to reach a solution faster than ADFs in many cases, although the quality of the ADF solution was often better.

- Ahluwalia [2] modified the GP-ADF algorithm by placing the RPBs of the programs in one population and each type of ADF into another, separate population. The populations evolved in parallel with the ADFs receiving fitness based on their use in RPBs and higher-level ADFs. In contrast to GP-ADF, Ahluwalia forced the ADFs in each population to perform the same job as others in the population. The method was also combined with Module Acquisition in a method called EDFs. EDFs outperformed both standard GP and

GP-ADF on two classification problems.

- Rodrigues et al. [97] extended Context-Free Grammar GP (CFG-GP) [123] by enabling the grammars to use ADFs. In a comparison, GP-ADF and the new method (GGGP with ADFs) performed much the same as each other and better than standard GP.

- Langdon [46] used a GP system with ADFs to evolve various functions on stacks and queues. The base functions indexed memory directly. The solutions were found to be correct but sub-optimal.

Other research on subroutines in GP

Banzhaf et al. [8] introduced a method for effectively evolving large genetic programs of several levels, from low-level modules to the high-level outer procedure. In essence this method was a modified crossover operator. Crossover was applied to the different levels separately with parent programs and subtrees being selected based on a differential fitness, determined by measuring the effect of replacing the subtree by a constant. Banzhaf et al. compared the method to standard GP on four continuous regression problems and two n-parity problems and showed an increase in performance on all problems.

Angeline and Pollack [6] described *Module Acquisition*. They defined a GP *module* as the portions of a subtree, from a genetic program, that are within a specified depth of the root of the subtree. Thus for a subtree that does not exceed the set depth, the module is simply the subtree itself. For a subtree that has branches exceeding the set depth, the module contains the subtree “clipped” to the set depth with unique variable names placed under the links broken by the clipping process. The method added two genetic operators to the GP system: compression and expansion. In order to evaluate a program, the module nodes within it are recursively expanded and the resulting tree is evaluated. The method evolved programs to solve two separate tasks: the tower of Hanoi with four disks on the initial stack and tic-tac-toe. Angeline and Pollack concluded that despite the frequent use of modules in programs, they were not examples of *modular programming*, noting that they shared "...more commonality with the *distributed representations* of procedural knowledge found in connectionist networks." The paper noted interesting emergent behaviour in the strategies used by the programs in solving the tasks. But no comparison was made to other methods or canonical GP.

Roberts et al. [96] developed *Subtree Encapsulation* as a similar method to Module Acquisition. In this method the depth limit of modules is removed so that all modules are subtrees and the compressed modules take no arguments.

Adaptive Representation through Learning (ARL) [105, 103, 104, 98, 100, 99, 107, 102] is a method developed by Rosca and Ballard during 1994 to 1997. Blocks in ARL are similar to the modules from Module Acquisition, but while the position of a module in the program is not constrained, ARL blocks are made from complete subtrees of a set depth. ARL obtains each block by replacing all terminal nodes in the subtree with variable nodes. Where module acquisition selects a module by randomly selecting a subtree in the population and making a module from it, ARL identifies all blocks of a set depth in the population and evaluates their fitness. It then uses this information to select only fit blocks. ARL assigns each block's fitness using either a specialized block-fitness function or a differential fitness function which measures the average change in fitness between a parent(s) and child(ren) in applying a genetic operator, where each parent does not contain the block but each child does.

Evolution progresses in *epochs*, each of which lasts some number of generations. At the end of each epoch, ARL appends the function set with atomic functions formed from the blocks that have highest fitness in that epoch. ARL has been applied to a range of problems but showed greatest performance improvement on the n-parity problem, where a natural block-fitness function consists of testing the block on lower order parity problems.

Woodward [132] presented theoretical results of the use of subroutines or modules in GP. He concluded that the size of a solution program in GP is independent of the primitive set, if modules are allowed. This is in contrast to the case without modules, where the complexity of the primitive set may affect the minimum possible size of a solution.

2.7.3 Current issues with empirical use of schemata in GP

Limited progress is being made in GP schema literature. While schema theory has proven difficult to extend and may produce results which are hard to interpret in real systems, the empirical study and use of schemata in GP evolutions suffers from a number of quite different complaints:

- Representation of a schema - the community needs a unifying form of schema and a set of tools for the use of that form of schema. The exact form of GP schema to use in experiments is far from clear and there are many studies in the literature and a great many more potentials which could be made by mix-and-match.
- Limits on scalability with expressive schemata - many of the empirical studies of schemata in GP use subtrees as their form of schema, others use terminals. It would be better to use a more expressive form of schema but doing so may often be prohibitively expensive.

2.8 Summary

This chapter presents an essential background to machine learning, evolutionary computation, GP and schemata in GP. A summary of the current state of affairs, relevant to this thesis topic, is as follows:

- GP has emerged as a popular and powerful algorithm in evolutionary computation which is a form of machine learning.

- Schemata have been referred to in the GP literature in several contexts: schema theory, subroutines in GP and empirical study of GP using schemata.
  - The form of schema used by each researcher follows no particular rule. A number of distinct entities have been used as GP schemata and no common tool-set has allowed researchers to compare their results between previous forms of schema.
  - Schema theory has yielded interesting results but it is unclear how these theoretical results relate to practice. The community needs more powerful tools analyzing schemata in practical GP evolutions.
  - The empirical study of GP using schemata suffers from an explosion of complexity when dealing with interesting schemata in interesting scales of evolution. Typically researchers avoid this difficulty by studying simple forms of GP schema.

This thesis provides two improvements on this situation:

- An advanced language for forms of schema. This language may describe many previous forms of schema.
- A set of algorithms and tools for analyzing schemata in a given population of genetic programs.

The next chapter begins the new work by defining our new form of schema language: the *match-tree form of schema language*.

Chapter 3

The match-tree form of schema language

3.1 Chapter introduction

For this thesis to be able to effectively analyze schemata occurring in GP populations, it must first clearly identify what is being analyzed. Thus this chapter addresses the question: what is a GP schema? While the question of what a GP schema looks like has been posed many times in the GP literature, no standard form of schema has emerged; this is in stark contrast to genetic algorithms where *similarity template* schemata [34] are typically used. The current state of the art in GP is a plurality of many incompatible forms. With the previously used GP forms of schema as a foundation, this chapter builds a single framework describing very rich sets of schemata, called *match-tree schemata*. This chapter also provides a mechanism for their specialization so that many new and existing forms of GP schema can be described as explicit specializations of the general form. The newly presented mechanism and representation of schema form is the *match-tree form of schema language*.

3.2 The problems with GP schemata

There are potentially as many forms of GP schema as there are types of patterns, limited only by researchers' imaginations.

As noted in chapter 2 in the context of Genetic Algorithms (GAs) the similarity template is typically used for schema research. This is not so of GP; the word schema in the context of GP though simple in its basic definition has on occasion meant: numeric terminals, subtrees, fragments which may possibly be rooted, rooted hyperschema, unordered fragments. In addition, all of these may be wrapped up into sets or multisets, limited to particular depth constraints or a range of other transforms. Each variation yields another slightly incompatible form of schema.

This multiplicity of GP forms of schema is problematic:

1. It is hard to make analysis tools that work on all forms of schemata since each form is incompatible with the other forms.

2. Research using one form of schema is hard to compare against research using a different form. To compare such research we need to settle on a common form of schema. Doing so may affect the experimental results and will involve a partial reimplementation of one or other of the experimental systems.

3. Few of the possibly useful forms of schema have been defined. Researchers must either conform to one of a set of very specific forms of schema or make their own slightly incompatible form, adding to the problem.

Another problem, related to those above, is that definitions of schema often lack precision. Definitions for even simple forms of schema have deceptive room for interpretation.

Because there is no agreement for a precise language for forms of schema, previous definitions of schemata typically rely on a great deal of information being implicit in the definition.

Some basic examples of the peculiar matching characteristics that are implicit in the definitions of various GP schemata follow:

- **Numeric terminals** may typically match an interval of values, rather than a particular value: for instance a numeric terminal node of value 2.0 in a schema will typically be matched by a program node of value 2.00001.

- **Schema functions often impose an order** on matches to their child arguments, although this is seldom described in the schema form definition; for instance the program subtree $(+\ 1\ 2)$ will *typically* not match the "subtree" schema $(+\ 2\ 1)$.

- **Additional program children**, after the last schema child, may or may not invalidate a match; for instance, does the program subtree $(+\ 1\ 2)$ match the "fragment" schema $(+\ \#)$?

The community needs a language for forms of schema precise enough that, when specifying a forms of schema, none of these essential details could be left out. Such a language could solve many of the above problems, if it were machine readable and experiments on schemata in GP could be run using any specified form of schema.

3.3 What is a schema?

A schema in GP is a pattern specifying a component or part of a program that can occur in many different programs. This thesis will say a program *matches* a schema when the schema *occurs in* the program. For example, Koza [42] defined his schemata as sets of subtrees; a program matches the schema if and only if it contains each member of the set as a subtree.

GAs schemata are inherently different from GP schemata.

3.3.1 GAs schemata

A typical GA schema is a chromosome with an additional *don't care* symbol in the alphabet [34]. An individual matches the schema if for each bit in its chromosome the corresponding schema bit has the same value or is a *don't care*.

Disregarding the fitness function, typically each bit of a GA chromosome is independent of the others; one bit may flip without affecting other bits. Therefore, it makes sense that the *don't care* bits of a similarity template cover one bit each and are independent of each other.

One may imagine a different scenario in which the GA system ignores each second bit based on the value of the previous bit. In this case a different schema representation might arise, enforcing don't care bits wherever bits will certainly be ignored since the value of those bits is never important they could be ignored safely. Since the bits of this second representation are no longer independent this second representation benefits from a more complex schema representation.

The case for greater complexity of schemata is strong in the context of GP.

3.3.2 GP schemata

One may imagine a particular GP system where program trees have a fixed shape and all nodes are independent. An example is complete trees of depth 3 where the range of values each node may take is independent of the value of any other node. In this system a good form of schema may be similar to the basic GAs form: a schema is a full binary-tree of depth three on a primitive set including an additional *don't care* primitive which, being placed at any node in the schema tree, specifies the equivalent node of a matching program may be of any value.

But typically GP implementations are very different to this simple system and program nodes may be highly dependent on each other: The very existence of each child node depends on the type of the parent node. For instance, changing the node to a terminal removes any children. This dependence of nodes leads to the benefits of the many complex forms of GP schema.

3.4 Schema in this thesis

This thesis adds to the multiplicity of forms by defining a new form of schema, the *match-tree form of schema*. But this thesis defines this form of schema specifically for use by a very new and highly useful structure: the *match-tree form of schema language*. Rather than adding to the problem, the match-tree form of schema provides unification of many previous forms of schema.

The match-tree form of schema describes a very rich set of schema. As a very general form it subsumes many of the forms of schema from the literature and lots of others that could be invented. It captures a general notion of a "program component". The match-tree form of schema language is a language for specifying useful, meaningful subsets of match-tree schema. Indeed, many forms of schema from the literature may be represented in this language.

By being implemented using the match-tree form of schema, rather than any single form, the analysis tool provided by this thesis avoids all of the problems raised in earlier sections.

3.5 Match-tree schemata

The match-tree form of schema is a very general form of schema which is to be specialized for actual use. In particular, it is more general than the forms of schema used in the literature, so let us create a candidate form.

3.5.1 A straw-man general form of schema

Table 3.1 re-examines most previously defined forms of schema in tree-based GP, excluding those incompatible with standard tree-based GP.

Table 3.1: Forms of schema used in relevant tree-based GP research

Used by **Name of schema**	**Schema is a** **collection of…**	**Collection type**
Daida et al [18] Ephemeral Random Constants	Numeric Terminal	Singleton
Koza [42] Schemata	Subtree	Set
Rosca et al [108] Rooted-tree schemata	Fragment	Singleton
Smart and Zhang [114] Unordered-fragment schemata	Unordered-fragment	Singleton
O'Reilly and Oppacher [70] Schemata	Fragment	Set of pairs <Fragment, count>
Poli et al [85] Hyperschemata	Hyperschema	Singleton

Singleton schemata

All schemata in table 3.1 are some sort of collection of: numeric terminal, subtree, fragment, unordered fragment or hyperschema. For the most part, these basic forms follow a natural hierarchy.

The hierarchy of the items in the "Schema is a collection of…" column:

1. A numeric terminal is a terminal.

2. A terminal is a subtree of depth one.
3. A subtree is a fragment without don't care terminal ("#") nodes.

4a. A fragment is a hyperschema without don't care function ("=") nodes.

4b. A fragment may be represented by an unordered-fragment with use of singleton "arg_n" functions.

Therefore, all numeric terminals, terminals, subtrees and fragments are hyperschemata. In addition all numeric terminals, terminals, subtrees and fragments may be represented as unordered-fragments.

So this section briefly defines another form of schema, called "unordered-hyperschemata":

> 1An unordered-hyperschema is a hyperschema with no regard paid to the order of any function node's children.

With this definition, the list has a final item:

5. Any unordered-fragment or hyperschema may be represented as an unordered-hyperschema, possibly using singleton "arg_n" functions to enforce the order of some or all functions' arguments.

Therefore, all schemata in the "Schema is a collection of..." column of table 3.1 may be represented by unordered-hyperschemata.

Collections of base schemata

Note that even unordered-hyperschemata would not generalize over the schemata used by Koza [42]. For that sets of unordered-hyperschema are required.

Finally, noting that the schema of O'Reilly et al may be alternately expressed as multisets of fragments which is a form of schema that meets our stated goal of being able to specialize to any of the forms of schema in the literature is: *multisets of unordered-hyperschemata*.

Straw man cut down

This newly defined form of schema is of very limited use, presenting us immediately with one important problem: it is brittle. It is likely that some researchers will invent a form of schema not "covered by" this definition, for instance by introducing don't care functions which may match more than one program node at a time. In response the fragile method must define a new, even more general form of schema.

So what form of schema is not only more general than known forms but also the forms yet to be invented. This thesis' answer, the *match-tree form of schema*, lies close to the concept of object-orientated programming. Each node of a match-tree schema encapsulates functions which determine whether it occurs at a given program node. Thus where previous schema nodes had only properties, like a label, match-tree schema nodes also have behaviour, like which program nodes labels they can match.

Match-tree schemata are very general and will be more resistant to the whims of future schema researchers than any previous form of schema.

3.5.2 Matching behaviour in rooted-fragments

To illustrate the matching behaviour required by a schema node, this section gives the example of rooted-fragments. Any particular rooted-fragment occurs in a program if for each node in the fragment there is a corresponding node in the program such that the mapping of schema nodes to program nodes obeys certain rules:

1. Children map to distinct children:

    - If a fragment node s maps to a program node p, then the children of s map to children of p.
    - No two or more children of s map to the same child of p.
    - All children of s map to some child of p.

- It may be the case that the i^{th}child of the schema node must map to the i^{th}child of p in which case the fragment is ordered.

2. Program node and schema node have the same label, except for don't cares: if a schema node s maps to a program node p, then either s is a don't care node or the label of s is the same as the label of p.

3. Don't care schema nodes have no children.

4. The root schema node maps to the root program node.

These rules exactly specify the rooted-fragment form of schema and one can imagine tweaks which would generalize to hyperschemata by relaxing rule three or specialize to subtrees by removing the "except for don't cares" from rule two.

Other than rule four which can be shared by all forms of schema by clever use of a root schema node there are two forms of matching behaviour on display: rule one, which determines acceptable mappings of a schema node's children and rules two and three, which determine how the schema node's properties must match the program node's properties.

Therefore, each match-tree schema node has two functions which determine its behaviour:

- **Label-matching:** the *label-match* function determines a schema node's matching behaviour based on the type of a given program node. For instance, some schema nodes will only match feature nodes and others will only match arithmetic functions.

- **Child-matching:** the *child-match* function determines a schema node's matching behaviour based on which children or descendents of a given program node match which children of the schema node. For instance, a schema function node s may require the first child of a potentially matching program node p match the first child of s and the second child of p match the second child of s and so on for all schema nodes, additional program nodes may be ignored or it may be required that the schema node and program node have the same number of children.

The following subsection describes the match-tree form of schema in detail.

3.5.3 The match-tree form of schema

Each match-tree schema is a tree of nodes. Each match-tree schema node s has a label $s.v$ and three associated functions:

- $s.f_n$ is a *label-match* function to determine whether a given schema node label matches a given program node label.
- $s.f_c$ is a *child-match* function to determine whether a given schema node's children match a given program node's descendents.
- $s.f_{c_s}$ is a *child-selection* function used for optimization which is described in section 3.6.1.

General concept: how a program matches a match-tree schema

As a concept, each schema a duality: the schema can be viewed either as a component which may be shared between programs or as the set of programs sharing this component. Therefore, a schema may be defined by a function taking a program which determines whether the schema occurs in the program. With match-tree schemata, this function producing a Boolean value indicating whether a given program matches a given schema may be implemented by a recursive procedure using the label-match and child-match functions of the schema nodes.

The subtree rooted at a schema node s is said to occur at a program node p if two conditions are met:

- The label-match function $s.f_n(L_s, L_p)$ returns `true`, indicating that the program node label L_p matches the schema node label L_s. For instance, an addition program node matches an addition schema node and would also match a don't care schema node.

- The child-match function $s.f_c$ returns `true`, indicating that the subtree rooted at the program node matches the schema node's children. $s.f_c$ can involve finding a match between child nodes of the schema node and nodes of the subtree rooted at the program node and ensuring that the structure of the children of the schema node matches the structure of the corresponding program nodes.

This formulation covers a wide range of schemata including ordered fragments and unordered fragments. It also ensures that the matching behaviour is made explicit in the schema definition.

3.5.4 Label-match functions

For a schema node s, $s.f_n$ is a Boolean valued *label-match function* of the form:

$$f_n(L_s, L_p) \rightarrow \mathcal{B}$$

where L_s is a schema node label and L_p is a program node label.

f_n tests whether the given labels match. The schema node s matches a program node having labels L_p only if $f_n(s.v, L_p) = \text{true}$.

Two useful label-match functions are:

- **Exact match ("!"):** returns `true` if and only if the two label arguments have the same value.

  A exact match label-match function with prefix v is referred to in this thesis by a schema node with label "v!".

- **Don't care with prefix ("#"):** returns `true` if and only if the program node label starts with the schema node label.

A don't care label-match function with prefix v is referred to in this thesis by a schema node with label "v#".

3.5.5 Child-match functions

For a schema node s, $s.f_c$ is a Boolean valued *child-match function* of the form:

$$f_c(s, p, M) \rightarrow \mathcal{B}$$

where s is a schema node, p is a program node and M is a set of pairs of matching program nodes and schema nodes.

f_c tests if the children of a given match-tree schema node s match the descendents of a given program node p. The third argument of f_c is a set of pairs (M) associating children of s with matching nodes from the subtree at p. M includes a pair $< s_c, p_d >$ only if the child s_c of s occurs at the program node p_d in the subtree rooted at p.

f_c may be used to specify a wide range of child-matching behaviour. The following are examples of child-match functions that could be used as $s.f_c$ for some schema node s:

- **Same number of children:** returns true if and only if the program node p has the same number of children as s.

- **Children at same indexes:** returns true if and only if p has at least the same number of children as s and for each i'th child s_i of s and the i'th child p_i of p, $< s_i, p_i >$ is in M.

- **Distinct children match:** returns true if and only if for each child s_i of s there exists a distinct child p_j of p such that $< s_i, p_j >$ is in M.

- **Distinct descendents match:** returns true if and only for each child s_i of s there exists a distinct descendent p_j of p such that $< s_i, p_j >$ is in M.

3.6 A matching algorithm

This subsection describes an algorithm which determines whether a given program subtree matches a given match-tree schema subtree using the functions at each node in the schema subtree.

A program node p matches a schema node s if its *label* matches and its *children* match.

- To determine if the label of a program node p matches that of a schema node s, the algorithm evaluates $s.f_n(s.v, p.v)$, where $p.v$ is the label of node p. A result of true would indicate a match, whereas a result of false indicates that the program node and schema node have incompatible labels.

- The matching algorithm uses $s.f_c$ to determine if the descendents of p, and possibly including p itself, match the children of s.

  – $s.f_c$ requires as input s, p and a set of the matches of p's descendents to s's children.

    If s has no children, then the final argument is empty. If s has children, then the algorithm recursively determines which nodes in the subtree at p match each child of s and constructs a set of matching pairs which is passed as the third argument of $s.f_c$.

Thus through this recursive procedure the algorithm determines if any given program subtree matches any given schema subtree. Finally, a given program matches a given schema if and only if its root subtree matches the schema's root subtree.

The algorithm is given in pseudocode 3.1. BasicMatches algorithm does not exploit an optimizations which would increase its efficiency considerably. The algorithm constructs pairs of schema node children and program node descendents, even if the child-match function will not use

```
FUNCTION BasicMatches(Schema subtree $s$, Program subtree $p$)

  IF $s.f_n$(label of root of $s$, label of root of $p$) $=$ false
    RETURN false
  ELSE
    $M$ is a set of pairs from $\{< S_N, P_N >\}$ initially empty
    FOR EACH child $s_c$ of $s$
      FOR EACH descendent subtree $p_d$ of $p$ including $p$ itself
        IF BasicMatches($s_c, p_d$)
          $M = M \cup \{< s_c, p_d >\}$
    RETURN $s.f_c(s, p, M)$

END
```

Pseudocode 3.1: BasicMatches

them. The optimization uses child selection functions to only construct pairs that might be used.

3.6.1 Child-selection functions

It is often the case that a schema node's child-match function ignores some elements of the set of pairs passed as its third argument. For example, a given child-match function may only ever look at the pairs containing direct children of p and ignore those of deeper descendents. Other child-match functions only ever look at pairs of children at the same index, matching first program child with first schema child, second with second, and so on. Child-selection functions use knowledge of this behaviour to improve the efficiency of the matching algorithm without affecting the actual matching behaviour.

A child-selection function f_{c_s} is of the form:

$$f_{c_s}(s, p) \rightarrow \{< S_N, P_N >\}$$

where s is a schema node, p is a program node, S_N is the space of schema nodes and P_N is the space of program nodes.

The child-selection function $s.f_{c_s}$ which is derived from child-match function f_c will, given a program subtree p, return the pairs of schema nodes an program nodes that are "of interest to" $s.f_c$.

The algorithm is given in pseudocode 3.2. In the above pseudocode,

```
FUNCTION AdvancedMatches(Schema subtree $s$, Program subtree $p$)

  IF $s.f_n$(label of root of $s$, label of root of $p$)
    $M$ is a set of pairs from $\{< S_N, P_N >\}$ initially empty
    FOR EACH pair $< s', p' > \in s.f_{c_s}(s, p)$
      IF AdvancedMatches($s', p'$)
        $M = M \cup \{< s_{\prime}, p' >\}$
    RETURN $s.f_c(s, p, M)$
  ELSE RETURN false

END
```

Pseudocode 3.2: AdvancedMatches

instead of recursing on each pair with a child of s and a descendent of p, the child selection function $s.f_{c_s}$ gives the set P' which includes only those pairs which could potentially affect the output of the child-match function $s.f_c$. The match function recurses on only these "interesting" pairs.

The optimizations do not affect the output of the function and a call to `AdvancedMatches` returns exactly the same result as an equivalent call to `BasicMatches`. By the definition of child-selection function, the set returned by $s.f_{c_s}$ has all pairs which may affect the output of the child-match function $s.f_c$. Thus if M is the set of pairs with a descendent p_d of

p and a child s_c of s such that p_d matches s_c, then $s.f_c(s.f_{c_s} \cap M)$ which is the effective operation of this function will always equal $s.f_c(M)$ which is the effective operation of `BasicMatches`. Therefore `AdvancedMatches` produces the same output as `BasicMatches`.

3.7 Potential child-match functions

There is no limit on the potential complexity of child-match functions. This thesis defines only a small subset of these possibilities, including the following:

- **Child at index matches – "cind":** returns true if and only if each schema node child matches the program node child at the same index. That is, p must have at least as many children as s and for each node s_i, the i^{th}child of the implicit schema node s, $< s_i, p_i >\in M$, where p_i is the i^{th}child of p.
  - The corresponding f_{c_s} returns the set of pairs $< s_i, p_i >$ with s_i from the first child to the last child of s and p_i is the corresponding child of p.
- **Child at index matches, same number of children – "cindx":** returns true if and only if the above "cind" function would return true and p and s have the same number of children. Thus p has as many children as s, each of which matches the schema child at the same index.
  - The corresponding f_{c_s} is the same as for the "cind" function.
- **Distinct children match – "cdist":** returns true if and only if p has at least as many children as s and there is some order of all children of p as $p_i, i \in \{1, 2, 3, \ldots\}$ such that for s_i, the i^{th}child s, it is true that $< s_i, p_i >\in M$. Thus the relative order of the program node children and the schema node children is unimportant but there must

be a one-to-one mapping from some subset of the program node's children to all of the schema node's children.

- As any pair of program node child to schema node child could potentially affect the result of the function, the corresponding f_{c_s} returns the set of all pairs $< s_c, p_c >$ where p_c is a child of p and s_c is a child of s.

- **Distinct children match, same number of children – "cdistx":** returns true if and only if the above, "dist", function would return true and p and s have the same number of children. Thus p has as many children as s and there is a one-to-one mapping from the children of p to matched children of s.

  - The corresponding f_{c_s} is the same as for the "cdist" function.

- **Label, index pairs match – "cpind":** returns true if the schema node children of each label match in order the program node children of the same label: for each child s_i of the schema node if s_i is the k^{th}child with the label $s_i.v$ and p_j is the k^{th}child of the program node with label $s_i.v$ then $< s_i, p_j >$ is in M.

  If all labels of the schema node children are different then "cpind" is equivalent to "cdist". The all labels of the schema node children are the same then "cpind" is equivalent to "cind".

  - The corresponding f_{c_s} will sort the children of p and s by label and return corresponding pairs.

- **Label, index pairs match, same number of children – "cpindx":** $f_c(p, M)$ returns true if and only if the above, "cpind", function would return true and p and s have the same number of children.

  - The corresponding f_{c_s} is the same as for the "cpind" function.

- **Descendents match – "desc_d_{min}_d_{max}":** defined for $d_{min}, d_{max} \geq 0$, $f_c(p, M)$ returns true if and only if for each child s_c of s, there is a pair $< s_c, p_d >$ in M, where p_d is a descendent of p in the depth range $[d_{min}, d_{max}]$ with depths starting from 0 at the root.
  - The corresponding f_{c_s} returns the set of pairs $< s_c, p_d >$ where s_c is a child of s and p_d is a descendent of p in the function's depth range as above.
- **Distinct descendents match – "ddist_d_{min}_d_{max}":** is similar to the "desc_d_{min}_d_{max}" function but requires that each child of s be matched by a distinct descendent of p. It shares the "desc_d_{min}_d_{max}" function's f_{c_s} function.
- **Match any – "any":** f_c simply returns true without looking at its arguments.
  - The corresponding f_{c_s} returns an empty set since no pair would ever affect the f_c's output.

3.8 The match-tree form of schema language

A schema is a set of programs which share the same pattern. A "form of schema" is a set of schemata that share the same kind of structure. For example, rooted-ordered-fragments is a particular form of schema. Many researchers have defined forms of schema.

The match-tree form of schema, given in the previous sections, is an extremely general form of schema and would not be useful directly for an analysis of GP programs. This thesis makes no commitment to any single more specific, and therefore more useful, form of schema but instead provides a language for forms of schema by which a variety of forms of schema can be defined.

The *match-tree form of schema language* is a language for specifying forms of schema as subsets of the match-tree form of schema. This language represents *match-tree forms* by vectors of disjunctive node patterns expressed as strings. Just as a program *matches* a match-tree schema, a match-tree schema is said to *belong to* a match-tree form. Specifically, a match-tree schema belongs to a match-tree form if there is an "acceptable" mapping from its nodes to the nodes of the match-tree form. The process is similar to how a program matches a match-tree schema, but the rules for determining which mappings are acceptable are very different.

3.8.1 Match-tree form representation

A match-tree form is defined by a vector of disjunctive node patterns. Each node pattern can be viewed as a pattern for schema nodes. A schema *belongs* to a match-tree form if the root node of the schema node *agrees with* the first node pattern. Each disjunct has the same general format.

The disjuncts

Which match-tree schema nodes *agree* with a particular disjunctive node pattern of a match-tree form node pattern is dependent on information contained in each disjunct. Each such disjunct t is associated with the following fields:

- $t.v$ is a node label
- $t.f_n$ is a label-match function.

  For a schema node to agree with a disjunct t, $s.f_n(s.v, p.v) \implies t.f_n(t.v, p.v)$ for any program node label $p.v$. That is, there may be no program node label which the schema label-match function accepts but that the form label-match function rejects.
- $t.f_c$ is a child-match function.

For a schema node to agree with a disjunct t, $s.f_c(s, p, M) \implies t.f_c(s, p, M)$ for any program node p and set of pairs M. That is, there may be no possible arguments which the schema child-match function accepts but that the form child-match function rejects.

- $t.c_{num}$ is a range on the whole numbers, for instance $[0, 0]$ or $[1, \infty]$.

  $t.c_{num}$ specifies a range on the number of children of a schema node. For a schema node to agree with a disjunct t, then the number of children of the schema node must be within $t.c_{num}$

- $t.c_{index}$ is the index of some node pattern of the match-tree form. For a schema node s to agree with a disjunct t, each of the children of s must agree with the disjunctive node pattern at index $t.c_{index}$ of the match-tree form containing t.

3.8.2 Character string representation

The analysis tools described later in this thesis need a specification of a match-tree form. For the purposes of being passed to the analysis tools, each match-tree form may be represented as a string.

The form, as a vector of disjunctive node patterns $D_1, D_2, D_3, \ldots, D_n$, is represented as "$[1 : S(D_1)\ 2 : S(D_2)\ 3 : S(D_3)\ \ldots\ n : S(D_n)]$" where $S(D)$ is the string representation of node pattern D. $S_D(D)$ for a node pattern D with disjuncts $\{d_1, d_2, d_3, \ldots, d_m\}$ takes the form "$(S(d_1)S(d_2)S(d_3)\ldots S(d_m))$" where $S(d)$ is the string representation of disjunct d. $S_d(d)$ for a disjunct d takes the form "$< d.v, d.f_n.\mathrm{name}, d.f_c.\mathrm{name}, [d.c_{num}.\mathrm{min}, d.c_{num}.\mathrm{max}], d.c_{index} >$" where $d.f_n.\mathrm{name}$ is a well-known name for the node-match function $d.f_n$, $d.f_c.\mathrm{name}$ is a well-known name for the child-match function $d.f_c$ and $d.c_{num} = [d.c_{num}.\mathrm{min}, d.c_{num}.\mathrm{max}]$.

3.8.3 Example match-tree forms

Table 3.2 lists various forms of schema reported in the research literature, previously presented in table 3.1, and a match-tree form string representation for each. The table also lists newly created forms.

Table 3.2: Forms of schema used in relevant tree-based GP research and equivalent match-tree forms

Daida et al – **"Ephemeral Random Constants"** [1:(<,#,desc_0_∞,[1,1],2>) 2:(<,isnumeric,any,[0,0],2>)]
Koza – **"Schemata"** [1:(<,#,desc_0_∞,[1,∞],2>) 2:(<,!,cindx,[0,∞],2>)]
Rosca et al – **"Rooted-tree schemata"** [1:(<,#,any,[0,0],1> <,!,exact,[0,∞],1>)]
Smart et al – **"Unordered-fragment schemata"** [1:(<,#,desc_0_∞,[1,1],2>) 2:(<,#,any,[0,0],2> <,!,dist,[0,∞],2>)]
O'Reilly et al – **"Schemata"** [1:(<,#,distdesc_0_∞,[1,∞],2>) 2:(<,#,exact,[0,0],2> <,!,exact,[0,∞],2>)]
Poli et al – **"Hyperschemata"** [1:(<,#,any,[0,0],1> <,!,exact,[0,0],1> <,#,exact,[1,∞],1> <,!,exact,[1,∞],1>)]
Thesis – **"Multisets of unordered-hyperschemata"** [1:(<,#,distdesc_0_∞,[1,∞],2>) 2:(<,#,any,[0,0],2> <,!,exact,[0,0],2> <,#,exact,[1,∞],2> <,!,exact,[1,∞],2>)]

Table 3.2 shows a few of the following constructs in the string representations of match-tree forms:

- Setting the first node pattern to "1:(<,#,desc_0_∞,[1,1],2>)", makes the form into a non-rooted form.

  For example "[1:(<,#,desc_0_∞,[1,1],2>) 2:(<,isnumeric,any,[0,0],2>)]" (non-rooted numeric terminals) is the corresponding non-rooted form for the match-tree form "[1:(<,isnumeric,any,[0,0],1>)]" (rooted numeric terminals).

- Setting the first child of the root node to "1:(<,#,distdesc_0_∞,[1,∞],2>)" makes the form into a non-rooted multiset.

  For example "[1:(<,#,distdesc_0_∞,[1,∞],2>) 2:(<,isnumeric,any,[0,0],2>)]" (multisets of non-rooted numeric terminals) is the non-rooted multiset of the match-tree form "[1:(<,isnumeric,any,[0,0],1>)]" (rooted numeric terminals).

- "i:(<,#,exact,[0,0],i> <,!,exact,[0,∞],i>)" describes ordered fragments. The first disjunct matches any terminal as a *don't care* node and the second disjunct matches any function or terminal of a particular value.

- "i:(<,#,any,[0,0],i> <,!,exact,[0,0],i> <,#,exact,[1,∞],i> <,!,exact,[1,∞],i>)" describes one interpretation of hyperschemata. The first disjunct is the *don't care* terminal, the second disjunct is the non-*don't care* terminal, the third disjunct is the *don't care* function and the fourth disjunct is the non-*don't care* function.

  Alternately, hyperschema may be represented by "i:(<,#,any,[0,0],i> <,!,exact,[0,∞],i> <,#,exact,[1,∞],i>)" but section 4.6 will show that such a form would be non-conjunctive and relatively difficult for the system described by this thesis to work with.

3.9 Chapter summary

Achieving the main thesis goal requires a precise definition of schemata in the context of GP. But, even though much GP research has used the concept, the water is far from clear; the state-of-the-art in GP schemata still consists of numerous, largely incompatible forms of schema.

One option is for this thesis to choose from amongst the forms of schemata available, which would certainly involve choosing various forms of schema with each requiring quite different implementations for experimentation. This thesis chooses a second option: that this thesis define a "unifying" form of schema, compatible with several other forms in the literature. This chapter defined both the very general "cover-all" form of schema, the *match-tree form of schema*, and a language for specifying subsets of this general form called the *match-tree form of schema language*. The specified subsets of the match-tree form of schema are called *match-tree forms*. This combination of the very general match-tree form of schema and corresponding match-tree form of schema language provide both flexibility in matching behaviour and precision of definition.

Match-tree schemata have a very flexible matching mechanism: each schema node independently specifies its matching behaviour. The move is similar to the move to object-oriented code in that the node itself encapsulates behaviour as well as data. Each schema node s is associated with functions which determine two types of behaviour:

- The first has been carried out implicitly by previous forms of schemata. It is this behaviour that determines that a feature node does not match an addition node but does match a don't care node.

  The schema node uses a *label-match function* and a label to make this type of behaviour explicit in the definition of the schema.

- The second is largely missing from many previous definitions of GP schemata. It is this behaviour that determines that in one instance program (+ 1 2) does match schema (+ 2 1) and in another instance it does not because the order of arguments differs between the programs.

The schema node uses a *child-match function* to make this type of behaviour explicit in the definition of the schema.

Previous forms of schemata have behaviours that are implicit or simply unstated in their definitions, prompting questions such as:

- Do numeric terminals in fact match a range of labels?
- Does the order matter when matching children?
- Does it matter to a program node/schema node match that the program node has more children than the schema node?

The reason for these behaviours being unstated may be linked to where they are set: the definition of the form of schema. It is simply very difficult to cover all the details in the few sentences of the average schema definition, therefore at times they are simply left out.

The match-tree form of schema language provides a language to precisely define a subset of the very general match-tree form of schema. This allows each of the relevant forms of schema used in the GP literature to be tabulated in table 3.2 represented as machine-readable strings as well as many undiscovered forms.

The analysis tool developed in this thesis will require a form of schema to be passed as an argument and in this respect the definition of the match-tree form of schema language breaks new ground; not only do match-tree schemata have a representation but so do *match-tree forms* of schema. Thus for instance, a researcher may experiment with hyperschemata simply by presenting an implementation with a string representing hyperschemata.

Building on this foundation, the next chapter presents the core of our analysis method for match-tree schemata in populations of genetic programs: maximal schemata. Maximal schemata can be used as a proxy for many types of analysis by efficiently grouping and counting schemata.

Chapter 4

Maximal schemata

4.1 Chapter introduction

This thesis develops a method to analyze schemata occurring in programs. Key to the newly developed method is that the analysis is not of *schemata* or of *programs* but of the relationship between schemata and programs.

This chapter precisely defines this core concept: the grouping and counting of schemata that are in exactly the same set of programs or alternatively, the grouping and counting of sets of programs matching the same sets of schemata. Grouping schemata and sets of programs is of fundamental importance; the sorts of empirical analysis of schemata this thesis aims for will involve dealing with prohibitively large sets of schemata. As an example, even single programs commonly match more distinct hyperschemata than there are atoms in the universe. This chapter provides a way to effectively analyze such sets of schemata by executing an equivalent analysis of sets of structures which *represent* the prohibitively large sets.

4.2 The problem stated

To analyze the programs matching each schema in a given set of schemata a researcher identifies and performs analysis on each schema and its matched programs. One implementation of such analysis iterates over the set of schemata, performing analysis on each schema. This approach, referred to here as the *naive approach*, is often impractical; there are often simply too many schemata in the set of all schemata. The following subsections determine upper bounds on how many schemata there could be in the case of a trivial population, containing only one program, p, that is a full n-arity tree of depth d, totalling N nodes for different forms of schema.

4.2.1 Example 1: restrictive-ordered subtrees

Suppose the schemata we are interested in are the restrictive-ordered-subtrees of table 6.4 in chapter 6. To perform analysis on the schemata in our program an analysis may iterate over the set of all schemata occurring in the program. There would be a maximum of N subtrees with one rooted in each node of the program. Therefore there would be a maximum of N schemata, although there could be fewer if some subtrees were identical. There would in all normal circumstances be a manageable upper limit on the number of schemata produced.

The situation is similarly simple when looking at other basic forms of schema, like numeric terminals.

But the case is more complex when looking at more interesting forms of schema because there can be many more schemata of the form. The following subsections present the case for fragments and hyperschemata.

4.2.2 Example 2: rooted-ordered-fragments

Let us look at a more complex form of schema: rooted-ordered-fragments. This subsection derives a formula for $N_{rof}(a, d)$ to calculate the maximum number of rooted-ordered-fragments occurring in a program tree of depth d and arity a. To do so it uses induction starting from the base case of a tree of depth 1 with a single node. The inductive step obtains the value for a tree of any depth, given the value for a tree of the previous depth.

Base case: tree of depth one

The number of rooted-ordered-fragments in a tree of depth one is two; any fragment matching a single node is either the node itself or a *don't care* node.

Therefore, for any arity a: $N_{rof}(a, 1) = 2$.

Inductive case: deeper tree

Adding one level to the tree increases the number of possible fragments.

Consider each child branch of the root node of a program tree of depth d and arity a. Each such child branch could match $N_{rof}(a, d-1)$ distinct fragments. The root node of a fragment occurring in the program could be a *don't care* terminal or it could have children such that each child occurs in the corresponding child of the root program node. These child fragments are independent of each other, thus for depths greater than 1 the following formula expresses the possible number of fragments occurring in the program:

$$N_{rof}(a, d) = \begin{cases} 2 & (d = 1) \\ N_{rof}(a, d-1)^a + 1 & (d > 1) \end{cases}$$

Plugging the values into a calculator produces some large numbers, given in table 4.1. In general, the maximum number of rooted-ordered-fragments occurring in a single program of arity a and depth d is in the order of $2^{a^{d-1}}$. For anything but a small tree, that is less than six deep and has less than three arity, the figures are too massive to allow the naive approach used on subtrees in the previous section.

Table 4.1: Worst case numbers of fragments occurring in a single a arity program tree of depth d.

$N_{rof}(a,d)$	$a=2$	$a=3$
$d=1$	2	2
$d=2$	5	9
$d=3$	26	730
$d=4$	677	$3.9\times\ 10^{8}$
$d=5$	45833	$5.9\times\ 10^{25}$
$d=6$	$2.1\times\ 10^{11}$	$2.0\times\ 10^{77}$
$d=7$	$4.4\times\ 10^{22}$	$8.5\times\ 10^{231}$
$d=8$	$1.9\times\ 10^{45}$	very large

For example, the naive analysis of the set of all fragments of a tree with arity three and depth six which is a moderately sized tree by typical GP standards, iterates over a set with close to as many members as current knowledge places the number of hydrogen atoms in the observable universe at about 3×10^{79}. Clearly, it would not be possible to increase the depth to seven!

It should be noted that there may be and probably are fewer fragments than this in any given program tree for the following reasons:

- The tree may not be "full"; there may be nodes at depth less than d that have arity less than a.
- Some of the fragments may be identical and therefore some fragments may have been counted twice or more. Thus the true value of the count of fragments would be smaller than that produced by N_f.

But one can easily form a pathological program tree that matches exactly $N_{rof}(a,d)$ fragments as any complete tree in which each node has a distinct label.

A program tree may match considerably more schemata of an expressive form like hyperschemata than it does rooted-ordered-fragments.

4.2.3 Other mitigation methods in GP

Past literature analyzing schemata in genetic programming has used various methods to avoid enumerating such large sets of schemata. Typically these methods fall into one of two categories:

1. Use a simple form of schemata like subtrees, thereby reducing the number of schemata. All schemata may then be enumerated using the naive approach without incurring excessive cost.

2. Use a complex form of schemata but don't enumerate them all. Instead, take a sample of the schemata of a size within computation limits; the sample may be taken by random sampling or may be directed toward schemata considered interesting.

Either of these methods is unacceptable here: the first would require the use of too simple a form of schema, which would place considerable constraint on the scope of analysis done. The second may put considerable constraint on the bias- and noise- immunity of analysis: which schemata should be enumerated? Should it be the biggest? or "best"? If so, what does "best" mean with reference to schemata? Should the analysis instead take a random sample?

4.3 The new method

This thesis avoids the two methods above altogether, by using a third:

3. Group and count schemata which for the purposes of desired analysis are the same and enumerate all such groups. The method performs analysis on only the group programs but produces results equivalent to the possibly impractical naive approach.

Key to the new method is that it analyzes not schemata and not programs but the relationship between the two; the new method analyzes either *schemata shared by given programs* or the *programs sharing a given schema*. For instance the multitude of schemata in a single genetic program is of limited interest when enumerated but a researcher may be interested in counting the number of schemata the program shares with other given programs.

Let $S(P, f)$ be the set of all schemata of form f occurring in each of a set of programs P. Further, let $P(s, P_0)$ be the set of all programs from population P_0 matching schema s.

The first of this thesis' groupings partitions schemata by their $P(s, P_0)$:

> Group together schemata of a form f which occur in exactly the same programs from the given population P_0.

That is, group any two schemata, s_1, s_2 of form f for which $P(s_1, P_0) = P(s_2, P_0)$ into a representing datum $G_s(P, f)$ where $P = P(s_1, P_0) = P(s_2, P_0)$, and each representing datum summarizes all the schemata which fall into the group, that is the set of schemata $\{s \in f | P(s, P_0) = P\}$. The nature of this summary is a set of most specific schemata in the group and a count structure of how many members of the group have specified properties. An example would be a count of how many schemata there are of each order.

The second grouping looks to the sets of programs matching given schemata and in this case the new method groups sets of programs rather than schemata.

> Group together subsets of the given population P_0 which match exactly the same set of schemata of form f.

Let a *program subset* be a subset of an implicit population P_0. In this grouping, the set of *program subsets* $\mathcal{P}(P_0)$ is partitioned by $S(P, f)$. That is, the method groups any two program subsets, $P_1, P_2 \subseteq P_0$ for which $S(P_1, f) =$

$S(P_2, f)$ into a representing datum $G_P(S, P_0)$ where $S = S(P_1, f) = S(P_2, f)$ and each representing datum summarizes all the program subsets which fall into the group. The nature of this summary is its largest member and a count of how many members of the group there are or alternatively a count of how many program subsets there are of each cardinality.

Thus the new method needs structures able to adequately represent a group of program subsets G_P or a group of schemata G_s. This thesis develops two structures to be used as base-level representatives: *maximal program subsets* and *maximal schemata*.

4.3.1 Maximal program subsets

This thesis defines a maximal program subset as follows:

> A *maximal program subset* $P_m(s, P_0)$ for some schema s and population P_0 is the set of all programs from P_0 which match s.

Thus a maximal program subset is a subset P_m of P_0. It is maximal in the sense that for any form f including the schema s and any program subset P, $P \supset P_m \Leftrightarrow S(P, f) \subset S(P_m, f)$ since some member of P doesn't match s and P_m must match any schema matched by a superset of P_m.

For analysis of a schema s, $P_m(s, P_0)$ provides adequate representation of s's *occurs in* relationship to the population P_0. Indeed, the definition of maximal program subset goes a way toward representing groups of program subsets which match the same schemata. But there may be program subsets which are not represented by maximal program subsets. A different structure represents these program subsets: the *representative program subset*.

4.3.2 Representative program subsets

This thesis defines *representative program subsets* as follows:

> A *representative program subset* $P_r(S, P_0)$ for some set of schemata S and population P_0 is the set of all programs from P_0 which match every schema in S.

Thus the definition of a *representative program subset*, the programs of which must match all of a set of schemata, is a generalization of the definition of a *maximal program subset* the programs of which must match a single schema. Any maximal program subset for some schema s and population P_0 is a representative program subset for the set of schemata $\{s\}$ and population P_0. Though not always the case, for many forms of schemata including all *conjunctive* forms of schemata any representative program subset for a set of schemata S and a population P_0 will be a maximal program subset for a schema in S and population P_0.

For any given program subset, there will be a representative program subset that matches exactly the same set of schemata and this representative program subset could be found as $P_r(S(P, f), P_0)$, that is the set of programs matching each schema occurring in P.

There is a subtle difference between maximal program subsets and representative program subsets: for some representative program subsets P and forms f, for each $s \in S(P, f)$ there may be a superset of P such that s occurs in all programs of the superset. Thus although P is representative, it is non-maximal. An example uses subtree schemata: let $P_0 = \{A = (+\ 1), B = (+\ 1\ 2), C = (+\ 2)\}$ and f be the space of ordered-subtrees rooted at depth>1. The program subset $\{B\}$ matches subtrees {1,2}, both of which occur in other programs. But no other program subset matches both the subtrees 1 and 2. Thus P is representative but non-maximal.

4.3.3 Maximal schemata

A maximal schema is a similar concept to a maximal program subset and has the following definition:

> A schema s_m is a *maximal schema* with respect to some program subset P and form of schema f, if s_m is a schema of form f that occurs in each program of P where no schema that is more specific than s_m also occurs in each program of P.

Thus a maximal schema is a schema of a given form f and occurring in each program of some program subset P, which is maximal in the sense that each schema that is more specific than s_m occurs in fewer programs. That is for any set of programs P_0 and schema s it is true that $s \text{ moreSpecificThan } s_m \Leftrightarrow P(s, P_0) \subset P(s_m, P_0)$.

The previous subsection showed that for a given schema s and population P_0 there is a single maximal program subset $P_m(s, P_0)$. Often this is the case for maximal schemata; for many interesting forms of schema, including most used in this thesis' analyses, there will always be exactly one maximal schema for a given program subset. Typically such forms are *conjunctive* as defined later in section 4.6. But this is not the case for all forms of schema and there may be many maximal schemata of a given form occurring in a single given program subset.

4.3.4 Representative sets of maximal schemata

Representative sets of schemata are to representative program subsets as maximal schemata are to maximal program sets. Where representative program subsets represent program subsets which may not be represented by any schema, representative sets of schemata represent schemata which may have no unique most specific representing schema.

This thesis defines a representative set of schemata as follows:

> A *representative set of schemata* $S_r(P, f)$ is for some program subset P and form of schema f the set of schemata which are maximal with respect to P, f.

While there may be no unique maximal schema with respect to a given form f and program subset P, there is a unique representative set of schemata.

Exactly one representing set of schemata represents any one given schema s of form f and this representing set of schemata may be found as $S_r(P(s, P_0), f)$. That is the set of all maximal schemata with respect to the programs matching the schema s and the form of schema f.

4.3.5 Maximal and representative pairs

For convenience when referring to the maximal and representative groupings, this thesis defines a *maximal pair* and a *representative pair* as follows:

> A *maximal pair* $r_m = < s_m, P_m >$, said to be maximal with respect to a population P_0 and form of schema f, is a pair containing a maximal program subset P_m and a maximal schema s_m, where $P_m = P_m(s_m, P_0)$ and s_m is maximal with respect to P_m and f.
>
> A *representative pair* $r_r = < S_r, P_r >$, said to be representative with respect to a population P_0 and form of schema f, is a pair containing a representative program subset P_r and representative set of schemata S_r, where $P_r = P_r(S_r, P_0)$ and $S_r = S_r(P_r, f)$.

In addition, to support analysis we require that each representor grouping carry a count structure of its represented program subsets or schemata. The format of this count structure depends on the type of analysis and in the simplest case is a single count of how many program subsets are represented by each representative program subset or how many schemata are represented by each representative set of schemata. In a more complex case, which is used the experiments of this thesis, the count structure has the count of represented program subsets of each cardinality or the count of represented schemata of each order. Other more complex count structures are also possible.

Thus this thesis constructs its representing structures as follows:

- $G_s(P, f) = < S_m(P, f), C_s >$ where $S_m(P, f)$ is defined above and C_s is a count structure of represented schemata.

- $G_P(s, P_0) = < P_m(s, P_0), C_P >$ where $P_m(s, P_0)$ is defined above and C_P is a count structure of represented program subsets.

Which may be combined into one overall representing structure for both:

- $G_r = < r_r, C_s, C_P >$ where r_r is a representative pair with respect to P_0 and f, C_s is a count structure of represented schemata and C_P is a count structure of represented program subsets.

4.3.6 How many?

It is very difficult to precisely specify how many maximal program subsets or maximal schemata there are and this thesis makes no attempt at tight analytical bounds on these values. Some very loose bounds are as follows:

- A bound on the numbers of maximal program subsets, representative program subsets and representative pairs is the number of program subsets: $2^{|P_0|-1}$.

- There may be more maximal schemata and maximal pairs than this, though the number will be less than $2^{|P_0|-1} \times N_{ms}$, where N_{ms} is the maximum number of schemata in any representative set of schemata.

  For many of the forms of schema used for this thesis' experiments $N_{ms} = 1$ and for all of this thesis' experiments $N_{ms} \leq N$ where N is the maximum number of nodes in a program tree. This thesis expects that in most cases the bulk of program subsets and schemata will be non-representative. That is, most program subsets will match exactly the same set of schemata as some superset of the program subset and most schemata will occur in the same programs as some more specific schema.

- While, finding closer analytic bounds on the number of maximal and representative structures is outside this thesis' scope, chapter 6 attempts to characterize the number of maximal pairs empirically.

4.4 Analysis using the new method

The new method presented by this thesis assumes one of two broad structures for the analyses performed:

- An analysis which analyzes each subset of the input set of programs, aggregating some calculation on the most specific schemata which occur in the programs of the program subset. Section 4.4.1 describes this form of analysis.

- An analysis which analyzes each schema of a given form, aggregating some calculation on the subset of P_0 matching the schema. Section 4.4.2 describes this form of analysis.

These two structures cover a wide range of analyses. Crucially, most such analyses may be refactored to have complexity dependent on the number of representative pairs, rather than the potentially massive numbers of schemata or program subsets.

4.4.1 Program subset analysis

Often program subset analyses may be expressed similarly to pseudocode 4.1. The analysis proceeds as an iteration over the power-set of P_0. Each program subset is analyzed using the size of the program subset and the most specific schemata of the form that it matches. The results are aggregated into the final analysis result. Constraints are placed on the types of analysis possible, for example the program subset may not perform analysis on its members' fitnesses. But it is easy to think of interesting analyses which fit this general framework.

```
FUNCTION PSAnalysis(Set of programs $P_0$, Form of schema $f$)

  Initialize an aggregate $A$
  FOR EACH $P \subseteq P_0$
    Compute a value $v$ based on analysis of the
      most specific schemata of form $f$ occurring in all programs of $P$ and
      the cardinality of $P$
    Aggregate $v$ into $A$
  RETURN $A$

END
```

Pseudocode 4.1: PSAnalysis

Using the new method's representative groups, we may refactor the algorithm of pseudocode 4.1 as having an outer loop over representative pairs and an inner loop over program subset cardinalities. The function is presented in pseudocode 4.2. The function operates as follows: the outer loop iterates over each representative program subsets P.

Any program subset will be represented by some representative program subset and the two will share the same set of schemata S. The inner loop iterates over program subset cardinalities z. The analysis result for cardinality z and schemata S is repeatedly aggregated into A, once for each program subset of cardinality z that is represented by P. Thus rather than a single loop with $2^{|P_0|}$ iterations, we have an outer loop with a relatively small $|\hat{P}_r|$ iterations and an inner loop limited to $|P_0|$ iterations. For simple aggregates like `max`, `min`, `sum`, `mean` and `variance`, the final aggregation over large numbers of represented program subsets may be done in constant time with the complexity of the equivalent analysis having the relatively small number of representative program subsets as its largest term.

```
FUNCTION PSAnalysisMPS(Set of programs $P_0$, Form of schema $f$)
  $\hat{P}_r =$ representative program subsets with respect to $P_0, f$
  Initialize aggregate $A$
  FOR EACH $P \in \hat{P}_r$
    FOR EACH cardinality $z$ of a program subset represented by $P$
      Compute a value $v$ based on analysis of the
        most specific schemata of form $f$ occurring in all programs of $P$
        and the cardinality $z$
      Aggregate $v$ into $A$ once per program subset of cardinality $z$
        that is represented by $P$
  RETURN $A$
END
```

Pseudocode 4.2: PSAnalysisMPS

4.4.2 Schema analysis

An alternative to the above analyses is analysis of individual schemata rather than program subsets. Pseudocode 4.3 presents a basic analysis algorithm.

```
FUNCTION SchemaAnalysis(Set of programs $P_0$, Form of schema $f$)
  $S$ = the set of all schemata of form $f$
  Initialize aggregate $A$
  FOR EACH $s \in S$
    Compute a value $v$ based on analysis of
      the subset of $P_0$ matching $s$ and
      the order of $s$
    Aggregate $v$ into $A$
  RETURN $A$
END
```

Pseudocode 4.3: SchemaAnalysis

The analysis proceeds as an iteration over the set of all schemata of the form f. Each schema is analyzed using its order (number of non-don't-care nodes) and the set of matching programs from P_0. The results are then aggregated into the final analysis result. As for the previous subsection, constraints are placed on the types of analysis possible, for example the schema may not perform analysis on its depth. But here too it is easy to think of interesting analyses which fit this general framework.

Using the new method's representative groups, we may refactor the algorithm of pseudocode 4.3 as having an outer loop over representative pairs and an inner loop over schema orders. The function is presented in pseudocode 4.4. The function operates as follows: The outer loop iterates over each representative set of schemata S.

```
FUNCTION SchemaAnalysisMS(Set of programs $P_0$, Form of schema $f$)

  $\hat{S}_r$ = representative sets of schemata with respect to $P_0, f$
  Initialize aggregate $A$
  FOR EACH $S \in \hat{S}_r$
    FOR EACH order $z$ of a schema represented by $S$
      Compute a value $v$ based on analysis of
        the subset of $P_0$ matching the schemata in $S$ and
        the order $z$
    Aggregate $v$ into $A$ once for each schema of order $z$
      that is represented by $S$
  RETURN $A$

END
```

Pseudocode 4.4: SchemaAnalysisMS

Any schema will be represented by some representative set of schemata and the two will share the same set of programs P from P_0. The inner loop iterates for each schema order z. The analysis result for cardinality z and programs P is repeatedly aggregated into A, once for each schema of order z that is represented by S. Thus rather than a single potentially expensive loop, we have an outer loop iterating over a presumed relatively small set of $|\hat{S}_r|$ items and an inexpensive inner loop limited by program size. As in the previous subsection, for simple aggregates like `max`, `min`, `sum`, `mean` and `variance`, the final aggregation over large numbers of represented schemata may be done cheaply with the complexity of the equivalent analysis having the relatively small number of representative sets of schemata as its largest term, rather than the number of all matched schemata.

4.5 Lattice forms of schema

For each maximal program subset P, $S_r(P, f)$ is a non-empty set of maximal schemata belonging to form f that occur in the programs of a program subset P. In important special cases of the form f it is the case that $|S_r(P, f)| = 1$ for every program subset P.

This is always the case for a specific class of forms of schema used widely in the remaining thesis. This thesis refers to forms in this class as *lattice* forms of schema. Lattice forms of schema must have two specific behaviours:

- The *Directed Acyclic Graph* (DAG) having as nodes the schemata of the form f, with the addition of a schema s_∞ (which occurs in no programs) and s_0 (which is more general than any conceivable schema) and having as edges the generality relation between those schemata, must be a *lattice*. That is, it must have the following properties:
    - For any two schemata s_a and s_b of the form, there *must* be a *join schema* $s_a \vee s_b$ which is the unique most general schema of

the form more specific than both s_a and s_b. Note that the join schema may be s_∞. For any distinct s_a and s_b, $s_a \vee s_b$ is more specific than both s_a and s_b and any other schema of the form that is also more specific than both s_a and s_b will certainly be more specific than $s_a \vee s_b$.

  – For any two schemata s_a and s_b of the form, there *must* be a *meet schema* $s_a \wedge s_b$ which is the unique most specific schema of the form more general than both s_a and s_b. Note that the meet schema may be s_0. For any distinct s_a and s_b with a meet schema $s_a \wedge s_b$, $s_a \wedge s_b$ is more general than both s_a and s_b and any other schema of the form that is also more general than both s_a and s_b will certainly be more general than $s_a \wedge s_b$.

- For any population P_0 and any two schemata s_a and s_b, it is true that $P_m(s_a \vee s_b, P_0) = P_m(s_a, P_0) \cap P_m(s_b, P_0)$. That is, the *join schema* for the two schemata must occur in any program from P_0 matching both s_a and s_b. Note that since it is more specific than s_a and s_b it could not match programs not matched by both s_a and s_b.

These basic requirements have flow-on effects and dealing with lattice forms of schema can be significantly easier than general forms of schema. Useful properties include:

- $|S_r(P, f)| \leq 1$ for any program subset P and form f.

  To see that this is true, suppose for some program subset P, $S_r(P, f)$ has two members: s_a and s_b. The join schema $s_a \vee s_b$ is more specialized than both and matches all programs which match both s_a and s_b and therefore matches all programs in P. Therefore neither s_a nor s_b is maximal with respect to P and f since the more specific $s_a \vee s_b$ could also be maximal with respect to P and f.

  Alternately, it may be that no schemata match all the programs of P and therefore $S_r(P, f) = \{\}$.

- If P_a and P_b are two maximal program subsets, then $P_a \cap P_b$ is either empty or maximal.

  As shown above, for any program subset P which matches any schema of a given form f there is a unique maximal schema with respect to P and f. Let $s_m(P, f)$ return this schema given a program subset and a form, that is the only member of $S_r(P, f)$.

  Given two maximal program subsets P_a and P_b, the join maximal schema $s_{a \vee b}$ is $s_m(P_a, f) \vee s_m(P_b, f)$. To show that, if it is non-empty, $P_a \cap P_b$ is maximal, it is enough to show that there is some schema which occurs in all programs of $P_a \cap P_b$ but no other program of P_0. $s_{a \vee b}$ is such a schema. By the second requirement of a lattice form of schema, $P_m(s_{a \vee b}) = P_a \cap P_b$. Note that since P_a is maximal it is true that $P_a = P_m(s_m(P_a, f))$, and similarly $P_b = P_m(s_m(P_b, f))$. Therefore $P_a \cap P_b$ is maximal. Note that the schema $s_{a \vee b}$ is not necessarily a maximal schema.

- If s_a and s_b are two maximal schemata, then if $s_a \wedge s_b$ exists in the form f, it is also maximal.

  In order to show that $s_a \wedge s_b$ is maximal, it is sufficient to show there is at least one program subset which matches $s_a \wedge s_b$ but no more specific schema. This subsection shows that $P_m(s_a, P_0) \cup P_m(s_b, P_0)$ is such an program subset.

  By its definition, $s_a \wedge s_b$ is the unique most specific schema more general than both s_a and s_b. Since s_a is maximal no superset of $P_m(s_a, P_0)$ matches any schema more specific than s_a, and since $|S_r(P_m(s_a, P_0), f)| = 1$, all schemata other than s_a occurring in all programs of $P_m(s_a, P_0)$ are more general than s_a. Similarly for s_b, also, all schemata other than s_b occurring in all programs of $P_m(s_b, P_0)$ are more general than s_b.

  Therefore, all schemata occurring in all programs in $P_m(s_a, P_0) \cup P_m(s_b, P_0)$ must be more general than both s_a and s_b. $s_a \wedge s_b$ is the unique most specific schema which is more general than both s_a and s_b. Therefore $s_a \wedge s_b$ is maximal since any other schema occurring in $P_m(s_a, P_0) \cup P_m(s_b, P_0)$ must be more general than $s_a \wedge s_b$. Note that $P_m(s_a, P_0) \cup P_m(s_b, P_0)$ is not necessarily maximal.

4.6 Conjunctive forms of schema

This thesis terms a specific subset of the class of lattice forms *conjunctive forms of schema*. All of the forms used in this thesis' experiments are either conjunctive forms of schema or based on conjunctive forms of schema.

Conjunctive forms of schema are important since schemata may be represented as sets of schema components and this step is vital to the implementation of many of the algorithms of the new method. Conjunctive forms of schema are relatively easy to work with, primarily since each schema of a conjunctive form may be represented as a set and for any two schemata of a conjunctive form there is a unique meet which may be easily found as the intersection of the two schemata when represented as sets.

A form of schema f is "conjunctive" if:

- Each schema may be expressed as a set of *schema components* derived from the form f.

  Schema components may be said to *occur in* programs, just as with schemata. Schema components may be more general, more specific or take on any relation to any other schema component.

- A given schema occurs in a program p if and only if p matches each schema component in the schema's set of schema components.

- For any two schemata s_a, s_b with non-disjoint schema component sets C_a, C_b there is either a schema in the form with $C_a \cap C_b$ as its schema component set or there is no schema in the form with any non-empty subset of $C_a \cap C_b$ as its schema component set.

- For any two schemata s_a, s_b with non-disjoint schema component sets C_a, and C_b, there is either a schema in the form with $C_a \cup C_b$ as its schema component set or such a schema would occur in no conceivable program.

All conjunctive forms of schema are lattice forms of schema:

1. For any two distinct schemata s_a, s_b with non-disjoint schema component sets C_a, C_b, if a schema s_c with schema component set $C_a \cap C_b$ exists in the form then $s_a \wedge s_b = s_c$, otherwise $s_a \wedge s_b = s_0$.

    - s_c is the most specialized schema which is more general than or equal to both s_a and s_b. $C_a \cap C_b$ is a subset of both C_a and C_b and therefore s_c is more general than s_a and s_b. All other schema for which this is true must have schema component sets which are subsets of $C_a \cap C_b$. In the case that $C_a \cap C_b$ is not in the form, there is no schema more general than both C_a and C_b and thus s_0 is used as a placeholder.

2. For any two distinct schemata s_a, s_b with non-disjoint schema component sets C_a, C_b, if a schema s_d with schema component set $C_a \cup C_b$ exists in the form then $s_a \vee s_b = s_d$, otherwise $s_a \vee s_b = s_\infty$.

    - $s_a \vee s_b$ is the most general schema which is more specific than or equal to both s_a and s_b. $C_a \cup C_b$ is a superset of both C_a and C_b and therefore s_d is more specific than s_a and s_b. All other schema for which this is true must have schema component sets which are supersets of $C_a \cup C_b$. In the case that $C_a \cup C_b$ is not in the form, there is no program which matches both C_a and C_b and thus the only schema more specific than both is s_∞.

3. The final requirement for the form to be a lattice form is for $P_m(s_a \vee s_b, P_0) = P_m(s_a, P_0) \cap P_m(s_b, P_0)$.

- Because $s_a \vee s_b = s_a \cup s_b$, the requirement follows logically: the set of programs matching all schema components in both s_a and s_b equals the intersection of the set of programs matching all schema components in s_a with the set of programs matching all schema components in s_b.

Some forms of schema lend themselves to this deconstruction into sets of schema components:

- Rooted-ordered-fragments.
- Rooted-ordered-hyperschemata

Other forms of schema may not have an easy decomposition to schema components:

- Any non-rooted schema, for instance: subtrees, numeric terminals and ordered-fragments.
- Rooted-unordered-fragments. Unordered schemata are seldom conjunctive.

4.6.1 Conversion to conjunctive forms of schema

Any non-conjunctive form of schema f has a corresponding conjunctive form f_{conj} where schemata of the form f_{conj} are sets of schemata of the form f. The schema components of the new, conjunctive form are the schemata of the old, non-conjunctive form. Therefore, for instance, sets of rooted-unordered-fragments is a conjunctive form of schema.

4.7 De-rooted conjunctive forms of schemata

Amongst the classes of non-conjunctive forms of schema, there is a relatively easy special case: conjunctive forms of schema used in a non-rooted way which are termed *de-rooted conjunctive forms*. Examples of de-rooted conjunctive forms are: ordered-subtrees (the non-rooted form of ordered-programs), ordered-fragments (the non-rooted form of rooted-ordered-fragments) ordered-hyperschemata and many others. Each conjunctive form of schema has a de-rooted conjunctive form.

In terms of match-tree schemata, a de-rooted conjunctive form f_{nr} for conjunctive form f_r has all branches from f_r but a new root branch, typically having a `desc` child-match function and pointing to f_r's root branch in its c_{ind} member. The `desc` child-match function causes the root branch of f_r to be passed all subtrees of the input programs, rather than the root programs, effectively making the form non-rooted by allowing the form to match any subtree. The `desc` node may also filter the nodes to, for instance, only those at a certain depth.

Thus the most part of the de-rooted conjunctive form is just the equivalent conjunctive form with the only non-conjunctive part of the form being a node pattern with a single `desc` disjunct. In fact, given an input set of programs P_0, the maximal schemata of the de-rooted conjunctive form with respect to P_0 are almost exactly the same as the set of maximal schemata of the relevant conjunctive form with respect to S_0, the set of subtrees from P_0 selected by the non-rooted form's `desc` node child-match function. There are, however, two key differences:

1. Each schema of the non-rooted form has a root node matching the `desc` node, this descendent node is not featured in schemata of the rooted form.

2. Some maximal schemata of the rooted form, run over subtrees, may not be maximal in the non-rooted form.

This second condition may only occur in one case: where for some schema s there is another maximal schema s_m in the same programs and some subtree of s_m is the same as or more specific than s using the rooted form.

Where to the rooted form they are the quite different schemata since they have different roots, to the non-rooted form s is more general than s_m.

4.7.1 Example one

As an example, let the rooted conjunctive form be programs and let the de-rooted conjunctive form be subtrees.

In a trivial case, the program set P_0 has only one program: $(+\ 1)$. Clearly, the maximal subtree is the program itself: $(+\ 1)$.

In order to use the algorithms of the previous sections, this simple analysis runs them on the subtrees of P_0, $S_0 = \{(+\ 1), 1\}$. The maximal programs with respect to S_0 are the subtrees themselves: $\{(+\ 1), 1\}$.

A final step removes each entry that is a more general than another entry, where the two are in the same programs. Thus removing the "1" entry, the analysis arrives at the correct singleton set of maximal subtrees.

4.7.2 Example two

Let the conjunctive form be rooted-ordered-fragments with the de-rooted conjunctive form being ordered-fragments. Let P_0 have two programs: $P_0 = \{(+\ 1\ 2\ 3), (+\ (+\ 1\ a)\ a\ 3)\}$

The set of subtrees S_0 is $\{1, 2, 3, a, (+\ 1\ 2\ 3), (+\ 1\ a), (+\ (+\ 1\ a)\ a\ 3)\}$ Running one of the previously described algorithms on S_0 using the rooted-ordered-fragment form the algorithm finds the following maximal schemata: $1, 2, 3, a$, $(+\ 1\ 2\ 3)$, $(+\ 1\ a)$, $(+\ 1\ \#)$, $(+\ \#\ \#\ 3)$, $(+\ (+\ 1\ a)\ a\ 3)$ and $(+\ \#\ a)$

Of these schemata:

- 2 and $(+\ 1\ 2\ 3)$ are in the first program only.
- $1, 3$, $(+\ 1\ \#)$ and $(+\ \#\ \#\ 3)$ are in both programs.
- a, $(+\ 1\ a)$, $(+\ (+\ 1\ a)\ a\ 3)$, $(+\ \#\ a)$ are in the second program only.

Thus the maximal ordered-fragments found after removing more general schemata are $(+\ 1\ 2\ 3)$, $(+\ 1\ \#)$, $(+\ \#\ \#\ 3)$ and $(+\ (+\ 1\ a)\ a\ 3)$.

There is one maximal program subset, that with both programs, which has more than one maximal schema which shows that the form used is non-conjunctive.

An important example using this conversion is for non-rooted forms of otherwise conjunctive forms of schema. Such forms are very seldom conjunctive since they are non-rooted but may be made conjunctive by regarding schemata as being sets of the non-rooted form. For instance, "sets of ordered-fragments" is the set form of the non-rooted form of the otherwise conjunctive form of "rooted-ordered-fragments". This thesis terms such forms *de-rooted conjunctive forms of schema* and are the only non-conjunctive forms of schema used in this thesis' experiments.

4.8 Chapter summary

This chapter presents the core concept of the new method: rather than reducing the complexity of the population to analyze or the form of schema to analyze, the new method reduces the algorithmic complexity of the analysis algorithm. It presents algorithms dependent, not on the very large sets of program subsets and schemata but on a presumed smaller set of *representative pairs*. Each such representative pair could potentially represent vast numbers of schemata and program subsets.

With certain restrictions on the type of analysis performed the chapter refactored a *naive* analysis, that iterates over the prohibitively large set of all schemata or program subsets, into an equivalent and achievable analysis of representative structures. The chapter defined general algorithms for achieving this analysis. In addition, this chapter defined *lattice*, *conjunctive* and *de-rooted conjunctive* forms of schema as useful classes of form of schema.

The following chapter defines several algorithms used to find these representative structures for a given population and a conjunctive or a de-rooted conjunctive form of schema, and in doing so it completes the new method.

Chapter 5

Algorithms of the new method

5.1 Chapter introduction

This thesis gives a method to analyze the match-tree schemata in sets of GP programs. The previous chapter defined the concepts of *maximal schema, maximal program subset, representative program subset, representative sets of subtrees*. These structures enable the compression of large numbers of schemata and program subsets in a population down to the ones required for typical analysis. This chapter presents algorithms to find these maximal and representative structures from a set of programs and a form of schema which is expressed as a match-tree form.

The combined system made up of the algorithms of this chapter takes a form of schema and a population of programs as input and produces an *annotated DAG of maximal pairs* as output. A flow diagram of this system is given in figure 5.1 The diagram shows that there are several steps in making the DAG and several classes of objects passed as input to and produced as output from each of these steps. The objects include:

- f: a form of schema specified as a match-tree form. This main system accepts only conjunctive match-tree forms. Section 5.9 gives an algorithm which generalizes this system to include de-rooted conjunctive

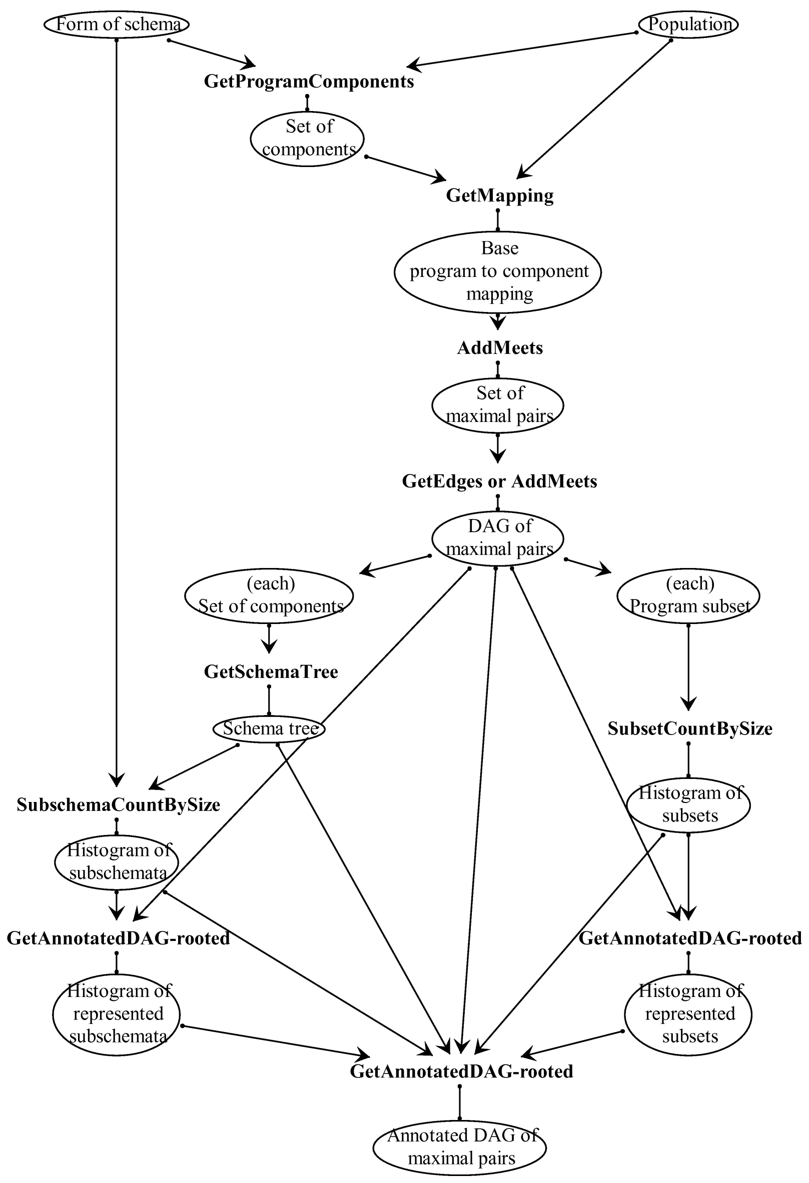

Figure 5.1: A flow diagram of the functions (bold text) and classes of objects (text in ellipses) which are used to produce the annotated DAG of maximal pairs.

forms of schema.

- P_0: a population to be analyzed. Together with the form of schema, this specifies the input to the system.
- C_0: a set of schema components. C_0 is returned by GetSchemaComponents which is described in section 5.4. The ability to describe any schema as a set of schema components is key to the new method. C_0 has all the schema components required to describe the maximal schemata for form f and population P_0.
- M: either a mapping from schema components to sets of programs or a mapping from programs to sets of schema components. M serves as a base for one of the AddMeets variants, which use it to construct a meet-semi-lattice which has the maximal pairs as nodes.
- A set of maximal pairs, each with a maximal program subset and a maximal schema. It is constructed by one of the variants of GetMaximalDAG of section 5.3, and has all maximal pairs with respect to f and P_0.
- R: an anti-transitive DAG of maximal pairs. R is constructed by either GetMaximalDAG while creating the set of maximal pairs or GetEdges of section 5.6. While R has no transitive edges, it has a path going from each maximal pair r to each maximal pair more general than r.
- A schema tree for each maximal schema. GetSchemaTree of section 5.7 constructs the tree from each maximal schema's set of schema components.
- A count structure for each maximal schema s counting the schemata of each order that are generalizations of s. SubschemaCountBySize of section 5.8 constructs this count structure from the maximal schema's set of schema components.

- A count structure for each maximal schema s counting the schemata of each order that are represented by s.
- A count structure for each maximal program subset counting its subsets of each cardinality. It is constructed by SubsetCountBySize of section 5.8.
- A count structure for each maximal program subset counting the subsets of each cardinality that it represents.
- The final annotated DAG of maximal pairs. Each maximal pair is made up of a maximal schema and a maximal program subset. In addition, each maximal schema s is annotated with:
  - s_{tree} which is the representation of s as a tree.
  - s_{cfull} which is a histogram of the number of subschemata of each order.
  - s_{crep} which is a histogram of the number of represented subschemata of each order.

  Each maximal program subset P is annotated with:
  - P_{cfull} which is a histogram of the number of subsets of each order.
  - P_{crep} which is a histogram of the number of represented subsets of each order.

5.1.1 Schema components

Schema components in this thesis are *rooted paths of schema nodes*. Noting that match-tree schemata are *trees of schema nodes*, any schema could be constructed as the union of a set of schema components.

Each schema node s has a label $s.v$ and several other fields as follows:

- $s.f_{disjunct}$ is the schema form disjunct used to make the schema node. Each schema node is associated with a single disjunct, and all children of the schema node are associated with a disjunct in the form's node pattern pointed to by $s.f_{disjunct}.c_{index}$.

- $s.f_n$ is a node-match function. In particular $s.f_n = s.f_{disjunct}.f_n$.

- $s.f_c$ is a child-match function. In particular $s.f_c = s.f_{disjunct}.f_c$.

- $s.P_{sub}$ is a set of program subtrees from programs in P_0 matching s. The label of the root node of each program subtree in $s.P_{sub}$ must match s. That is for each program subtree p in $s.P_{sub}$, if $p.v$ is the label of the root of p then $s.f_n(s.v, p.v)$. The children of the root node of each program subtree in $s.P_{sub}$ must match s if given some mapping from children of the schema node to descendents of the program node. That is for each program subtree p in $s.P_{sub}$, for some set of pairs M $s.f_c(s, p, M)$.

- $s.P$ is the set of programs from P_0 containing the subtrees in $s.P_{sub}$.

  The set $s.P$ with all programs from the population P_0 which match the schema component which ends in s. The set $s.P$ was used in the previous subsection to identify a mapping between programs and schema components.

A useful duality emerges from the use of schema components: a program subset is a set of programs, and a schema is a set of schema components. Also, the programs that match a schema are found as the programs that match each of the schema's schema components, and the schema components that occur in a program subset are found as the schema components that occur in each of the program subset's programs.

While there are typically many schemata, there are typically significantly fewer schema components. For conjunctive forms of schema with simple node-match functions subsection 5.4 presents an efficient algorithm finding the schema components. The algorithm takes a form of schema and a population of programs.

5.2 GetAnnotatedDAG-rooted: overall algorithm

Given a population P_0 and a form of schema f, the GetAnnotatedDAG-rooted algorithm presented in this section returns the *annotated DAG of maximal pairs*. Each maximal pair is annotated with counts of its represented program subsets and its represented schemata and a tree representation of its maximal schema. This algorithm works exclusively with conjunctive match-tree forms of schema, although later in section 5.9 an algorithm will generalize this algorithm to de-rooted conjunctive match-tree forms of schema.

The algorithm, presented in pseudocode 5.1, outputs a DAG R of pairs. Each pair $r \in R$ has a maximal schema r_s and a maximal program subset r_P. Further, each maximal schema s in any pair has a count of subschemata s_{cfull}, a count of represented subschemata s_{crep} and a representation as a tree s_{tree}. Each maximal program subset P has a count of subsets P_{cfull} and a count of represented subsets P_{crep}.

The operation of the GetAnnotatedDAG-rooted function is as follows:

- The function first passes the form of schema f and the population P_0 to the GetMaximalDAG algorithm, presented in section 5.3, producing R as the DAG of maximal pairs with respect to form f and population P_0.
- The function finds P_{cfull} for each maximal program subset P having a count of subsets of P of each cardinality. These counts are found using SubsetCountBySize which is presented in section 5.8.
- P_{crep} is found for each maximal program subset P as the count of the subsets represented by P. A simple mechanism is used: any

```
FUNCTION GetAnnotatedDAG-rooted ($f, P_0$)

  // $f$ is a match-tree form of schema
  // $P_0$ is a set of programs
  $R$ =GetMaximalDAG($f, P_0$)
  FOR EACH $< s, P >\in R$ from small $P$ to large $P$
    $P_{cfull}$ =SubsetCountBySize($P$)
    $P_{crep} = P_{cfull} - \sum_{<s',P'>\in R, P'\subset P} P'_{crep}$
  FOR EACH $< s, P >\in R$ from large $P$ to small $P$
    $s_{tree}$ =SchemaTree($s$)
    $s_{cfull}$ =SubschemaCountBySize($s_{tree}$)
    $s_{crep} = s_{cfull} - \sum_{<s',P'>\in R, P'\supset P} s'_{crep}$
  RETURN $R$

END
```

Pseudocode 5.1: GetAnnotatedDAG-rooted subset that is not represented by some other subset of P must be represented by P itself. Therefore P_{crep} is found as the sum of P'_{crep} over all maximal subsets of P subtracted from P_{cfull}.

- In a similar way, the function finds s_{crep} for each maximal schema s as the sum of represented counts of maximal subsets subtracted from a count of all subschemata s_{cfull} of s. Each s_{cfull} is found using SubSchemaCountBySize which is presented in section 5.8.

- The function uses SchemaTree, which is presented in section 5.7, to identify the tree representation of each maximal schema s.

GetAnnotatedDAG-rooted provides a way to obtain a DAG of maximal pairs along with information about each maximal pair, given a conjunctive match-tree form of schema and a population of programs. The following sections describe the functions that this overall function depends on.

5.3 GetMaximalDAG: maximal pairs as a DAG

This section describes an algorithm that is core to the operation of the overall system. The algorithm constructs a DAG of the maximal pairs, given a form of schema and a population of programs. There are several variations in how this task may be approached.

There are several options for the operation of GetMaximalDAG:

- 2 directions of a mapping between programs and schema components. The mapping may be from programs to schema components, or from schema components to programs.

- 2 variants of the AddMeets algorithm: IntersectMeet and PromoteIncrement.

- 2 ways to find the edges of the graph using GetEdges.

But each combination of these options defines an exhaustive algorithm which returns exactly the same DAG of maximal pairs.

The main four variations are called IntMPS, IntMS, ProMPS and ProMS.

The first two variations – IntMS (Intersect Maximal Schemata) and IntMPS (Intersect Maximal Program Subsets) – use an *exhaustive intersect* approach. The IntersectMeet algorithms find the meets, the intersections, of maximal program subsets or maximal schemata, by starting from a base set of large program subsets or schemata and exhaustively intersecting subsets of the base set:

- IntMS-simple/IntMS using a mapping from program to schema components, and IntersectMeet.

  The IntMS algorithms start with a base set of large maximal schemata and iteratively find smaller and smaller maximal schemata. This is equivalent to finding larger and larger program subsets.

- IntMPS-simple/IntMPS using a mapping from schema components to programs, and IntersectMeet.

  The IntMPS algorithms start with a base set of large maximal program subsets and iteratively find smaller and smaller maximal program subsets. This is equivalent to finding larger and larger schemata.

The other two variations – ProMS (Promote Maximal Schemata) and ProMPS (Promote Maximal Program Subsets) – use an *increment-promote* approach. The PromoteIncrement algorithms work in the opposite direction to the IntersectMeet algorithms, starting from small maximal program subsets or schemata and successively adding to them:

- ProMS-simple/ProMS using a mapping from schema components to programs, and PromoteIncrement. The ProMS algorithms start with a base set of general maximal schemata and find larger and larger maximal schemata. This is equivalent to finding smaller and smaller program subsets. The algorithm also constructs each maximal schema's respective maximal program subset.

- ProMPS-simple/ProMPS using a mapping from program to schema components, and PromoteIncrement. The ProMPS algorithms start with a base set of small maximal program subsets and find larger and larger maximal program subsets. This is equivalent to finding smaller and smaller schemata. The algorithm also constructs each maximal program subset's respective maximal schema.

It may be that a researcher desires only output nodes with specific properties, that is, researchers may wish to *filter* the output. Each base algorithm has a *direction*, either progressing from large program subsets (small schemata) to small program subsets (large schema) or vice-versa. Some algorithms (ProMS and IntMPS) allow us to efficiently filter the output to just the small maximal schemata and other algorithms (ProMPS and IntMS) can efficiently filter the output to just the large maximal schemata. Because of the way they work, the ProMS and ProMPS algorithms implement this type of filter well while IntMS and IntMPS implement it less efficiently.

5.3.1 GetMaximalDAG **common outer function**

The function GetMaximalDAG is presented in pseudocode 5.2. GetMaximalDAG operates as follows:

- First, it constructs the set of schema components to be used to make the maximal schemata. This operation is highly dependent on the form, and subsection 5.4 describes how to construct the schema components for conjunctive forms of schema.

- The construction of each schema component also constructs the set of programs from P_0 it occurs in. The second step in the algorithm

```
FUNCTION GetMaximalDAG ($f, P_0$)
  // $f$ is a match-tree form of schema
  // $P_0$ is a set of programs
  $C_0$= GetSchemaComponents($\textit{empty path}, P_0, 1, f, 1$)
  $M$ = a mapping $\mathcal{A} \rightarrow \{\mathcal{B}\}$, which is either
    a mapping from each $p \in P_0$ to its matched schema components from $C_0$ or
    a mapping from each $c \in C_0$ to its matching programs from $P_0$
  $R$ = AddMeets($M$)
  IF elements of $R$ are sets of $\mathcal{B}$'s
    $R' = \{\}$
    FOR EACH $B \in R$
      $A = \{\}$
      FOR EACH $< a' \rightarrow B' > \in M$ where $B' \supseteq B$
        $A = A \cup \{a'\}$
      $R' = R' \cup < A, B >$
    $R = R'$
  IF $R$ is a set, not a DAG
    $R$ =GetEdges($R$)
  RETURN $R$

END
```

Pseudocode 5.2: GetMaximalDAG

constructs a mapping M between programs and schema components based on this set for each schema component.

In one alternative of GetMaximalDAG, M maps each program p from P_0 to the subset of C_0 with elements that occur in p. In this case the rest of the algorithm refers to programs as type "$\mathcal{A}$" and refers to schema components as type "$\mathcal{B}$". In the alternative of GetMaximalDAG, M maps each schema component c from C_0 to the matching programs of P_0 that was constructed with c. In this case $\mathcal{A}$'s and $\mathcal{B}$'s are reversed in the rest of the algorithm, which refers to schema components as type "$\mathcal{A}$" and refers to programs as type "$\mathcal{B}$".

Either of these alternatives results in the same final DAG of maximal pairs.

- One of the variants of AddMeets from section 5.5 uses M to construct either a set or DAG of either maximal pairs or maximal sets of $\mathcal{B}$'s.
- If the return value R of the selected AddMeets variant is a set of maximal sets of $\mathcal{B}$'s, then GetMaximalDAG uses a nested loop to extend each maximal set of $\mathcal{B}$'s with the matching maximal set of $\mathcal{A}$'s, thus creating a set of maximal pairs.
- If R is a set, then the edges required to make it an anti-transitive DAG are added by GetEdges which is presented in section 5.6

Despite having several options for how it operates, the returned value of GetMaximalDAG is always the same anti-transitive DAG of maximal pairs, given the same form f and population P_0.

5.4 GetSchemaComponents: schema components

The first operation in the algorithm GetMaximalDAG constructs a set C_0 with schema components. This subsection presents a function GetSchemaComponents obtaining C_0 using a recursive procedure to find all possible paths of schema nodes from the root of a schema to some node in the tree. The algorithm is initially called as GetSchemaComponents(*empty path*, $P_0, 1, f, 1$) where P_0 is a population of programs and f is a match-tree form of schema. The function accumulates a set C and after the function returns C contains the schema components to be used for C_0.

The algorithm works on each disjunct n of one of the disjunctive node patterns of the form f. For each such disjunct, the algorithm adds a new schema component c consisting of a new schema node s, based on the disjunct n, appended to a supplied parent schema component c_{par}. Some schema components may be more general than other schema components but in the same programs, and these schema components are safely removed since they are not needed; any maximal schema containing the more general schema component would also contain the more specific schema component.

GetSchemaComponents is presented in pseudocode 5.3. The operation of the function is as follows:

- The function contains an outer loop over all form disjuncts of the given tree.

- Each such node n may form the basis of a schema node to be appended to the path c_{par}.

  This part of the algorithm uses an operation derived from the node-match function $n.f_n$. The operation obtains the schema node labels matching a given program node label $p.v$, that is the $s.v$'s such that $n.f_n(s.v, p.v)$. This operation is dependent on $n.f_n$.

  - For the *exact string match* node-match function $s.v = p.v$.

  - For the *don't care with prefix* node-match function there is one $s.v$ for each possible prefix of $p.v$.

  - For a plain *don't care* node-match function $s.v =$"".

```
FUNCTION GetSchemaComponents ($c_{par}, P, i_P, f, t$)

  // $c_{par}$ is a schema
  // $P$ is a set of program subtrees
  // $i_P$ is a schema node child index (as in $c_c$ is the $i_P$ th child of $c_p$)
  // $f$ is a match-tree form
  // $t$ is a disjunctive node pattern in form $f$.
  Let $C$ be a global output set of schema components
  $S = \{\}$
  FOR EACH disjunct $n$ in $t$
    $P' =$ each $p \in P$ having a number of children in the range $n.c_{num}$
    $L = \{\}$
    FOR EACH program $p$ in $P'$
      $L = L \cup$ each schema node label $v$ with $n.f_n(v, p.v)$
    FOR EACH schema node label $v$ in $L$
      $v.P = \{p : p \in P', n.f_n(v, p.v)\}$
    FOR EACH schema node label $v$ in $L$
      if no $v' \in L$ is more specific than $v$ where $v.P = v'.P$
        $s.v = v$
        $s.f_n = n.f_n$
        $s.f_c = n.f_c$
        $s.f_{disjunct} = n$
        $s.P_{sub} = v.P$
        $s.P =$ programs from $P_0$ containing the subtrees in $s.P_{sub}$
        $S = S \cup \{s\}$
  FOR EACH $s$ in $S$
    if no $s' \in S$ is more specific than $s$ where $s.P = s'.P$ and $s.f_c = s'.f_c$
      $c = s$ appended to $c_{par}$ as child number $i_P$
      $C = C \cup c$
      FOR EACH $i$ th partition $P_c$ of child nodes of nodes in $s.P_{sub}$ using $s.f_c$
        GetSchemaComponents($c, P_c, i, f, s.f_{disjunct}.c_{index}$ th node pattern in $f$)
  RETURN $C$

END
```

Pseudocode 5.3: GetSchemaComponents

  - For other node-match functions the operation is more complex, and it may even result in an infinite set. An example is when numeric schema nodes which match a continuous range of program node labels are used.

- $v.P$ is the set of program subtrees matching each label v.

- The function creates schema nodes out of the most specific labels for each distinct set of program subtrees

- The function creates schema components out of the most specific schema nodes, appended to the parent schema component c_{par}. The resultant schema component c is added to the output set C.

- For each such schema node s the function then uses an iterator derived from the child-match function $s.f_c$. This iterator is dependent on the operation of the child-match function and steps through the partitions of children for a given set of program nodes, at each child index i. The following are examples of this operation:

  - For the ordered `cind` child-match function the i^{th}partition has the i^{th}children of the program nodes.
  - For the partly-ordered `cpind` child-match function each partition associates program node children that are the j^{th}child with label v, for each possible combination of j and v

- The function recurses to find the possible extensions to the schema component c.

5.5 AddMeets: get meets from mapping

The functions of this section, collectively called "AddMeets", are given a base mapping $M \in \mathcal{A} \rightarrow \{\mathcal{B}\}$ provided by GetMapping, where either $\mathcal{A}$'s are programs and $\mathcal{B}$'s are schema components or vice-versa. Each variant provides a meet-semi-lattice closure of the mapped-to sets of $\mathcal{B}$'s as either maximal program subsets or maximal schemata. In addition, some also produce the maximal pairs and some also produce the edges of the DAG containing these pairs.

There are two variants, each having a quite different mode of operation:

- IntersectMeet, presented in section 5.5.1, uses the intersection operator to find each meet set of $\mathcal{B}$'s.

  Any maximal set of $\mathcal{B}$'s is the intersection of some sets mapped to in M. IntersectMeet performs the intersection over all such combinations of sets mapped to in M.

- PromoteIncrement, presented in section 5.5.2, uses a *promote-increment* approach.

  Adding any schema component to a maximal schema s and promoting to its representing maximal schema leads to a maximal schema which is more specific than s. Adding any program to a maximal program subset P and promoting to its representing maximal program subset leads to a maximal program subset which is a proper superset of P.

  The PromoteIncrement algorithms employ this *promote-increment* method using a function which, given a maximal schema or maximal program subset, finds all more specific maximal schemata or larger maximal program subsets respectively.

The makeup of this returned set may take one of three forms depending on the function used:

- A set of sets, having the members of the meet-semi-lattice closure of the sets mapped to in M. These are the maximal schemata or maximal program subsets.

- A set of pairs, having each member B of the meet-semi-lattice closure of the sets mapped to in M, paired with the set of elements which in M map to a superset of B. These are the set of pairs each with a maximal schema and its matching maximal program subset.

- An anti-transitive directed-acyclic-graph (DAG) with each maximal pair arranged either with a path from each schema to each more specific schema or with a path from each schema to each more general schema.

5.5.1 IntersectMeet: exhaustive intersection

Summary

The algorithms of this subsection return, given a mapping of $\mathcal{A}$'s to sets of $\mathcal{B}$'s, the meet-semi-lattice closure of intersections on those sets. That is, the returned set has as a member the intersection of the mapped to sets in each subset of the input mapping. Therefore, the returned set is a superset of the set of mapped-to sets of the input mapping and contains the nodes of a meet-semi-lattice by the subset relation. Though often implied, the empty set is not stored in the output set.

This section describes two algorithms of varying efficiency:

- First, the simpler algorithm IntersectMeet-simple is presented.

- Next, IntersectMeet-hash is presented, being identical in input and output but with an expected improvement in efficiency.

IntersectMeet-simple

This simpler implementation of IntersectMeet adds to an output set $\hat{B}$, which is initially empty, by two nested loops.

The combination of these loops provides the meets for each set B mapped to in M with each meet made from intersections of previously seen B's.

The function adds each such meet to $\hat{B}$ which, at the start of the inner loop, is the minimal meet-semi-lattice which is a superset of the set of previously seen B's.

IntersectMeet-simple is presented in pseudocode 5.4. The operation of

```
FUNCTION  IntersectMeet-simple ($M$)

  // $M$ is a mapping from $\mathcal{A}$'s to sets of $\mathcal{B}$'s
  Set of sets of items $\hat{B} = \{\}$
  FOR EACH mapping $a \to B \in M$
    Mark all mappings in $\hat{B}$ as old
    FOR EACH $B' \in \hat{B}$
      IF $B'$ has been marked as old
        $\hat{B} = \hat{B} \cup \{B \cap B'\}$
    $\hat{B} = \hat{B} \cup \{B\}$
  RETURN($\hat{B}$)

END
```

Pseudocode 5.4: IntersectMeet-simple

the function is as follows:

- The outer loop iterates over the members of the mapping M.
- For each mapped-to set B the inner loop iterates over each B' in an accumulated output set $\hat{B}$.
- Given each combination of B and B', the function simply adds their intersection to the output set unless the B' is "new" and $B \cap B'$ would equal B'.
- Lastly, each B is itself added to the output set at the end of its outer loop.

Let us assume that at the start of the inner loop the set $\hat{B}$ holds the smallest meet-semi-lattice which is a superset of that portion of the mapped-to sets of M so far seen as B in the outer loop. The inner loop adds each intersection of B with a member of this minimal meet-semi-lattice to $\hat{B}$. After this process $\hat{B}$ must contain the meet between B and each previously found meet. Therefore $\hat{B}$ has a meet between B and each subset of the previously seen B's.

The function does not add a set to $\hat{B}$ unless it is the meet for some subset of the mapped-to sets of M, therefore, after the function adds B itself to $\hat{B}$, $\hat{B}$ is the smallest meet-semi-lattice which is a superset of that portion of the mapped-to sets of M so far seen as B, including the current B in the outer loop.

Therefore, the previously assumed loop invariant holds for the next iteration of the outer loop.

The set $\hat{B}$ would typically be implemented as a hash-set for fast member-inclusion testing.

Complexity

Let the mapping M have $|A|$ members and the mapped to sets in M be no larger than $|B|$. The outer loop iterates over $|A|$ sets of items. If R is the set returned, the inner loop iterates through worst case $|R|$ sets of items.

Each iteration of the inner loop performs an intersection of two sets of items. If the size of the largest set is $|B|$ then this process has worst case complexity $O(|B|)$. The result is added to a hash set with complexity $O(|B|)$ to calculate the hash value for the set of items and $O(1)$ to add to the hash set assuming no collisions.

Thus a worst case complexity for the algorithm is $O(|A||R|(|B| + (|B| + 1)))$ or about $O(|A||B||R|)$.

IntersectMeet-hash

The operation of IntersectMeet-hash is similar to that of IntersectMeet-simple. The function has two parts: a setup phase and a loop along the lines of that in IntersectMeet-simple.

The setup phase assigns each $\mathcal{B}$ a random integer $b_{randhash}$ and the function later uses this integer to provide a fast way to test properties about the intersection between two sets of items.

The looping phase iterates over the sets mapped to in M. For each such member B the algorithm identifies each previously found meet B' where $B' \not\subset B, B \not\subset B', B' \cap B \neq \{\}$. The meet $B' \cap B$ to the output set. The key improvement of IntersectMeet-hash over IntersectMeet-simple is the speedy test for these "interesting" meets and therefore the avoidance of performing an expensive intersection for the many "uninteresting" members of R. In particular, two methods should improve this algorithm's relative efficiency:

- The random integers assigned to each item may be used to very quickly identify if a particular pair of sets of $\mathcal{B}$'s have a meet of interest.

- A hash set efficiently prevents any given meet from being identified twice.

IntersectMeet-hash is presented in pseudocode 5.5. The operation of the function is as follows:

- The initial setup phase of the algorithm sets up $b_{randhash}$ for each $\mathcal{B}$ as a *large* random integer

  In particular, $b_{randhash}$ must to be large enough that for any two sets B_1 and B_2 expected in the output, in all but pathological cases: $\sum_{b \in B_1} b_{randhash} = \sum_{b \in B_2} b_{randhash} \Leftrightarrow B_1 = B_2$.

  The sum of these random hashes over the items making the intersection of two sets of items B and B' will be zero if they are disjoint and

```
FUNCTION IntersectMeet-hash ($M$)

  // $M$ is a mapping from $\mathcal{A}'$s to sets of $\mathcal{B}'$s
  Set of sets of $\mathcal{B}'$s $\hat{B}$ = all mapped-to sets in $M$
  FOR EACH $b$ in any member of $\hat{B}$
    SET FIELD $b_{randhash}$ =large pseudo-random number
  Set of sets of $\mathcal{B}'$s $R = \{\}$
  FOR EACH $B \in \hat{B}$
    FOR EACH $B' \in R$
      SET FIELD $B'_h = 0$
    FOR EACH $b \in B$
      FOR EACH $B' \in R$ with $b$ as a member
        $B'_h = B'_h + b_{randhash}$
    Hash set of sets of items $H_m = \{\}$
    FOR EACH $B' \in R$
      IF $B'_h \neq 0 \wedge B'_h \neq \sum_{b \in B} b_{randhash}$
        IF no $B'' \in H_m$ has $B''_h = B'_h$
          IF no $B'' \in R$ has $\sum_{b \in B''} b_{randhash} = B'_h$
            $H_m = H_m \cup \{B'\}$
    FOR EACH $B' \in H_m$
      $R = R \cup \{B \cap B'\}$
    $R = R \cup \{B\}$
  RETURN($R$)

END
```

Pseudocode 5.5: IntersectMeet-hash

it must be that $B \subseteq B'$ or $B' \subseteq B$ if this sum is equal to the equivalent sum over the items of B or B' respectively. In addition, if the sums for two meets are found to be the same, we may assume that the meets themselves are the same.

Each of these cases indicates that the meet is "uninteresting" and the lack of these conditions indicates it is "interesting".

- The next phase of the algorithm iterates over each mapped-to set B in M, with the aim to add B to a set R, as well as the meet of B with each previous member of R. In this way IntersectMeet-hash works similarly to IntersectMeet-simple.
- While IntersectMeet-simple found each meet explicitly, IntersectMeet-hash does this in several steps:
  - For each previously found meet B', the hash for the meet of B and B' is found as the sum of $b_{randhash}$ for each item b in $B \cap B'$.
  - The function adds to the hash set H_m each B' which has a meet indicating that: B and B' are non-disjoint, neither B nor B' is a subset of the other and the meet would form an as yet unidentified meet.

    Since two sets of items B', B'' which have the same hash, that is $B'_h = B''_h$, will have the same meet with B therefore the hash set stores only one set of $\mathcal{B}'$s for each distinct B'_h.
- Iterating through the set H_m, the actual meets are found and stored in R.

Finally, the function adds B itself to R.

Complexity

Let the mapping M have $|A|$ members and a limit of $|B|$ on mapped-to set size. Also let the output set have $|R|$ members.

The initial setup phase, assigning a random number to each $\mathcal{B}$ has complexity $O(|A||B|)$.

The main part of the algorithm is performed for each mapped-to set B from mapping M, a total of $|A|$ times.

Each $B' \in R$ has constant time initialization performed with complexity $O(|R|)$. The next loop, finding the B'_h for each of these sets may be implemented with complexity $O(|R||B|)$.

Assuming constant-time additions to a hash set and tests for inclusion of a hash in a hash set, the next loop over the members of R proceeds with worst case complexity $O(|R|)$.

The final inner loop proceeds in $O(|R||B|)$ time since H_m may have at most $|R|$ members and identifying the meet $B \cap B'$ has worst case complexity $O(|B|)$. No given meet may occur in H_m in more than on iteration of the outer loop through B since each of these meets is new.

Summing up the algorithm has a total worst-case complexity of $O(|A||B| + |A|(|R| + |R||B| + |R|) + |B||R|) = O(|A||B| + |A||B||R| + |B||R|)$

Comparing to the complexity of IntersectMeet-simple, we find the same worst case complexity of $|A||B||R|$. Despite this, the algorithm is expected to be more efficient in practice.

5.5.2 PromoteIncrement

The algorithms in this subsection find maximal program subsets and maximal schemata by exploiting a property of what this thesis means by "maximal": any proper superset a of a maximal schema a_m is represented by a maximal schema which is itself a proper superset of a_m. Similarly, any proper superset a of a maximal program subset a_m is represented by a maximal program subset which is itself a proper superset of a_m. By using this property, combined with a method for "promoting" the schema/program subset a to its maximal representative, the algorithms of this section may efficiently find the desired set of maximal program subsets/schemata.

In terms of the meet-semi-lattice, the meet of a collection of sets is the largest set which has each set in the collection as a superset, that is the intersection of the sets. Any set which is a one-step addition to a given meet is be represented by a proper superset of the original meet.

One considerable advantage of these algorithms over those described previously in section 5.5.1 is that, with little extra complexity, they may arrange the output sets of maximal program subsets and maximal schemata into an anti-transitive directed acyclic graph (DAG) with edges indicating a general/specific relation. The new method requires the edges of this DAG in order to count the number of program subsets or schemata represented by each maximal program subset or maximal schema and this count in turn allows more powerful analysis. By contrast, the algorithms of section 5.5.1 require an additional expensive algorithm to identify the edges in this DAG.

This section describes two algorithms equivalent in input and output:

- A simple algorithm, PromoteIncrement-simple.
- A more complicated algorithm, PromoteIncrement-hash with improved complexity.

PromoteIncrement-simple

PromoteIncrement-simple is the simpler of the two algorithms utilizing this promote/increment approach to finding the meet-semi-lattice of maximal schemata and maximal program subsets. The main function PromoteIncrement-simple in turn calls a recursive function recurse on singleton set $\{a\}$ for each key a found in the given map M. recurse then promotes this singleton set to its representing maximal set, A_m, which is stored in the output set R. In the following *increment* step the recurse function recurses on each possible one-step addition to the input singleton set A. Each one-step-addition set is represented by a proper superset of the A_m from the calling instance of recurse.

Thus recurse promotes the one-step-addition set and the resultant representing set is added to R. recurse then recurses on each minimally larger set.

PromoteIncrement-simple is presented in pseudocode 5.6. The operation of the PromoteIncrement-simple is as follows:

```
FUNCTION  PromoteIncrement-simple ($M$)

  // $M$ is a mapping from $\mathcal{A}'$s to sets of $\mathcal{B}'$s
  Set of pairs $R = \{\}$
  FOR EACH $< a \rightarrow B > \in M$
    recurse($\{a\}$)
  RETURN($R$)

END
FUNCTION  recurse($A$)

  // $A$ is a set of sets of $\mathcal{A}'$s
  initialize set $B_m$
  FOR EACH $a$ in $A$
    $B_m = B_m \cap \{b : b \in B, < a \rightarrow B > \in M\}$
  FOR EACH $b$ in $B_m$
    $A_m = A_m \cap \{a :< a \rightarrow B' > \in M, B' \supseteq B_m\}$
  IF $< A_m, B_m > \notin R$
    $R = R \cup < A_m, B_m >$
    FOR EACH $< a \rightarrow B > \in M$ where $a \notin A_m$
      recurse($\{a\} \cup A_m$)

END
```

Pseudocode 5.6: PromoteIncrement-simple

- The main function, PromoteIncrement-simple, presents the recursive procedure with each singleton set of $\mathcal{A}'$s.

- The recursive function, recurse takes a set of $\mathcal{A}$'s, A, which is not necessarily maximal.
- recurse then finds the maximal set of $\mathcal{B}$'s for A as B_m.
- A similar operation finds the maximal set of $\mathcal{A}$'s for B_m. This process produces A_m which must represent A. The process also, usefully, finds it's mate B_m. If A_m is a maximal schema, B_m will be the largest program subset matching all schema components of A_m. If A_m is a maximal program subset, B_m will be the largest set of schema components occurring in all programs of A_m.
- The recurse function recurses on one-step-additions to the maximal set A_m. It also restricts unnecessary repeated calls by testing whether a given A_m has previously been found and only recursing if it hasn't.

Each of the initial singleton sets may or may not be maximal, but any set of $\mathcal{A}$'s that is maximal must be a superset of one or more of the singleton sets. Any maximal set of $\mathcal{A}$'s must either represent one of these singleton sets or must be a proper superset of a maximal set which represents one of these singleton sets. Thus given that the function recurse finds all maximal sets of $\mathcal{A}$'s which are supersets of the input set, the outer call to PromoteIncrement-simple will find all desired maximal sets of $\mathcal{A}$'s.

Complexity

Let the mapping M have $|A|$ members and a limit of $|B|$ on mapped-to set size. Also let the output set have $|R|$ members.

The IF statement in recurse ensures that the function doesn't recurse unless A_m is new to the output set, thus the statement after the IF is called exactly $|R|$ times. Therefore, recurse is called no more than $|R||A|$ times.

Each call to recurse entails finding B_m ($O(|A||B|)$) then finding A_m (also $O(|A||B|)$). Thus the initial part of recurse, called $|R||A|$ times, contributes $O(|R||A|^2|B|)$ to the complexity.

The statements after the if are called $|R|$ times with $O(|A|)$ each time and thus do not contribute to the overall worst case complexity.

Thus the complexity of the call to PromoteIncrement-simple is $O(|R||A|^2|B|)$ where there are $|R|$ maximal pairs found, there are $|A|$ $\mathcal{A}$'s in M and there are $|B|$ $\mathcal{B}$'s mapped to in M.

Let the mapping M have $|A|$ members and a limit of $|B|$ on mapped-to set size. Also let the output set have $|R|$ members.

The IF statement in recurse ensures that the function doesn't recurse unless A_m is new to the output set, thus the statement after the IF is called exactly $|R|$ times. Therefore, recurse is called no more than $|R||A|$ times.

Each call to recurse entails finding B_m ($O(|A||B|)$) then finding A_m (also $O(|A||B|)$). Thus the initial part of recurse, called $|R||A|$ times, contributes $O(|R||A|^2|B|)$ to the complexity.

The statements after the if are called $|R|$ times with $O(|A|)$ each time and thus do not contribute to the overall worst case complexity.

Thus this complexity of the call to PromoteIncrement-simple is $O(|R||A|^2|B|)$ where there are $|R|$ maximal pairs found, there are $|A|$ $\mathcal{A}$'s in M and there are $|B|$ $\mathcal{B}$'s mapped to in M.

PromoteIncrement-hash

PromoteIncrement-hash is similar in operation to the previously described algorithm, also utilizing a promote/increment approach to finding the desired maximal schemata and maximal program subsets. But the increment step has been enhanced to improve the algorithm's complexity.

As with PromoteIncrement-simple, the main function PromoteIncrement-hash in turn calls a recursive function recurse on a number of base sets, however, in contrast to the singleton sets used by PromoteIncrement-simple PromoteIncrement-hash uses *minimally maximal* sets of $\mathcal{A}'s$, that is the smallest maximal proper supersets of the input set, which are found using a function findMinimals.

recurse is only ever called with a maximal set as input. It then finds the respective maximal set of $\mathcal{B}'$s as B_m, thus completing the maximal pair $< A_m, B_m >$. The returned set contains each such pair found. In the case where the recurse function hasn't yet been called with this A_m, the function recurses on each minimally maximal proper superset of A_m. These supersets are found using the function findMinimals.

PromoteIncrement-hash is presented in pseudocode 5.7. The operation of PromoteIncrement-hash is as follows:

- The function is passed a mapping from $\mathcal{A}'$s to sets of $\mathcal{B}'$s, where either $\mathcal{A}'$s are programs and $\mathcal{B}'$s are schema components or vice-versa.
- As for the previously described IntersectMeet-hash function, each $\mathcal{B}$ passed to the function via the mapping M is annotated with a "large random integer". This integer should be large enough that the sum of the annotations for two typical sets of $\mathcal{B}'$s will be equal only if the sets are equal.
- The findMinimals function, presented in pseudocode 5.8, is used to find the very smallest maximal pairs and the recurse function is used to add these to the output DAG R and find all more general maximal pairs if the $\mathcal{A}'$s are program, or all more specific maximal pairs if the $\mathcal{A}'$s are schema components.
- The recurse function takes a maximal set of $\mathcal{B}'$s as B_m and a pair r_{par} as the parent if any of the maximal pair containing B_m.
- The function first checks whether the pair containing B_m exists in the DAG. If it has, the function simply adds an edge to r_{par} from this pair and exits.
- Otherwise, the maximal pair is new.
  - The function finds the respective maximal set of $\mathcal{A}'$s, A_m, for B_m.

```
FUNCTION PromoteIncrement-hash ($M$)

  // $M$ is a mapping from $\mathcal{A}'$s to sets of $\mathcal{B}'$s
  $A_0 = \{a :< a \rightarrow B >\in M\}$
  $B_0 = \{b \in B :< a \rightarrow B >\in M\}$
  FOR EACH $b \in B_0$
    SET FIELD $b_{randhash}$ = large random integer
  Set of pairs $R = \{\}$
  FOR EACH $B_m \in$ findMinimals($\{\}, B_0$)
    recurse($B_m$,null)
  RETURN($R$)

END
FUNCTION recurse($B_m, r_{par}$)

  // $B_m$ is a set of $\mathcal{B}'$s
  // $r_{par}$ is a pair with a set of $\mathcal{A}'$s and a set of $\mathcal{B}'$s
  IF for some $A'$, $< A', B_m >\in R$
    IF $r_{par}$ is non-null, put edge from $< A', B_m >$ to $r_{par}$ in DAG
  ELSE
    $A_m = \cap_{b \in B_m} \{a :< a \rightarrow B' >\in M, B' \supseteq B_m\}$
    $R = R \cup < A_m, B_m >$
    IF $r_{par}$ is non-null, put edge from $< A_m, B_m >$ to $r_{par}$ in DAG
    FOR EACH $B'_m \in$findMinimals($A_m, B_m$)
      recurse($B'_m, < A_m, B_m >$).

END
```

Pseudocode 5.7: PromoteIncrement-hash

- It then adds an edge from the new pair to r_{par} if any.
- It then recurses on each maximal subset of B_m. These sets are found using the findMinimals function.

  Each such set is maximal, is a subset of B_m. Each set is also *minimally smaller*, that is for each set B'_m of $\mathcal{B}'$s found by getMinimals, there is no other maximal set of $\mathcal{B}'$s which is both a subset of B_m and a superset of B'_m.

The function findMinimals, used by PromoteIncrement-hash, is presented in pseudocode 5.8. The operation of the function is as follows:

```
FUNCTION findMinimals($A, B$)

  // $A$ is a set of $\mathcal{A}'$s
  // $B$ is a set of $\mathcal{B}'$s
  FOR EACH $a \in A_0 \backslash A$
    SET FIELD $a_{hash} = \sum_{b \in B \cap B' : < a \rightarrow B' > \in M} b_{randhash}$
  MAKE $\hat{A} \in \{\{\mathcal{A}\}\}$ as grouping of members of $a \in A_0 \backslash A$ by $a_{hash}$ value
  Set of sets of $\mathcal{B}'$s $\hat{B} = \{\}$
  FOR EACH $A' \in \hat{A}$
    CHOOSE any $a' \in A'$
    $\hat{B} = \hat{B} \cup \{B \cap B' :< a' \rightarrow B' > \in M\}$
  Remove each member $B_m$ from $\hat{B}$ where $\exists B_n \in \hat{B} : B_m \subset B_n$
  RETURN $\hat{B}$

END
```

Pseudocode 5.8: findMinimals

- The function groups each $a \in A_0 \backslash A_m$ by the subset of input set B to which it maps in M.

  As A is maximal and B has all $\mathcal{B}'$s which are mapped to by all elements of A, there could be no such a which maps to all of B. Instead, the set for any given a is the maximal set of $\mathcal{B}'$s for $A \cup \{a\}$.

- $\hat{A}$ groups possible a's into those with different matched subsets of B.
- The construction of $\hat{B}$ then explicitly finds each such subset of B.
- A final step removes members of $\hat{B}$ which are subsets of some other member of $\hat{B}$.

Complexity

The main body of the recurse function executes exactly once for each maximal pair for a total of R times. It finds A_m from B_m at a cost of $O(|A_0||B_0|)$. It then calls findMinimals.

The findMinimals function gets the hash values for the members of $A_0\backslash A$ at worst case complexity $O(|A_0||B_0|)$. Done by a hash set, grouping may be done in $O(|A_0|)$. Finding the maximal set of $\mathcal{B}'$s for each member of $\hat{A}$ takes $O(|B_0|)$ and thus constructing $\hat{B}$ has worst case complexity $O(|A_0||B_0|)$. The step removing subsets from $\hat{B}$ may be done in $O(|A_0||B_0|)$ time by identifying the largest containing set in $\hat{B}$ for each member of $A_0\backslash A$.

Thus the call to recurse has worst case complexity $O(|A_0||B_0|)$ and the call to PromoteIncrement-hash has worst case complexity $O(|R||A_0||B_0|)$ where there are $|R|$ maximal pairs, there are $|A_0|$ $\mathcal{A}'$s in M and there are $|B_0|$ distinct $\mathcal{B}'$s in the mapped-to sets of M.

The main body of the recurse function executes exactly once for each maximal pair for a total of R times. It finds A_m from B_m at a cost of $O(|A_0||B_0|)$. It then calls findMinimals.

The findMinimals function gets the hash values for the members of $A_0\backslash A$ at worst case complexity $O(|A_0||B_0|)$. Done by a hash set, grouping may be done in $O(|A_0|)$. Finding the maximal set of $\mathcal{B}'$s for each member of $\hat{A}$ takes $O(|B_0|)$ and thus constructing $\hat{B}$ has worst case complexity $O(|A_0||B_0|)$. The step removing subsets from $\hat{B}$ may be done in $O(|A_0||B_0|)$

time by identifying the largest containing set in $\hat{B}$ for each member of $A_0 \backslash A$.

Thus the call to recurse has worst case complexity $O(|A_0||B_0|)$ and the call to PromoteIncrement-hash has worst case complexity $O(|R||A_0||B_0|)$ where there are $|R|$ maximal pairs, there are $|A_0|$ $\mathcal{A}$'s in M and there are $|B_0|$ distinct $\mathcal{B}$'s in the mapped-to sets of M.

5.6 GetEdges: edges of the DAG

This section presents an algorithm to find the edges of a graph based on the subset relation which are to be used alongside the variants of AddMeets which return a set of maximal pairs rather than the required DAG of maximal pairs. Specifically, the nodes of the graph are input as a set of maximal pairs and the output graph is an anti-transitive DAG where there is a path from each maximal pair $< s, P >$ to each other, more general, maximal pair $< s', P' >$ where $P \subset P'$ and therefore $s \supset s'$.

The algorithm assigns the edges from each maximal pair $< s, P >$ in turn, descending on program set size. An inner loop identifies each program subset which is a superset of P and adds it as an outgoing edge from $< s, P >$ if none of its subsets is also a superset of P.

Since the outer loop moves from large program set size to small, the edges from each maximal pair that has a superset of P as its program subset must have been found by the time P is visited. The inner loop moves from small program set size to large so that a basic marking scheme inhibits the addition of an edge from a node a to a node a if the edge from a to a subset of b exists.

The function is presented in psuedocode 5.9. The algorithm uses a count P'_{share} to test whether a given set P' is a superset of P, resulting in slightly improved efficiency over the use of a more standard subset/superset test.

```
FUNCTION GetEdges ($R$)

  // $R$ is a set of maximal pairs
  FOR EACH $< s, P >\in R$ from large $P$ to small $P$
    FOR EACH $< s', P' >\in R : |P'| > |P|$
      SET FIELD $P'_{share} = 0$
      SET FIELD $P'_{minimal}$ =true
    FOR EACH $p \in P$
      FOR EACH $< s', P' >\in R : p \in P' \wedge |P'| > |P|$
        $P'_{share} = P'_{share} + 1$
    FOR EACH $< s', P' >\in R : |P'| > |P|$ from small $P'$ to large $P'$
      IF $P'_{share} = |P|$
        IF $P'_{minimal}$ =true
          ADD EDGE from $< s, P >$ to $< s', P' >$
        FOR EACH $< s'', P'' >$ with an incoming edge from $< s', P' >$
          $P''_{minimal}$ =false
  RETURN $R$

END
```

Pseudocode 5.9: GetEdges

5.7 GetSchemaTree: tree representation of a schema

This section gives another core algorithm of the new method. The AddMeets algorithm produces as output a DAG of maximal pairs, each with a set of programs and a schema represented as a set of schema components. The algorithm of this section obtains a tree representation for the schema of a maximal pair, given its set of schema components and its form of schema.

The input to the original call to the GetSchemaTree function is a set of schema components C each member of which was originally produced by the GetSchemaComponents algorithm of the previous subsection. Each schema component is in fact a path of schema nodes. The set of schema components in C is produced by the AddMeets function as the set representation of a schema and combine in the function of this subsection to form the tree representation of the schema. It must be noted that not all subsets of the complete set of schema components for a given form and a given population makes up a valid schema. For example, if two schema components from C_0 have incompatible values, for instance "+" for one and "-" for the other, at their root then they may not be combined into a schema tree. However in no conceivable program would two such schema components exist and therefore AddMeets would not produce any one maximal schema that contains both.

The function given in pseudocode 5.10 takes two arguments: C and a schema node c_{par}. C is assumed to form a valid schema. Given a parent schema node in s_{par}, the function finds all most specific schema components which extend this parent. Initially the base level schema components are used. It then recurses on each found schema component and the final tree is the smallest tree of schema nodes which is more specific than each passed schema component when viewed as a path of schema nodes.

The operation of the algorithm is as follows:

- The set C_{ch} is formed as all schema components ending in a potential

```
FUNCTION   GetSchemaTree ($C, s_{par}$)

  // $C$ is a set of schema components. Each is a schema tree rooted path
  // $s_{par}$ is a schema tree node
  $C_{ch}$ = members of $C$ with $s_{par}$ as the penultimate schema node
    or singleton paths from $C$ if $s_{par}$ =null
  $C'_{ch}$ = most specific members of $C_{ch}$
  $S_{ch}$ = final nodes in paths from $C'_{ch}$
    (that is each path no more specific path in $S'_{ch}$)
  FOR EACH child $s_{ch}$ in $S_{ch}$
    GetSchemaTree($C$, $s_{ch}$)
  IF $s_{par}$ ≠null
    $s_{par}$.children = $S_{ch}$ and is sorted and filtered based on $s_{par}.f_{disjunct}.f_c$
      (e.g. if $s_{par}.f_{disjunct}.f_c$ =cind, order by matching subtree child index)
  ELSE RETURN the only member of $S_{ch}$

END
```
Pseudocode 5.10: GetSchemaTree

child of s_{par}. If this is the first call to the function, then C_{ch} is formed as the set of root level schema components.

- The set C'_{ch} is formed from C_{ch} by removing schema components that are more general than other schema components. The child index of position of each node in each path of schema nodes matters in testing whether it is more general than or more specific than another schema node. During this step more-general schema components like *don't-care* nodes can be removed if there is a specific node like a "+" node at the same position.

- The set S_{ch} is formed as the end nodes of paths in C'_{ch} and thus forms the children of s_{par} ,although some may be discarded in a later step.

- The function recurses, thus finding for each child s_{ch} the set of children $s_{ch}.\mathrm{children}$. The children are in order as required by the child-match function $s_{ch}.f_{disjunct}.f_c$

- The children in S_{ch} are either returned or they are sorted, filtered and stored as the children of s_{par}. On the outermost call to the function for a valid schema this set will be a singleton containing the schema tree. The sorting and filtering process depends on the child-match function for the form disjunct which made the schema component of s_{par}, which is found as $s_{par}.f_{disjunct}.f_c$. Most child-match functions in conjunctive schemata impose some order on the children of a schema node. For example, it may impose the same order as the children of matched program nodes. Some may impose filters. For example, some child-match functions may disregard children after the first empty child index. This step allows the child-match function to impose these kinds of specific behaviour.

Thus the function gives an efficient way to form the tree representation of a schema from the set representation of the schema that was produced by the AddMeets function.

5.8 Counts of subschemata and subsets

This section presents two algorithms, one to count subsets and another to count subschemata.

The first function, SubsetCountBySize, provides a histogram by subset size of the number of subsets of a given program subset. This value, the number of subsets of a given set P that are of size i, is readily available mathematically as $\binom{|P|}{i}$ and SubsetCountBySize is accordingly simple. The function is presented in psuedocode 5.11. The second more

```
FUNCTION SubsetCountBySize ($P$)

  // $P$ is a set of programs
  Array $A$
  FOR $i : 1 \leq i \leq |P|$
    $A[i] = \binom{|P|}{i}$
  RETURN $A$

END
```

Pseudocode 5.11: SubsetCountBySize

complex algorithm, SubschemaCountBySize, provides a histogram, by schema size, of the number of valid subschemata of a given schema. To do this the function uses the previously defined GetSchemaTree to obtain a tree representation of the given schema and then annotates each subtree with the histogram of variations possible of that particular subtree. Thus histogram obtained for the root is our desired output.

The histogram of variations for any subtree may be found using the following:

- The histograms for the children of the subtree-root schema node n.
- The form disjunct n_f used by the subtree-root schema node n.

- The alternate disjuncts which are more general than n_f in the disjunctive node pattern from the form containing n_f.
- The number of potentially matching node values more general than s, using the disjuncts' label-match functions.
- The type of child-match behaviour exhibited by the disjuncts.

Since this information does not depend on values of nodes in the schema other than in the subtree at n, dynamic programming may be used, moving from the leaves of the schema tree to the roots, annotating each node with a histogram, resulting in the desired histogram for the root.

The function is presented in psuedocode 5.12. The operation of the function is as follows:

- The algorithm consists of a dynamic programming main function SubschemaCountBySize which obtains the histogram for each subtree by using the countForSubtree subroutine.
- countForSubtree obtains the histogram of subschemata for a given tree or subtree t assuming that the histograms for child subtrees have been found and the exact form disjunct which produced the schema node at the root s of t is available as $s.f_{disjunct}$.
- The function loops over all possible more general schema nodes, including combinations of:
  - The match-tree form disjunct f.
  - The node size z.
  - The labels allowed by the disjunct's label-match function.
  - The children allowed by the disjunct's child-match function.
- The function adds the count A', taking into account all these factors, to an overall count in A. Each count is a histogram with an integer for each possible order of schema.

```
FUNCTION SubschemaCountBySize ($t$)

  // $t$ is a schema tree
  FOR EACH subtree $t'$ of $t$ from leaves to root
    $t'_{hist}$ =countForSubtree($t'$)
  RETURN $t_{hist}$

END
FUNCTION countForSubtree (Tree $t$)

  Array $A$
  $s = t_{root}$
  FOR EACH disjunct $f$ in the disjunctive node pattern containing $s.f_{disjunct}$
    FOR EACH size $z$ of schema node
      $N_s$ = number of nodes of size $z$ allowed by label-match function $f.f_n$ ...
        which are the same as or more general than $s$
      IF $N_s > 0$
        FOR EACH combination $T$ of child subtrees of $s$ which ...
          ... would be allowed by child-match function $f.f_c$
          Array $A'$
          $A'[z] = S_n$
          FOR EACH $t' \in T$
            FOR EACH $k$: $A'[k] = \sum_i A'[i]t'_{hist}[k-i]$
          $A = A + A'$
  RETURN $A$

END
```

Pseudocode 5.12: SubschemaCountBySize

Thus the function gives an efficient way to count the subschemata of a given schema.

5.9 De-rooted conjunctive forms

It is important to note that the algorithms presented so far work exclusively with conjunctive match-tree forms of schema as defined previously in section 4.6. Non-conjunctive forms of schema prove relatively difficult to work with for a number of reasons:

- Any set of schema of a conjunctive form has a single *meet schema* which is the most specific schema of the form that is more general than each member of the set. Sets of schemata of a non-conjunctive form may have more than one most specific schema that is more general than each member of the set.

  Each conjunctive form of schema has a representation for schemata where each schema is a set of schema components and, using this representation, the meet schema for a given set of schemata is the intersection of the members of the set. By contrast this is seldom the case for non-conjunctive forms of schema. There seems no similarly efficient way to find the meet schema or schemata of two or more schemata for non-conjunctive forms.

- Where there is an efficient algorithm to find the string representation of a schema of a conjunctive form, no efficient algorithm was found to perform this task in general over non-conjunctive forms.

- Where there is an efficient algorithm to count the number of schemata represented by a given maximal schema of a conjunctive form, no efficient algorithm was found to perform this task in general over non-conjunctive forms.

Significant effort went into constructing a method to analyze non-conjunctive forms of schema, resulting in a system able to find maximal schemata from any match-tree form. But the method proved too inefficient and complex to present here.

Amongst the classes of non-conjunctive forms of schema there is a relatively easy special case: the de-rooted conjunctive forms defined in the previous chapter. Finding the maximal schemata of a de-rooted conjunctive form of schema f on population P_0 may be treated as similar to finding the sets of maximal schemata of the matching conjunctive form for f run on the subtrees of P_0. The GetAnnotatedDAG-de-rootedalgorithm, given in pseudocode 5.13, performs this process and generalizes the GetAnnotatedDAG-rooted algorithm to de-rooted conjunctive forms of schema. GetAnnotatedDAG-de-rooted is given a de-rooted conjunctive form of schema as f_{nr} and a set of programs as P_0. It finds the representative pairs with respect to a given population P_0 for de-rooted conjunctive forms of schema.

The function operates as follows:

- It uses GetAnnotatedDAG-rooted to find the maximal pairs R_r for the relevant rooted-conjunctive form of schema f_r with respect to the subtrees Q_0 of the programs in P_0.

  Each of the desired maximal schemata with respect to form f_{nr} and set of programs P_0 may be found in R_r.

- Each maximal pair for the conjunctive form will have a schema s_r and a set of subtrees Q_r of programs in P_0. For each such s_r the algorithm finds as P_r the set of programs that match s_r. This is found as the subset of P_0 with some subtree in Q_r.

  P_r will be returned as a representative program subset.

- GetAnnotatedDAG-non-rooted then uses the function AddMeets-DAG described in subsection 5.5 to both group the schemata from the maximal pairs in R_r into those with the same P_r and to construct a meet-semi-lattice from the program sets. This results in an anti-transitive

```
FUNCTION GetAnnotatedDAG-de-rooted ($f_{nr}, P_0$)

  // $f_{nr}$ is a de-rooted match-tree form of schema
  // $P_0$ is a set of programs
  LET $f_r$ be the rooted conjunctive form for $f_{nr}$
  $n =$ desc node at root node of root branch of $f_{nr}$
  $Q_0 =$ subtrees of program in $P_0$ at depth specified by $n$
  $R_r =$GetAnnotatedDAG-rooted($f_r, Q_0$)
  $M$=Map: schema $\rightarrow$ set of programs.
  FOR EACH $< s_r, Q_r > \in R_r$
    $P_r =$ the subset of $P_0$ with each program having a subtree in $Q_r$
    ADD MAPPING $s_r \rightarrow P_r$ to $M$
  $R =$AddMeets-DAG($M$, selected variant of AddMeets)
  FOR EACH $< S, P > \in R$ from small $P$ to large $P$
    $P_{cfull} =$SubsetCount($P$)
    $P_{crep} = P_{cfull} - \sum_{<s',P'>\in R, P'\subset P} P'_{crep}$
  FOR EACH $< S, P > \in R$ from large $P$ to small $P$
    $S_{crep} = \sum_{s\in S:(\nexists <S',P'>\in R: P'\subset P, s\in S')} s_{crep}$
    REMOVE each $s$ from $S$ where $\exists s' \in S, s'$ more specific than $s$
  RETURN $R$

END
```

Pseudocode 5.13: GetAnnotatedDAG-de-rooted

DAG of pairs, each with a set of schemata and a program subset. The meet nodes added may be seen to represent sets of schemata but no single schema on its own.

Though the call to AddMeets-DAG is expensive, it is essential since the later steps in this algorithm require the graph R be a meet-semi-lattice.

- The function finds the program subset counts in the same way as GetAnnotatedDAG-rooted.

- The function finds the counts of schemata represented by a set of schemata S as the sum of the counts of schemata represented by the individual schemata in S.

- Members of S are then removed if they are more general than some other member of S and therefore are not maximal since they are more general than another schema that occurs in the same programs.

- In a final step, the most specific schemata from pairs with supersets of P as their program subsets are added to S. This ensures that the set S has all of the most specific schemata occurring in all its programs.

  The inclusion of this step in the algorithm depends on whether the desired schemata are really the sets of schemata, for example "sets of subtrees", in which case it should be included or if they are the individuals in the sets of schemata, for example "subtrees", in which case it should be omitted.

- Thus this function provides a way to find the representative pairs with respect to a population of programs and a de-rooted conjunctive form of schema.

5.10 Chapter summary

This chapter has provided several algorithms involved in the new method presented by this thesis. Each adds some component to the overall system of enumerating the DAG of representative pairs given a population and form of schema. This DAG contains all the representative program subsets, representative sets of schemata, maximal program subsets and maximal schemata.

The algorithms fall into several broad classes:

- Overall algorithms, presented in sections 5.2 and 5.9, which invoke and coordinate the lower-level algorithms. One algorithm works with conjunctive forms of schema and another works with de-rooted conjunctive forms of schema. The representation of forms of schema is as match-tree forms.
- Algorithms preparing a base mapping for which the meet semi-lattice closure is the set of maximal program subsets or maximal schemata.
- Algorithms finding the meet semi-lattice closure of a given base mapping which are called the AddMeets variants.
- An algorithm finding the edges of an anti-transitive DAG based on the subset relation, given this DAG's nodes as sets.
- An algorithm finding the schema components of a given form, occurring in a given population.
- An algorithm finding the tree representation of a schema, given its schema components.
- Algorithms counting subsets and subschemata.

Together, these algorithms produce a DAG of representative pairs, which may be used by the algorithms given previously in chapter 4. Of these, the core algorithms are the AddMeets variants. Each variant does the same job and produces the same maximal pairs, but the different variants are expected to have different complexity in practice. Part of the next chapter, the first to show the new method in use, will compare the many options for this part of the new method. The following two chapters will present the results of numerous experiments using the new method.

Chapter 7 will demonstrate the efficacy of the new method using sample analyses. Beforehand, the following chapter will characterize how the new method works in practice, over many values for several parameters such as the population size, program size and AddMeets variant used.

Chapter 6

Characterizing experiments

6.1 Chapter introduction

Previous chapters have presented a powerful new method for analysis of GP schemata. Chapter 3 defined the new *match-tree form of schema language* and later chapters defined a method to analyze match-tree schemata shared between the programs of any given population.

This chapter aims to characterize the new method by assessing the time requirements, the space requirements and the size of the output set, over a range of parameter values like population size, program size and generation number.

6.2 Experimental setup

This chapter provides summaries of the results of many runs of the new method's algorithms, the "schema engine", using as input populations produced during evolution by a GP system, the "GP engine".

To provide consistency between the different experiments, whenever a parameter like program size was altered, the remaining parameters assumed default settings. Table 6.1 on page 143 lists default GP engine settings. Table 6.2 on page 145 lists default schema engine settings.

Each set of parameters was run over 50 randomized trials, achieved by supplying an otherwise identical GP setup with different initial random number generator seeds. All results presented are either the distribution over all the trials, quartiles over the trials or the median of the trials if only one value is given. The mean value is not given in the results presented here due to the existence for all parameter settings of outliers of unknown value, typically at least one trial of the 50 would exceed the "early-stopping" limits on the number of representative pairs or the time allowed by the schema engine. The existence of outliers still allows for the calculation of the distribution, median and quartiles of the trials but disallows the accurate calculation of the mean and standard deviation of the trials.

Some figures have "run distribution" as the x axis label, which means that the results of many runs were sorted, and the figure shows this ascending line of results from multiple runs.

6.2.1 GP engine settings

The GP engine served to provide the schema engine with the sorts of populations that a typical GP system might produce as the target of analysis. Table 6.1 tabulates the default settings used. The GP system used, VGP4 which was written by and freely available from the author, uses standard subtree crossover and mutation with the exception that crossover produces one offspring rather than two. Which of the two children typically produced by crossover is randomly chosen. The two tasks include the often used "Wisconsin Breast Cancer Database" dataset ("BCW") and a symbolic regression task "Sphere" chosen for its suitability for producing building blocks.

The "BCW" task is commonly used for testing genetic programming systems as one of several "standard" datasets. Its use here is to show the

Table 6.1: Common GP engine settings

Parameter	**Default value**
GP system	"Victoria Genetic Programming" (*VGP4*)
Task (fitness function)	(BCW) Wisconsin Breast Cancer Database – (January 8, 1991) (9 features, 699 patterns) (Sphere) symbolic regression to $x^2 + y^2 + z^2$ – (3 features, 2000 patterns)
Selection	Tournament with size 4
Mutation	Standard subtree, rate = 30%
Crossover	Standard subtree, rate = 60%
Reproduction rate	remaining 10%
Population size	101
Minimum program size	5 nodes
Maximum program size	60 nodes
Function set	*addition, subtraction, multiplication, division, if*
Function arity (except *if*)	2.5. Half of the nodes have arity 2, and half arity 3
Terminal set	Numeric and feature
Numeric terminal block size	0.1

performance of the system on real-world data. Several other commonly used datasets could equally have been used.

The "Sphere" task involves producing the function close in phenotype to $F1^2 + F2^2 + F3^2$ for three features and there are several partial solutions of increasing fitness, especially including combinations of $F1^2$, $F2^2$, and $F3^2$. The fitness function for the Sphere task randomizes the actual points in feature space each generation so that no over-fitting could occur.

The GP engine used tournament selection with a tournament size of four. Mutation rate and crossover rate were 30% and 60% respectively. Elitism copied the top 10% of programs from any given generation to the next.

The experiments use the unusual population size of 101 programs, rather than 100. Other experiments also use population sizes like 51 and 501. The reasoning behind the extra 1 is to allow us to look at, for example programs of rank 1, 26, 51, 76 and 101, while still including the best and worst programs. For similar reasons each run recorded 101 generations including the random initial as generation 0 then 1-100.

Since the arity of programs' function nodes was a parameter changed between runs and runs used an arities of up to four, programs were *node-limited* rather than *depth-limited*; programs were subject to a minimum number of nodes of five and a default maximum number of nodes of 60 and up to 80 in some experiments. Using a high arity even a small depth limit could produce very large programs. The problem is avoided by using a node-limiting setting.

The *numeric terminal block size* refers to the "quantum" of the random number generator used to determine the value of an allocated numeric terminal. For example, if the block size is 0.1 then the values 0.1 and 1.6 could be allocated, but not 0.35 or 1.601, although all these values would be possible with a setting of 0.001.

The division function was protected with sufficiently small divisors producing a quotient of zero. Along with addition and multiplication, both the subtraction and division functions allowed more than two arguments and in all these cases the operation associates from the left to right. As an example the program $(-\ 1\ 2\ 3\ 4)$ in Lisp syntax computes the expression $(((1-2)-3)-4)$.

The "if" function takes three arguments and returns the second argument when the first argument is negative and the third argument otherwise. Experiments include this function in order to allow discontinuity in the output of programs using it.

6.2.2 Schema engine settings

While the GP engine aims to produce typical GP populations, the Schema engine implements the new method presented in this thesis. The schema engine was run on populations provided by the GP engine without actually influencing the production of those populations by the GP engine in any way. The schema engine has a number of parameters and default settings. Table 6.2 tabulates these parameters.

Table 6.2: Common schema engine settings

Parameter	**Default value**
Schema engine	PhDVgp2 (built for this thesis)
Label-match functions	Don't care with no prefix (written #) Exact string match (appended !, e.g. "F1!")
Child-match functions	`cind, cindx, cpind, any, desc1`
Generation	0, 20 or 40
addMeets algorithm variant	ProMP
Forms of schemata	Rooted-ordered-fragments
Max. num. representative pairs	200000

The author developed the schema engine used, "PhDVgp2", specifically for this thesis. PhDVgp2 is freely available under an open-source licence. The schema engine allows forms to have two types of label-match function and five types of child-match function:

- *Don't care* label-match function: matches any program node.

- *Exact string match* label-match function: matches any program node with the same value as a string other than the trailing exclamation mark. For example the schema node ":1.0!" would match the program node ":1.0" but not ":1.01" or ":1".

- `cind` child-match function: schema node s matches any program node p where p has at least as many children as s and each i^{th}child of s matches the i^{th}child of p.

- `cindx` child-match function: schema node s matches any program node p where p has the same number of children as s and each i^{th}child of s matches the i^{th}child of p.

- `cpind` child-match function: schema node s matches any program node p where p has at least as many children as s and for each label v of a child node of s, the i^{th}child of s that has the label v matches the i^{th}child of p that has the label v.

  The `cpind` function gives a medium between the *ordered* behaviour of `cind` and purely *unordered* behaviour, called *partly-ordered* behaviour. Forms using `cpind` may still be conjunctive, where using purely unordered child-match function typically leads a form to be non-conjunctive.

- `any` child-match function: matches any program node.

- `desc1` child-match function: schema node s matches any program node p where the single child of s matches some descendent of p. The depth at which this match may occur is within a given range and some settings may allow p itself to be matched to the child of s. In this thesis this depth range was set to admit any descendent of p or p itself.

  The `desc1` child-match function was only ever used as the root node of schemata, effectively making a form using it non-rooted. For example prepending a `desc1` node appropriately in the form representation of rooted-ordered-fragments may produce the form representation of ordered-fragments.

Many variants of the addMeets algorithm were implemented. Section 6.6 presents a comparison of the variants and other sections use the default variant of ProMP. Note that all variants would produce the same set of representative pairs; changing the variant used should affect only the time taken and the memory used.

For most experiments, a limit of 200,000 representative pairs was in place. The schema engine stopped runs which exceeded this number and assigned the data-point as "unknown but large". A notable exception is the experiments of section 6.6 where there was no limit of numbers of representative pairs but rather a time limit of five minutes.

6.3 Lines of verifying experiments

This chapter has a number of experiments characterizing how this thesis' methods work in practice. These experiments test the method in four ways. The first set of experiments, presented in section 6.4, aim to verify that the many different variants of the addMeets algorithm agree, that is, each algorithm to find the DAG of representative pairs given a form and population finds exactly the same DAG of representative pairs as any other variant given that form and population.

The second set of experiments, presented in section 6.5, characterize the new method with default parameters. The parameters varied between runs included the trial and the generation number. Several measures of each combination of generation and trial are presented:

- Number of representative pairs.
- Time to find the representative pairs and breakdown of time taken by each process.
- Memory required by each run of the schema engine.
- Size of result on disk and breakdown by type of object stored.

The third set of experiments, presented in section 6.6, compare the time taken by the different addMeets-variant algorithms in finding each representative pair DAG.

The fourth set of experiments, presented in section 6.7, characterize the new method by varying a single parameter at a time, other than trial and generation, including:

- Maximum program size (from 20 nodes to 80 nodes).
- Population size (from 26 programs to 501 programs)
- Function arity (from two to four)

6.4 Equivalence of different addMeets variants

The first set of experiments aimed to verify that each variant of the addMeets algorithm from section 5.3 of the previous chapter produced the exact same DAG of representative pairs.

Each algorithm was run independently on over 3300 variations of parameters:

- 2 tasks (BCW and Sphere)
- 3 forms (rooted-ordered fragments, rooted-partly-ordered-subtrees, rooted-ordered-hyperschemata)
- Populations from 11 generations from each run of evolution (the initial random population and populations from generations 10 to 100 in intervals of 10)
- 50 randomized trials supplying different random seeds to the GP engine

Each run derived two hash values, each 64 bits, from its results:

1. A hash over the representative pairs, not including the DAG edges.

2. A hash over the representative pairs, including the DAG edges.

The first of these would be expected to be the same for two runs only if they produced the same set of representative pairs, while the second of these would be expected to be same for two runs only if they produced the same DAG of representative pairs.

Eight algorithms were tested: ProMS, ProMS-simple, ProMPS, ProMPS-simple, IntMS, IntMS-simple, IntMPS, and IntMPS-simple. To test the basic algorithms, each algorithm was run, finding a set of representative pairs then using two alternatives to find the DAG edges: CD and PD as defined in section 5.6 of the previous chapter. In addition, the advanced algorithms ProMS, ProMPS, IntMS, and IntMPS were run in a way optimized to find the DAG edges along with the representative pairs.

Thus for each set of parameters the experiments of this section compared 20 variants of addMeets: 8 algorithms, each with a CD variant and a PD variant, plus 4 more advanced algorithms. Rather than a limit on the number of representative pairs found, the experiments were early stopped after five minutes. Algorithms which went over this time limit did not contribute to the results. This time limit was seldom reached and only ever for the difficult rooted-ordered-hyperschema form of schema.

The happy result is that there were no discrepancies between any of the variants, either in representative pair set hash or representative pair DAG hash; any two runs on the same set of parameters, where both runs finished, produced exactly the same DAG of representative pairs excepting the negligible possibility of a hash collision. This is a good indication that the variants would return the same result given new, unseen input. This result could also be seen as an indication that the variants of the algorithm work correctly since it is less likely they would all work incorrectly in the same way.

6.5 Characterizations with default parameters

The second set of experiments saw the schema engine run with default parameters over two tasks, two population sizes, eleven forms of schema listed in table 6.3, 50 random seeds and 21 generations including the initial random population then generations 5-100 in steps of five. In addition, for five of the seeds the schema engine was run on each of the 101 generations. Thus the schema engine was run a total of 63800 times, taking a matter of

Table 6.3: Forms used to characterize the method with default parameters

Form	**Match-tree Lisp representation**
Ordered-programs	[1:(<,!,cind,[0,∞],1>)]
Restrictive-ordered-programs	[1:(<,!,cindx,[0,∞],1>)]
Rooted-partly-ordered-subtrees	[1:(<,!,cpind,[0,∞],1>)]
Rooted-ordered-fragments	[1:(<,#,any,[0,0],1> <,!,cind,[0,∞],1>)]
Rooted-restrictive-ordered-fragments	[1:(<,#,any,[0,0],1> <,!,cindx,[0,∞],1>)]
Rooted-ordered-hyperschemata	[1:(<,#,any,[0,0],1> <,#,cind,[1,∞],1> <,!,cindx,[0,0],1> <,!,cindx,[1,∞],1>)]
Ordered-subtrees	[1:(<,#,desc1,[1,1],2>) 2:(<,!,cind,[0,∞],2>)]
Restrictive-ordered-subtrees	[1:(<,#,desc1,[1,1],2>) 2:(<,!,cindx,[0,∞],2>)]
Partly-ordered-subtrees	[1:(<,#,desc1,[1,1],2>) 2:(<,!,cpind,[0,∞],2>)]
Ordered-fragments	[1:(<,#,desc1,[1,1],2>) 2:(<,#,any,[0,0],2> <,!,cind,[0,∞],2>)]
Restrictive-ordered-fragments	[1:(<,#,desc1,[1,1],2>) 2:(<,#,any,[0,0],2> <,!,cindx,[0,∞],2>)]

a few days on cluster of fifteen identical PCs.

This section lists the results, characterising the numbers of objects found in subsection 6.5.1, the time it took to find them in subsection 6.5.2 and the space consumed on disk in subsection 6.5.3, and in memory in subsection 6.5.4.

6.5.1 Numbers of representative pairs

The new method summarizes the full set of schemata down to a set of representative pairs. The number of such pairs affects both the size of the summary produced by the new method on disk and the time taken to analyze using the new method.

This subsection presents the results of experiments characterizing how many representative pairs are typically produced by the new method. The number of representative pairs for a given population and form of schema is the same as the number of maximal program sets and is the same as the number of maximal schemata if a rooted form is being used. Each representative pair of a non-rooted form may hold more than one maximal schema.

Figures 6.1, 6.2, 6.3, and 6.4 present plots for this value at generation 40 for eleven forms of schema.

Looking at these figures it is clear that the distribution for all schemata is highly biased. Many runs had small numbers of representative pairs and maximal schemata with a few having very large numbers. For the rooted forms of schemata very few of the runs exceeded the upper limit of 200,000 representative pairs and those that did could at times have more than 1,000,000 representative pairs. The non-rooted forms of ordered-subtrees and restrictive-ordered-subtrees also have few large values. But the distribution for non-rooted forms of schema is more biased with many small and few large numbers of representative pairs.

The different forms of schema had very different numbers of representative pairs with median values over 50 trials presented in table 6.4 on page 152. As expected, there are many more representative pairs for non-rooted forms of schema than for rooted forms of schema. Also as the complexity and expressiveness of a form of schema increases, for instance

Table 6.4: Median number of representative pairs found for different forms of schema for population sizes 51 programs and 101 programs

Form	**Median representative pairs**	
	51 progs	101 progs
Restrictive-ordered-programs	664	1,698
Ordered-programs	742	1,915
Restrictive-ordered-subtrees	3,620	15,784
Rooted-partly-ordered-subtrees	5,266	22,341
Rooted-restrictive-ordered-fragments	6,612	24,971
Rooted-ordered-fragments	6,893	27,681
Rooted-ordered-hyperschemata	19,453	109,747
Ordered-subtrees	19,875	$\geq$200,000
Partly-ordered-subtrees	$\geq$200,000	$\geq$200,000
Ordered-fragments	$\geq$200,000	
Restrictive-ordered-fragments	$\geq$200,000	

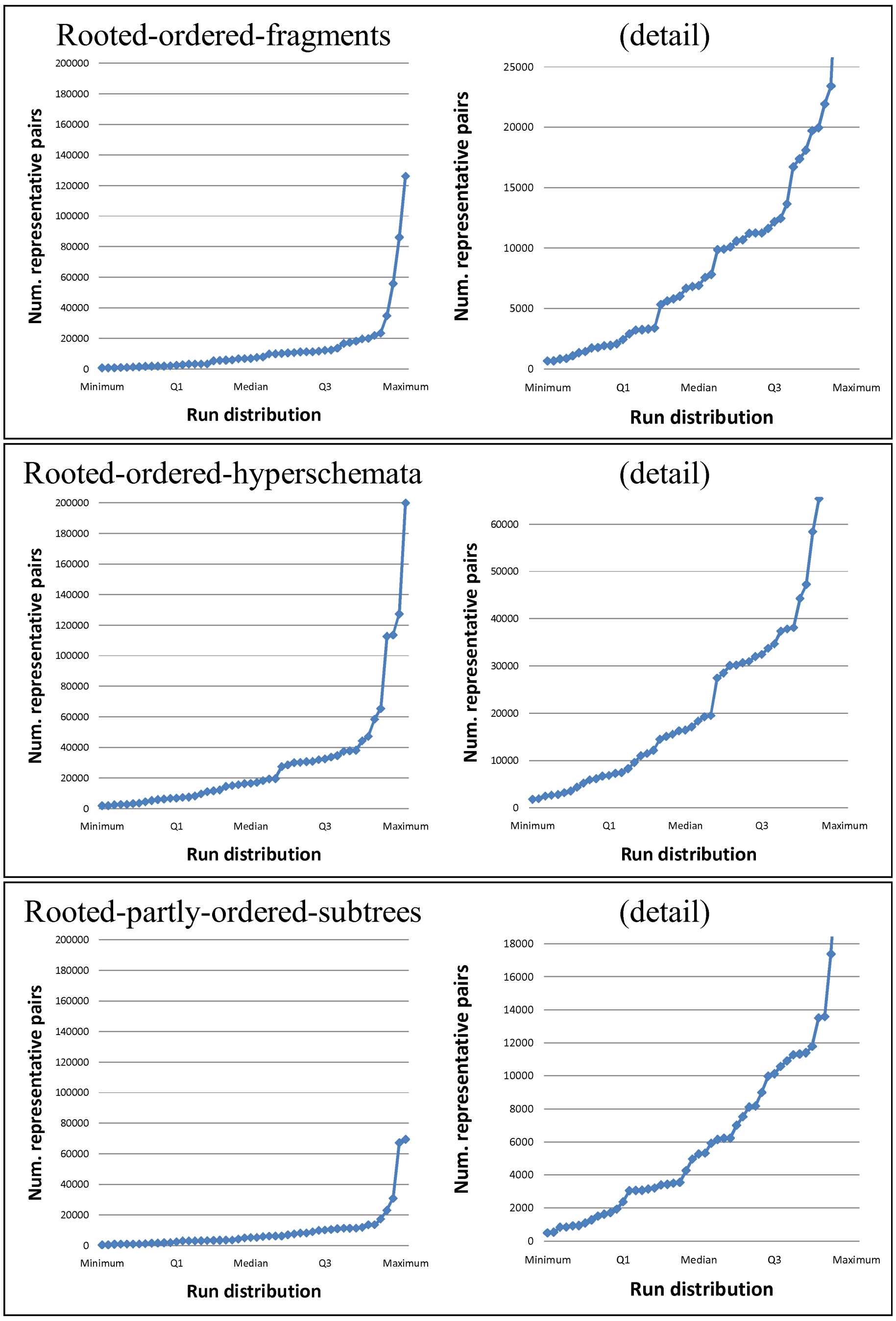

Figure 6.1: Distribution of numbers of representative pairs at generation 40, population 51, task BCW for three forms of schema.

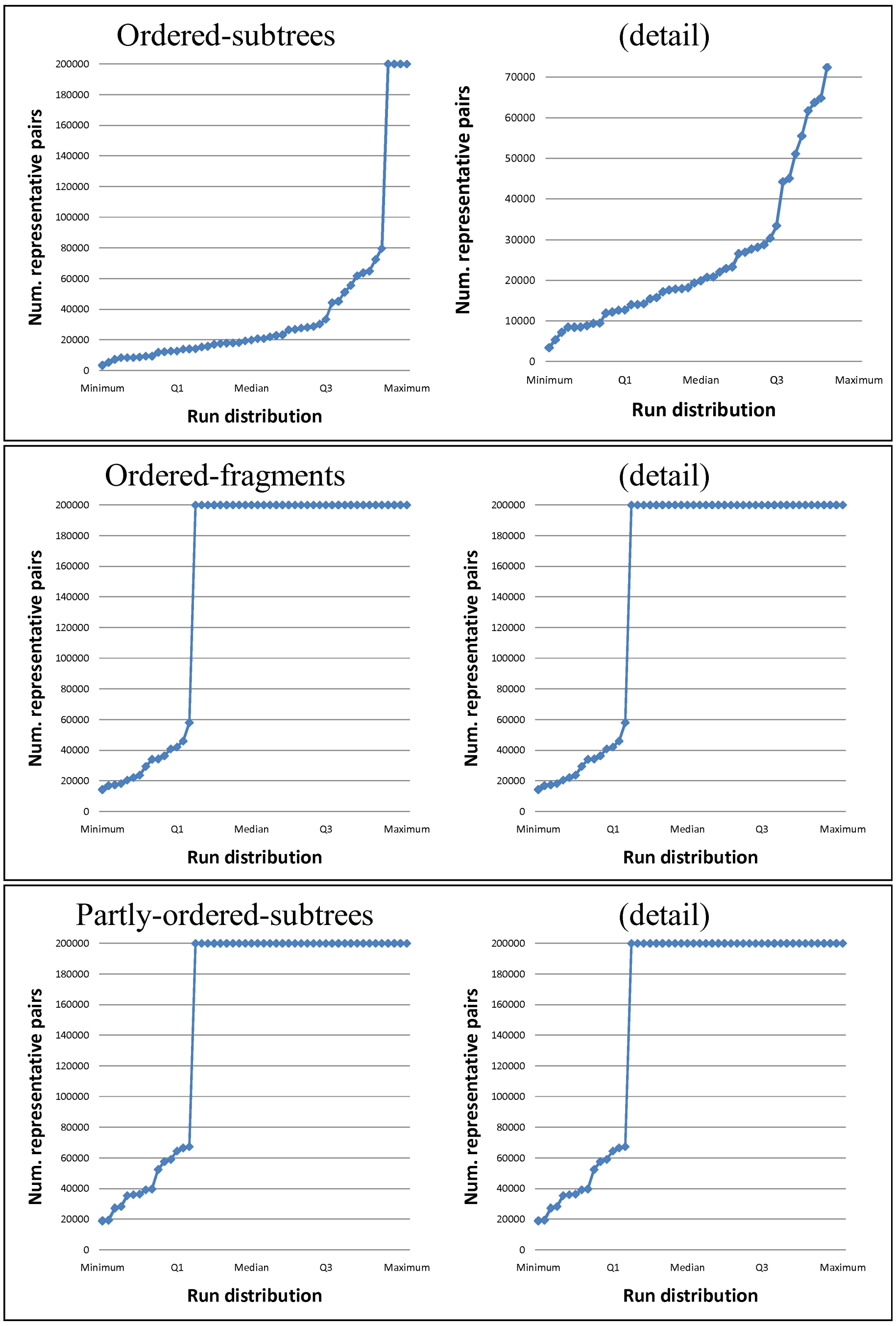

Figure 6.2: Distribution of numbers of representative pairs at generation 40, population 51, task BCW for next three forms of schema.

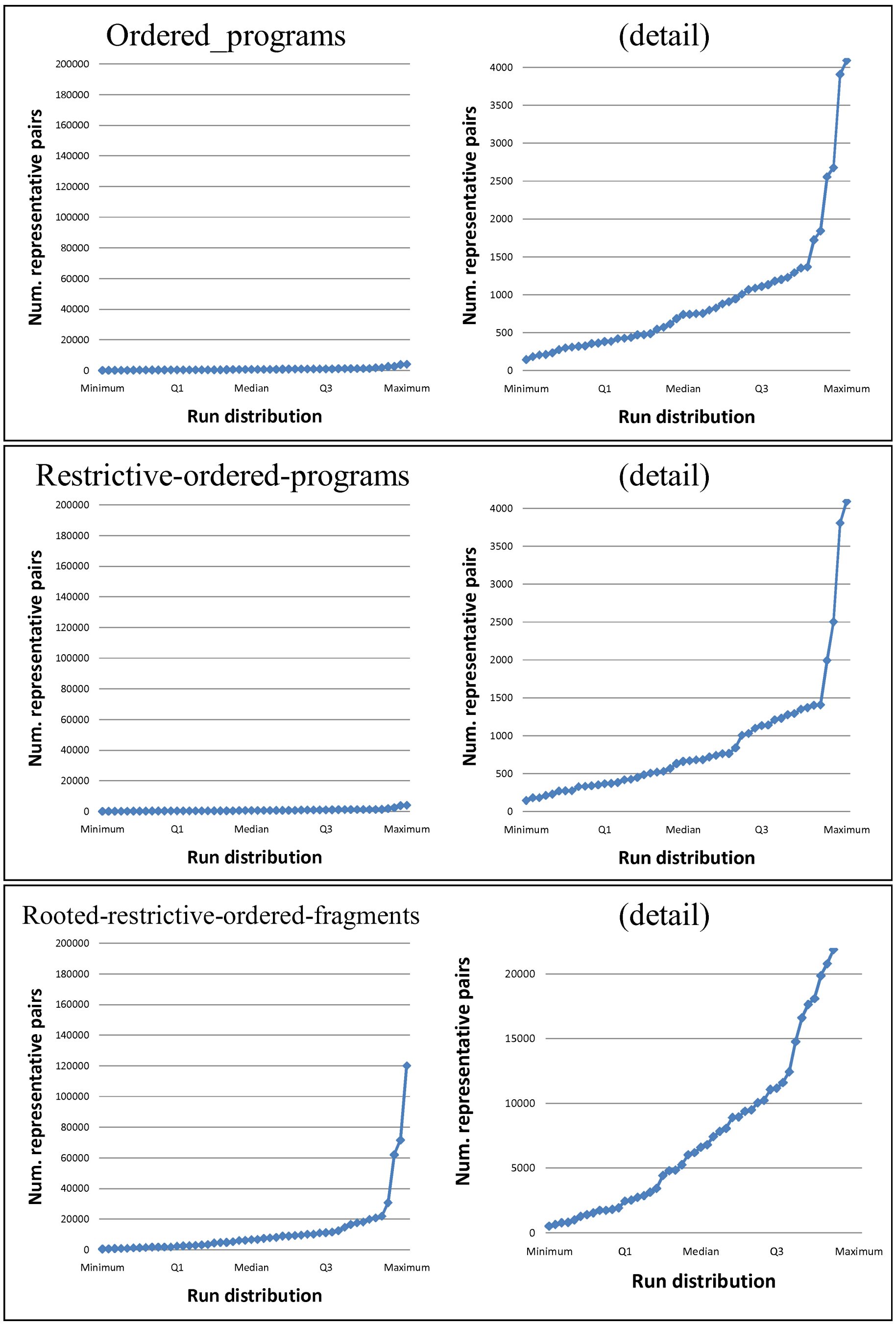

Figure 6.3: Distribution of numbers of representative pairs at generation 40, population 51, task BCW for next three forms of schema.

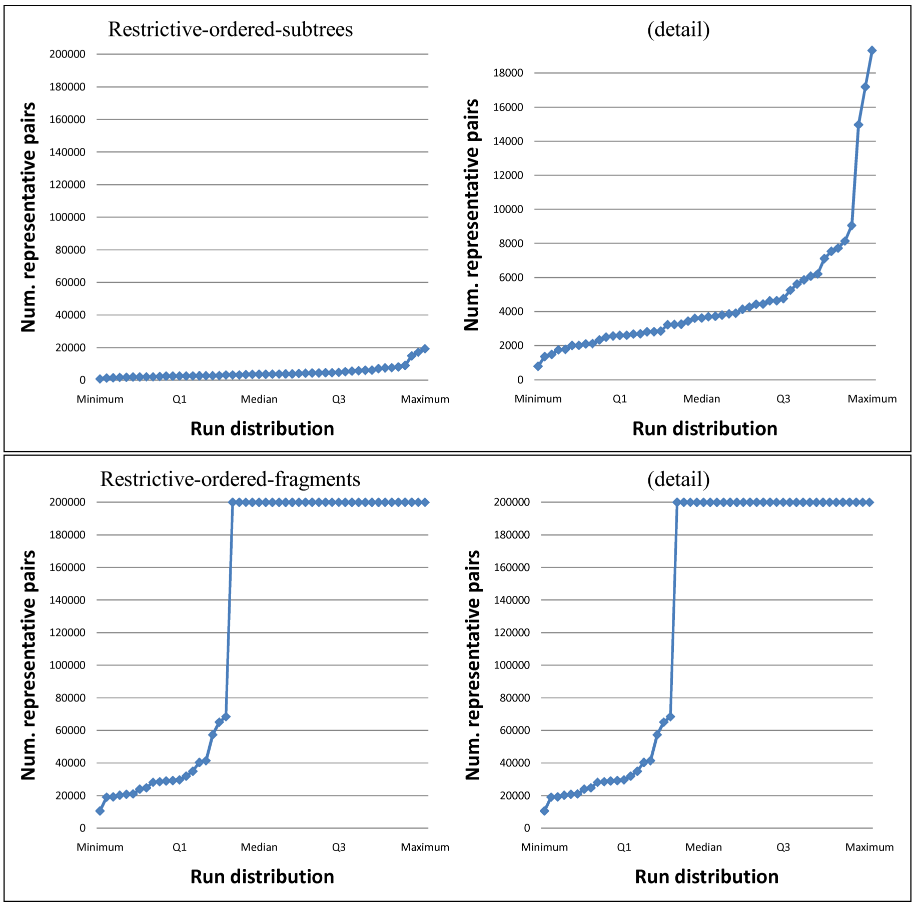

Figure 6.4: Distribution of numbers of representative pairs at generation 40, population 51, task BCW for remaining two forms of schema.

from subtrees to fragments to hyperschemata, so too does the number of representative pairs found.

One interesting feature of the data is the seemingly excessive number of representative pairs of restrictive forms of schema, most noticed with "restrictive-ordered-programs". Logically, each maximal schema of this form *must* be the same as one of the programs and thus there is a strict limit on how many schemata there may be as it must be less than or equal to the population size. Investigation shows that for the restrictive forms, most representative pairs represented object subsets but no schemata of the form. This is also seen with non-rooted forms, although for a very different reason.

In the case of restrictive forms of schemata, the sets of schema components held by these non-schema-representing representative pairs are not valid as schemata since, even though there are programs which match each schema component individually, no program matches the schema made from the set as a whole. An example is the two programs $(+\ 1), (+\ 2)$ under the "restrictive-ordered-programs" form of schema. The set of schema components $\{+\}$ is the intersection of the sets $\{+, (+\ 1)\}$ and $\{+, (+\ 2)\}$ and is found as part of the meet-semi-lattice, but this intersection describes a schema "$+$" which does not occur in any of the programs in the population although it would if the form where loosened to "ordered-programs". For the restrictive forms of schema, a great many of these representative pairs with "invalid" schemata were found, while the numbers for "valid" schemata was as times relatively small.

In the case of non-rooted forms of schemata, representative pairs may be found which do not represent any schemata. However in these cases the schemata occurring in each such representative pair's program set are "valid" but are simply represented by representative pairs which hold more programs.

To show the trend over time, figures 6.5 and 6.6 present the quartiles of the numbers of representative pairs for each generation from the initial generation to the 100th, at intervals of five for six key forms of schema. The five series of the graph are, from top, the maximum, third quartile, median, first quartile and minimum results over fifty runs.

The figure shows a marked increase in the number of representative pairs as the run goes on but generally this trend plateaus at about generation 30 to 40. This may be caused by the programs of the population sharing more large schemata. Since the algorithms of the thesis' method search for patterns that are common between programs, they must do more work to process the later generations, in which programs share large schemata, than to process early generations, in which few programs share even small schemata. This increase in work is exhibited as an increase in the number of maximal pairs.

It is interesting to note that, while the number of maximal subtrees has plateaued by generation 20, the number of representative pairs for the more expressive forms of schema continue to increase until later in the run. Also, while the number of maximal subtrees stays relatively constant after the initial increase, the numbers of the more expressive schemata can at times vary markedly. For instance, the median number of rooted-ordered-hyperschemata at generation 85 (11,812) almost halves the same statistic for just 10 generations before (21,873).

For each of these figures, there is an equivalent figure for the *sphere* task. These figures exhibit very similar overall trends and look very similar to the figures presented, so those figures are omitted from this thesis.

6.5.2 Time taken

This subsection aims to characterize how long the algorithm takes to perform each of the processes involved in finding the representative pair DAG and annotate it with schema tree representations and represented schema and program set counts.

Figures 6.7 and 6.8 presents stacked plots for these values at generation

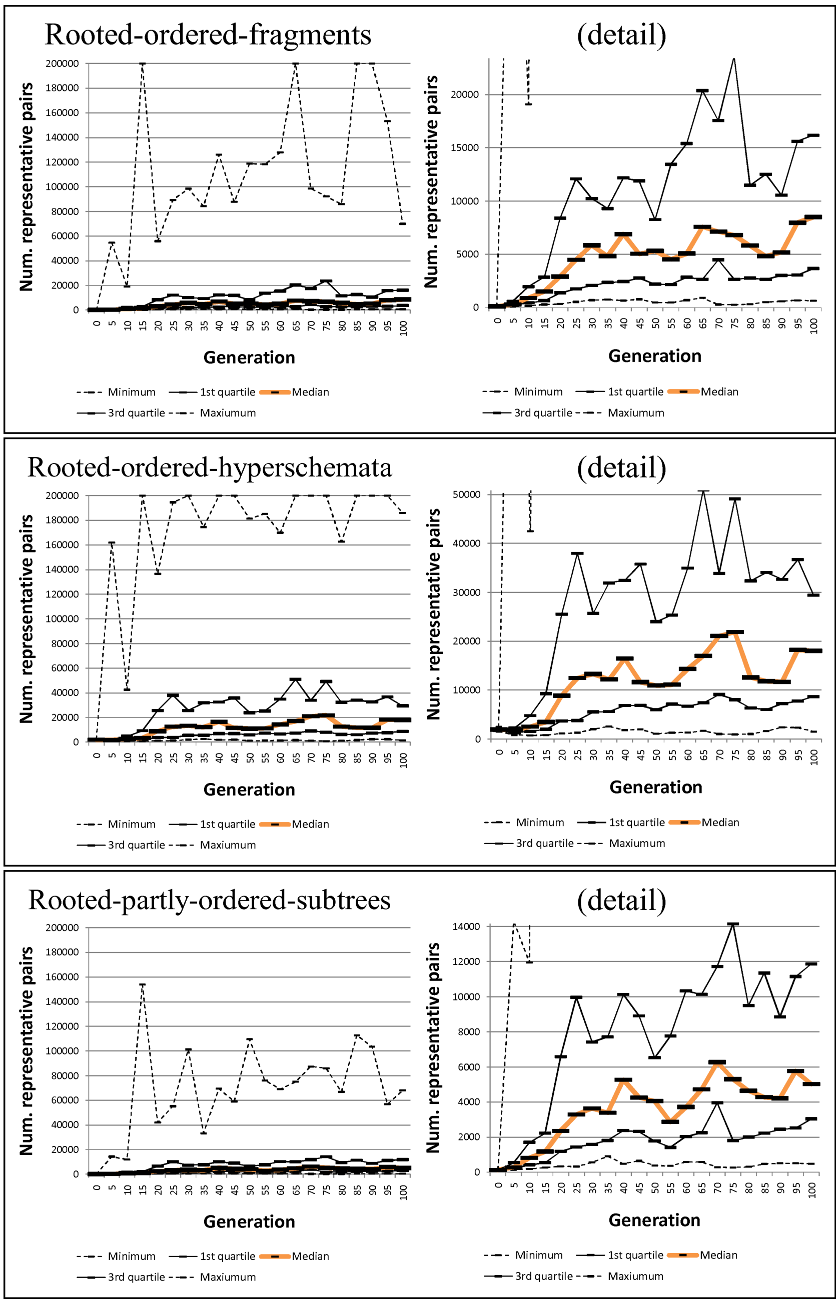

Figure 6.5: Distribution of the number of representative pairs over each generation, population 51, task BCW for three forms of schema.

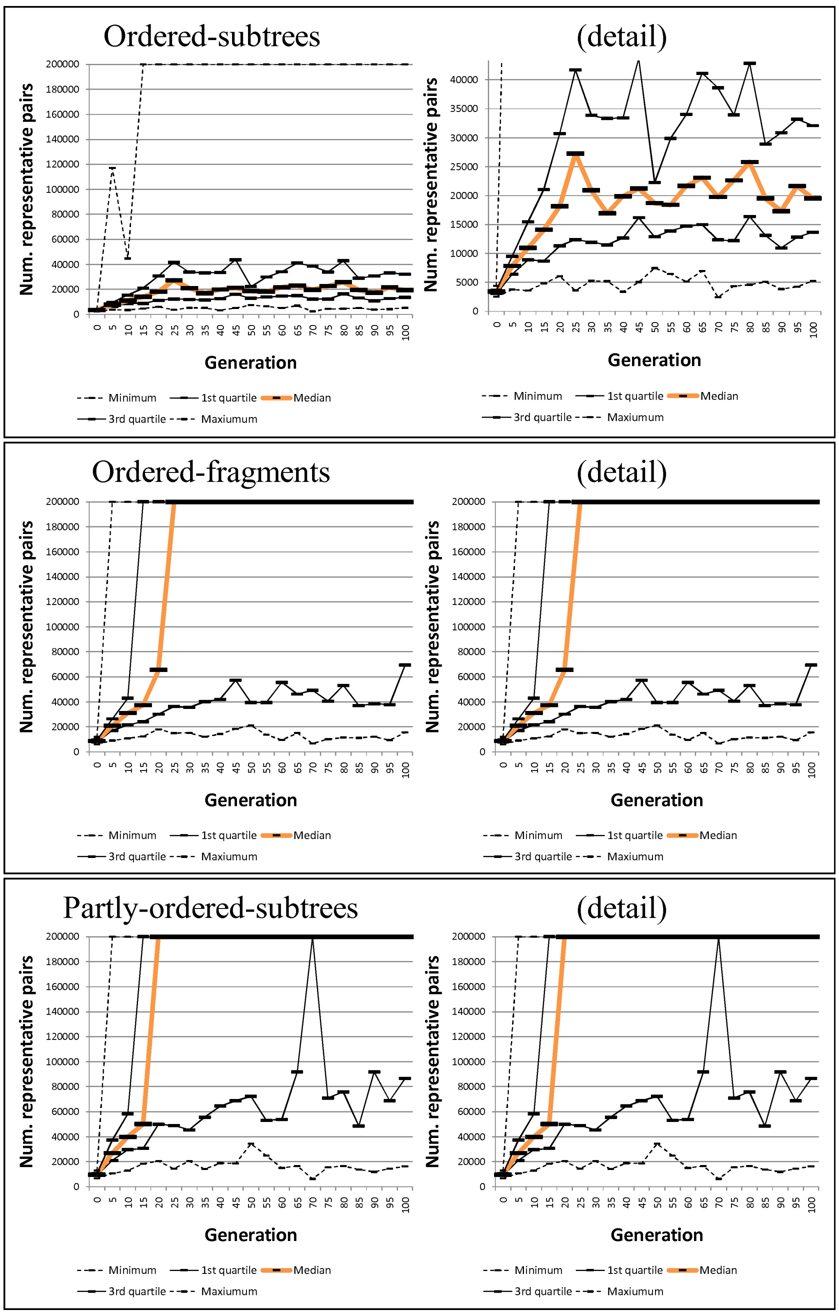

Figure 6.6: Distribution of the number of representative pairs over each generation, population 51, task BCW for next three forms of schema.

40 for six key forms of schema.

In the figures, the stacked plot shows the time taken by the following sub-processes, from top down:

- Finding for each maximal program subset the count of represented program subsets.
- Conversion from conjunctive form to de-rooted conjunctive form, presented for non-rooted forms only.
- Finding for each maximal schema the count of represented schemata, given the count of more general schemata.
- Finding for each maximal schema the count of more general schemata.
- Finding for each maximal schema the tree representation of that schema.
- Finding the maximal schemata and maximal program subsets as a DAG, given a population and form of schema.

The figures clearly show that for the ProMS variant of the addMeets algorithm used the time to find the DAG of representative pairs is not the slowest part of the whole process and takes typically about 10% of the time taken by the whole process. Instead, the sub-processes finding for each representative pair histograms with counts of represented schemata and program subsets were the most significant parts of the process in terms of time taken. These two sub-processes most often took over half the time taken by the process as a whole.

The sub-process finding the histogram of counts of more general schemata for each maximal schema also took a significant fraction of the total time, about twice to three times the fraction taken to find the DAG of representative pairs. The sub-process finding the tree representation for each schema was relatively fast.

For the non-rooted forms, the most expensive part of the total process was the conversion from the conjunctive form on subtrees to the de-rooted

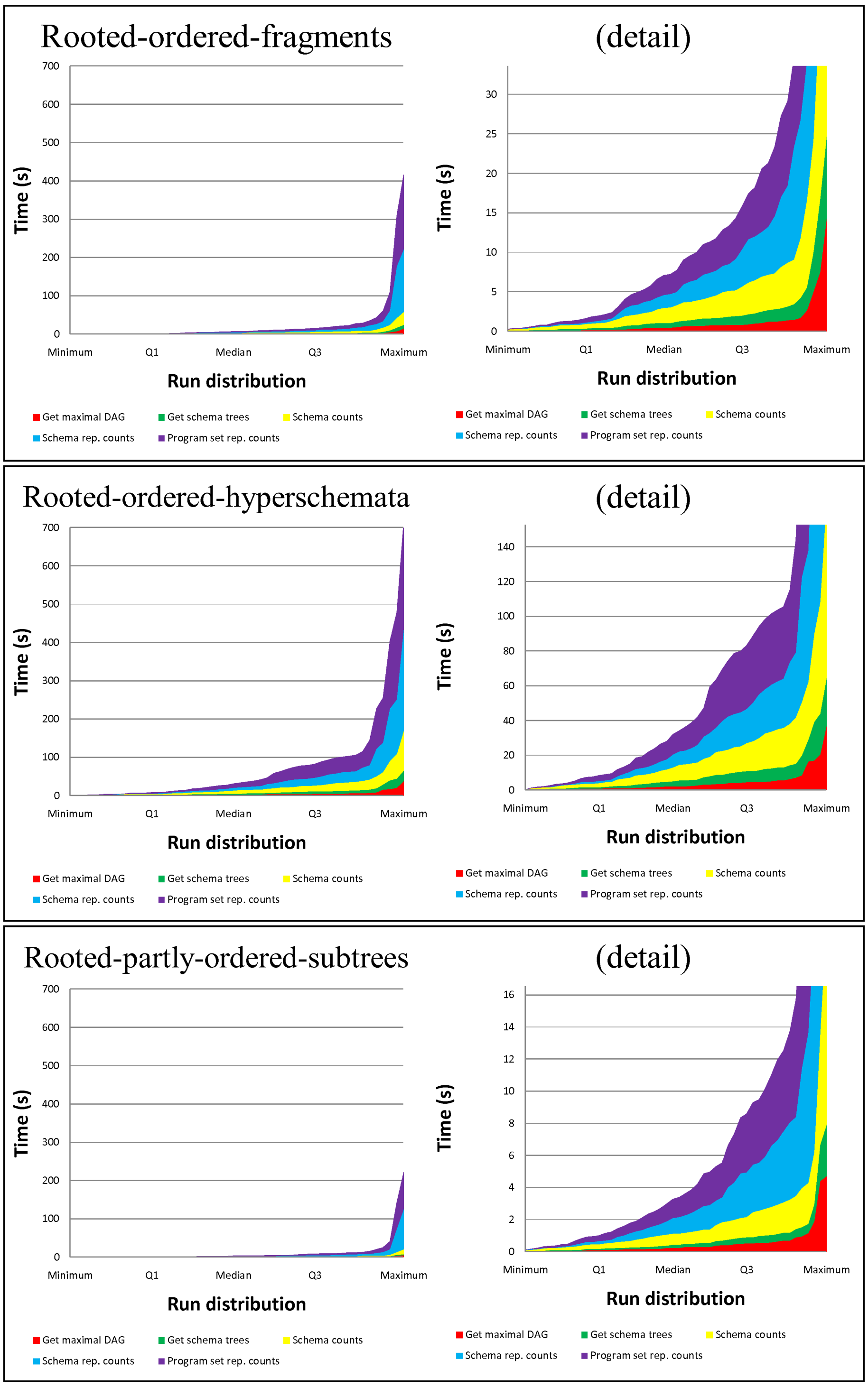

Figure 6.7: Distributions of time taken by processes at generation 40, population 51, task BCW for three forms of schema.

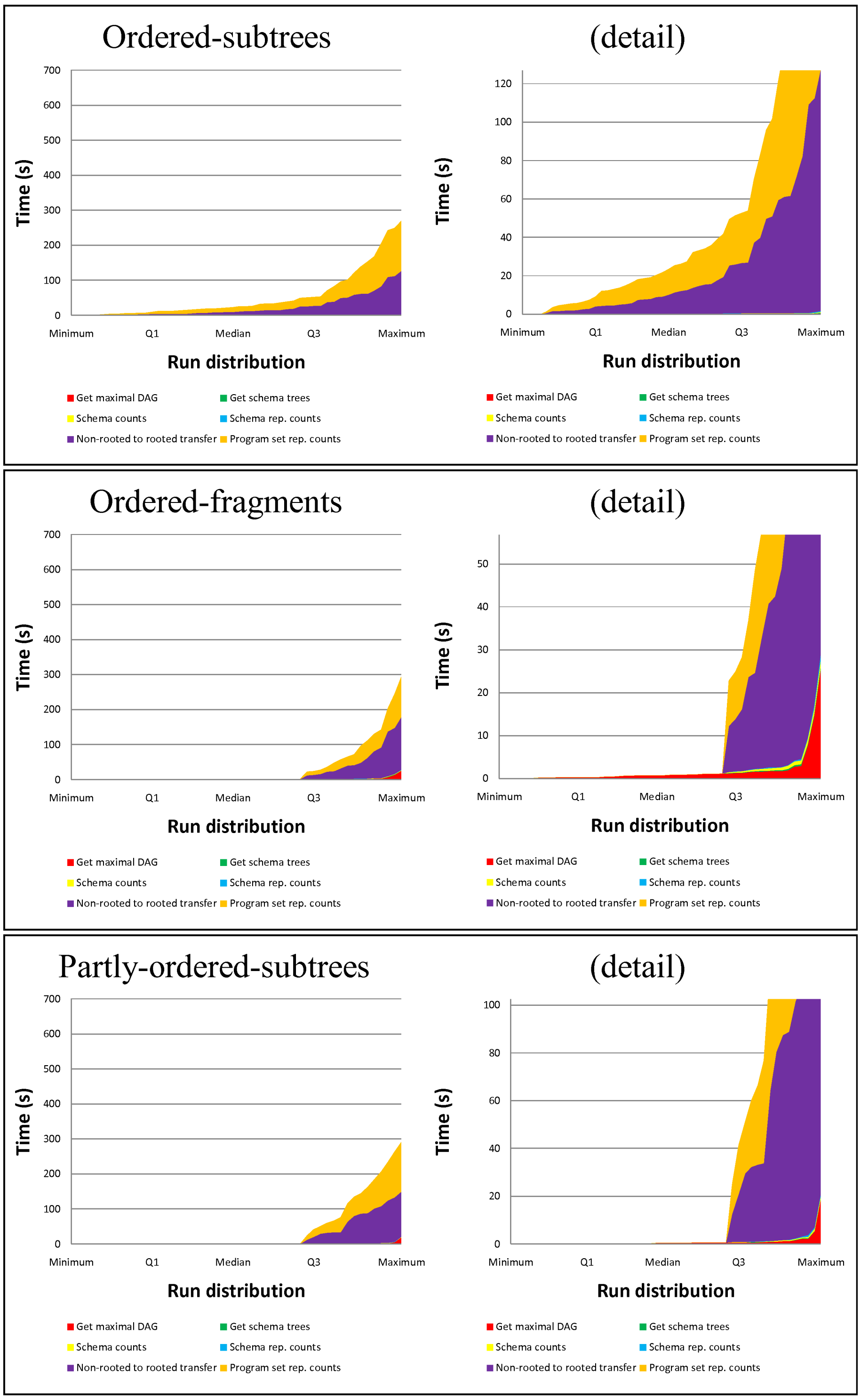

Figure 6.8: Distributions of time taken by processes at generation 40, population 51, task BCW for next three forms of schema.

conjunctive form on programs. This conversion entails running the addMeets function on the set of representative pairs for the conjunctive form on subtrees and this proves to be a significant cost.

The total time has a similar distribution to that of the number of representative pairs presented in the previous subsection. In fact, the number of representative pairs is highly correlated to the time taken by the method. Figure 6.9 presents a scatter plot of the time taken to find the full annotated DAG of representative pairs, versus the size of this DAG with one point per run each fifth generation. The figure shows high correlation between the time take by the new method and the number of representative pairs found. Typically the method took 15 seconds to find a total of 15,000 representative pairs for all forms. If the total number of representative pairs was greater, then the system required more time per representative pair. For instance 22,000 representative pairs took on average about 30 seconds. Similarly, the method proves more efficient when finding smaller DAGs of representative pairs.

To show the trend over time, figure 6.10 presents stacked plots of the medians over 50 trials of these values for each generation from the initial generation to the 100th, at intervals of five for six key forms of schema. The figure shows high correlation in outline to figure 6.5 from the previous subsection. The figure also shows that the various sub-processes involved in running the new method have similar complexity with respect to the number of representative pairs found since each occupies a similar fraction of the whole throughout the run.

6.5.3 Output file size

Each time the schema engine was run, an output file stored the population, schema components and representative pairs as a DAG. This output file contains enough information to allow analysis of the population and form using the new method. This subsection aims to characterize the size

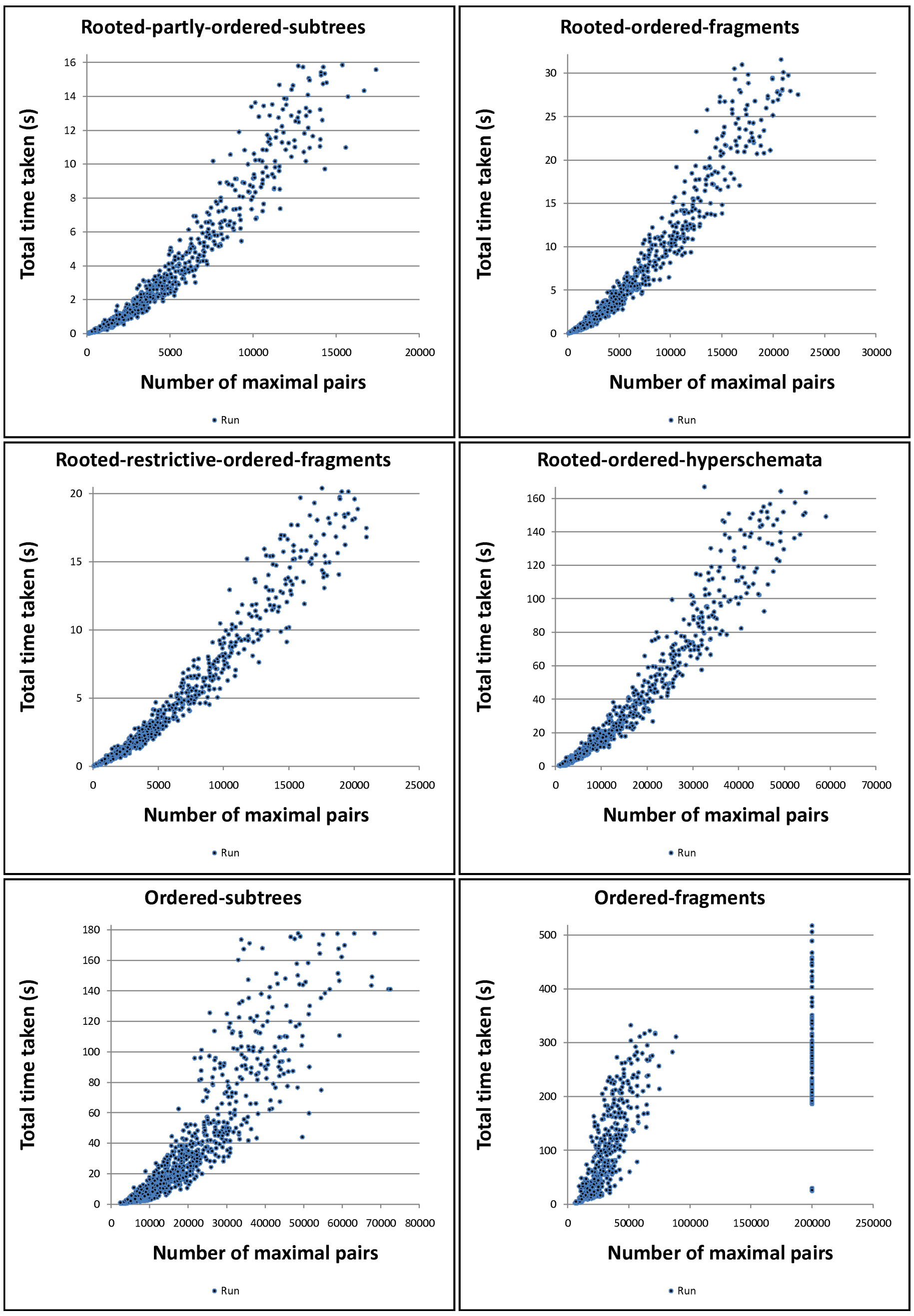

Figure 6.9: Time taken to find representative pairs versus number of representative pairs, population 51, task BCW for six forms of schema.

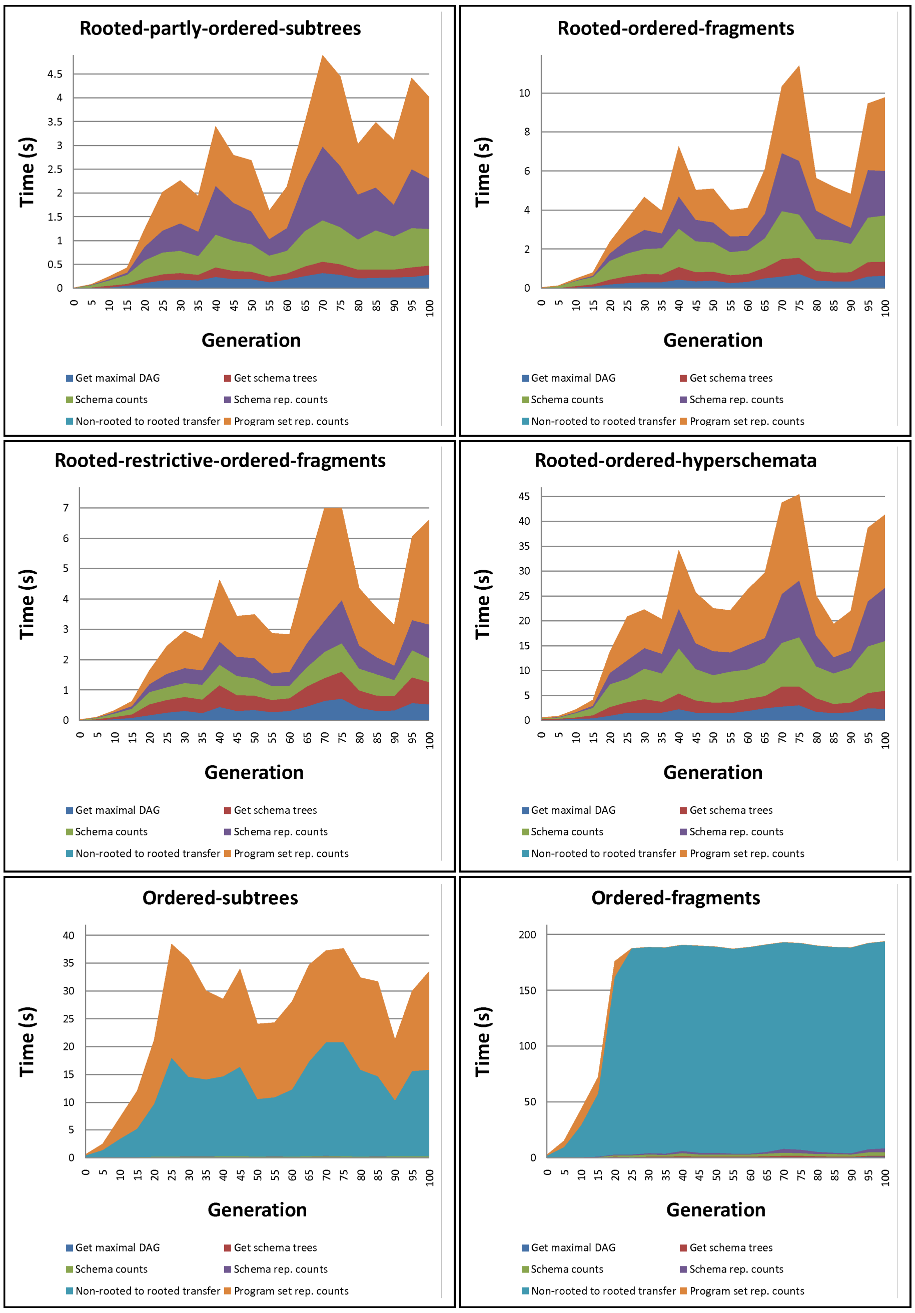

Figure 6.10: Time taken by processes over each generation, population 51, task BCW for six forms of schema.

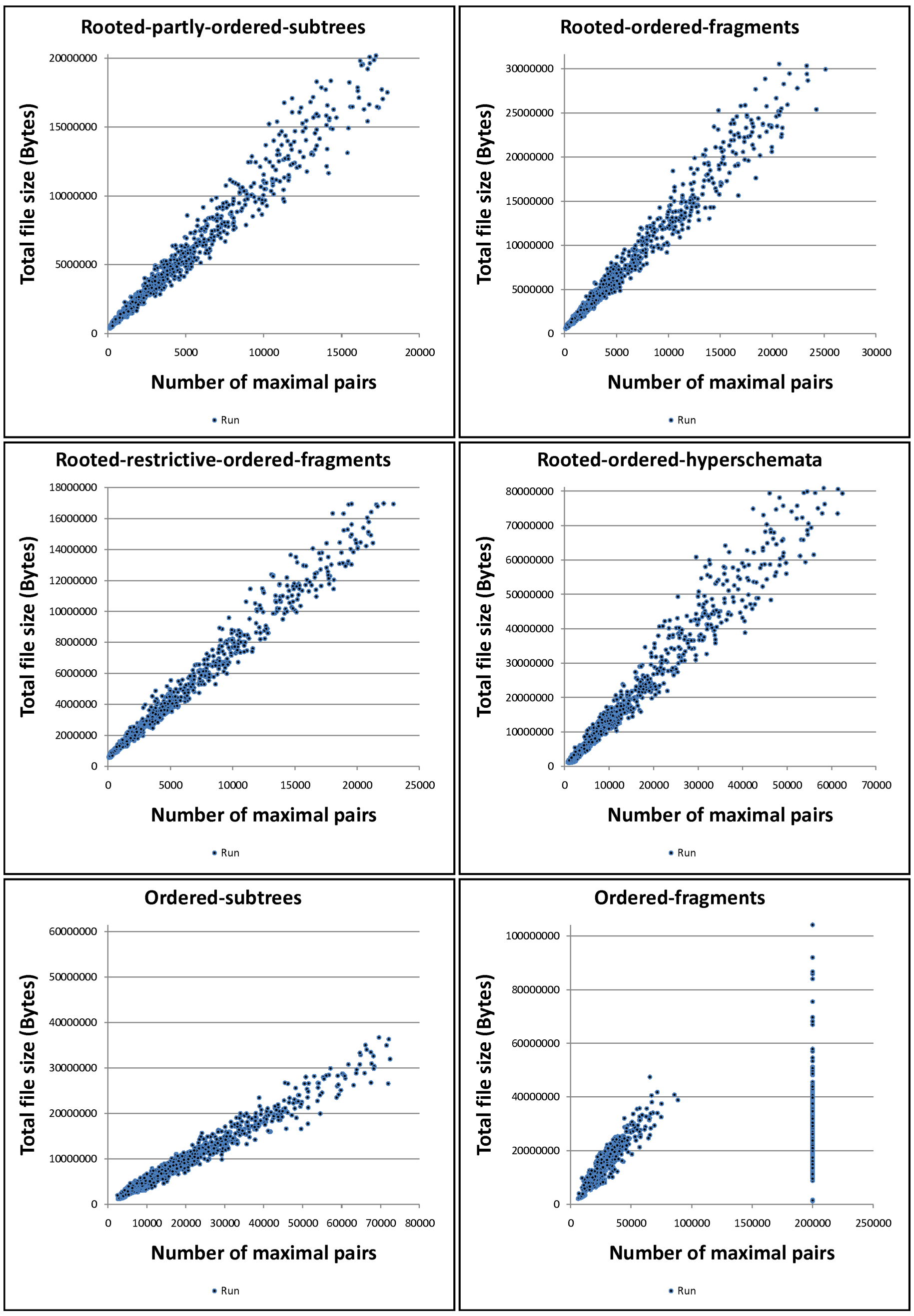

Figure 6.11: Disk space used versus number of representative pairs, population 51, task BCW for six forms of schema.

of the output file produced by the new method as an indication of typical disk space requirements. Figure 6.11 presents a scatter plot of the uncompressed output file size on disk, versus the number of representative pairs with one point per run each fifth generation. The plots show proportional correlation between the output file size and the number of representative pairs. The per-maximal-pair disk usage ranges between about 480 bytes for ordered-subtrees to about 1,400 bytes for rooted-ordered-hyperschemata. This difference reflects that each maximal schema stores a set of schema components and the size of this set depends on the expressiveness of the form of schema; for ordered subtrees there are few schema components and for rooted-ordered-hyperschemata there are many.

Typically the output files compressed to 10% of their original size under the standard `zip` format compression. Also, some of the information in the file could be superfluous in some setups. An example is where the schema components are not stored since they would seldom be used in analysis, which would result in significant savings in disk space.

6.5.4 Memory used

This subsection aims to characterize the maximum memory used by the method as it finds the representative pairs DAG. As the *addMeets* algorithm runs, it allocates memory to store the representative pairs and for other uses. This subsection reports the maximum amount of memory thus allocated during runs of the algorithm. Figure 6.12 presents a scatter plot of the maximum memory allocated by the method to store representative pairs and interim results during the sub-process of finding the representative pairs DAG, versus the number of representative pairs with one point per run each fifth generation.

The figure shows that for rooted forms of schema, the memory allocated is about 3,000 bytes per representative pair. But for non-rooted forms there is no obvious correlation between RAM bytes and number of representative pairs. This is likely caused by the inclusion of a sub-process changing the conjunctive form on subtrees to a de-rooted conjunctive form on programs. This process returns a final set of representative pairs which may be significantly smaller than the original set of representative pairs. It may occur that before this sub-process the algorithm allocates a significant memory for many representative pairs and after this sub-process most representative pairs have been removed. Another effect of this two step process is shown clearly by the plot for ordered-fragments, which has a clear line of points at 200,000 representative pairs on the x-axis. The system imposed the limit of 200,000 representative pairs on both the first representative pairs sub-process, for the conjunctive form on subtrees, and the second representative pairs sub process, for the de-rooted conjunctive form on programs. This limit was often reached by the former even when the latter would have resulted in fewer final representative pairs.

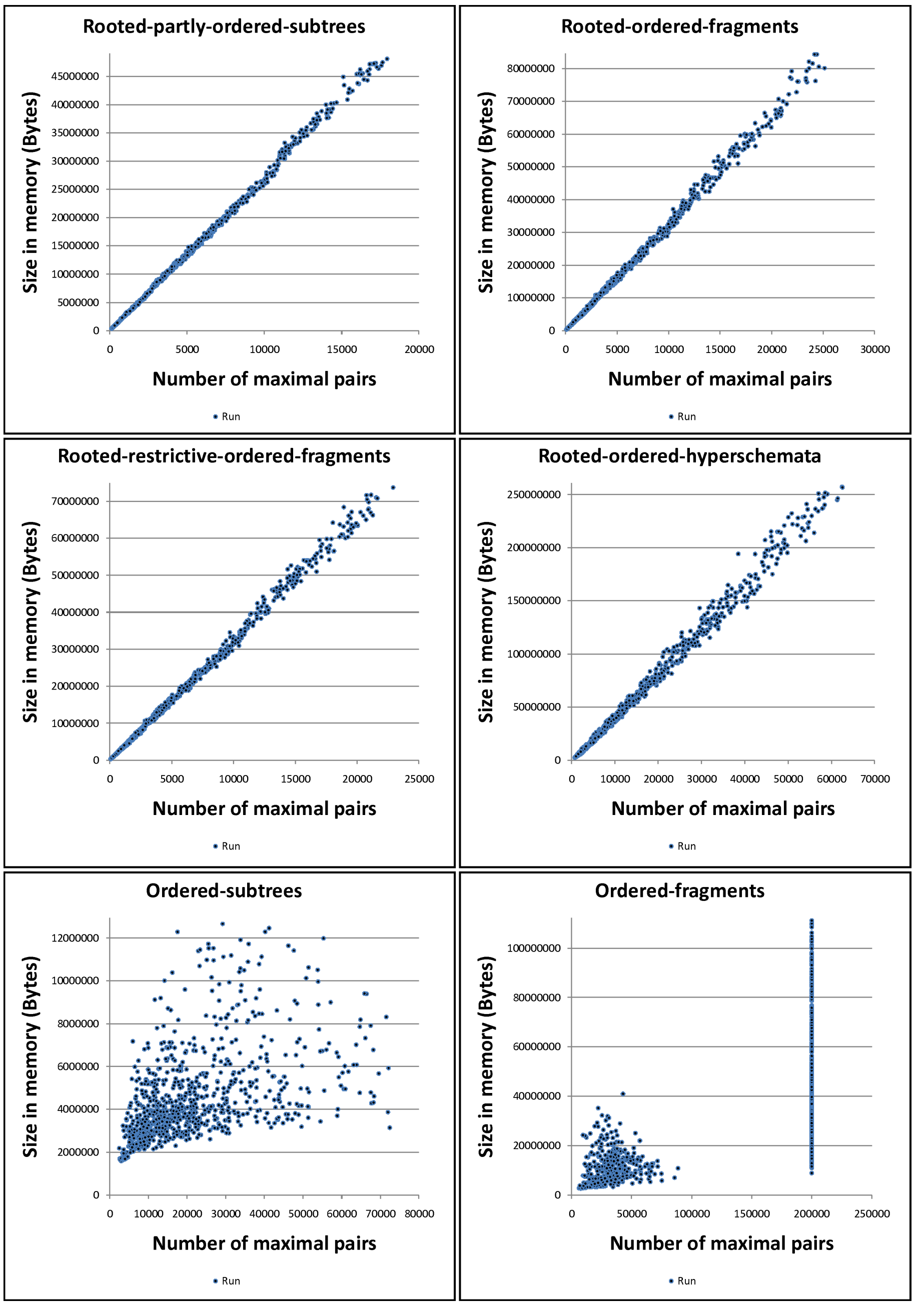

Figure 6.12: Memory (RAM) used versus number of representative pairs, population 51, task BCW for six forms of schema.

6.6 Comparing the different algorithms

The third set of experiments saw the method run with default parameters but using different algorithms to find the representative pairs.

Figure 6.13 presents plots for the time taken to find the representative pairs as an un-annotated DAG, without schema or program subset counts or schema tree representations, for four optimized algorithms, at generation 40 for task *BCW* and three key forms of schema.

Figure 6.14 shows the equivalent plots for the *Sphere* task. The figures shows that some algorithms were significantly faster than others. In particular ProMP is typically the fastest of the four, ProMPS being next, then IntMPS with IntMP often significantly slower than the other three. This justifies the use of the ProMP algorithm throughout the other experiments of this chapter and the next chapter.

All four algorithms compared find both the representative pairs and the edges between them as an anti-transitive DAG. These edges are most easily found when using the "promote-increment" approach as used by ProMP and ProMPS and causes slightly more slow-down of IntMP and IntMPS. This slowdown may explain a little of the time difference between the "promote-increment" algorithms and the "intersect" algorithms.

The difference between IntMP and IntMPS is most marked when using the rooted-ordered-hyperschemata form of schema, an effect caused by the large numbers of schema components in this expressive form of schema. As the numbers of schema components increases, the schema component sets representing the maximal schemata also grow and performing intersects on these sets is a major contributor to the complexity of the *IntMP* algorithm. The difference is least marked for the rooted-partly-ordered-subtrees form of schema where there are typically relatively few schema components.

The difference is greater on the "BCW" task than on the "sphere" task with the latter showing slightly less difference between the algorithms,

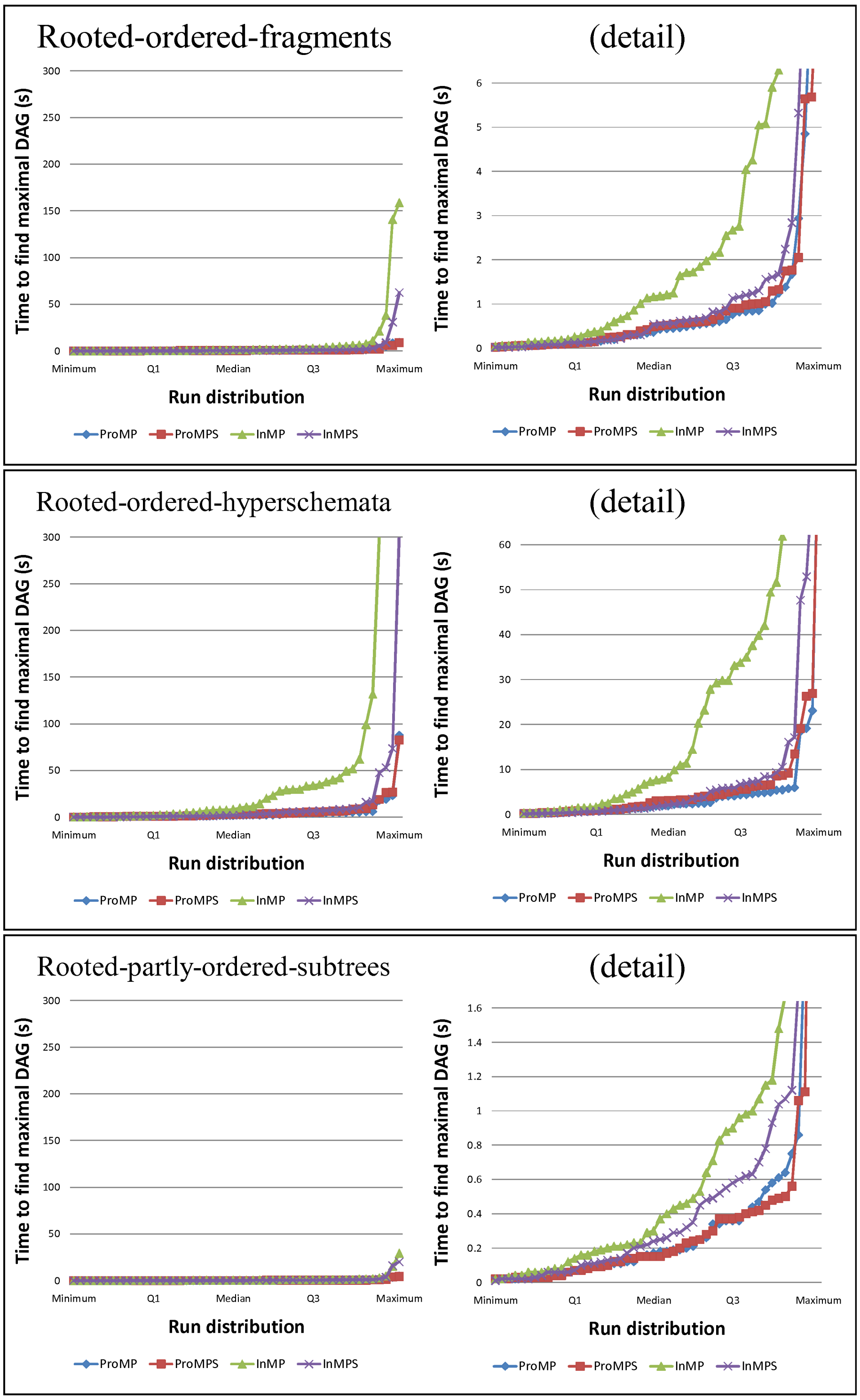

Figure 6.13: Time taken to find representative pairs as a DAG for different algorithms, at generation 40, population 51, task BCW for three forms of schema.

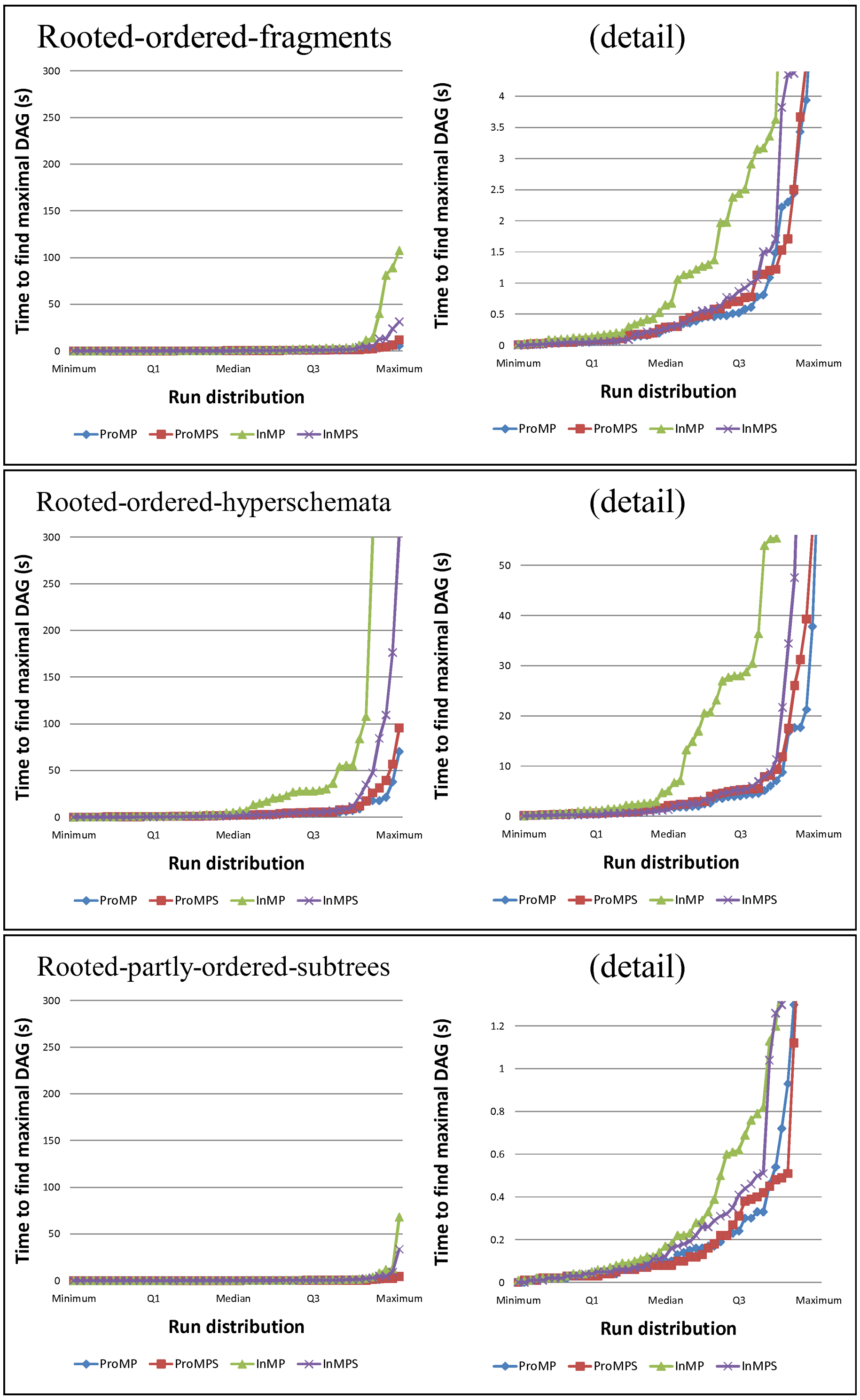

Figure 6.14: Time taken to find representative pairs as a DAG for different algorithms, at generation 40, population 51, task *Sphere* for three forms of schema.

especially for the rooted-partly-ordered-subtrees form of schema.

Figure 6.15 shows the time taken to find the representative pairs as an un-annotated DAG for different options of the ProMP algorithm. The options given in the plots are:

- **Set only**: the time to find the set of representative pairs without obtaining the DAG edges.
- **Get DAG from set (c)**: the time to find the set of representative pairs, then obtain the DAG edges using GetEdgesS presented in section 5.6 from chapter 5. This variant of the algorithm find the DAG edges by identifying which schemata are subsets of which other schemata.
- **Get DAG from set (p)**: the time to find the set of representative pairs, then obtain the DAG edges using GetEdgesPS presented in section 5.6 from chapter 5. This variant of the algorithm find the DAG edges by identifying which program subsets are subsets of which other program subsets.
- **Set and DAG (optimized)**: the time to find the set of representative pairs, as well as the edges, at the same time using an optimized algorithm.

The figures show clearly that producing the DAG is an expensive task when performed separately but may be efficiently integrated into the process of finding the representative pairs themselves.

For the expressive rooted-ordered-hyperschemata form of schema, the difference between the "Get DAG from set (c)" and "Get DAG from set (p)" options is more marked than for the other forms. Similarly to the difference between the IntMP and IntMPS algorithms, this is likely caused by the larger schema component sets for this more expressive form of schema.

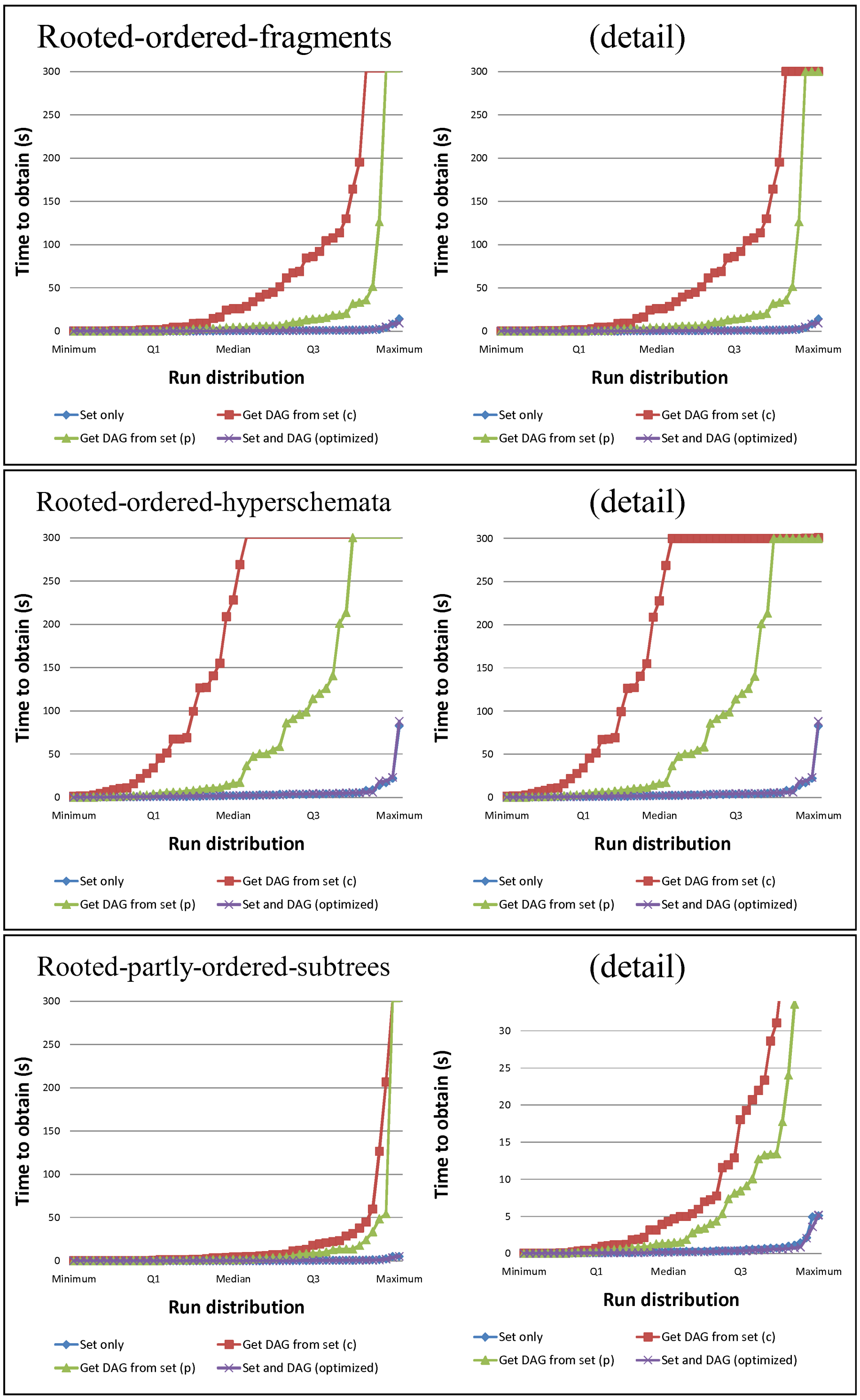

Figure 6.15: Time taken to find representative pairs as a DAG for different options of the ProMP algorithm, at generation 40, population 51, task *BCW* for three forms of schema.

6.7 Characterizations by varying single parameters

The fourth set of experiments aimed to characterize the new method over various input parameters, including: population size, program size in nodes and function node arity. The method was run with default settings other than the one changed parameter for all combinations of: two tasks, one population size (51), two forms of schema (rooted-partly-ordered-subtrees and rooted-ordered-fragments), 50 random seeds and 21 generations (initial generation to 100th in steps of five). In addition, five of the seeds were run over 101 generations (initial generation to 100th).

Thus the schema engine was run a total of 5800 times per changed parameter value. The following parameter values were used:

- Population size: 26, 51, 76, 101, . . . , 451, 476, 501
- Program size (maximum node count): 20, 30, 40, 50, 60, 70, 80
- Function node arity (other than `if`): 2, 2.25, 2.5, 2.75, 3, 3.25, 3.5, 3.75, 4

  (A fractional arity a indicates the GP engine created a proportion of functions with the floor of a as their arity and created a proportion of functions with the ceiling of a as their arity. Thus an arity of 2.25 indicates that three quarters of function nodes, other than `if` nodes, have arity 2 and one quarter have arity of 3.

6.7.1 Population size

An important feature of the new method is that it can analyze medium-scale populations of medium-sized programs. This section attempts to characterize the new method over a range of population sizes, up to 501 programs. The method was run with default parameters, other than varying the population size from 26 to 501 in steps of 25.

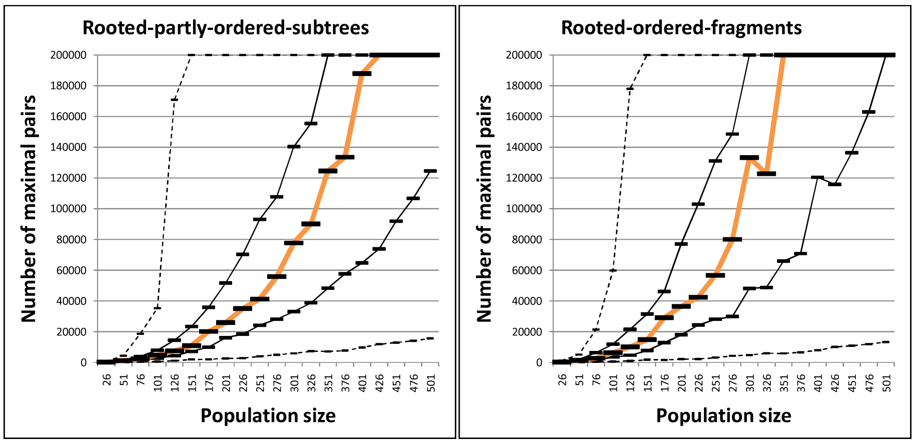

Figure 6.16: Effect of the population size on the distribution of numbers of representative pairs at generation 40, task BCW for two forms of schema.

Figure 6.16 shows how the population size affects the number of representative pairs by population size. The figure has two plots, one for rooted-ordered-fragments and the other for rooted-partly-ordered-subtrees. Each plot has quartiles over 50 trials per population size of the number of representative pairs, capped at 200000, versus population size.

The figure shows a polynomial relationship between the size of population and the number of representative pairs. The regressed bases for this relationship are $|R| \propto |P_0|^{2.37}$ for rooted-partly-ordered-subtrees and $|R| \propto |P_0|^{2.29}$ for rooted-ordered-fragments. In these equations R is the output set of representative pairs for the median of the 50 trials and P_0 is the input set of programs. The results used least-squares polynomial regression to find the closest a and b in $|R| = a|P_0|^b$.

Thus the typical number of maximal pairs found quadruples as the population size is doubled. Extrapolating, a population size of 1000 would be expected to have a median number of maximal rooted-ordered-fragments of about 1,300,000 at generation 40.

To show the trend over time, figure 6.17 presents the median number of representative pairs found versus generation for the following population

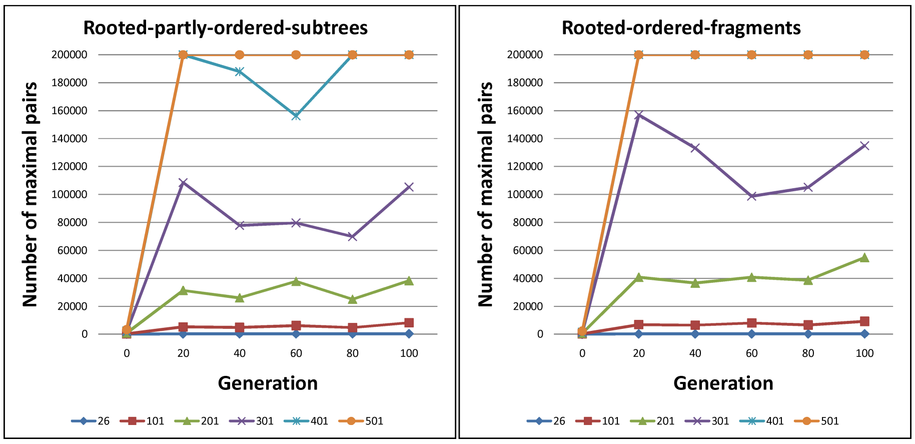

Figure 6.17: Median number of representative pairs versus generation for different population sizes, task BCW for two forms of schema.

sizes: 26, 101, 201, 301, 401 and 501.

The figure shows the interesting trend that, while the number of representative pairs grows steadily for small population sizes, for larger population sizes it can peak early in the run and stays at a similar level as the run progresses. In particular, the series for a population size of 301 has a maximum at generation 20 and immediately starts to decline, rising again after about generation 80. This may indicate that the number of representative pairs has a somewhat stable value through for a given population size.

6.7.2 Program size

This section attempts to characterize the new method over a range of program sizes. The experiments placed a node-count limit on programs rather than a tree-depth limit. The method was run with default parameters, other than varying the maximum program size from 20 to 80 in steps of 10.

Figure 6.18 shows how the program size affects the number of repre-

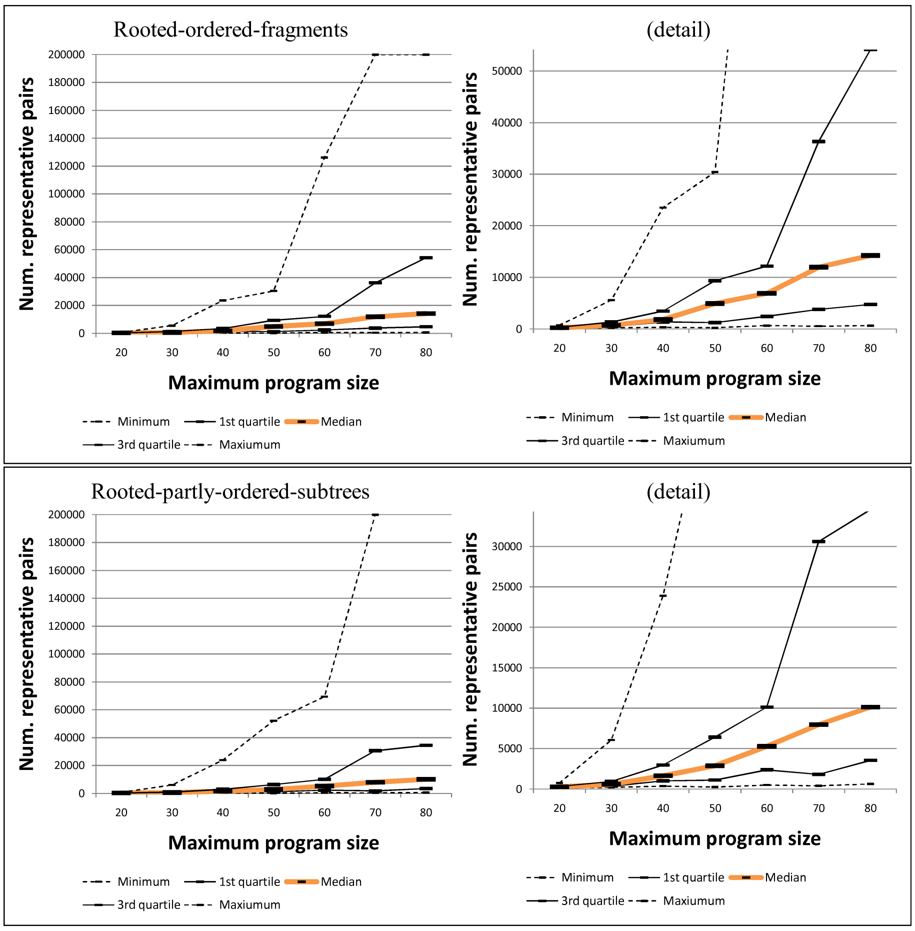

Figure 6.18: Effect of program size on the distribution of numbers of representative pairs at generation 40, task BCW for two forms of schema.

sentative pairs. The figure has two plots. Each plot gives quartiles over 50 trials per program size of the number of representative pairs, which was capped at 200000, versus program size.

The figure shows a marked increase in the number of representative pairs given an increase in the size of programs. Interpreting this increase as polynomial and again using least-squares regression, the regressed bases for this relationship are $|R| \propto {N_{nodes}}^{2.77}$ for rooted-partly-ordered-subtrees and $|R| \propto {N_{nodes}}^{3.08}$ for rooted-ordered-fragments. In these formulae R is the output set of representative pairs for the median of the 50 trials and N_{nodes} is the maximum size of a program.

Perhaps even more striking than the increase of the median statistic is the increase of the upper quartile and maximum runs. At about 70 nodes, the upper quartile series shows marked increase. Polynomial regression of the upper-quartile series for rooted-ordered-fragments gives $|R| \propto {N_{nodes}}^{3.68}$, significantly larger in exponent than the same for the median series. Polynomial regression of the maximum series for rooted-ordered-fragments up to generation 60 with 126,178 representative pairs gives $|R| \propto {N_{nodes}}^{4.46}$, another marked increase in exponent over the upper quartile series. This increase in exponent would indicate that, were the experiment to further increase the size of programs, it would expect the runs with the most representative pairs to have *many* more representative pairs.

To show the trend over time, figure 6.19 presents the median number of representative pairs found versus generation for differing program sizes. In contrast to the trend shown previously in figure 6.16, where larger population sizes gave a steep but brief increase in the number of representative pairs as the run progressed, figure 6.19 gives a slower and more sustained rise in the number of representative pairs for larger maximum program sizes as the run progressed. Similar trends are shown for the two forms of schema. It may be that while all generations share a population size and thus a population size affects all generations evenly, code-growth causes later generations to be more affected by a large program size.

6.7.3 Function arity

Some GP systems have varying numbers of arguments to functions, for instance, an addition node may have 1, 2, 3, 4 or more child arguments. It is expected that the number of representative pairs has correlation to

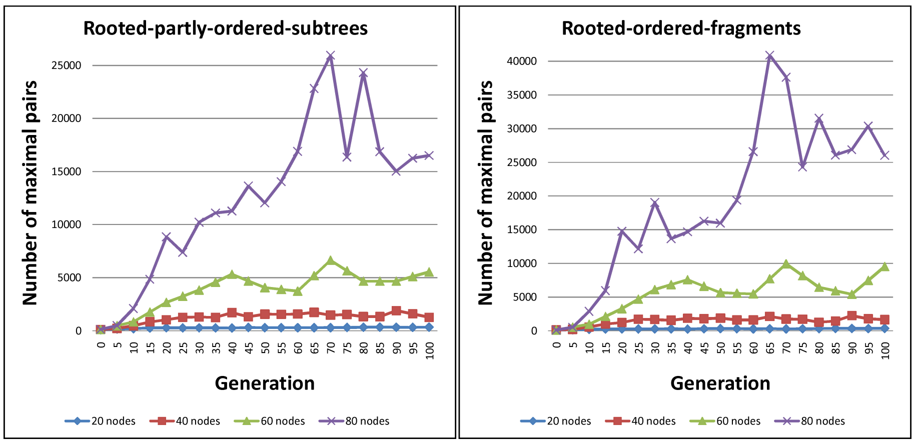

Figure 6.19: Median number of representative pairs versus generation for different program sizes, task BCW for two forms of schema.

this *arity* value. In the experiments of this subsection, the method was run with default parameters other than varying the function arity from 2 to 4 steps of 0.25. The arity was set at the start of each run and influenced the desired number of children for the variable arity functions created by mutation or in the initial population. Each population may still contain a mixture of arities of function nodes for any given primitive since the arity value set here gives only an upper bound on the preferred number of children assigned to any given function node.

Figure 6.20 shows how the function arity affects the number of representative pairs. The figure has two plots. Each plot gives quartiles over 50 runs per function arity of the number of representative pairs, which was capped at 200000, versus function arity.

The figure shows that there is indeed a correlation between the number of representative pairs and the arity of the function nodes with higher arities in general leading to more representative pairs.

For an arity of 2, the number of representative pairs is about half that when the arity is 2.5, the default setting for experiments in this thesis. Thus we could expect that a system which uses binary functions could perform

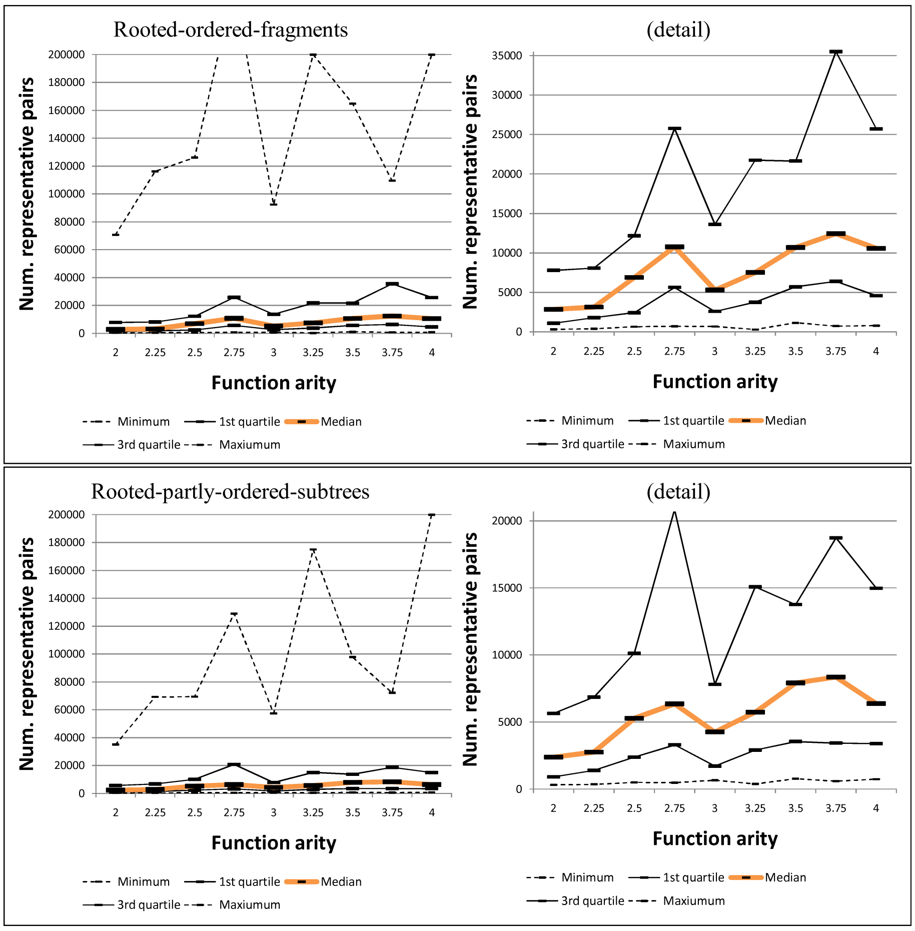

Figure 6.20: Effect of function arity on the distribution of numbers of representative pairs at generation 40, task BCW for two forms of schema.

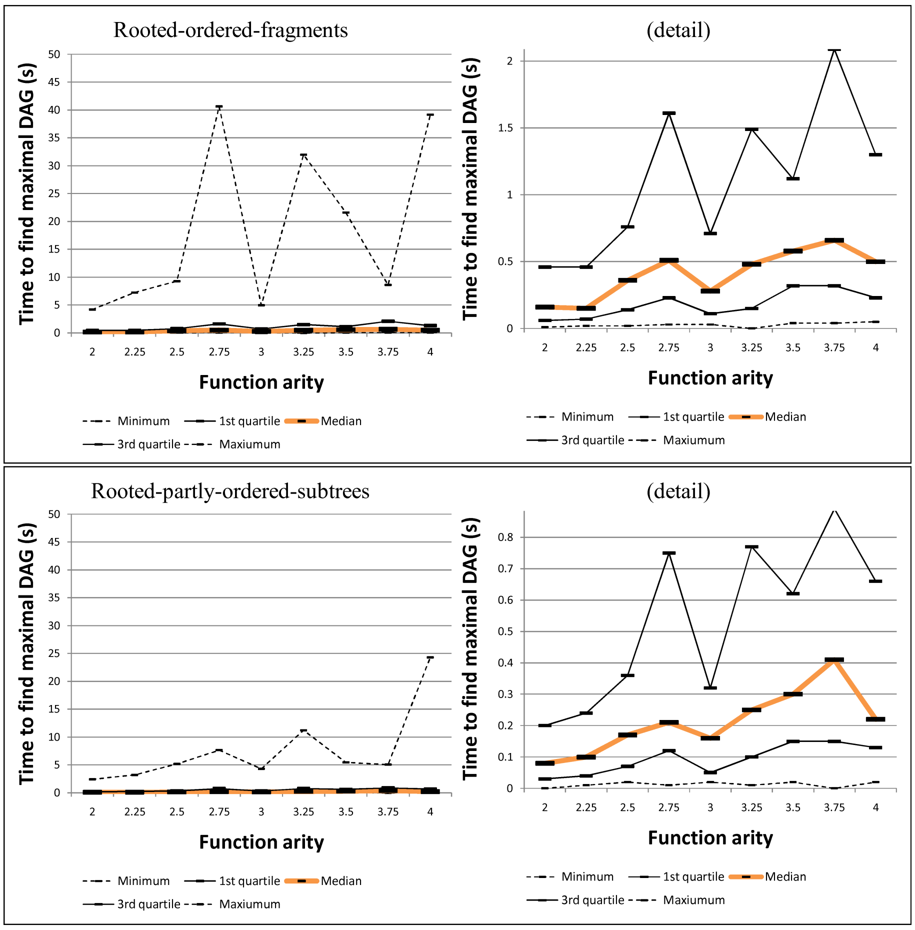

Figure 6.21: Effect of function arity on the distribution of times taken to find the representative pairs at generation 40, task BCW for two forms of schema.

faster and take less memory than the default setup presented here.

To show this effect of function arity on the time taken by the new method, figure 6.21 shows the time taken versus the function arity. The figure has two plots, each with quartiles of the time taken to find the unannotated representative pair DAG versus function arity.

As expected, the plots of figure 6.21 have much the same trends as the plots of figure 6.20 with the new method taking about half the time to process populations with binary functions than the populations produced with the default arity of 2.5. Figures 6.20 and 6.21 are quite noisy and have with a significant dip at arity 3. This may be due to the use of fractional arity affecting the evolutionary process; where integer arity produces many functions with the same arity, fractional arity produces a population of many function arities and this may change the dynamics of the GP evolution and the matched schemata. Further experiments would be required to determine whether the noise is just noise or symptomatic of the system used.

6.8 Discussion

The experiments of this chapter aim to test and characterize the new method analyzing GP populations in a "typical" GP system. Thus the GP engine parameters used are standard as far as possible. No one system could represents all GP setups that the method might be used on and indeed a useful advantage of this empirical method over theoretical schema theory methods is that it may be used in *unusual* GP setups. Nevertheless, this chapter does give useful information on the general trends one may encounter in using the new method.

Section 6.4 gives good evidence that each of 20 variants of the *addMeets* algorithm, which is core to the new method, gives exactly the same result as each other given the same input population and form. Each experiment derived a 64-bit hash from the DAG of representative pairs. The experiments of this section compared these hashes between the variants for each population and form. Though hash collisions could possibly occur, it is highly improbable.

There are other algorithms of the new method which are harder to test in this way since only one variant exists. For example, there is only one algorithm to get the tree representation of a schema and only one algorithm to get the histograms of represented program subsets for a representative program subset. These other algorithms were tested visually, by using them on simpler cases and testing that they produced the correct results.

The second set of experiments provides us with expected distributions for the default experimental setup, of: the number of representative pairs produced by the new methods, the time taken by the new method and the memory footprint of the new method both in memory and on disk. Most striking is the correlation of the time taken and memory footprint, of the new method with the number of representative pairs found. An approximately linear relationship is shown in most plots of figures 6.9, 6.11 and 6.12 showing that the defining characteristic affecting any given run of the schema engine is the number of representative pairs produced.

The number of representative pairs found in any given trial varied widely. Many runs had small numbers, while a few had very large numbers. But, when compared with the total number of schemata or program subsets, the number of representative pairs is tiny and proves to be a good summary for analysis of even moderately large populations of moderately large programs. As expected, a major influence on the number of representative pairs was the form of schema used. Non-rooted forms of schema were found to produce a great deal more representative pairs than rooted forms.

Of the tested variants of the addMeets algorithm, ProMP appears quickest. The form of schema used also influences the relative speeds of the algorithms slightly through the increasing or decreasing of the size of the schema components set which defines each schema; some variants like IntMS suffer if these sets get too large, where some like IntMPS are indifferent.

Section 6.7 gives characterizations of the new method with varied parameters, including: population sizes up to 501 programs, program sizes up to 80 nodes and function arities up to 4 arguments. For the two forms of schema used, rooted-ordered-fragments and rooted-partly-ordered-subtrees, the new method exceeded the limit of 200,000 representative pairs about

25% of the time for populations of 301 60-node programs. Program size was seen to be more influential on the number of representative pairs than population size; while the number of representative pairs is approximately quadratic on the number of programs in the population, it is approximately quartic on the size of the programs in nodes. Raising the function arity was also found to slightly increase the number of representative pairs. Many GP setups use binary functions for addition, subtraction, multiplication and division and the results of section 6.7.3 suggest that in these setups there would typically be about half the number of representative pairs for the same population size, program size and form of schema.

Another varied parameter which is not listed in the experiments of section 6.7 is the numeric terminal block size, which was found to have no significant effect on the number of representative pairs. Lowering this parameter should in theory increase the *total* number of schemata occurring in the population, through numeric terminals with slightly different values being considered different for small numeric terminal block sizes.

All experiments were run on two tasks: "BCW" and "sphere". Most of the chapter's figures are for the "BCW" task since these figures reported very similar trends to the equivalent figures for the "sphere" task. As expected, the influence of the task performed by the "GP engine" had little bearing on coarse "schema engine" performance statistics such as number of representative pairs and time taken.

6.9 Chapter summary

This chapter empirically characterized the performance of the new method. The experiments of this chapter aimed to test the new method in several ways:

- To indicate whether each variant of the algorithm producing the representative pairs as a DAG gives exactly the same DAG of representative pairs as output.

- Characterize the typical number of representative pairs given as output.
- Characterize the typical time taken by the method and the typical memory footprint of the method both on disk and in RAM.
- Indicate correlations between the number of representative pairs output the time taken or memory footprint both on disk and in RAM of the method.
- Characterize the time requirements of the new method for different variants of the algorithms for the new method.
- Characterize the time and space requirements of the new method over different parameter values, including: population size, program size and function arity.
- Identify trends in these statistics over generation number.

As may be expected, the number of representative pairs drives both the time taken by the new method as a whole and its memory footprint with approximately linear relationships to time-taken, bytes-on-disk and bytes-in-RAM. Importantly, the number of representative pairs was found to be manageably small for most parameter settings with a limit of 200,000 pairs found to be an acceptable setting for all rooted forms of schemata when using a population size of 51 and for rooted-partly-ordered-subtrees and rooted-ordered-fragments in populations up to 301. But there was a large increase in the number of maximal schemata by using a non-rooted form of schema. The limit of 200,000 representative pairs was too small for most non-rooted forms of schema tested.

These small numbers of representative pairs compare well to the total numbers of schemata in the populations, which at times would easily exceed 10^{100}, justifying the use of the new method over the naive iteration over all schemata.

The number of representative pairs was found to be heavily influenced by three parameters being quadratic on population size, quartic on program size and highly dependent on form of schema.

Little or no correlation was found between the number of representative pairs and the numeric terminal block size. A small positive correlation was found between the number of representative pairs and the function arity.

Where this chapter characterized the new method, the next chapter assesses its efficacy by providing some potentially useful and previously impractical analyses of GP runs using the new method.

Chapter 7

Efficacy experiments

7.1 Chapter introduction

In contrast to the previous chapter which aimed to characterize the new method, this chapter aims to show that is useful by presenting a range of simple analyses of GP runs.

The method presented in this thesis provides a tool for the analysis of GP evolutions which is powerful in these key ways:

- The tool generates useful statistics over the otherwise prohibitively large set of schemata occurring in a set of programs.
- The tool may be tailored to many forms of schema, through accepting a match-tree form as a parameter.

This chapter presents the results in graphical form of several statistics which are likely to be of interest to the GP community, starting with simple statistics and ending with more complex analyses. The statistics presented include:

- Section 7.3 gives the total number of schemata occurring in any program in the population.

- Section 7.4 gives the size of the largest schema occurring in any set of n programs for various values of n.
- Section 7.5 gives the size of the largest schema occurring in a set of n programs, averaged over all such sets of programs for various values of n.
- Section 7.6 gives the distance between each pair of programs, where the distance is the percentage of shared schemata.
- Sections 7.4 and 7.6 also have analyses on *generational-populations* described in section 7.2.1 below.

The chapter presents these results on a wide range of schemata which are presented to the system as match-tree forms:

- Rooted-ordered-fragments (*r-o-fragments*)
- Rooted-ordered-hyperschemata (*r-o-hyperschemata*)
- Rooted-partly-ordered-subtrees (*r-po-subtrees*)
- Ordered-subtrees (*o-subtrees*)
- Ordered-fragments (*o-fragments*)
- Rooted-restrictive-ordered-fragments (*r-x-o-fragments*)

7.2 Experimental setup

The experimental setup is very similar to that of the previous chapter. Tables 6.1 and 6.2 in that chapter give default GP and schema engine settings respectively.

Most results given are the median result over fifty randomized trials. Exceptions are graphs giving a distribution over the fifty trials and the results given in section 7.6 which are for a single run of evolution.

7.2.1 Generation populations

There are broadly two ways to construct the set of programs, used as input into the method, from the series of populations at each generation of a run of evolution.

Mostly, the experiments of this chapter use the populations themselves as input. Some experiments use populations obtained by grouping together programs of differing generation rather than rank. This thesis will refer to these sets of programs as *generational-populations*. They provide a useful contrast between the good programs and the bad programs, at each generation.

7.3 Number of schemata of each size

This section presents a very simple, yet deceptively hard to obtain, statistic on a population of programs: the number of schemata of a given form in any one or more programs of the population.

Chapter 5 gave an algorithm for finding the number of schemata in a single program, however, the case for multiple programs is significantly more difficult since the analysis must count each schema only once no matter how many programs it is in. Using the method of this thesis, the task is simple: sum the numbers of represented schemata for each maximal schema. Since each schema is represented by exactly one maximal schema, the new method arrives trivially at this otherwise difficult statistic.

Method

Given a set of programs and a form of schema, the representative sets of schemata are found. Each representative set of schemata is annotated with a histogram containing the count of represented schemata of each size.

The total number of schemata of size z is the sum over all representative sets of schemata of the count of represented schemata of size z.

Using the analysis algorithms of chapter 4, the analysis is trivial. It is an analysis over schemata with the following two defining functions:

- $\mathsf{AnSc}(P, z_s) = 1$
- $\mathsf{Agg}(A, A', c) = A + A' * c$

The AnSc function provides a count of 1 for each schema. The aggregate function Agg sums these counts by adding c times this count per c schema.

Results

Figure 7.1 gives the numbers of schemata of each order (that is, number of non-don't-care nodes) as a median over fifty randomized trials for six forms of schema. As expected, the figure shows that the most important factor determining the number of schemata is the form used. There are three orders of magnitude more hyperschemata than any other form of schema. There were very similar numbers of r-o-fragments, r-po-fragments, and o-fragments (although there are more small o-fragments than either of the others). The r-x-o-fragments show heavy bias toward large schemata since their matching behaviour prevents low arity schema node functions from matching higher arity program node functions.

The stage in the run also affects the results with numbers of all forms of schemata increasing more than an order of magnitude from the first generation to the 60th. Interestingly, this increase in numbers of schemata toward later generations was more pronounced in the r-x-o-fragments than the other forms of schemata.

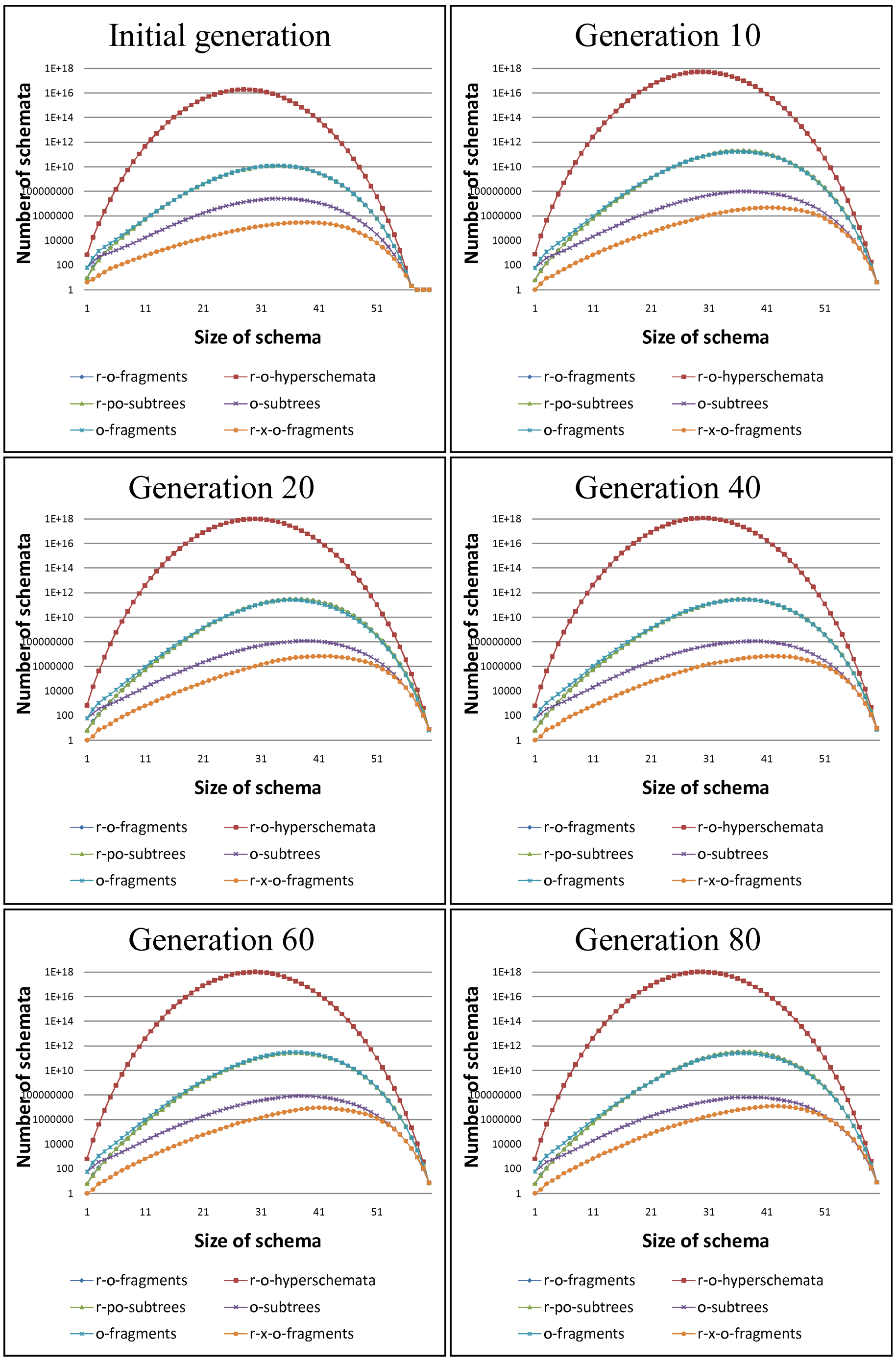

Figure 7.1: Total number of schemata occurring in any program in a population of 51 programs for various generation numbers and forms of schema. Note that the r-p-o-subtrees and r-o-fragments series overlap.

7.4 Largest matched schemata by k programs

This section presents another statistic readily available using the new method: the size of the largest schema in any set of k programs for different sizes k.

This statistic is of considerable interest:

- Evolution tends to stagnate with slow improvement in the quality of programs if too many programs are too similar and the population has *converged*. In this case programs may share some large schema.
- This statistic could be used in some form as a measure of diversity and correlated between good runs and bad to characterize its effect on performance.

Method

Where the previous analysis iterated over the set of schemata, this analysis iterates over the set of program subsets.

Using the analysis algorithms of chapter 4, the analysis is over program subsets with the following defining functions:

- $\mathsf{AnPS}(S, z_P) =$ if $z_P = k$ then return the size of largest schema in S, otherwise return 0
- $\mathsf{Agg}(A, A', c) = \mathrm{Max}(A, A')$

The aggregating function Agg returns the maximum value found by the function AnPS, which returns the size of the largest schema in the iterated program subset, so long as the program subset has cardinality k. Thus the analysis finds the largest schema in any program subset of size k.

Results

Figure 7.2 gives the trend of the largest schema size in a set of k programs, over generation number for four forms of schema. Each graph has gen-

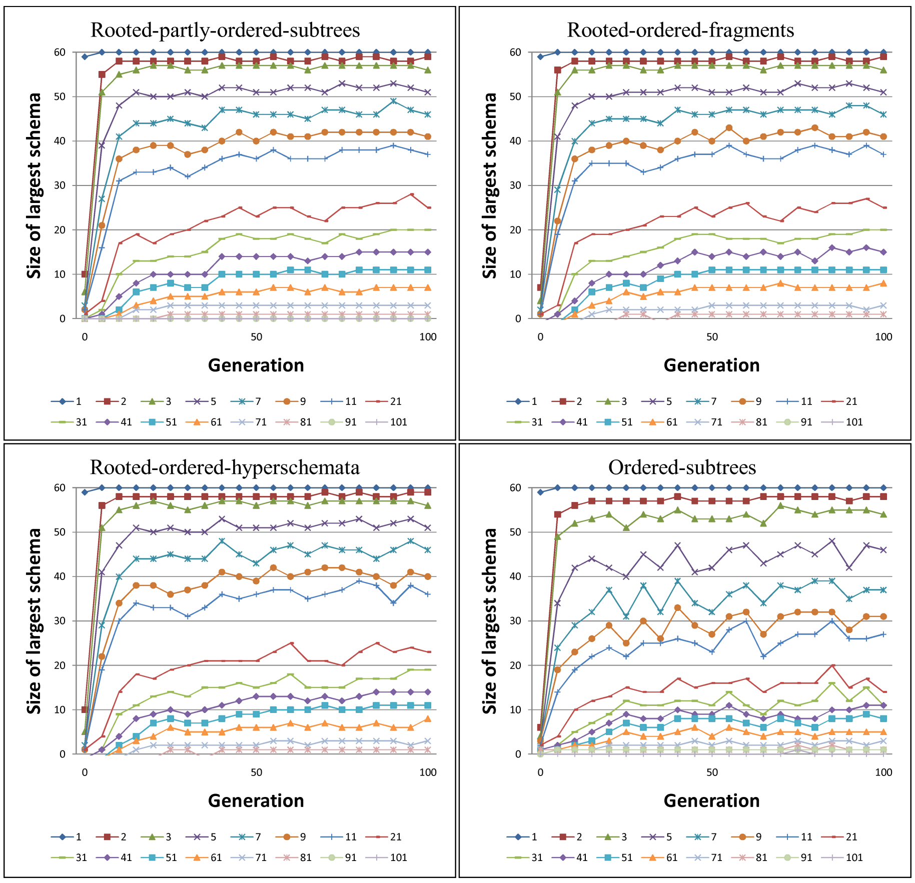

Figure 7.2: Size of largest schema occurring in k programs over generation for four forms of schema. Median values over 50 runs with population size 101.

eration on the horizontal axis and the size of the largest schema in any k programs on the vertical axis for various values k from 1 to the population size of 101 as the series. Thus we would expect the series for $k = 1$ to simply record the size of the largest program at each generation and the series at $k = 101$ to record the size of the largest schema occurring in all programs if any. A diverse population would be expected to have low values for most series and a highly converged population might have very high values for most series. The graphs record the median values over 50 randomized trials.

The figures are surprising; for the three rooted forms of schemata, even after only ten generations, very large schemata exist which are in many programs. For instance, there is a 44 node rooted-ordered-fragment which is in some set of 7 programs. Despite the very fast rise, the sizes don't change a lot later in the run. For instance, the largest rooted-ordered-fragment in generation 100 is still about 46 nodes. Though the runs very quickly developed large schemata, they did not fully converge since no large schemata occurred in large program subsets. Note that the results show the median over 50 trials.

The series for the centre values of k are lower for the ordered subtree form of schema than the other three forms, suggesting that the programs shared larger fragments and hyperschemata than they shared subtrees, in turn validating our interest in the more expressive forms of schema. Interestingly, the size of the largest subtree seems a noisier statistic than the size of the largest fragment, partly-ordered subtree or hyperschema. Subtrees seem to represent a more "fragile" form of schema than these other forms; if a program changes in a way that invalidates its match with a given schema s, the expressiveness of s's form of schema affects how much s must change to once again match the program. For the expressive form of hyperschemata all that may be required is a targeted insertion of a *don't care*, whereas for the less expressive form of subtrees a much greater modification to s may be required.

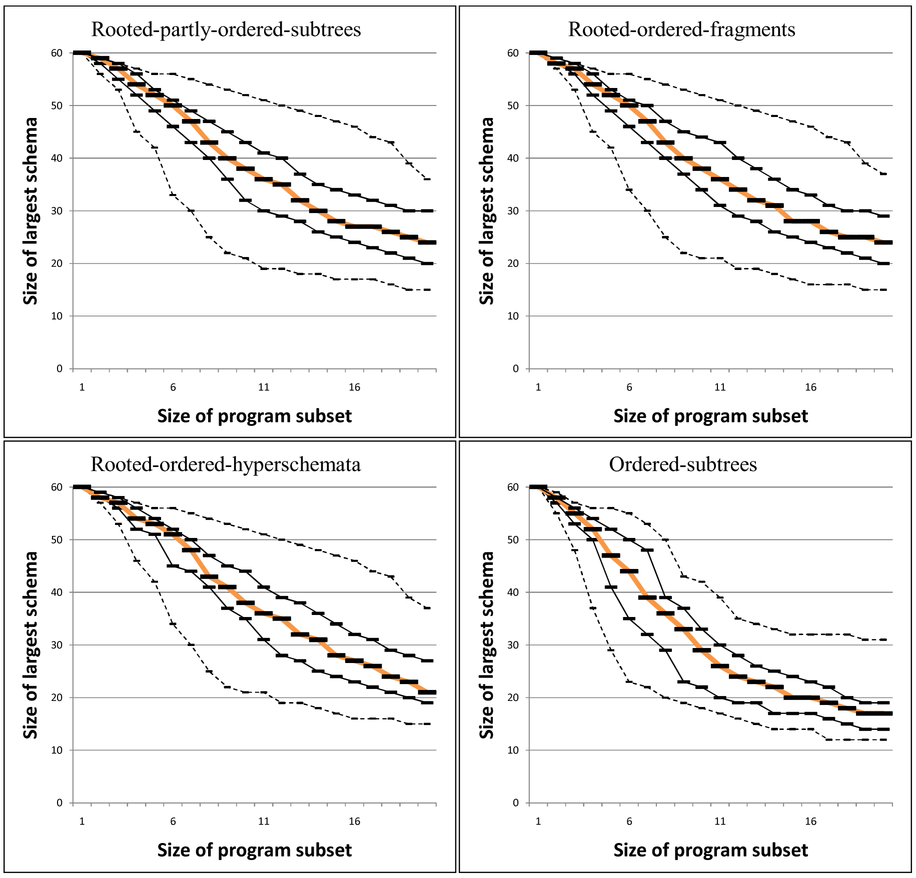

Figure 7.3: Quartiles over fifty runs of size of largest schema occurring in k programs (size of program subset) for four forms of schema at generation 40.

Figure 7.3 shows the distributions for the sizes of the largest schemata in k programs at generation 40 for different values of k. The five lines correspond to the maximum, upper quartile, median, lower quartile and minimum over the 50 randomized trials. As expected, all runs had some very large schema in at least one program. The three rooted forms of schema show similar distributions for larger values of k with ordered-subtrees showing a different trend. Where for the rooted forms at least one run has some over-40-node schema in some set of 18 programs, the maximum values for large k are smaller in the case of ordered-subtrees.

Figure 7.4 focuses of the case of rooted-ordered-fragments by showing graphs similar to those above but for the initial, 5th, 10th, 20th, 40th and 60th generations. As expected, the initial generation shows very small shared fragments for all runs; the random programs of the initial generation have not been combined by selection and crossover and thus have very little chance of the spontaneous sharing of schemata. Later generations show a general increase in the sizes of shared schemata as the run progresses. Interestingly, after the initial generations, the run with the largest shared schemata shows little variance in the size of those schemata with generation and has reached its peak by generation five. Most other runs show considerable increase in the size of the largest fragment in k programs as the run progresses.

The central half of the runs, from the lower quartile to the upper quartile, show very similar characteristics with very little difference in schema size over all values of k shown. Unexpectedly, this is especially true later in the run.

Figure 7.5, again dealing with rooted-ordered-fragments, analyzes generational-populations for various ranks. The six graphs of the figure show the distribution of the largest schemata shared by the programs of rank k at each generation from the initial generation to generation 100 with each graph showing data for a different value for k. Thus for instance, the figures could show if selection favours the sharing of material from good programs and suppresses the sharing of material from bad programs.

Indeed, this is exactly the trend seen in the results; the best programs share large schemata across many generations and for at least 75% of the trials, the worst programs share no large schemata across even a few generations. There are other interesting trends: the largest shared schemata line is surprisingly high for the worst programs, that is, while the worst programs in almost all runs shared few if any schemata across multiple runs, there is some run where even the worst programs shared a fragment of size 19 across 8 generations. This run has likely converged completely such that for several late generations all programs in the population share a sizeable fragment, although more analysis will be required to confirm this hypothesis.

Small sets of programs, that is small numbers of generations, show similar trends between programs of rank 11 and those of rank 31, indicating that there is some region of evolution where programs of the two quite different ranks share similar sized fragments. But for larger sets of programs, that is larger numbers of generations, the better programs share significantly larger schemata that the worse programs. This likely indicates that most runs have a period of convergence when even unfit programs share large schemata, that this period is typically at least 6 generations in length and that the shared schemata occur in fitter programs beyond this period of convergence.

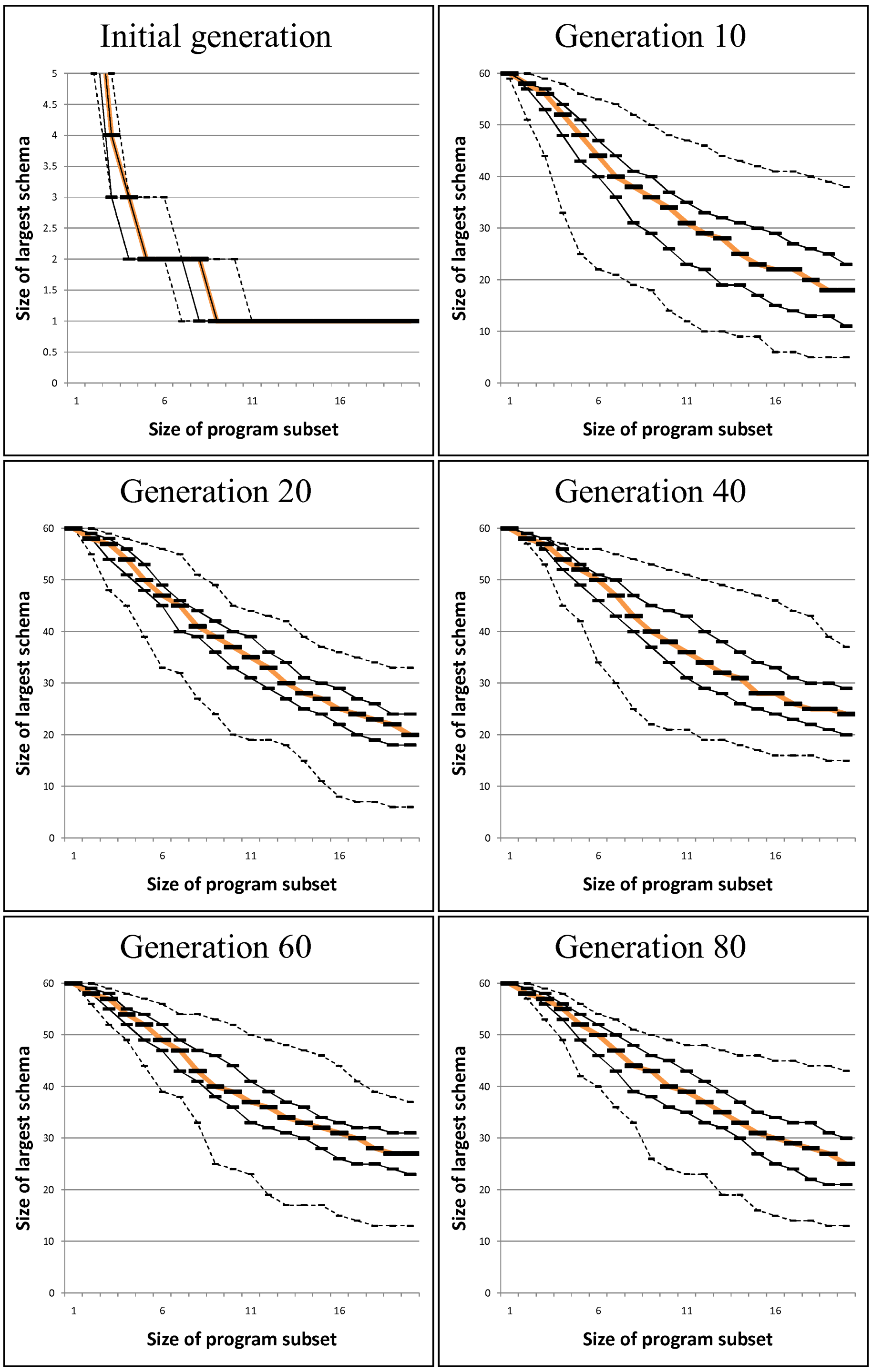

Figure 7.4: Quartiles over fifty runs of size of largest r-o-fragment occurring in k programs (size of program subset) for four forms of schema at various stages in the run.

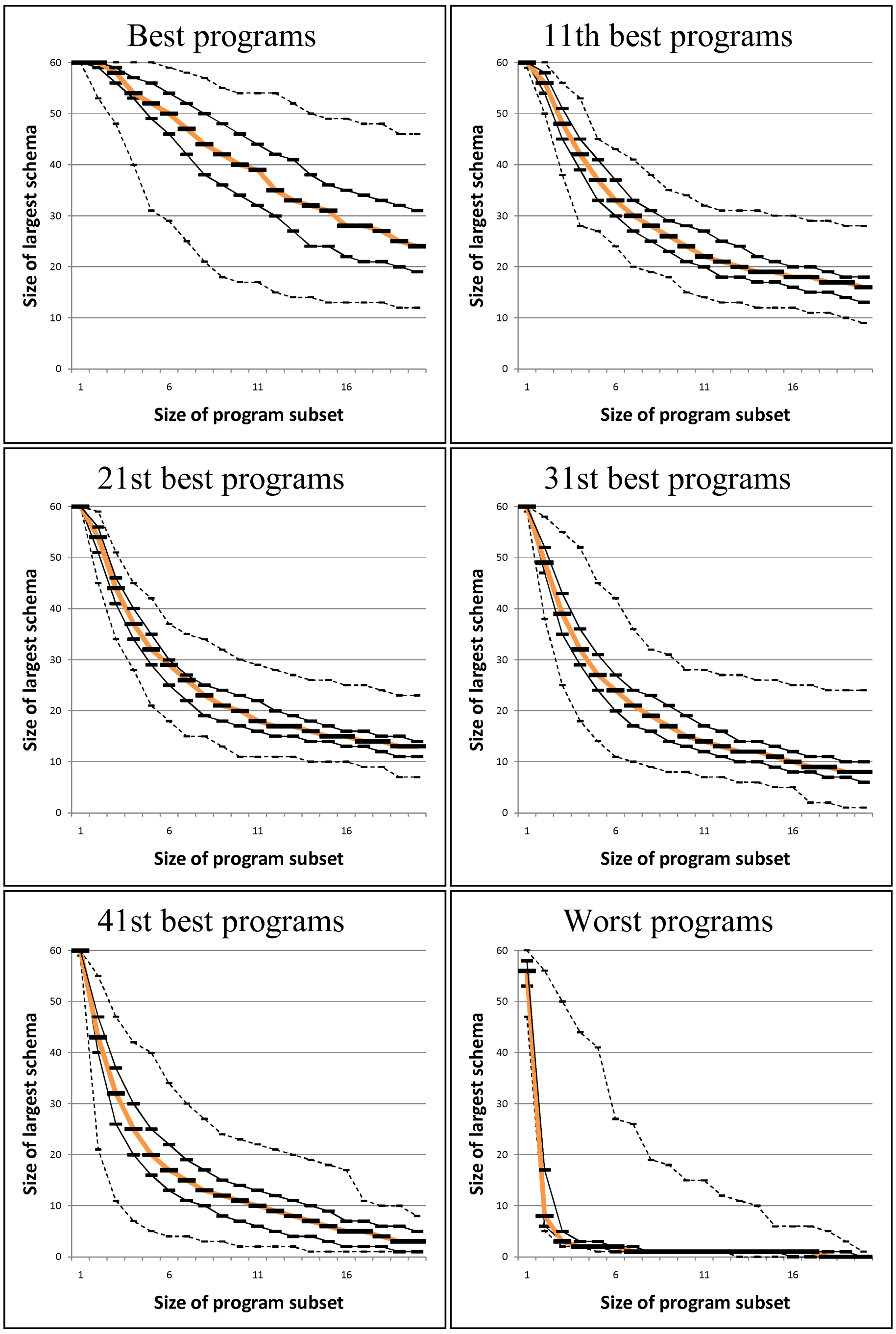

Figure 7.5: Quartiles over fifty runs of size of largest r-o-fragment occurring in k programs (size of program subset) for four forms of schema and generational-populations of various ranks.

7.5 Average size of maximal schemata

Two interesting statistics available using the new method that are closely related to the largest schema sizes of the previous subsection, are: the size of the largest schema occurring in each of a set of programs P of cardinality k averaged over all such sets P and the average number of programs matching schemata of order k. These statistics give insight into not just the very most popular schemata but the population at large; these statistics take into account each match between a schema and a program, which is not the case if just the largest schema is found.

This section presents results for these two statistics in single runs of evolution:

- The average maximal schema size – for a program subset P the experiment finds the size z of the largest schema occurring in all programs of P and the average such z over all P's of a given cardinality k.
- The average maximal program subset size – for a schema s the experiment finds the number of programs k matching s and the average such k over schemas s of a given order z.

It is expected that, where the population to converge, the resulting lack of diversity would push up both the average maximal program subset size and the average maximal schema size.

Method

The two statistics of this section are complementary with one iterating over schemata and one over program subsets.

Using the analysis algorithms of chapter 4, the average maximal schema size is found using the following defining functions:

- $\mathsf{AnPS}(S, z_P)$ = a pair containing a *sum* and a *count*: if $z_P = k$ then return <size of largest schema in $S, 1 >$, otherwise return $< 0, 0 >$

- Agg$(A, A', c) =< A_{sum} + cA'_{sum}, A_{count} + cA'_{count} >$

Thus each aggregated value is both a sum of the aggregated values from the analysis function AnPS and a count of how many sizes there were. The final average is found by dividing the sum by the count. AnPS gives the size of the largest schema occurring in each program subset if the size of that program subset is k, and a value of no effect otherwise. Thus the analysis provides the size of the largest schema matching a program subset P, averaged over each program subset P of cardinality k.

The average maximal program subset size is similarly found using the following defining functions:

- AnSc(P, z_s) = if $z_s = k$ then return $< |P|, 1 >$, otherwise return $< 0, 0 >$

- Agg$(A, A', c) =< A_{sum} + cA'_{sum}, A_{count} + cA'_{count} >$

The aggregating function again allows an average of the values of the analysis function, which in this case provides the size of the matching program subset for each analyzed schema of order k. Thus the analysis provides the number of programs matching a schema s, averaged over each schema s of order k.

Results

Figure 7.6 gives the trend of the average maximal schema size over all sets of k programs, by generation number for four forms of schema. Each graph has generation on the horizontal axis and the average maximal schema size on the vertical axis. The four series are the four sizes of program subsets used for the average.

Three of the four graphs show an interesting feature at exactly generation 64; the average maximal schema size very suddenly increases for all schema sizes, in some cases by a factor of 5 in one generation. It seems that

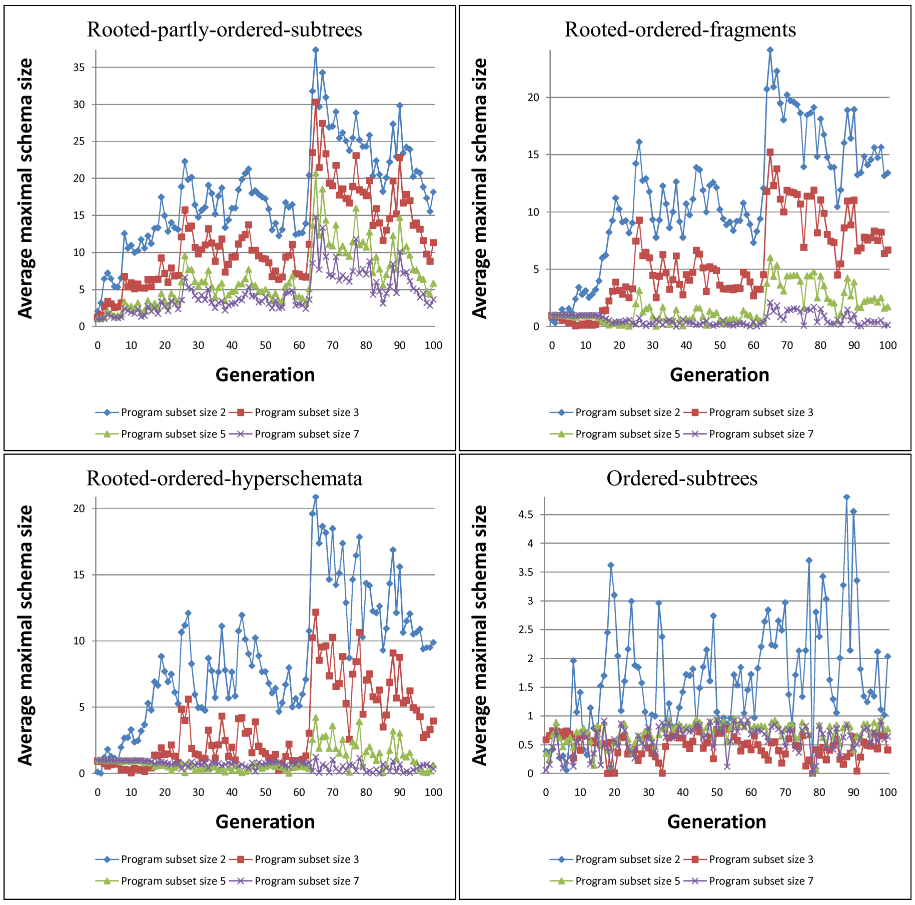

Figure 7.6: Average, over all program subsets of a given size, of the largest schema occurring in the program subset. Values are for the “seed 1” trial with population size 51.

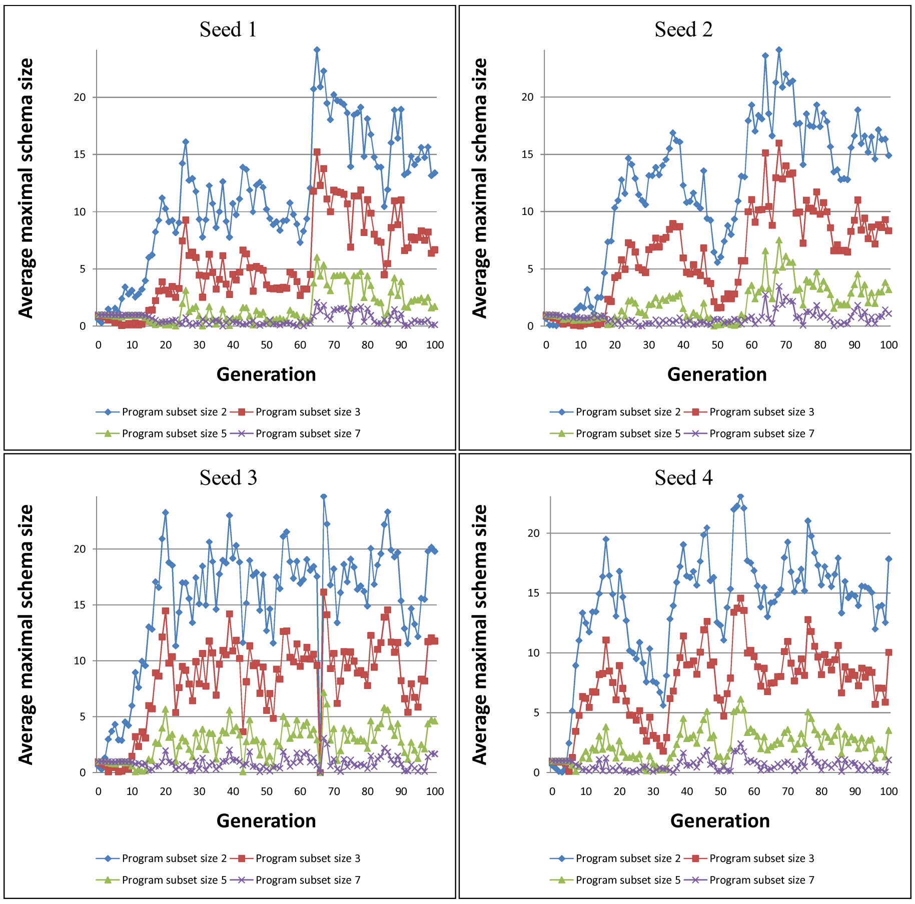

Figure 7.7: Average, over all program subsets of a given size, of the largest schema occurring in the program subset. Four trials (trials 1 to 4, population size 51).

at that exact generation some very large schemata suddenly became popular, occurring in at least 5 programs. But these schemata were not subtrees since the ordered subtree graph shows no such feature. The feature is most marked for the expressive hyperschemata form of schema.

Figure 7.7 shows similar graphs but rather than showing four forms of schema shows four trials. As before, each graph has generation on the horizontal axis and the average maximal schema size on the vertical axis.

The four series are the four sizes of program subsets used for the average.

The graphs each show marked different trends with the sudden increase feature only occurring in the previously shown “Seed 1” trial. All trials show small average maximal schema sizes in the earlier generations but where the “Seed 2” trial does not show marked increase until generation 17 the last two trials show rapid increase even before generation 9. The “Seed 3” trial especially shows no change in the series after about generation 15. Sections 7.6 and 7.7 give further analysis of this run, showing it has converged by generation 20.

Figure 7.8 shows the average number of matching programs over all schemata of a given size, by generation for four forms of schema. Each graph has generation on the horizontal axis and the average maximal program subset size on the vertical axis. The four series are the four sizes of schemata used for the average.

The sudden-increase feature at generation 64 is gone with this figure showing a noisy but steady increase in program subset size for all forms of schema for most of the run, up until a maximum is found at about generation 60 to 80. It is interesting to compare the different forms of schema; where very small subtrees are highly shared, very small hyperschemata are not. Perhaps this is because programs sharing a very small hyperschema must share material at the root while very small subtrees may occur in different programs even if the programs do not share a root. Order 7 subtrees occur in as many programs as the same sized hyperschemata. It seems that the larger, order 7, hyperschemata occur in few less programs than smaller, order 2, hyperschemata even early in the run. This may be due to hyperschemata of any of the given sizes occurring in on average few programs.

Partly-ordered-subtrees and fragments show a trend somewhere between the two extremes of hyperschemata and subtrees; small partly-ordered-subtrees occur in on average about 6 programs and small fragments about 5 programs.

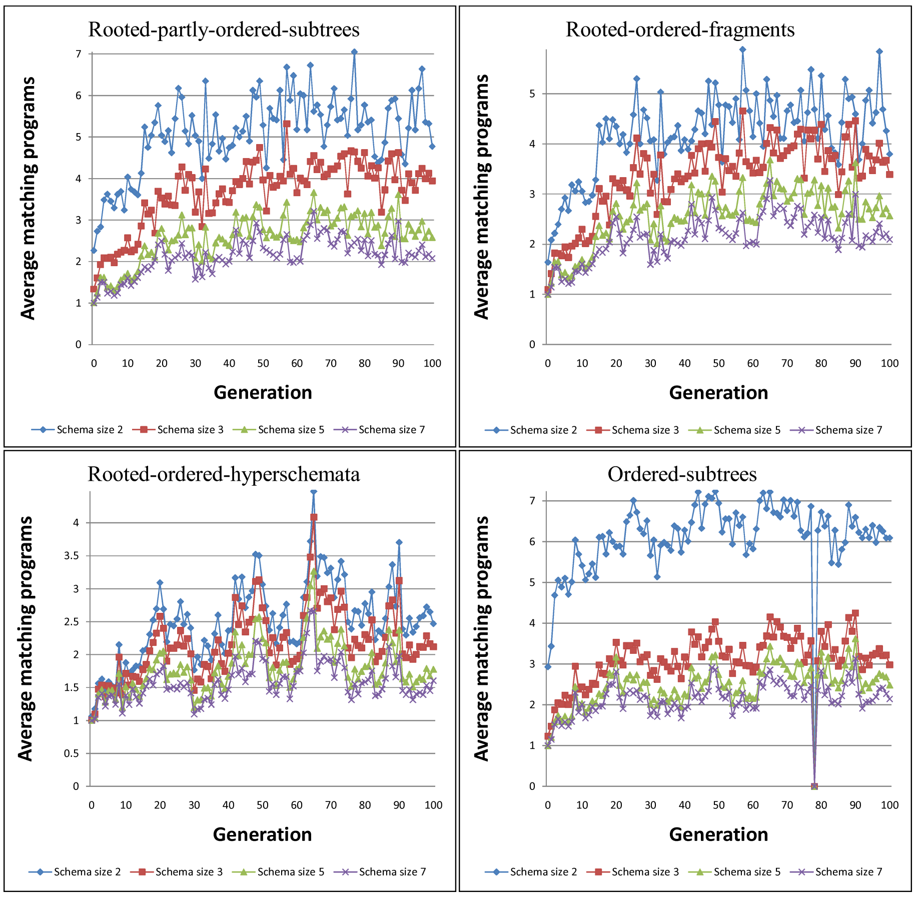

Figure 7.8: Average number of matching programs over all schemata of a given size. Values are for one run (trial 1 with population size 51).

7.6 Similarity between programs

This section gives an example of a more complex analysis possible using the new method. Schemata may be effectively used to measure the similarity between pairs or sets of programs. This section presents one way of forming a program similarity measure using the new method.

The similarity of two programs $S(a, b)$ is $\frac{N_{a \wedge b}}{N_{a \vee b}}$, where $N_{a \wedge b}$ is the number of schemata of a given form that occur in both a and b and $N_{a \vee b}$ is the number of schemata of a given form that occur in either a or b, or both a and b. If $a = b$ then this measure will give its maximum possible value of 1. If a and b share no schemata then the measure will give its minimum possible value of 0. Since there are typically many orders of magnitude more large schemata than there are small schemata, only schemata of a given size z_s are counted. By restricting to one size of schema the analysis avoids the possibility that the many large schemata of a large program would dominate over the few small schemata of a small program.

Another useful schema inclusion/exclusion filter is made possible by the method: since there may also be many uninteresting schemata which are in only one or two programs, the new method may be set to exclude or include schemata based on how many programs they occur in. In this section the parameter $\min_P$ sets a minimum number of programs a schema must occur in for it to count toward the program similarity calculation.

Method

This section presents similarities between each pair of programs a, b as the count of schemata shared by a and b divided by the count of schemata occurring in either a or b, or both a and b. This similarity value may be worked out from two statistics:

- $N_s(p)$: the count of schemata of size z_s occurring in program p and a total of at least $\min_P$ programs.

- $N_{\text{share}}(p_1, p_2)$: the count of schemata of size z_s occurring in both programs p_1 and p_2 and a total of at least $\min_P$ programs.

The count of schemata occurring in either a or b is found as $N_s(a) + N_s(b) - N_{\text{share}}(a, b)$.

To find $N_s(p)$ the analysis uses a similar method to that of section 7.3. Using the analysis algorithms of chapter 4 the analysis is over schemata with the following two defining functions:

- $\mathsf{AnSc}(P, z_s) = 1$ if $p \in P$ and $|P| \geq \min_P$, 0 otherwise
- $\mathsf{Agg}(A, A', c) = A + A' * c$

The aggregating function Agg is the sum function also found in section 7.3. In this case the aggregated analysis function returns 1 for schemata matching p and at least $\min_P - 1$ other programs, and returns 0 otherwise. Thus the analysis returns the count of these schemata. This core analysis is run for each program p in the input population.

A similar method finds $N_{\text{share}}(p_1, p_2)$ for each pair of programs. Using the analysis algorithms of chapter 4 the analysis is again over schemata with the following two defining functions:

- $\mathsf{AnSc}(P, z_s) = 1$ if $p_1 \in P$ and $p_2 \in P$ and $|P| \geq \min_P$, or 0 otherwise
- $\mathsf{Agg}(A, A', c) = A + A' * c$

The aggregating function Agg is the same sum function used previously. The aggregated analysis function returns 1 for schemata matching p_1 and p_2 and at least $\min_P - 2$ other programs, and returns 0 otherwise. Thus the analysis returns the count of these schemata. This core analysis is run for each pair of programs p_1, p_2 in the input population.

Thus defining P_0 as the population and R as the set of maximal pairs used for the analysis, the complexity of the analysis is $O(|P_0|^2|R|)$ since the analysis producing $N_{\text{share}}(p_1, p_2)$ has complexity $O(|R|)$ and this core analysis is run $|P_0|^2$ times.

The figures of this section present programs on both the *x* and *y* axes with the i^{th}program on the *x* axis also being the i^{th}program on the *y* axis. The ordering of the programs is vital to clearly presenting any clusters of similar programs that may exist and the natural ordering of programs by rank tends to hide these clusters by placing dissimilar programs with similar fitnesses next to each other.

For the plots of this section an alternative approach was used: once the similarity between each pair of programs was found, the programs are ordered by rank, and a function repeatedly optimized this order toward an order where similar programs are close together and dissimilar programs are far apart. This optimization proceeded for 50,000 iterations.

Each iteration randomly assigned a number $1 \leq n \leq 3$ and randomly shuffled n programs to new places in the order of programs. If this new order was found to have improved over the old order, then the new order was kept, otherwise the changes were rolled back. To test whether the new order was *better* the method used the following fitness function to reward orders for grouping similar programs together and dissimilar programs apart:

$$F(p_1, p_2, p_3, \ldots, p_{|P_0|}) = \frac{\sum_{i,j} (1 - \text{Similarity}(p_i, p_j)) * (i - j)}{\sum_{i,j} (i - j)^2}$$

where $\text{Similarity}(p_i, p_j)$ is the similarity of programs p_i, p_j as found by the analysis. This formula gives the *ordinary least squares* linear regression of program distance to difference in index and gives a high value if small distances correspond to small differences in index and large distances correspond to large separation of index.

Results

Figure 7.9 shows, for four forms of schema, the program similarities at generation 40 of one run of evolution (random trial “seed 1”).

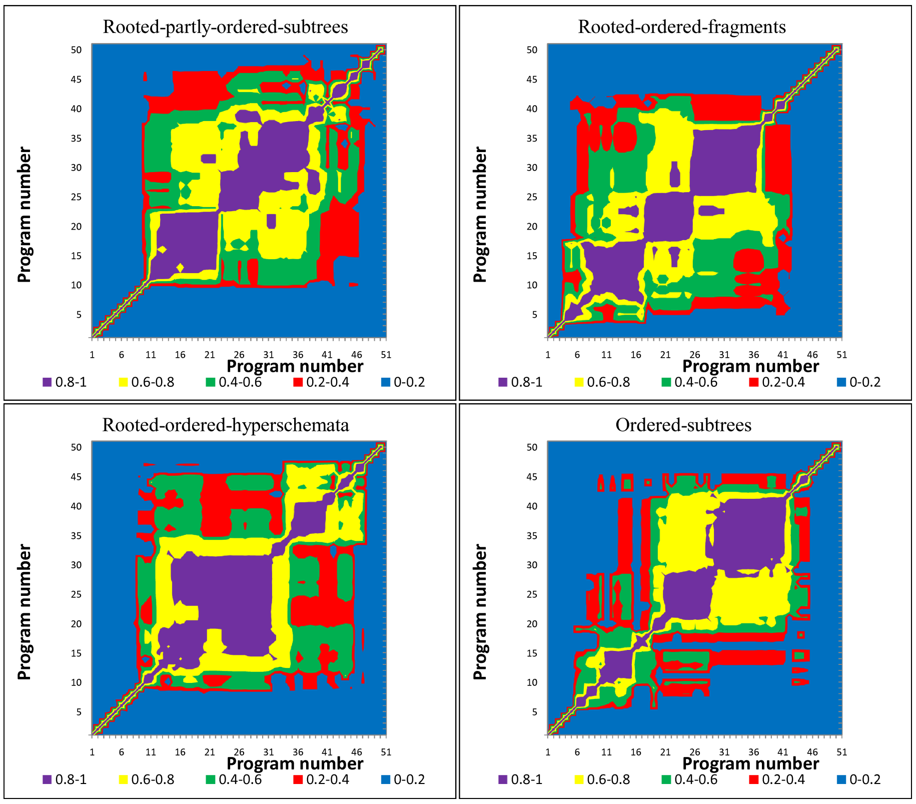

Figure 7.9: Program similarities between each pair of programs in generation 40 of one run with 51 programs for four forms of schema. $z_s = 5, \min_P = 1$

Only schemata with 5 nodes counted toward the similarity but the analysis placed no lower limit on the number of programs a schema must occur in.

The plots present the similarities as five scaled groups:

- Those pairs which share no schemata have the lowest value (blue, marked 0-0.2)

- The remaining groups hold the remaining pairs (red for low similarity, then green, then yellow and finally purple for high similarity) such that each group has the same number of pairs in it. This scaling step avoid non-linearities inherent to the schemata count based similarity calculation.

The figure shows clusters of similar programs in the population with each plot showing partly separated purple blocks of similar programs. Each plot shows many outer programs which are completely dissimilar in terms of schemata shared to all but one or two other programs. The purple diagonal line shows that all programs are similar to themselves.

The large central cluster in the plot for hyperschemata shows that most programs share many schemata of this very expressive form of schema. By contrast, the plot for the subtree form of schema shows less coherence with more distinct small clusters.

Figure 7.10 shows similar plots to figure 7.9 but counts only schemata occurring in at least 10 programs.

Comparing the plots, figure 7.10 has less fragmentation and more clear clusters of programs. This is especially true for the subtree and hyperschema forms of schema. In addition, much of the diagonal purple line is gone as these dissimilar programs match no schemata which also occur in at least 9 other programs.

Each plot shows at least one quite distinct square cluster of programs, which are mostly dissimilar to the other programs. Defining P as the set of programs which share some schemata with some other program, for the rooted forms of schemata all programs in P share some schemata with most other programs in P since the central, non-blue group is roughly a filled square. For the non-rooted subtree form of schema there is a small number of programs with share schemata with only a small subset of P.

A good use of this method is to show the state of the population at regular intervals as evolution progresses.

Figure 7.11 shows similar plots to figure 7.10 but shows the state of the population for each 10th generation and generation 5 during the "seed 1" evolution. The figure gives a good sense of how the evolution progressed:

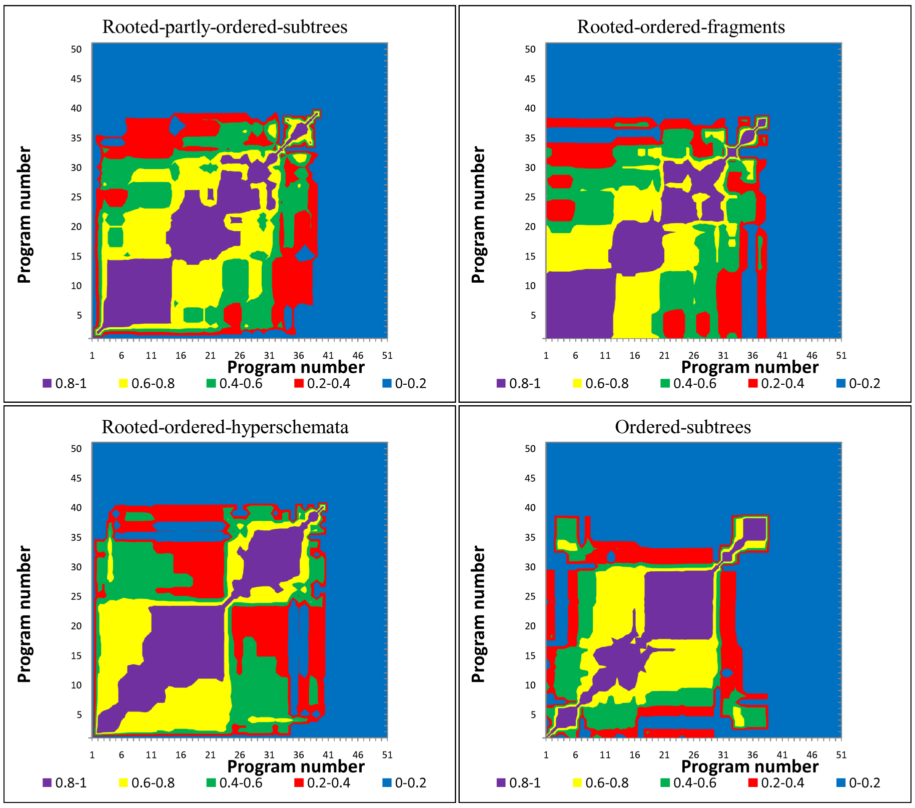

Figure 7.10: Program similarities between each pair of programs in generation 40 of one run with 51 programs for four forms of schema. $z_s = 5, \min_P = 10$

- The initial programs share no 5-node-sized schemata which occur in enough programs.
- After 5 generations, selection has produced five small (4 to 7 programs) clusters of similar programs but these clusters have not yet been combined.
- After 10 generations, the clusters remain but some have mixed with others with some similarity of programs from different clusters. But one cluster has both very high similarity of programs in the cluster

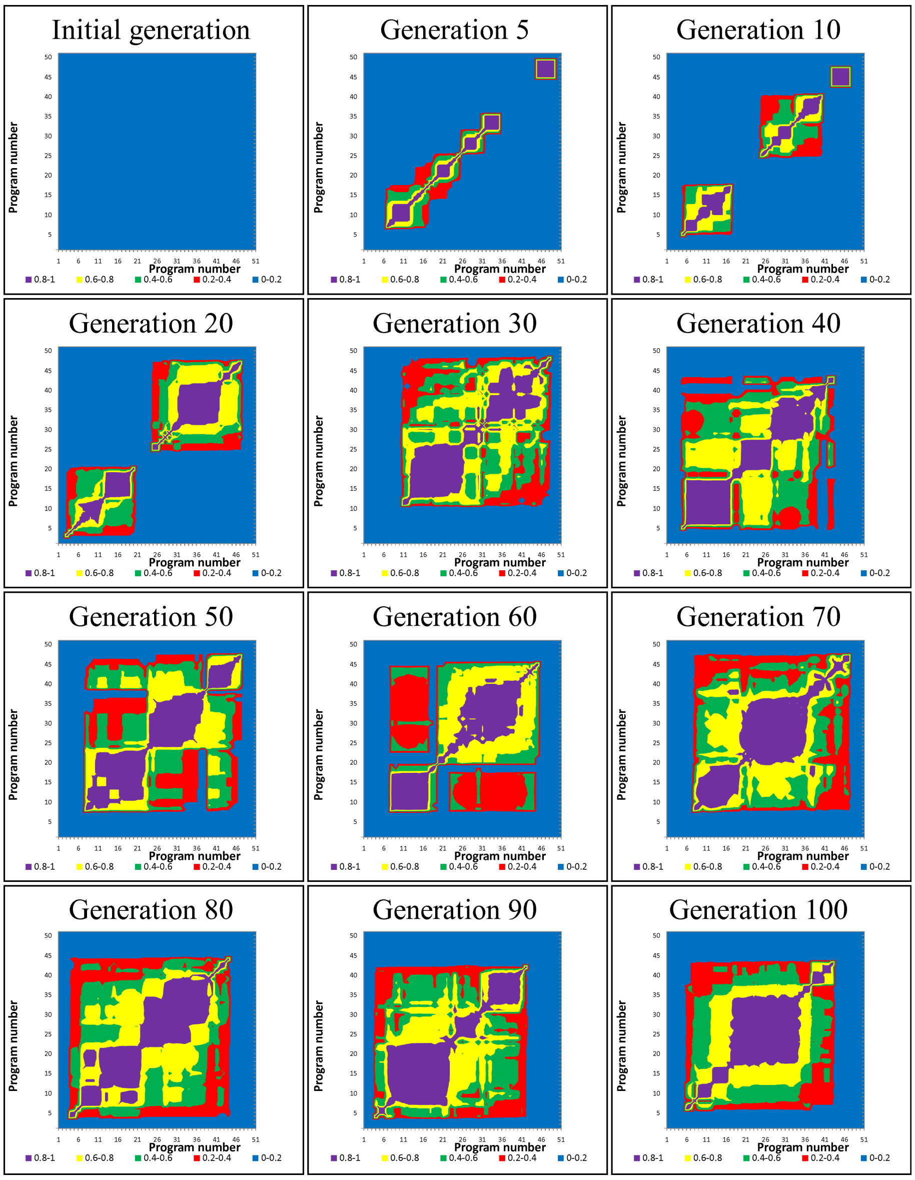

Figure 7.11: Program similarities between each pair of programs at various stages of one run with 51 programs for rooted-ordered-fragments. $z_s = 5, \min_P = 5$

and no similarity of any of these programs with programs outside the cluster.

- At 20 generations there is still no mixing of the two main clusters but these clusters have grown in size and have a core of very similar programs with a periphery of less similar programs.
- At 30 generations the two clusters have combined and diluted with no very similar programs but many combinations of less similar programs.
- Further generations again strengthen the boundaries of program clusters with generation 60 presenting two distinct groups of programs: one with a hard edge in which all programs are very similar, and one with a softer edge in which there are some less similar programs.
- After 100 generations, there is a dominant cluster with a core group of very similar programs and a periphery of less similar programs.

Figure 7.12 shows similar plots to figure 7.11 but for the "seed 3" evolution which section 7.5 found to have converged at generation 15.

Indeed, even by generation 5 the evolution looks different to the case for the "seed 1" evolution with one dominant soft cluster of programs and only one other smaller, hard-edged cluster of 9 programs. A large, soft-edged cluster which may be assumed to derive from this original cluster dominates the evolution right through to generation 100. While at times a small cluster emerges, it remains integrated into the single dominant cluster.

7.6.1 Non-determinism of the program ordering

The algorithm used to determine the order of programs for the figures is non-deterministic, and we would expect that two figures differing only by their initial random seed would differ in their overall look. Figures 7.13, 7.14,

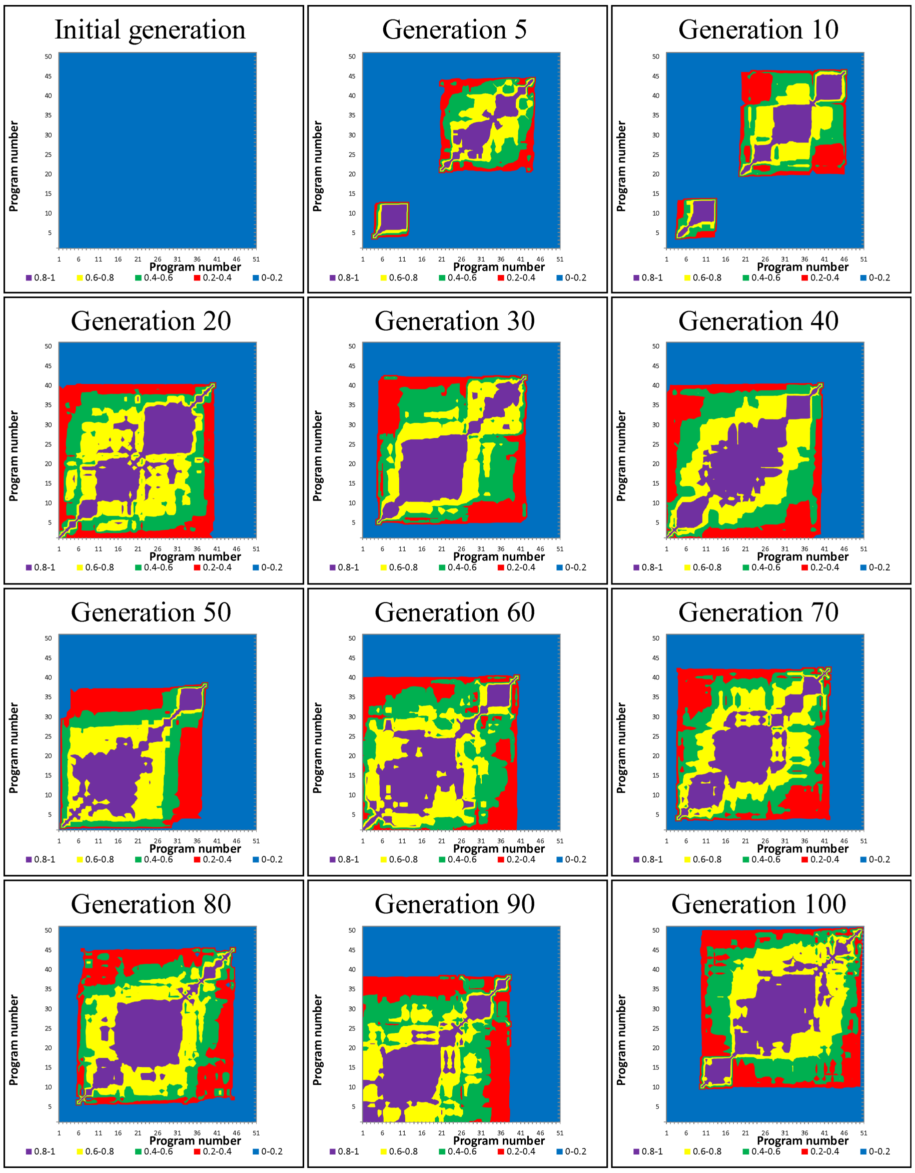

Figure 7.12: Program similarities between each pair of programs at various stages of one run with 51 programs for rooted-ordered-fragments. $z_s = 5, \min_P = 5$

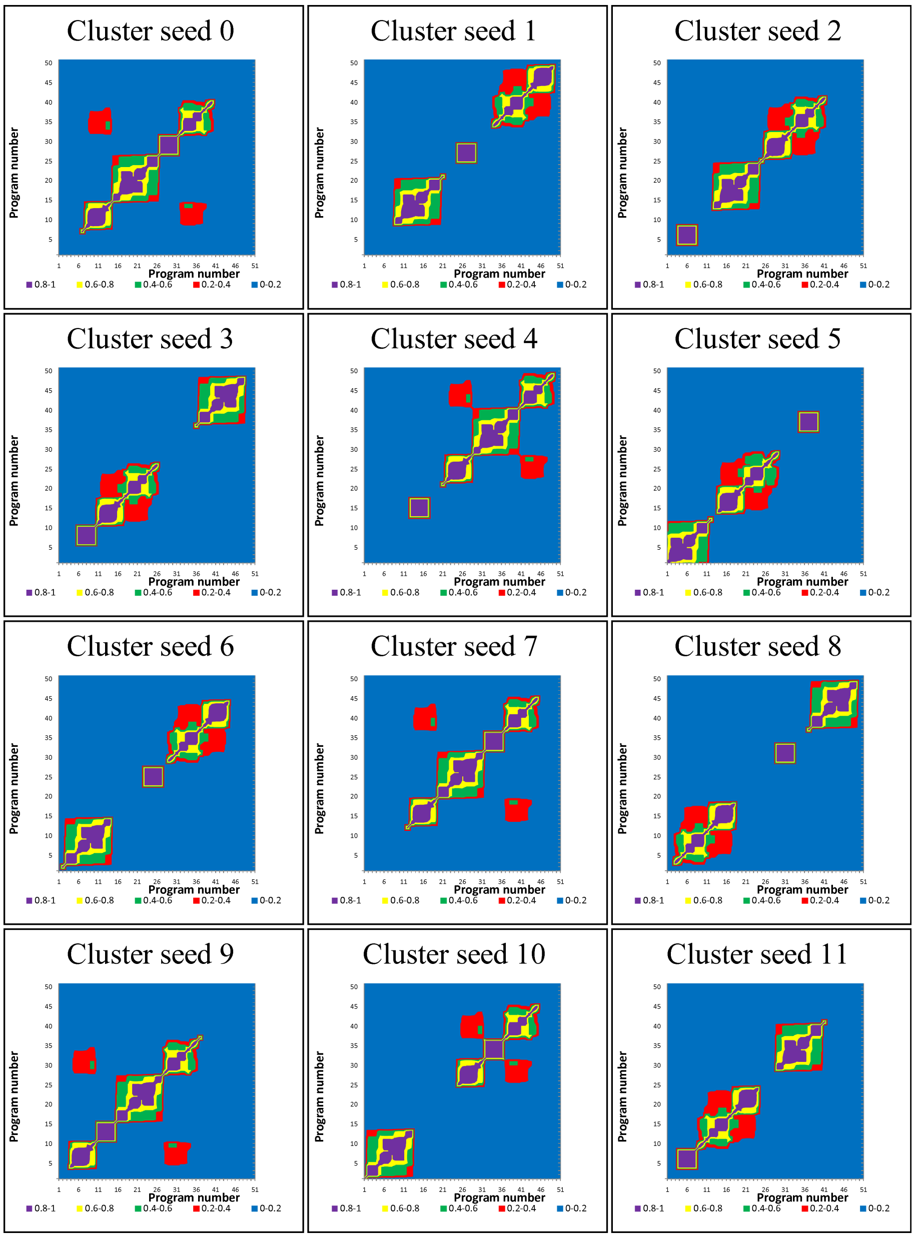

Figure 7.13: Program similarities between each pair of programs at generation 10 with 51 programs for rooted-ordered-fragments. $z_s = 5, \min_P = 5$. Plots differ only by the initial random seed used by the algorithm obtaining a program ordering.

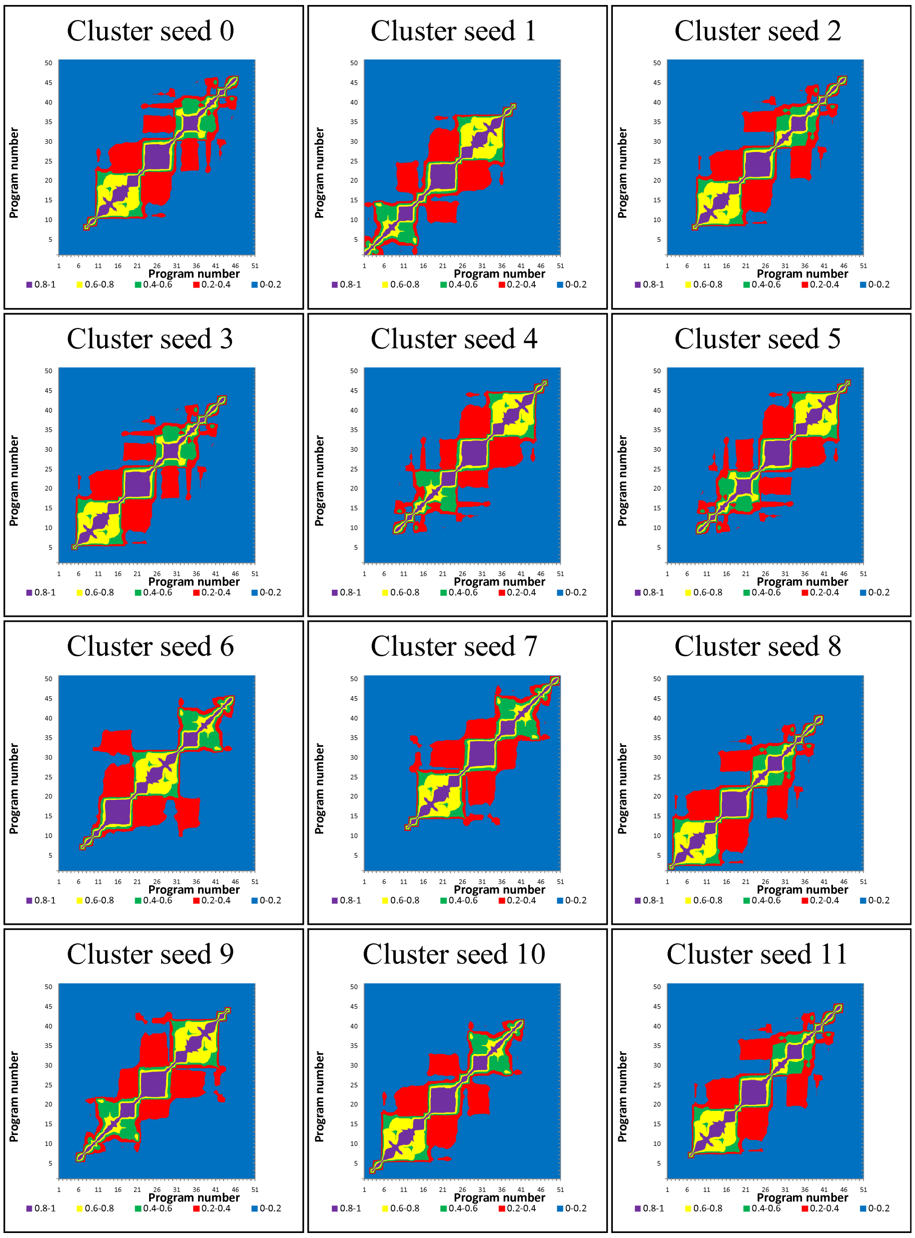

Figure 7.14: Program similarities between each pair of programs at generation 10 with 51 programs for rooted-ordered-fragments. $z_s = 5, \min_P = 5$. Plots differ only by the initial random seed used by the algorithm obtaining a program ordering.

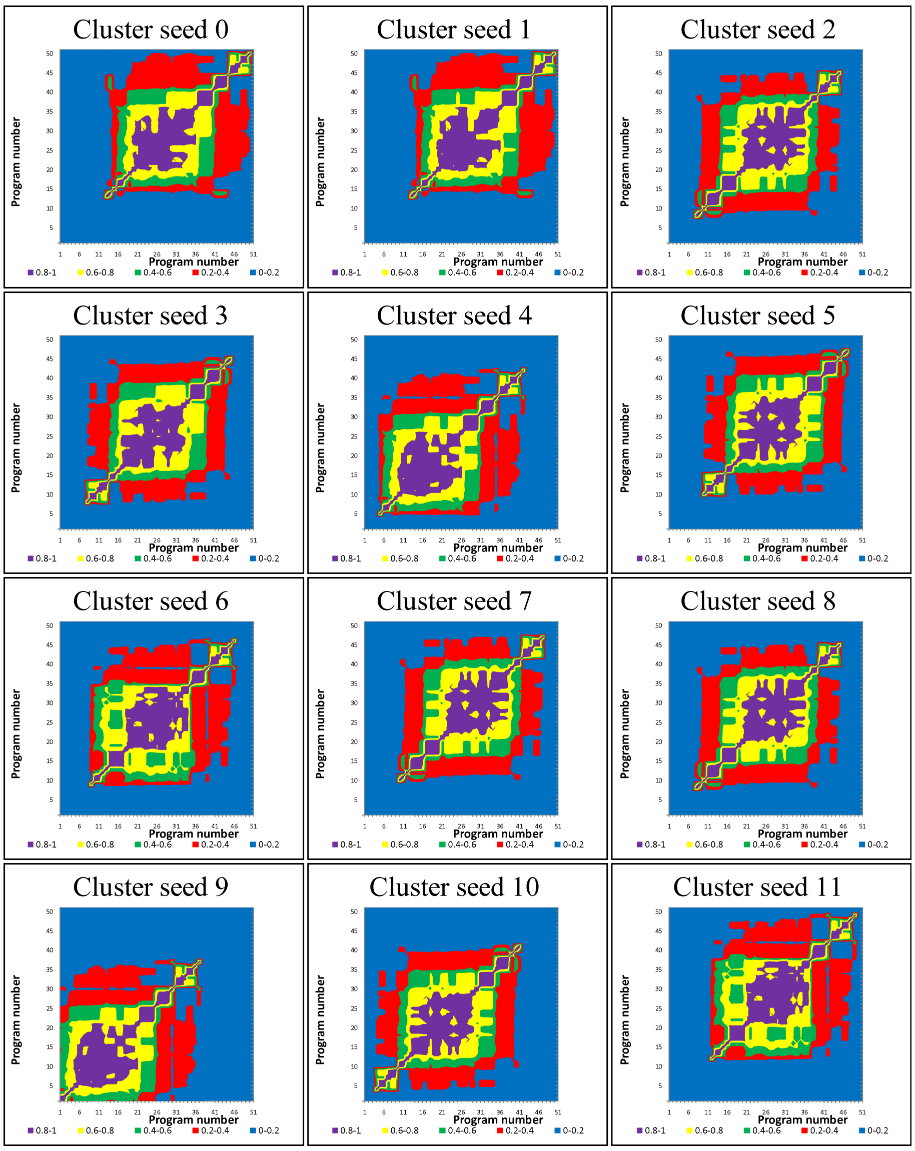

Figure 7.15: Program similarities between each pair of programs at generation 10 with 51 programs for rooted-ordered-fragments. $z_s = 5, \min_P = 5$. Plots differ only by the initial random seed used by the algorithm obtaining a program ordering.

and 7.15 show the effect of the non-determinism on the overall look of the figures. In each figure the plots differ only by the initial random seed used by the ordering algorithm. The figures themselves differ by generation, with figure 7.13 being generation 10 (see figure 7.11), figure 7.14 being generation 40, and figure 7.15 being generation 100.

As may be expected, while the plots of each figure differ in the placement of clusters, the nature of the inner, purple, clusters (that is, number of programs, hard or soft edge, internal speckling of lower similarity pairs, and so on) remains intact between plots.

The outer, red, fringe in more variable, with some plots showing different patterns. An example is the red fringe of "cluster seed 0" of figure 7.13 which looks different to that of "cluster seed 1" of the same figure. The program ordering algorithm is good at identifying the programs belonging to a cluster, but not at placing the cluster. The red fringe to shows similarities of pairs of programs that fall into different clusters, and therefore depends on cluster placement more than the purple which shows similarities of programs in the same cluster.

7.7 Clustering programs

This section gives an final example of an interesting analysis possible using the new method.

While the previous section effectively shows the clusters of programs in given populations, the graphs of this section show these clusters in a more explicit way, also bypassing the requirement to search for a *good* order of the programs.

The graphs of this section show bar graphs of two series:

- An orange series gives the size of the largest schema occurring in a given program and some minimum number of programs in total.

  Each orange bar marks a distinct program.

- A black series, divides adjacent clusters.

  For any given size of schema z on the y-axis, any two programs associate with two orange bars on the plot. If the two programs share a schema of size z such that the schema is in some minimum number of programs in total, then all orange and black bars between the programs are at least size z. If for two sets of programs no program in the first set shares such a schema with any program in the second set then the programs of each of these sets will be divided by a black bar of size less than z.

  Thus the black bars have a height such that they join the orange bars into clusters and the height of a black bar is the maximum size of schema for which all programs of the cluster share a schema of that size with some other program of the cluster. Were the population divided into two groups of programs which were similar to only programs in the same group, then there would be two peaks in the graph with a small sized black line dividing them

Similarly to the plots of the last section, the order of programs on the x axis is crucial to the plots of this section. But the method of this section relies less on random numbers to determine the order of programs. The method used ensures a grouping of programs such that neighbouring programs share large schemata as far as possible.

The overall effect is for clean and descriptive plots directly showing the clusters of similar programs in any given population, using schemata of a given form that are in some minimum number of programs.

Method

This method forms clusters based on the largest schema size occurring in pairs of programs. Thus on top of requiring the size of the largest schema $z_s(p)$ in any one program p for the orange series, the analysis requires the size of the largest schema $z_s(p_1, p_2)$ in each pair of programs p_1, p_2.

To find $z_s(p)$ this analysis uses a slightly different approach than that used in section 7.4, allowing us to filter the schemata to only those occurring in P. Using the analysis algorithms of chapter 4 the analysis is over schemata with the following two defining functions:

- AnSc$(P, z_s) = z_s$ if $p \in P$ and $|P| \geq \min_P$, or 0 otherwise
- Agg$(A, A', c) = \max(A, A')$

The aggregation function Agg returns the maximum value of the analysis function AnSc. This analysis function returns the order of the analyzed schema if it occurs in p and at least $\min_P - 1$ other programs, or returns a value of no effect otherwise. Thus the analysis provides the order of the largest such schema.

This core analysis is run for each program p in the input population.

A similar method finds $z_s(p_1, p_2)$ for each pair of programs. Using the analysis algorithms of chapter 4 the analysis is again over schemata with the following two defining functions:

- AnSc$(P, z_s) = z_s$ if $p_1 \in P$ and $p_2 \in P$ and $|P| \geq \min_P$, or 0 otherwise
- Agg$(A, A', c) = \max(A, A')$

The aggregation function Agg again returns the maximum value of the analysis function AnSc. In this case the analysis function returns the order of the analyzed schema if it occurs in p_1 and p_2 and at least $\min_P - 2$ other programs, or returns a value of no effect otherwise. Thus the analysis provides the order of the largest such schema.

This core analysis is run for each pair of programs p_1, p_2 in the input population.

Thus defining P_0 as the population and R as the set of maximal pairs used for the analysis, the complexity of the analysis is near $O(|P_0|^2|R|)$ since the analysis producing $z_s(p_1, p_2)$ has complexity $O(|R|)$ and this core analysis is run $|P_0|^2$ times.

The method to find the order of programs is as follows:

- The procedure proceeds from small schema sizes to large schema sizes by progressively dividing a base cluster, initially the whole population, into smaller clusters by finding the smallest possible subsets of a given cluster of programs where each pair of programs sharing a schema of a given size are placed in the same subset.
- For each schema size z, the method further divides the clusters of programs for size $z - 1$, or the whole population if $z = 0$, based on which programs share a schema of size z.
- The method orders the z sub-clusters of each $z - 1$ cluster from many programs on the left to few programs on the right.
- The process then proceeds to the next size until the clusters are singletons or no shared schemata are found.

Thus other than the ambiguous ordering of equal-sized clusters, the process is deterministic.

Results

Figure 7.16 shows twelve plots from various generations of the "seed 1" trial also used in the previous section. Each of the twelve plots shows the cluster graph of the population at that generation for rooted-ordered-fragment schemata occurring in at least 5 programs. The figure also clearly echoes the results of the previous section for the "seed 1" evolution: At first there are few shared schemata of any size but by generation 5 there are several distinct competing clusters. Later two main distinct clusters form, which by generation 30 have joined into one. These plots show with more certainty the sizes of the clusters and show clearly the sizes of the shared schemata in each cluster. For instance, the small third cluster seen at generation 10 of figure 7.11 has programs sharing only small schemata. The figure clearly identifies what happened at generation 64 of figure 7.6

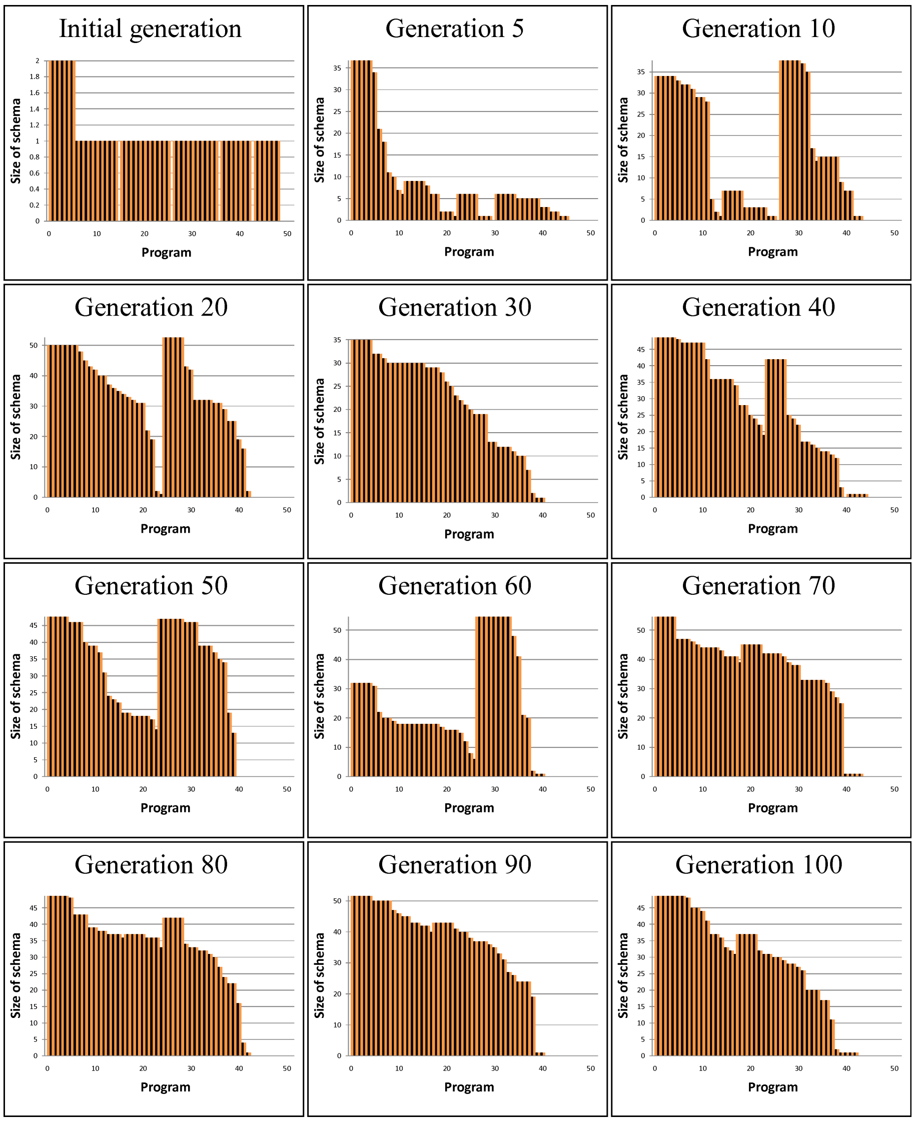

Figure 7.16: Program clusters for each 10thgeneration of the "seed 1" evolution using rooted-ordered-fragment schemata occurring in at least 5 programs.

of section 7.5 which saw the average size of schema for many sizes of program subset increase five-fold in one or two generations: a cluster that emerged around generation 40 and strengthened to cover half of the programs by generation 50 gained in number but decreased in program size. By generation 60 the cluster has many programs but with a small average size. By generation 70 the cluster has been replaced by the progeny of the smaller in number cluster containing significantly larger programs. From the previous result it may be assumed that this transition occurred at generation 64. Later generations show convergence with few clusters of appreciable size competing with the dominant cluster.

Figure 7.17 shows twelve similar plots for the ordered-subtree form of schema. The plots are distinctive in how they compare to those of the previous figure. For early generations they show quite a different picture to the rooted-ordered-fragments. Even by generation 5 many programs share sizable subtrees and nearly all programs share some subtree of size 3. The plots for generations 5 and 10 are not as broken up into distinct clusters as was the case for rooted-ordered-fragments. The plot for the initial generation may present a clue why: where few of the initial random programs shared any rooted-ordered-fragment of more than one node, most shared some subtree with two or three nodes. This trend carries over into generation 5 providing most programs with a base-line subtree similarity. Interestingly, after about generation 30 the plots for the two forms of schema are very similar. Perhaps by this generation the baseline similarity between random programs and its difference between the two forms of schema, has become less influential.

Figure 7.18 shows twelve plots from various generations of the “seed 3” trial which was found in the previous section to have converged early in the run. Each of the twelve plots shows the cluster graph of the population at that generation for rooted-ordered-fragment schemata occurring in at least 5 programs. Again, the figure echoes the results of the previous section. It shows that even by generation 5 a single large cluster of

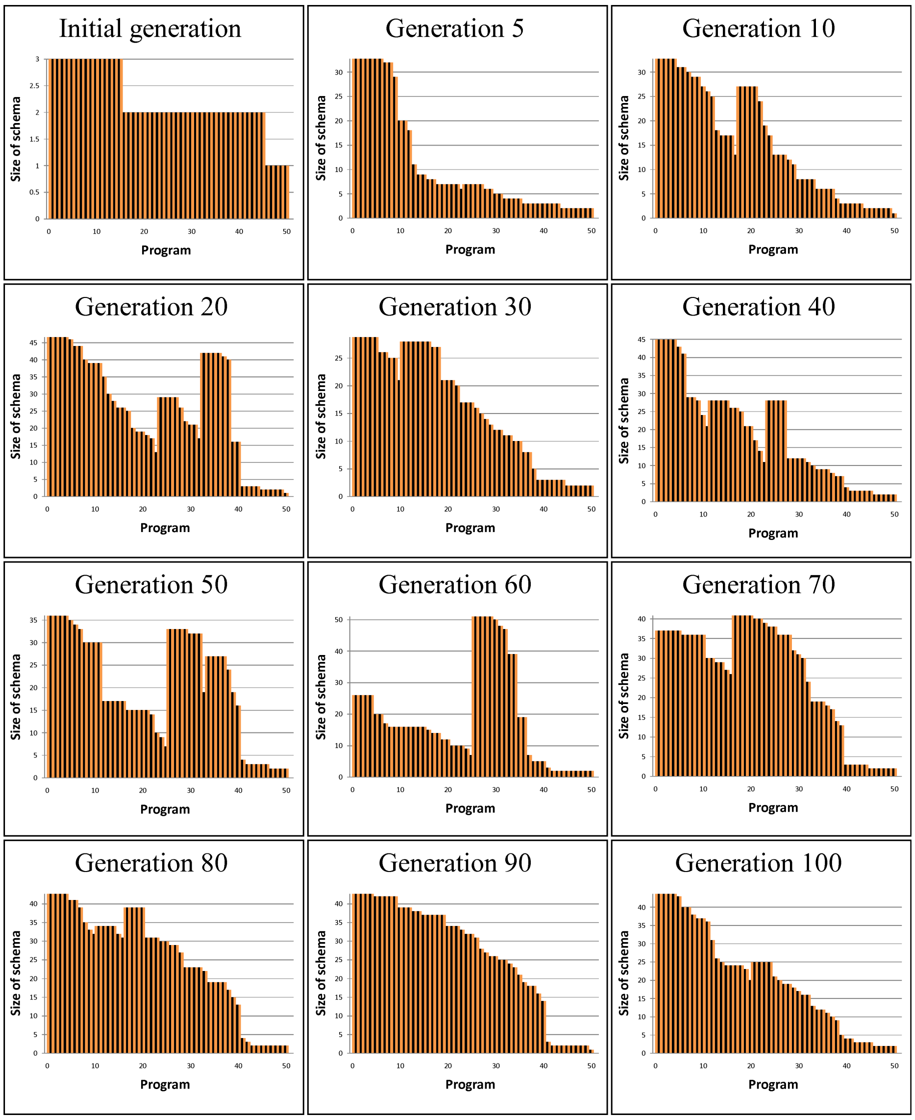

Figure 7.17: Program clusters for each 10thgeneration of the "seed 1" evolution using ordered-subtree schemata occurring in at least 5 programs.

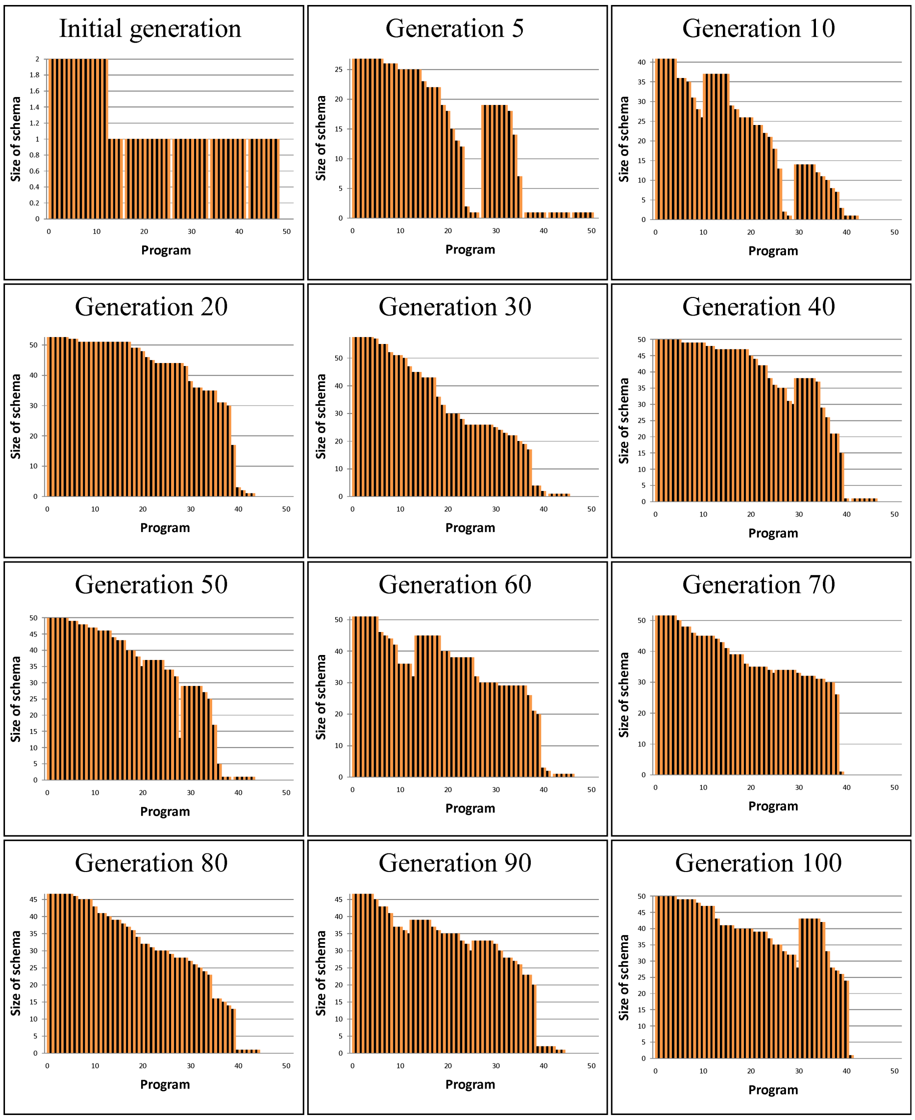

Figure 7.18: Program clusters for each 10th generation of the "seed 3" evolution using rooted-ordered-fragment schemata occurring in at least 5 programs.

large programs has emerged. The other competing cluster dwindles in size and by generation 20 the single dominant cluster has established itself and there is little change for the rest of the run. At times a competing cluster emerges but they are always reabsorbed into the main cluster.

The one major parameter to the analysis, other than form of schema, is the minimum number of programs a schema must occur in to be included in the cluster analysis. Figure 7.19 shows six plots with various settings for this parameter. As may be expected, the trivial case with no minimum number of programs shows a noisy graph with many small clusters. Increasing the number of required programs smoothes the graph and a large enough setting removes all smaller clusters.

Interestingly, while the leftmost major cluster for the "no limit" case has 20 programs, the leftmost major cluster for the "5 programs" case has 23 programs. This shows that the difference is not a simple smoothing of the graph.

7.7.1 Non-determinism of the program ordering

As for the method of the previous section, a non-deterministic algorithm is used to order the programs in each of the figures of this section, but where the algorithm of the previous section relies heavily on the random number generator, the algorithm used here uses it only to determine the order of two equal size clusters.

To test the effect of the non-determinism, the settings used to produce each of the twelve plots of figure 7.17 were used to produce twelve similar plots which differed only by the initial random number generator (RNG) seed used by the clustering algorithm. With careful visual inspection no difference was noticed between any two plots that differed only by RNG seed.

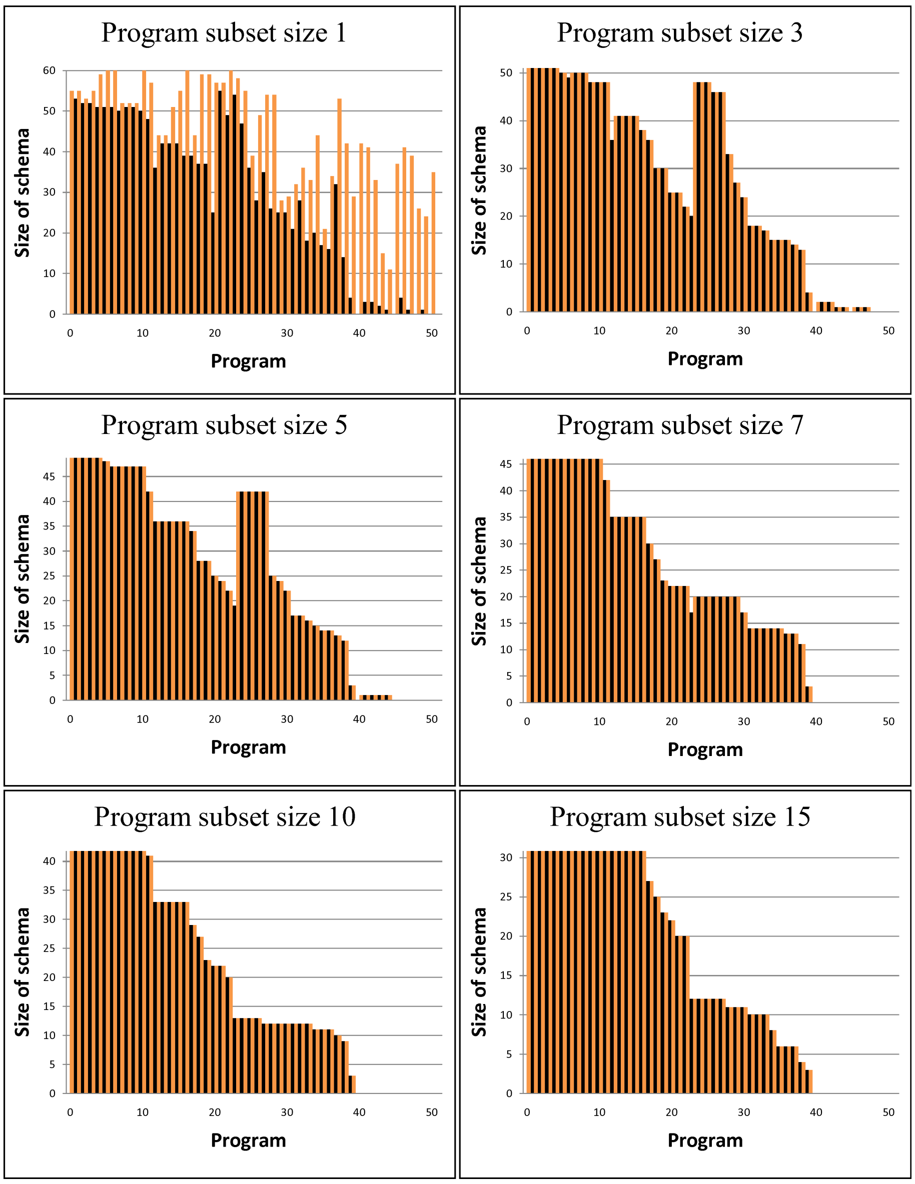

Figure 7.19: Program clusters for the 40th generation of the “seed 1” evolution using rooted-ordered-fragment schemata occurring in at least k programs for various values of k.

7.8 Discussion

This chapter presented a few sample analyses using the new method. The chapter presents five types of analysis, including three basic analyses of statistics on the schemata occurring in programs of a given population and two more complex analyses. The various analyses presented some results that were expected and some results that were quite unexpected:

- Section 7.3 presented the total number of schemata of each size occurring in any program of the given population.

  As expected, the analysis found few large or small schemata and many medium schemata. Each curve was slightly shifted toward larger schemata. The form of schema was a major factor in the number of schemata with orders of magnitude more schemata of a more expressive form like hyperschemata than of a less expressive form like fragments.

- Section 7.4 presented the size of the largest schema occurring in any k programs of a given population for various values of k.

  The expected trend was a slow growth in the size of the largest schemata in k programs for most sizes k. In contrast, the observed trend is a very fast rise in this statistic for all sizes k and relatively slow growth after about generation 20. By generation 50 the largest rooted-ordered-hyperschema in any 21 programs was unexpectedly large at about 20 nodes in size, not including *don't-care* nodes. The largest ordered-subtree was slightly smaller. Despite the trend of a fast rise at the start of the run, then steady-state for the remainder of the run, the plots do not suggest convergence since the distribution does not suggest that all programs are excessively similar.

- Section 7.4 also presented the size of the largest schema occurring in the r^{th}ranked programs over any k generations for various values of r and k.

This analysis used *generational populations*, where rather than analysing programs of different rank and common generation, analysis is on programs of common rank and different generation. As may be expected, fit programs shared significantly larger fragments than unfit programs. Interestingly, there was at least one trial out of the fifty randomized trials of figure 7.5 in which this was not the case; in these runs even the very most unfit programs of this trial sharing large fragments over many generations.

- Section 7.5 presented the average, over all program subsets of size k, of the largest matched schema.

  While the previous results were medians over 50 randomized trials, this section presented results for individual trials and thus showed in more detail the events and trends of a single run. The analysis showed very clear trends and events in some evolutions, including the clear trend that at about generation 18 the "seed 3" trial had relatively large shared schemata and showed little change in the size of these schemata for the rest of the run. This would seem clear indication that this trial converged early.

  The average largest matched schema for trial "seed 1" showed a very interesting event at generation 64 with the statistic jumping up to 4-fold in that one generation for rooted-hyperschemata and $k = 3$. This sudden increase is sustained later in the run. It is unclear from this graph why such an event would occur, although later analysis in sections 7.6 and 7.7 gave good evidence it was one tight-knit cluster of large programs succeeding a larger cluster of smaller programs.

- Section 7.6 presented similarities between programs, based on number of shared schemata.

  This section presented a matrix of similarities between programs. Each similarity was defined as the number of schemata shared by both programs, divided by the total number of schemata in either program. This very visual analysis gave a very fine grained view into a single run of GP by displaying how clusters of programs grew, shrank and merged through evolution. The plots of the section show another view of the event at generation 64 of the "seed 1" trial: one cluster of programs spectacularly gave way to another. They also backed up the results of section 7.5 by showing that the "seed 3" trial had effectively converged by generation 15, though with less precise timing.

- Section 7.7 presented clusters of programs, based on size of shared schemata.

  A cluster of programs was defined as a *connected* set of programs, that is, any two programs sharing a schema of the required size were placed in the same cluster. Thus this section presented an alternate view of the clusters of section 7.6, showing the size of shared schemata on a bar-plot. The same clusters were found in the two analyses but the plots of section 7.7 showed more clearly schema size as well as the cluster size shown in the plots of section 7.6.

This chapter shows several interesting analyses of GP using the new method, both of the general trend of a GP system over multiple runs, in sections 7.3 and 7.4, and of the trend of the GP system in a single evolution, in sections 7.5, 7.6 and 7.7.

The new method, though slow to use on an evolution, presents extremely fine-grained analysis of the shared schemata in the population in a way not at all practical on this scale by previous methods. This chapter presented only a small fraction of the analyses possible by using the method and particularly focussed on convergence of runs and clusters of programs. Other analyses could detect bloat or analyze wider properties of GP like the ability of crossover to successfully combine fit building blocks into fitter programs. In addition, it would be very interesting to look at the relationships between the schema statistics presented here and the fitness of populations and programs.

But even the basic analyses presented here are very interesting. The simple program distance used gives an intriguing view into evolution. One may imagine an animation showing the *macro* process of evolution in a way resembling the *micro* process of Conway's game of life [28]. The clustering of section 7.7 shows very plainly how many programs share how large a schema and would also be suited to animation.

The generation populations of section 7.4 nicely show the difference between fit and unfit programs, which is not apparent in the analyses of standard populations, which could be termed *rank populations*. Another avenue not yet explored is *seed-populations* where the population passed to the algorithm is made up of programs of the same rank and from the same generation but from different trials. Such an analysis would make independent the factors of rank and generation; in both rank and generational populations one or other of these factors is forced to play a role in the analysis by being different for each program in the population. All types of analysis presented in this chapter could, like clustering or average schema size, be made to work with generational populations and seed populations.

7.9 Chapter summary

The goal of this chapter was to present a few sample analyses using the new method, deriving results of interest to the GP community from a few runs of GP. The chapter aimed to show the efficacy of the new method, if only with a small number of relatively simple analyses.

Though most of the analyses presented are very difficult to replicate using previous methods, the new method performs these analyses without great difficulty even when analyzing rooted-hyperschemata in real-sized populations of 101 60-node programs. While there are many ways that these analyses may be extended and this chapter seeks only to provide a taste of what the new method can do, even the results of this chapter give interesting insights into: the potential of a run to converge, the extent of that convergence, the jostling for position of clusters of programs in evolution and the effect of these clusters on shared schemata.

Chapter 8

Conclusions

This chapter presents the conclusions of this thesis with reference to the goals of section 1.2.

This thesis had the following goal:

> produce a new method for the analysis of Genetic Programming (GP) by empirically analysing the schemata shared between programs

The new method should be able to analyze various forms of schema and be applicable to medium sized populations and programs. Another stipulation was that the method be deterministic in its results, not bending to the whims of chance and that the enabled analyses be potentially useful. The method that this thesis has presented achieves all these aims.

The new method has defined a new form of schema and a new language for forms of schema in order to be applicable over a range of forms of schema. The *match-tree form of schema* is a general form of schema which borrows from Object-Oriented programming to lend behaviour to each schema node, easily generalizing over all relevant surveyed forms of schema in the literature. The *match-tree form of schema language* provides a standardized way to refer to specializations of this extremely general form of schema and allows these specializations to be expressed. All relevant commonly used forms of GP schema and many as yet undiscovered forms, may be expressed as *match-tree forms*. The match-tree form of schema can be extended by adding *label-match* and *child-match* functions to the vocabulary used by the system. The experiments of this thesis have laid out a set of basic label-match and child-match functions.

The new method has been shown in the previous chapters to effectively analyze populations of GP genetic programs even up to 500 60-node programs, easily exceeding its target. Only a small subset of the forms expressible in the match-tree form of schema language are implemented in the system: the *conjunctive match-tree forms* which are ordered rooted forms of schema and *de-rooted conjunctive forms of schema* which are ordered forms of schema. Generalizing to non-rooted forms of schema significantly decreased performance and increased the chances of exceeding resource limits. No efficient algorithm was found to allow further generalization to non-conjunctive forms of schema, for instance *unordered-fragments*.

The new method is indeed deterministic, producing the same output for the same input. The previous chapter has presented some analyses enabled by the method which are likely to be of interest to the GP community. Future work promises to present many more.

8.1 Specific conclusions

The main goal of this thesis was to

> provide a method to perform efficient analysis on all the schemata occurring in an input set of genetic programs

This thesis achieved this goal by designing and building a system performing such analysis.

In experiments the analysis system proved efficient even at sizable scales of evolution. The new method groups schemata such that large volumes of similarly propertied schemata are summed up by a representative. In doing so the system can perform analyses on medium-sized populations of genetic programs which were previously impractical or impossible.

In particular, the new method meets all of the several qualifications to the main goal that were given in section 1.2.

- *Some of the analyses enabled by the method will be useful to researchers and the research community.*

  The new method presented in this thesis provides a base for a great many analyses. While chapter 7 gives the results of several analyses on GP populations using the new method, there are a great many more which are possible and can be explored in future work.

  The analyses of chapter 7 include previously impossible or difficult analyses which give insight into: convergence, the clustering of programs during evolution, and the difference in behaviour between different forms of schema as well as providing a fine-grained view into the process of GP evolution itself. Each of these areas is potentially of great interest to the GP community.

  To indentify and study convergence, experiments in section 7.4 looked at the *highest-order schema occurring in any set of k programs*. Using the new method, this statistic is found by performing an analysis on maximal program subsets that is equivalent to an analysis finding the size of the largest schema in each set of k programs from the population P_0. This useful analysis is made possible by grouping the huge set of schemata analyzed into a manageable set of representatives, giving orders of magnitude better efficiency over a naive analysis. Another similar analysis found the average size, over all sets of programs P of size k, of the largest schema occurring in all programs of P.

  The new method showed that in the runs of evolution studied total convergence did not occur even after 100 generations. But after only

20 generations, in the median of 50 randomized trials, there was already a 10-node fragment shared by almost half of the 101 programs of the population. After this steep increase of the size of the largest shared fragment the remainder of the run shows a steady but slow increase in the size of the largest fragment in k programs for most values of k. The largest rooted-ordered-hyperschemata and rooted-partly-ordered-fragments followed this similar trend of a fast initial increase followed by a slower rise for the remainder of the run.

In addition, the results of section 7.6 give insight into convergence by presenting a schema related distance between each pair of programs, at various points during the evolution. Figure 7.11 shows these distances as they evolve in one evolution. The clear clusters of programs seen in the figure are in a constant state of flux, and there is no apparent stagnation late in the run. But for a different evolution shown by figure 7.12 the difference is marked, with clear convergence little change in the arrangement of the clusters of programs after about generation 20.

- *The method provides an analysis on all schemata of the given form in the given programs. The result is not dependent on random numbers or selective sampling.*

  The new method defined by this thesis uses a deterministic algorithm, not relying on selective sampling and exactly emulates certain "global" analyses which if done in a "naive" way are typically impractical due to the large numbers of schemata. Section 4.4 gives methods by which a naive analysis which visits each schema in turn and analyses the programs the schema is in may be refactored into an equivalent and achievable analysis of *representative pairs*. The equivalent analysis will reliably return the same result as the naive analysis.

  By removing the need for selective sampling of schemata, this thesis removes the inevitable sampling bias inherent in sampling from a collection of trees. The result is an algorithm which has all the benefits of the global naive analysis but has practical and at times very good complexity even when analyzing the fragments in populations of 400 60-node programs.

- *The method provides analysis on complex and interesting forms of schemata*

  By using the "form of schema language" the new method may analyze a very wide range of forms of schema in one implementation. Included in the forms used for the experiments of chapters 6 and 7 are both rooted and non-rooted forms of schema, including subtrees, fragments, hyperschemata and the new form: *partly-ordered-subtrees* which has only limited ordering of function arguments. Previous literature has seldom dealt with unordered schemata and hyperschemata are typically seen as a highly general and highly interesting form of schema.

- *The new method should be able to analyze all schemata in 100 seven-deep programs. The analysis should be available on real world, non-toy problems*

  The experiments of chapter 6 included runs of the new method on populations of up to 501 60-node programs or up to 51 80-node programs, although at these scales of evolution many runs of the analysis exceeded resource limits. This thesis was able, even using a fairly unoptimized system, to reliably perform useful analysis on rooted-ordered or rooted-partly-ordered forms of schemata in populations of 300 60-node programs. Given that a seven-deep binary program has a maximum of 63 nodes, and that few typical populations of such programs would have an average program size of over 60 nodes, the new method exceeds the goal's requirement and is applicable to real-world, non-toy GP systems.

8.2 Other observations

Achieving the main goal involved achieving subsidiary goals: *Provide a language for forms of GP schema unifying common forms of GP schema.*

- There are many forms of schema in the literature and this thesis provides an important unifying contribution. The *match-tree form of schema language* is a language by which forms of schema may be represented. This is similar to forms of schema defining a way by which individual schemata may be represented. This language is expressive enough to represent all relevant existing forms of schema in the literature.

  The match-tree form of schema language derives its considerable expressivity from this thesis' definition of the *match-tree form of schema* with any form of schema represented by the match-tree form of schema language in fact defining a subset of the overall match-tree form of schema. This overall form of schema borrows a similar idea from object-oriented-programming; each node in a match-tree schema encapsulates behaviour allowing it to decide intelligently which program nodes it does or does not occur in. The functions providing this behaviour provide arbitrarily complex matching behaviour.

  The definition of a representation for forms of schema means a more rigorous way to define schemata. Too often literature defining a form of schema sums-up the form in few words and leaves out crucial details. Such details include whether the form is "ordered" or whether additional program children negate a match. In declaring the form of schema using the match-tree form of schema language, all such details are also declared.

- *Build the analysis method to be compatible with this language for forms of schema.*

  The system used for this thesis' experiments implements two broad classes of match-tree forms: conjunctive match-tree forms of schema and de-rooted conjunctive match-tree forms of schema. Though only allowing analysis of a small proportion of the forms expressible by the match-tree form of schema language, these two classes are enough to represent most forms of schema used in past GP literature with the notable exception of the seldom used "unordered" forms.

An analysis system which implements the match-tree form of schema language may be provided with a form of schema as an argument with no need to reimplement to specialize to a different form of schema. Indeed, the system used for this thesis' experiments takes as an argument the string representation of a form of schema, returning results valid for the represented form.

Characterize the method by using it for GP analysis in a range of situations.

- Since no close theoretical bounds on the complexity of the new method were found, there was a need to test the new method in practice. Chapter 6 put the new method through its paces over tens of thousands of parameter combinations, testing the time and space complexity in practice with default parameters and while varying parameters including generation number, variant of the algorithm, population size, program size and function arity.

  As expected, experiments found high correlation between the number of representative pairs, the grouping object which is the major artefact produced by the new method, and its running time and memory footprint both in RAM and on disk, with close to linear correlation in each case.

  The new method was found to slow as the run progressed and the number of schema shared by programs of the population increased. But the complexity of the new method was found to plateau after a few generations with sustainable complexity at least until generation 100.

> Agreeing with the deterministic nature of the new method, each variant of its core algorithm was found to produce the exact same output. Some variants were found to be significantly faster than others. The variant "ProMP" was the fastest overall.
>
> The method allows large population sizes more easily than large program sizes. It is approximately quadratic on population size and quartic on program size in nodes. Experiments found that function arity had a considerable effect on the complexity of the method with larger function arities leading to longer running time and larger memory footprint. A certain proportion of the runs would typically fail by exceeding memory limits, even when these limits were set very high. The majority of runs stayed well within reasonable limits.

The secondary goal of this thesis was to implement the method in C++ as a tool for analysis of GP and the thesis achieved this goal by the `PhDVgp` package which is available from the author under an open source licence[1]. The tool currently implements the subset of the new method required for experiments but is under further development to provide additional functionality.

8.3 Discussion of the uses of this research

I hope that this research provides the community with a very useful analysis tool, but what of its eventual use?

This analysis tool provides fine-grained information to any GP researcher dealing in small to medium scales of evolution, who wishes to "see" what their GP system is doing during evolution. The plots of the previous chapter show that very detailed information is present about the state of a run, and how the run is progressing. The researcher could be expected, for

[1]PhDVgp is currently hosted at sourceforge: *http://sourceforge.net/projects/phdvgp/*.

instance, to adjust the parameters of the system until most runs don't converge, and there are typically two or three soft edge clusters in the population.

Or should they be hard edge clusters? Or a mix? I expect that this tool could be used to look for the combination of clusters of programs which gives best expected performance. In evolution, the tool could be used to vet evolutions which are unpromising (for example, the "seed 5" evolution of the previous chapter could have been identified as converged quite early in the run), or to adjust parameters on the fly such that the run continues to be promising.

This thesis' experiments do not exploit a powerful property of the analysis method. To all the analyses, each schema was a number, however, the method actually identifies the exact largest schema (as a match-tree schema) for each set of programs in the population, as well as the set of programs that the schema occurs in. So most of the figures of the previous chapter could have been annotated with the string representations of the schemata they referred to (for instance, each cluster could be labelled with its prominent schema). Schemata have long been used as subroutines (for instance, as ADFs), and this method may find use as an automatic way to find commonly used subroutines that are in fit programs.

Are there building blocks in GP? If so, they are likely to be representable as schemata, and would have peculiar characteristics. I expect that this tool will not only shed considerable light on the behaviour of building blocks in evolution through plots like those in the previous chapter (are they the hard edge clusters?), but will be able to name suspected building blocks as match-tree schemata. It would be very interesting to inject such a suspected building block into evolutions other than its originator to test its range.

A highly optimized version of the tool could be run on the union of the populations of many generations of the same run of evolution. The result would be not only the common schemata between programs, but the path of these schemata through evolution. Combined with a graph showing the hereditary of each program, this would provide copious data on the propagation of schemata through evolution, guiding the search for better GP.

8.4 Directions for future work

Now that this thesis is complete, we look to the future. The following list gives a few ideas on how this research could be used and extended:

- More in depth analysis – the results of chapter 7 could be expanded to, for instance, show the clusters of the population each generation instead of each 10th generation. Some of the details are lost at the wider time-scale.

- Animated visualization of the results – some of the results would be ideally suited to animation, showing how the trends change by generation.

- Generation and seed populations – while the results of section 7.4 take a cursory glance at generation populations, these populations deserve a more thorough investigation. Seed populations, that is populations where individuals have common rank and generation and differ only by seed, may hold the key to finding objectively good building blocks and will be an immediate line of inquiry.

- More forms of schema – the forms of schema used in experiments do not use some of the more advanced features of the match-tree form of schema language, for instance, forms could use the partly ordered `pind` child-match function only for unordered functions like addition, and not for ordered functions like `if`. Other child-match functions are also possible.

- Work on generalizing the algorithms to take non-conjunctive forms of schema.
- In experiments, the non-rooted forms of schema performed badly and this is likely to be able to be corrected.
- Implement filters to set minimum and maximum sizes on the program-subset and schemata of interest – such filters would be easy to implement and would make the algorithms far more efficient.
- Use the method as part of evolution:
  - To detect when a run will go bad – experiments of chapter 7 may indicate that convergence is detectable early in the run, is fitness also detectable. If it is, we may save time by using a fast test to avoid a slow, fruitless run.
  - To detect when a diversity boost is needed – the new method can detect the diversity of a population. By detecting diversity during evolution we could provide tailored mutation and crossover rates.
  - Use the maximal schemata as functions – rather than using the maximal schemata for analysis, they may be suited for use as subroutines. The new method would provide a non-biased way to obtain these subroutines for future generations or evolutions.

So, the PhD student with his ice-cream scoop finds there is an entirely new iceberg under the one made into this thesis.

Bibliography

[1] AAMODT, A., AND PLAZA, E. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications 7*, 1 (1994), 39–59.

[2] AHLUWALIA, M., AND BULL, L. Coevolving functions in genetic programming. *Journal of Systems Architecture 47*, 7 (2001), 573–585.

[3] ALTENBERG, L. The evolution of evolvability in genetic programming. In *Advances in Genetic Programming*, J. Kenneth E. Kinnear, Ed. MIT Press, 1994, ch. 3, pp. 47–74.

[4] ALTENBERG, L. The schema theorem and Price's theorem. In *Foundations of Genetic Algorithms 3* (Estes Park, Colorado, USA, 1994), L. D. Whitley and M. D. Vose, Eds., Morgan Kaufmann, pp. 23–49. Published 1995.

[5] ANGELINE, P. J. Subtree crossover: Building block engine or macromutation? In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 9–17.

[6] ANGELINE, P. J., AND POLLACK, J. B. Coevolving high-level representations. July Technical report 92-PA-COEVOLVE, Laboratory for Artificial Intelligence. The Ohio State University, 1993.

[7] BADER-EL-DEN, M., AND FATIMA, S. Genetic programming for auction based scheduling. In *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010* (Istanbul, 2010), A. I. Esparcia-Alcazar, A. Ekart, S. Silva, S. Dignum, and A. S. Uyar, Eds., vol. 6021 of *LNCS*, Springer, pp. 256–267.

[8] BANZHAF, W., BANSCHERUS, D., AND DITTRICH, P. Hierarchical genetic programming using local modules. *InterJournal Complex Systems 228* (2000).

[9] BICKEL, A. S., AND BICKEL, R. W. Tree structured rules in genetic algorithms. In *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms* (MIT, Cambridge, MA, USA, 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, pp. 77–81.

[10] BISHOP, C. *Neural Networks for Pattern Recognition*. Oxford University Press, USA, 1995.

[11] BRAMEIER, M. *On Linear Genetic Programming*. PhD thesis, Fachbereich Informatik, Universit Germany, 2003.

[12] BROCK, O. Evolving reusable subroutines for genetic programming. In *Artificial Life at Stanford 1994*, J. R. Koza, Ed. Stanford Bookstore, Stanford, California, 94305-3079 USA, 1994, pp. 11–19.

[13] BROOKS, R. A. Artificial life and real robots. In *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life* (Cambridge, MA, USA, 1992), F. J. Varela and P. Bourgine, Eds., MIT Press, pp. 3–10.

[14] CIESIELSKI, V., AND LI, X. Experiments with explicit for-loops in genetic programming. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation* (Portland, Oregon, 2004), IEEE Press, pp. 494–501.

[15] CLERC, M. *Particle Swarm Optimization*, vol. 4. ISTE London, UK, 2006.

[16] COVER, T., AND HART, P. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory 13*, 1 (2002), 21–27.

[17] CRAMER, N. L. A representation for the adaptive generation of simple sequential programs. In *Proceedings of an International Conference on Genetic Algorithms and the Applications* (Carnegie-Mellon University, Pittsburgh, PA, USA, 1985), J. J. Grefenstette, Ed., pp. 183–187.

[18] DAIDA, J. M., BERTRAM, R. R., POLITO, J. A., AND STANHOPE, S. A. Analysis of single-node (building) blocks in genetic programming. In *Advances in Genetic Programming 3*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, Eds. MIT Press, Cambridge, MA, USA, 1999, ch. 10, pp. 217–241.

[19] DASGUPTA, D. *Artficial Immune Systems and Their Applications*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1998.

[20] DORIGO, M., MANIEZZO, V., AND COLORNI, A. Ant system: optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on 26*, 1 (2002), 29–41.

[21] DROSTE, S. Efficient genetic programming for finding good generalizing boolean functions. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 82–87.

[22] EIBEN, A., AND SMITH, J. *Introduction to Evolutionary Computing*. Springer Verlag, 2003.

[23] FELDMAN, J., AND BALLARD, D. Connectionist models and their properties. *Cognitive Science 6*, 3 (1982), 205–254.

[24] FERARIU, L., AND PATELLI, A. Multiobjective genetic programming for nonlinear system identification. In *9th International Conference on Adaptive and Natural Computing Algorithms, ICANNGA 2009* (Kuopio, Finland, 2009), M. Kolehmainen, P. Toivanen, and B. Beliczynski, Eds., vol. 5495 of *Lecture Notes in Computer Science*, Springer, pp. 233–242.

[25] FORSYTH, R. Beagle a darwinian approach to pattern recognition. *Kybernetes 10* (1981), 159–166.

[26] FUKUNAGA, A. S. Massively parallel evolution of sat heuristics. In *2009 IEEE Congress on Evolutionary Computation* (Trondheim, Norway, 2009), A. Tyrrell, Ed., IEEE Computational Intelligence Society, IEEE Press, pp. 1478–1485.

[27] FURUHOLMEN, M., GLETTE, K., HOVIN, M., AND TORRESEN, J. Coevolving heuristics for the distributor's pallet packing problem. In *2009 IEEE Congress on Evolutionary Computation* (Trondheim, Norway, 2009), A. Tyrrell, Ed., IEEE Computational Intelligence Society, IEEE Press, pp. 18–21.

[28] GARDNER, M. On cellular automata, self-reproduction, the garden of eden, and the game of life. *Scientific American 224*, 2 (1971), 112–117.

[29] GOLDBERG, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison–Wesley, Reading, MA, 1989.

[30] GREFENSTETTE, J. J., AND BAKER, J. E. How genetic algorithms work: A critical look at implicit parallelism. In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA'89)* (San Mateo, California, 1989), J. D. Schaffer, Ed., Morgan Kaufmann Publishers, Inc., pp. 20–27.

[31] HAYNES, T. Phenotypical building blocks for genetic programming. In *Genetic Algorithms: Proceedings of the Seventh International Conference* (Michigan State University, East Lansing, MI, USA, 1997), T. Back, Ed., Morgan Kaufmann, pp. 26–33.

[32] HAYNES, T. D. *Collective Adaptation: The Sharing of Building Blocks*. PhD thesis, Department of Mathematical and Computer Sciences, University of Tulsa, Tulsa, OK, USA, 1998.

[33] HOANG, T. H., ESSAM, D., MCKAY, B., AND HOAI, N. X. Building on success in genetic programming: Adaptive variation and developmental evaluation. In *Proceedings of the Second International Symposium on Computation and Intelligence, ISICA 2007* (Wuhan, China, 2007), L. Kang, Y. Liu, and S. Y. Zeng, Eds., vol. 4683 of *Lecture Notes in Computer Science*, Springer, pp. 137–146.

[34] HOLLAND, J. H. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[35] JONG, K. D. On using genetic algorithms to search program spaces. In *Genetic Algorithms and their Applications: Proceedings of the second international conference on Genetic Algorithms* (MIT, Cambridge, MA, USA, 1987), J. J. Grefenstette, Ed., Lawrence Erlbaum Associates, pp. 210–216.

[36] KAMEYA, Y., KUMAGAI, J., AND KURATA, Y. Accelerating genetic programming by frequent subtree mining. In *GECCO '08: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation* (Atlanta, GA, USA, 2008), M. Keijzer, G. Antoniol, C. B. Congdon, K. Deb, B. Doerr, N. Hansen, J. H. Holmes, G. S. Hornby, D. Howard, J. Kennedy, S. Kumar, F. G. Lobo, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, J. Pollack, K. Sastry, K. Stanley, A. Stoica, E.-G. Talbi, and I. Wegener, Eds., ACM, pp. 1203–1210.

[37] KINZETT, D., JOHNSTON, M., AND ZHANG, M. How online simplification affects building blocks in genetic programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (Montreal, 2009), G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C. B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J. F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. D. Penta, B. Doerr, T. Jansen, R. Poli, and E. Alba, Eds., ACM, pp. 979–986.

[38] KINZETT, D., JOHNSTON, M., AND ZHANG, M. Numerical simplification for bloat control and analysis of building blocks in genetic programming. *Evolutionary Intelligence 2*, 4 (2009), 151–168.

[39] KINZETT, D., ZHANG, M., AND JOHNSTON, M. Analysis of building blocks with numerical simplification in genetic programming. In *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010* (Istanbul, 2010), A. I. Esparcia-Alcazar, A. Ekart, S. Silva, S. Dignum, and A. S. Uyar, Eds., vol. 6021 of *LNCS*, Springer, pp. 289–300.

[40] KOZA, J. R. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89* (Detroit, MI, USA, 1989), N. S. Sridharan, Ed., vol. 1, Morgan Kaufmann, pp. 768–774.

[41] KOZA, J. R. A genetic approach to the truck backer upper problem and the inter-twined spiral problem. In *Proceedings of IJCNN International Joint Conference on Neural Networks* (1992), vol. IV, IEEE Press, pp. 310–318.

[42] KOZA, J. R. Genetic programming: On the programming of computers by means of natural selection. *Statistics and Computing 4*, 2 (1994).

[43] KOZA, J. R., III, F. H. B., ANDRE, D., AND KEANE, M. A. Automated design of both the topology and sizing of analog electrical circuits using genetic programming. In *Artificial Intelligence in Design '96* (Dordrecht, 1996), J. S. Gero and F. Sudweeks, Eds., Kluwer Academic, pp. 151–170.

[44] KRATOCHVIL, O., OSMERA, P., AND POPELKA, O. Parallel grammatical evolution for circuit optimization. In *Proceedings of the World Congress on Engineering and Computer Science, WCECS '09* (San Francisco, USA, 2009), S. I. Ao, C. Douglas, W. S. Grundfest, and J. Burgstone, Eds., vol. II, International Association of Engineers, Newswood Limited, pp. 1032–1037.

[45] KRONBERGER, G., WINKLER, S. M., AFFENZELLER, M., BEHAM, A., AND WAGNER, S. On the success rate of crossover operators for genetic programming with offspring selection. In *12th International Conference on Computer Aided Systems Theory, EUROCAST 2009* (Las Palmas de Gran Canaria, Spain, 2009), R. Moreno-Díaz, F. Pichler, and A. Quesada-Arencibia, Eds., vol. 5717 of *Lecture Notes in Computer Science*, Springer, pp. 793–800.

[46] LANGDON, W. B. Evolving data structures using genetic programming. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)* (Pittsburgh, PA, USA, 1995), L. Eshelman, Ed., Morgan Kaufmann, pp. 295–302.

[47] LANGDON, W. B. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines 1*, 1/2 (2000), 95–119.

[48] LANGDON, W. B., AND BANZHAF, W. Repeated patterns in tree genetic programming. In *Proceedings of the 8th European Conference on Genetic Programming* (Lausanne, Switzerland, 2005), M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, Eds., vol. 3447 of *Lecture Notes in Computer Science*, Springer, pp. 190–202.

[49] LANGDON, W. B., AND BANZHAF, W. Repeated sequences in linear genetic programming genomes. *Complex Systems 15*, 4 (2005), 285–306.

[50] LANGDON, W. B., AND HARMAN, M. Evolving a cuda kernel from an nvidia template. In *2010 IEEE World Congress on Computational Intelligence* (Barcelona, 2010), P. Sobrevilla, Ed., IEEE, pp. 2376–2383.

[51] LANGDON, W. B., AND POLI, R. Why ants are hard. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (University of Wisconsin, Madison, Wisconsin, USA, 1998), J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds., Morgan Kaufmann, pp. 193–201.

[52] LANGDON, W. B., AND POLI, R. Why "building blocks" don't work on parity problems. Tech. Rep. CSRP-98-17, University of Birmingham, School of Computer Science, 1998.

[53] LI, G., LEE, K.-H., AND LEUNG, K.-S. Evolve schema directly using instruction matrix based genetic programming. In *Proceedings of the 8th European Conference on Genetic Programming* (Lausanne, Switzerland, 2005), M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, Eds., vol. 3447 of *Lecture Notes in Computer Science*, Springer, pp. 271–280.

[54] LI, X., AND CIESIELSKI, V. An analysis of explicit loops in genetic programming. In *Proceedings of the 2005 IEEE Congress on Evolution-*

ary Computation (Edinburgh, UK, 2005), D. Corne, Z. Michalewicz, M. Dorigo, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, T. K. Chen, G. Raidl, A. Zalzala, S. Lucas, B. Paechter, J. Willies, J. J. M. Guervos, E. Eberbach, B. McKay, A. Channon, A. Tiwari, L. G. Volkert, D. Ashlock, and M. Schoenauer, Eds., vol. 3, IEEE Press, pp. 2522–2529.

[55] LUKE, S., AND SPECTOR, L. Evolving graphs and networks with edge encoding: Preliminary report. In *Late Breaking Papers at the Genetic Programming 1996 Conference Stanford University July 28-31, 1996* (Stanford University, CA, USA, 1996), J. R. Koza, Ed., Stanford Bookstore, pp. 117–124.

[56] MAJEED, H. A new approach to evaluate GP schema in context. In *Genetic and Evolutionary Computation Conference (GECCO2005) workshop program* (Washington, D.C., USA, 2005), F. Rothlauf, M. Blowers, and J Eds., ACM Press, pp. 378–381.

[57] MAJEED, H. *The Importance of semantic context in tree based GP and its application in defining a less destructive, context aware crossover for GP*. PhD thesis, University of Limerick, Ireland, 2007.

[58] MAJEED, H., RYAN, C., AND AZAD, R. M. A. Evaluating GP schema in context. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation* (Washington DC, USA, 2005), H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, Eds., vol. 2, ACM Press, pp. 1773–1774.

[59] MCKAY, R. I., NGUYEN, X. H., CHENEY, J. R., KIM, M., MORI, N., AND HOANG, T. H. Estimating the distribution and propagation of genetic programming building blocks through tree com-

pression. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation* (Montreal, 2009), G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C. B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J. F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. D. Penta, B. Doerr, T. Jansen, R. Poli, and E. Alba, Eds., ACM, pp. 1011–1018.

[60] McKay, R. I. B., Shin, J., Hoang, T. H., Nguyen, X. H., and Mori, N. Using compression to understand the distribution of building blocks in genetic programming populations. In *2007 IEEE Congress on Evolutionary Computation* (Singapore, 2007), D. Srinivasan and L. Wang, Eds., IEEE Computational Intelligence Society, IEEE Press, pp. 2501–2508.

[61] McPhee, N. F., Ohs, B., and Hutchison, T. Semantic building blocks in genetic programming. Working Paper Series Volume 3 Number 2, University of Minnesota Morris, 600 East 4th Street, Morris, MN 56267, USA, 2007.

[62] McPhee, N. F., Ohs, B., and Hutchison, T. Semantic building blocks in genetic programming. In *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008* (Naples, 2008), M. O'Neill, L. Vanneschi, S. Gustafson, A. I. E. Alcazar, I. D. Falco, A. D. Cioppa, and E. Tarantino, Eds., vol. 4971 of *Lecture Notes in Computer Science*, Springer, pp. 134–145.

[63] McPhee, N. F., and Poli, R. A schema theory analysis of the evolution of size in genetic programming with linear representations. Tech. Rep. CSRP-00-22, University of Birmingham, School of Computer Science, 2000.

[64] McPhee, N. F., and Poli, R. Using schema theory to explore interactions of multiple operators. In *GECCO 2002: Proceedings of the*

Genetic and Evolutionary Computation Conference (New York, 2002), W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, Eds., Morgan Kaufmann Publishers, pp. 853–860.

[65] McPhee, N. F., Poli, R., and Rowe, J. E. A schema theory analysis of mutation size biases in genetic programming with linear representations. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001* (COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 2001), IEEE Press, pp. 1078–1085.

[66] Miller, J. F., and Thomson, P. Cartesian genetic programming. In *Genetic Programming, Proceedings of EuroGP'2000* (Edinburgh, 2000), R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, Eds., vol. 1802 of *LNCS*, Springer-Verlag, pp. 121–132.

[67] Mitavskiy, B., and Rowe, J. E. A schema-based version of Geiringer's theorem for nonlinear genetic programming with homologous crossover. In *Foundations of Genetic Algorithms 8*, A. H. Wright, M. D. Vose, K. A. D. Jong, and L. M. Schmitt, Eds., vol. 3469 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 2005, pp. 156–175.

[68] Munawar, A., Wahib, M., Munetomo, M., and Akama, K. Hybrid of genetic algorithm and local search to solve max-sat problem using nvidia cuda framework. *Genetic Programming and Evolvable Machines 10*, 4 (2009), 391–415.

[69] Nordin, P., and Banzhaf, W. Evolving turing-complete programs for a register machine with self-modifying code. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*

(Pittsburgh, PA, USA, 1995), L. Eshelman, Ed., Morgan Kaufmann, pp. 318–325.

[70] O'REILLY, U.-M., AND OPPACHER, F. The troubling aspects of a building block hypothesis for genetic programming. In *Foundations of Genetic Algorithms 3* (Estes Park, Colorado, USA, 1994), L. D. Whitley and M. D. Vose, Eds., Morgan Kaufmann, pp. 73–88. Published 1995.

[71] O'REILLY, U.-M., AND OPPACHER, F. Using building block functions to investigate a building block hypothesis for genetic programming. Working Paper 94-02-029, Santa Fe Institute, 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA, 1994.

[72] PADMAN, R., AND ROEHRIG, S. A genetic programming approach for heuristic selection in constrained project scheduling. Tech. Rep. 95-30, H. John Heinz III School of Public Policy and Management, Carniege-Mellon University, 1995.

[73] PATELLI, A., AND FERARIU, L. Elite based multiobjective genetic programming in nonlinear systems identification. *Advances in Electrical and Computer Engineering 10*, 1 (2010), 94–99.

[74] POLI, R. New results in the schema theory for GP with one-point crossover which account for schema creation, survival and disruption. Tech. Rep. CSRP-99-18, University of Birmingham, School of Computer Science, 1999.

[75] POLI, R. Probabilistic schema theorems without expectation, recursive conditional schema theorem, convergence and population sizing in genetic algorithms. Tech. Rep. CSRP-99-3, University of Birmingham, School of Computer Science, 1999.

[76] POLI, R. Schema theory without expectations for GP and GAs with one-point crossover in the presence of schema creation. Tech. Rep.

CSRP-99-13, University of Birmingham, School of Computer Science, 1999.

[77] POLI, R. Exact schema theorem and effective fitness for GP with one-point crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)* (Las Vegas, Nevada, USA, 2000), D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, Eds., Morgan Kaufmann, pp. 469–476.

[78] POLI, R. Hyperschema theory for GP with one-point crossover, building blocks, and some new results in GA theory. In *Genetic Programming, Proceedings of EuroGP'2000* (Edinburgh, 2000), R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, Eds., vol. 1802 of *LNCS*, Springer-Verlag, pp. 163–180.

[79] POLI, R. A macroscopic exact schema theorem and a redefinition of effective fitness for GP with one-point crossover. Tech. Rep. CSRP-00-1, University of Birmingham, School of Computer Science, 2000.

[80] POLI, R. Microscopic and macroscopic schema theories for genetic programming and variable-length genetic algorithms with one-point crossover, their use and their relations with earlier GP and GA schema theories. Tech. Rep. CSRP-00-15, University of Birmingham, School of Computer Science, 2000.

[81] POLI, R. Exact schema theory for genetic programming and variable-length genetic algorithms with one-point crossover. *Genetic Programming and Evolvable Machines 2*, 2 (2001), 123–163.

[82] POLI, R. General schema theory for genetic programming with subtree-swapping crossover. In *Genetic Programming, Proceedings of EuroGP'2001* (Lake Como, Italy, 2001), J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds., vol. 2038 of *LNCS*, Springer-Verlag, pp. 143–159.

[83] POLI, R. Exact schema theorems and markov chain models for genetic programming and variable length genetic algorithms. Report 330, Dagstuhl, Germany, 2002.

[84] POLI, R., AND LANGDON, W. B. An experimental analysis of schema creation, propagation and disruption in genetic programming. In *Genetic Algorithms: Proceedings of the Seventh International Conference* (Michigan State University, East Lansing, MI, USA, 1997), T. Back, Ed., Morgan Kaufmann, pp. 18–25.

[85] POLI, R., AND LANGDON, W. B. A new schema theory for genetic programming with one-point crossover and point mutation. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 278–285.

[86] POLI, R., AND LANGDON, W. B. A review of theoretical and experimental results on schemata in genetic programming. Technical Report CSRP-97-27, University of Birmingham, B15 2TT, UK, 1997.

[87] POLI, R., AND LANGDON, W. B. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation 6*, 3 (1998), 231–252.

[88] POLI, R., LANGDON, W. B., AND O'REILLY, U.-M. Analysis of schema variance and short term extinction likelihoods. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (University of Wisconsin, Madison, Wisconsin, USA, 1998), J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds., Morgan Kaufmann, pp. 284–292.

[89] POLI, R., AND MCPHEE, N. F. Exact GP schema theory for headless chicken crossover and subtree mutation. Tech. Rep. CSRP-00-23, University of Birmingham, School of Computer Science, 2000.

[90] POLI, R., AND MCPHEE, N. F. Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. Tech. Rep. CSRP-00-14, University of Birmingham, School of Computer Science, 2000.

[91] POLI, R., AND MCPHEE, N. F. Exact schema theory for GP and variable-length GAs with homologous crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* (San Francisco, California, USA, 2001), L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, Eds., Morgan Kaufmann, pp. 104–111.

[92] POLI, R., MCPHEE, N. F., AND ROWE, J. E. Exact schema theory and markov chain models for genetic programming and variable-length genetic algorithms with homologous crossover. *Genetic Programming and Evolvable Machines 5*, 1 (2004), 31–70.

[93] POLI, R., AND STEPHENS, C. R. The building block basis for genetic programming and variable-length genetic algorithms. *International Journal of Computational Intelligence Research 1*, 2 (2005), 183–197.

[94] POLI, R., STEPHENS, C. R., WRIGHT, A. H., AND ROWE, J. E. A schema theory based extension of Geiringer's theorem for linear GP and variable length gas under homologous crossover. In *Foundations of Genetic Algorithms VII* (Torremolinos, Spain, 2002), K. A. D. Jong, R. Poli, and J. E. Rowe, Eds., Morgan Kaufmann, pp. 45–62.

[95] PRICE, G. R. Selection and covariance. *Nature 227, August 1* (1970), 520–521.

[96] ROBERTS, S. C., HOWARD, D., AND KOZA, J. R. Evolving modules in genetic programming by subtree encapsulation. In *Genetic Programming, Proceedings of EuroGP'2001* (Lake Como, Italy, 2001), J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds., vol. 2038 of *LNCS*, Springer-Verlag, pp. 160–175.

[97] RODRIGUES, E., AND POZO, A. Grammar-guided genetic programming and automatically defined functions. In *Advances in Artificial Intelligence: 16th Brazilian Symposium on Artificial Intelligence, SBIA 2002, Proceedings* (Porto de Galinhas/Recife, Brazil, 2002), G. Bittencourt and G. L. Ramalho, Eds., vol. 2507 of *LNAI*, pp. 324–333.

[98] ROSCA, J. Towards automatic discovery of building blocks in genetic programming. In *Working Notes for the AAAI Symposium on Genetic Programming* (MIT, Cambridge, MA, USA, 1995), E. V. Siegel and J. R. Koza, Eds., AAAI, pp. 78–85.

[99] ROSCA, J., AND BALLARD, D. H. Evolution-based discovery of hierarchical behaviors. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)* (1996), AAAI / The MIT Press, pp. 888–894.

[100] ROSCA, J. P. Genetic programming exploratory power and the discovery of functions. In *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming* (San Diego, CA, USA, 1995), J. R. McDonnell, R. G. Reynolds, and D. B. Fogel, Eds., MIT Press, pp. 719–736.

[101] ROSCA, J. P. Analysis of complexity drift in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Con-*

ference (Stanford University, CA, USA, 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 286–294.

[102] ROSCA, J. P. *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, Department of Computer Science, The College of Arts and Sciences, University of Rochester, Rochester, NY 14627, USA, 1997.

[103] ROSCA, J. P., AND BALLARD, D. H. Genetic programming with adaptive representations. Tech. Rep. TR 489, University of Rochester, Computer Science Department, Rochester, NY, USA, 1994.

[104] ROSCA, J. P., AND BALLARD, D. H. Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning* (1994), Morgan Kaufmann, pp. 251–258.

[105] ROSCA, J. P., AND BALLARD, D. H. Learning by adapting representations in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence* (Orlando, Florida, USA, 1994), IEEE Press, pp. 27–29.

[106] ROSCA, J. P., AND BALLARD, D. H. Complexity drift in evolutionary computation with tree representations. Technical Report NRL5, University of Rochester, Computer Science Department, Rochester, NY, USA, 1996.

[107] ROSCA, J. P., AND BALLARD, D. H. Discovery of subroutines in genetic programming. In *Advances in Genetic Programming 2*, P. J. Angeline and J. K. E. Kinnear, Eds. MIT Press, Cambridge, MA, USA, 1996, ch. 9, pp. 177–202.

[108] ROSCA, J. P., AND BALLARD, D. H. Rooted-tree schemata in genetic programming. In *Advances in Genetic Programming 3*, L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, Eds. MIT Press, Cambridge, MA, USA, 1999, ch. 11, pp. 243–271.

[109] RYAN, C., MAJEED, H., AND AZAD, A. A competitive building block hypothesis. In *Genetic and Evolutionary Computation – GECCO-2004, Part II* (Seattle, WA, USA, 2004), K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, Eds., vol. 3103 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 654–665.

[110] SASTRY, K., O'REILLY, U.-M., GOLDBERG, D. E., AND HILL, D. Building block supply in genetic programming. In *Genetic Programming Theory and Practice*, R. L. Riolo and B. Worzel, Eds. Kluwer, 2003, ch. 9, pp. 137–154.

[111] SHAN, Y., MCKAY, R. I., BAXTER, R., ABBASS, H., ESSAM, D., AND HOAI, N. X. Grammar model-based program evolution. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation* (Portland, Oregon, 2004), IEEE Press, pp. 478–485.

[112] SIGAUD, O., AND WILSON, S. Learning classifier systems: a survey. *Soft Computing-A Fusion of Foundations, Methodologies and Applications 11*, 11 (2007), 1065–1078.

[113] SMART, W., ANDREAE, P., AND ZHANG, M. Empirical analysis of GP tree-fragments. In *Proceedings of the 10th European Conference on Genetic Programming* (Valencia, Spain, 2007), M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, and A. I. Esparcia-Alcázar, Eds., vol. 4445 of *Lecture Notes in Computer Science*, Springer, pp. 55–67.

[114] SMART, W., AND ZHANG, M. Empirical analysis of schemata in genetic programming using maximal schemata and MSG. In *2008 IEEE World Congress on Computational Intelligence* (Hong Kong, 2008), J. Wang, Ed., IEEE Computational Intelligence Society, IEEE Press, pp. 2983–2990.

[115] SPEARS, W., DE JONG, K., BAECK, T., FOGEL, D., AND DE GARIS, H. An overview of evolutionary computation. In *Machine Learning: ECML-93* (1993), Springer, pp. 442–459.

[116] SPECTOR, L. Simultaneous evolution of programs and their control structures. In *Advances in Genetic Programming 2*, P. J. Angeline and J. K. E. Kinnear, Eds. MIT Press, Cambridge, MA, USA, 1996, ch. 7, pp. 137–154.

[117] TANJI, M., AND IBA, H. Program optimization by random tree sampling. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation* (Montreal, 2009), G. Raidl, F. Rothlauf, G. Squillero, R. Drechsler, T. Stuetzle, M. Birattari, C. B. Congdon, M. Middendorf, C. Blum, C. Cotta, P. Bosman, J. Grahl, J. Knowles, D. Corne, H.-G. Beyer, K. Stanley, J. F. Miller, J. van Hemert, T. Lenaerts, M. Ebner, J. Bacardit, M. O'Neill, M. D. Penta, B. Doerr, T. Jansen, R. Poli, and E. Alba, Eds., ACM, pp. 1131–1138.

[118] TURING, A. M. Intelligent machinery. In *Machine Intelligence*, B. Meltzer and D. Michie, Eds., vol. 5. Edinburgh University Press, Edinburgh, UK, 1969, ch. 1, pp. 3–23.

[119] VAFAEE, F., XIAO, W., NELSON, P. C., AND ZHOU, C. Adaptively evolving probabilities of genetic operators. In *Seventh International Conference on Machine Learning and Applications, ICMLA '08* (La Jolla, San Diego, USA, 2008), IEEE, pp. 292–299.

[120] VANNESCHI, L., CASTELLI, M., AND SILVA, S. Measuring bloat, overfitting and functional complexity in genetic programming. In *GECCO '10: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation* (Portland, Oregon, USA, 2010), J. Branke, M. Pelikan, E. Alba, D. V. Arnold, J. Bongard, A. Brabazon, J. Branke, M. V. Butz, J. Clune, M. Cohen, K. Deb, A. P. Engelbrecht, N. Krasnogor, J. F. Miller, M. O'Neill, K. Sastry, D. Thierens, J. van Hemert, L. Vanneschi, and C. Witt, Eds., ACM, pp. 877–884.

[121] VEKARIA, K., AND CLACK, C. Schema propagation in selective crossover. In *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conference* (Orlando, Florida, USA, 1999), S. Brave and A. S. Wu, Eds., pp. 268–275.

[122] WALKER, M. Evolution of a robotic soccer player. *Research Letters in the Information and Mathematical Sciences 3*, 1 (2002), 15–23.

[123] WHIGHAM, P. A. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* (Tahoe City, California, USA, 1995), J. P. Rosca, Ed., pp. 33–41.

[124] WHIGHAM, P. A. A schema theorem for context-free grammars. In *1995 IEEE Conference on Evolutionary Computation* (Perth, Australia, 1995), vol. 1, IEEE Press, pp. 178–181.

[125] WHIGHAM, P. A., AND DICK, G. Implicitly controlling bloat in genetic programming. *IEEE Transactions on Evolutionary Computation 14*, 2 (2010), 173–190.

[126] WHITLEY, L. D. Fundamental principles of deception in genetic search. *Foundations of Genetic Algorithms* (1990), pp. 221–241.

[127] WILKERSON, J. L., AND TAURITZ, D. Coevolutionary automated software correction. In *GECCO '10: Proceedings of the 12th annual*

conference on Genetic and evolutionary computation (Portland, Oregon, USA, 2010), J. Branke, M. Pelikan, E. Alba, D. V. Arnold, J. Bongard, A. Brabazon, J. Branke, M. V. Butz, J. Clune, M. Cohen, K. Deb, A. P. Engelbrecht, N. Krasnogor, J. F. Miller, M. O'Neill, K. Sastry, D. Thierens, J. van Hemert, L. Vanneschi, and C. Witt, Eds., ACM, pp. 1391–1392.

[128] WILSON, G. C., AND HEYWOOD, M. I. Context-based repeated sequences in linear genetic programming. In *Proceedings of the 8th European Conference on Genetic Programming* (Lausanne, Switzerland, 2005), M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, Eds., vol. 3447 of *Lecture Notes in Computer Science*, Springer, pp. 240–249.

[129] WONG, M. L., AND LEUNG, K. S. Applying logic grammars to induce sub-functions in genetic programming. In *1995 IEEE Conference on Evolutionary Computation* (Perth, Australia, 1995), vol. 2, IEEE Press, pp. 737–740.

[130] WONG, P., AND ZHANG, M. Numerical-node building block analysis of genetic programming with simplification. Tech. Rep. CS-TR-06-15, Computer Science, Victoria University of Wellington, New Zealand, 2006.

[131] WONG, P., AND ZHANG, M. Effects of program simplification on simple building blocks in genetic programming. In *2007 IEEE Congress on Evolutionary Computation* (Singapore, 2007), D. Srinivasan and L. Wang, Eds., IEEE Computational Intelligence Society, IEEE Press, pp. 1570–1577.

[132] WOODWARD, J. R. Modularity in genetic programming. In *Genetic Programming, Proceedings of EuroGP'2003* (Essex, 2003), C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds., vol. 2610 of *LNCS*, Springer-Verlag, pp. 254–263.

[133] YUEN, C. C. Selective crossover using gene dominance as an adaptive strategy for genetic programming. MSc intelligent systems, University College, London, UK, 2004.