CHAPTER 1

Introduction

This chapter gives an overview of the motivations, goals and results of the thesis. Section 1.1 first introduces a general definition of program discovery, an inductive search problem that has been pursued with a diverse set of approaches. After motivating the study of program discovery, it describes the precise framework for program discovery used by the particular algorithm that this thesis focuses upon. The algorithm is named Genetic Programming (GP) [69]. For the remainder of the thesis we adopt GP's restricted version of program discovery for the purposes of exposition, focus and clarity.

GP is introduced in Section 1.2 using the background of Genetic Algorithms (GAs). GP is an adaptive search algorithm that is based upon neo-Darwinian concepts of evolution. We have chosen to study GP because it is a robust, successful program discovery algorithm. Chapter 2 provides a detailed description of GP and its related literature.

In Section 1.3 the goal of the thesis, to analyse and improve GP, is elaborated upon and accompanied by a high level description of how it is approached and solved. Section 1.4 succinctly lists the contents of the remaining chapters of the thesis.

1.1. Program Discovery

The goals of program discovery are:

Given a set of input-output pairs or formal specification of behaviour, produce a computer program that

1. non-trivially computes correct outputs for the inputs of each test case. Non-

trivial computation implies that the program does not directly map from inputs to outputs by means of some sort of table. Rather, the program is an encoding of some algorithm.

2. computes outputs in such a way that, if the inputs have been representively chosen, the program will compute correct outputs for novel inputs.

A diverse set of approaches to solving program discovery problems, including Automatic Programming (e.g. [11, 8, 98]), Inductive Logic Programming (e.g. [84, 93, 94]), and evolution-based algorithms different from GP (e.g. [29, 110, 16, 30, 40, 109]) have been pursued. A exposition and comparison of them is beyond the scope of this thesis. The focus of this thesis is to provide a systematic analysis of one program discovery algorithm called Genetic Programming. Therefore, in Section 1.1.3, we state a restricted version of program discovery which we shall assume in the remainder of the thesis.

1.1.1. Program Discovery as an Induction Problem

In so far as a correct program must be induced from test cases, it is obvious that program discovery is an induction task. Therefore, program discovery inherits all the merits and pitfalls of inductive reasoning. While it is "creative" and ampliative by suggesting new hypotheses to link the outputs to the inputs, inductive hypotheses are always falsifiable and their ability to generalize out of sample (i.e., beyond the suite of test cases in the case of program discovery) is intrinsically linked to how well the sample represents the entire problem domain. The sample needs to be representative of the entire space, if possible. By definition, program discovery is accomplished without the introduction of new primitives, however, this does not preclude the definition of new "concepts" to replace the original ones. This means that, although there is a finite set of inductive hypotheses consisting of all possible combinations of them up to a prespecified maximum depth parameter, the set of inductive hypotheses is very large and potentially very expressive. The finiteness of this set is constraining only if a fundamental behaviour (i.e. one that is more simple than any primitive in the primitive set) can not be expressed by any combination.

1.1.2. Motivation for Studying Program Discovery

What generally makes program discovery special and different from other inductive search tasks is, quite obviously, that its solutions are expressed as programs. A program is a useful solution because it is an algorithm; in other words, a specification of behaviour that works for a general class of problems because it is parameterized by the use of variables. Finding algorithms is more difficult than finding a single solution but obviously it is also more useful since generalized solutions work for an entire class of tasks. Programs can encode high level program semantics such as rules, logic, iteration, recursion, and sequence as well as simple numbers and numerical relationships.

What makes a program discovery algorithm useful and, thus, important to study, is the fact that many problems from a wide range of domains can be translated into program discovery problems and, should the algorithm succeed, these same problems will directly have solutions. Figure 1 illustrates this concept. It is also thoroughly emphasized and demonstrated in the book "Genetic Programming: on the programming of computers by means of natural selection" by John R. Koza. Koza states:

> A wide variety of seemingly different problems from many different fields can be recast as requiring the discovery of a computer program that produces some desired output when presented with particular inputs. That is, many seemingly different problems can be reformulated as problems of program induction. [69, pg. 3]

In Chapter 2 of his book Koza supplies a table that lists 13 problem domains, describes how the concept of a computer program has an analogy in each, and names the analogous program inputs and outputs of the domain. For example,

- in optimal control, a control strategy is the equivalent of a computer program. The control strategy input is state variables and the output is a control variable.
- in planning, a plan mimics a program by using sensor or detector values as input and producing effector actions.
- in sequence induction, a mathematical expression plays the role of a program using an index position as input. Its output is a correct sequence element.

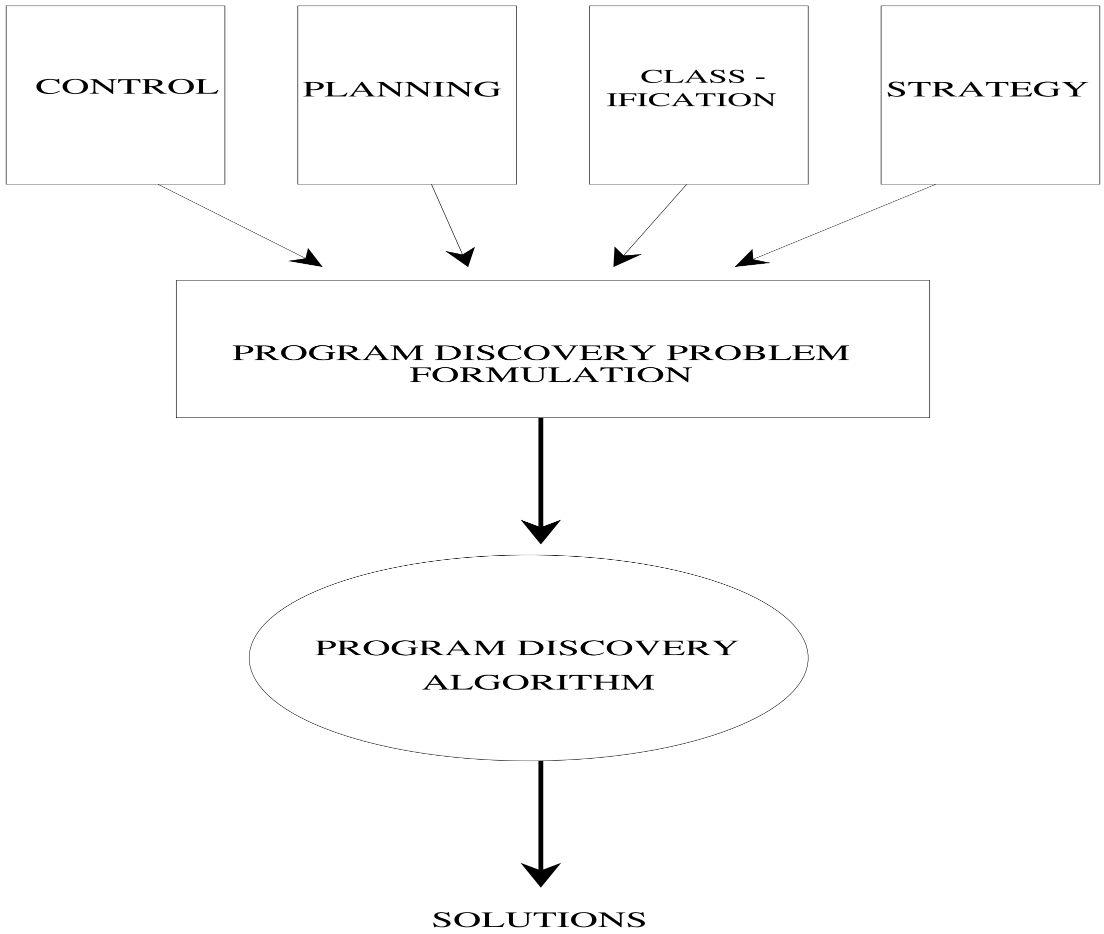

Figure 1. The value of Program Discovery: When many problems can be reformulated as program discovery, a program discovery algorithm is advantageous.

- in symbolic regression a mathematical expression is the equivalent of a program that, using independent variables, derives dependent variables.
- in game playing strategies, a strategy is a program that uses game and move information as input and produces output that is the direction of moves.
- in empirical discovery and forecasting, a model is a form of program which manipulates independent variables to output dependent variables.

The key to a program induction reformulation is to recognize that, despite different terminology, within the domain of interest there exists a basic need for some algorithm to provide solutions for multiple instances of a problem.

Reformulating tasks into program discovery problems means that, in the course of the search for a correct program, many candidate programs will have to be executed and assessed a fitness value. The execution of the programs may take place in a simulated version of the problem domain or each program may actually be assessed by using it in the actual domain. A simulation approach in program discovery is no different from simulation approaches elsewhere; care must be taken to authenticate the simulated environment so that it relays valid information on performance.

1.1.3. Thesis Definition of Program Discovery

In the GP framework of program discovery, the program discovery algorithm is supplied by the task designers (i.e., the persons who have chosen this approach program discovery as a means of solving their problem) with a "test suite", a set of "primitives", and a "fitness function".

A "test suite" consists of test cases which are each a specific example of a problem described in terms of inputs and desired outputs.

"Primitives" are functions or variables that can be used by the algorithm to compose a program. Each composition of primitives (or program) is a candidate solution to the posed program discovery problem. The set of primitives must be chosen so that it has the capacity to represent actual actions and objects (or, operators, operands, results) that occur or exist in the problem domain. For example, in the task of block stacking by a manipulator arm, the operators could be represented by the functions `remove-top-block`, `place-block-on-stack`, `find-next-needed-block`, etc. and the operands could be represented as the functions `next-needed-block`, `top-correct-block`. Fitting the `place-block-on-stack` primitive together with the `next-needed-block` primitive would form a syntactically correct invocation of the `place-block-on-stack` function with its formal parameter being bound to the result of the `next-needed-block` function. In this example, a single primitive directly corresponds to an action or object of the problem domain but this is not necessary. Instead, the correspondence can be achieved by using a primitive set from which combinations of primitives correspond to actions or objects.

The set of primitives should be "closed" in the following sense: all primitives which use parameters must be able accept as actual parameters any primitive in the set or its result.

A "fitness function" is a measure that can be applied to any program that is a candidate solution simply by executing the program with the inputs of every test case bound one at a time to its input variables and then measuring, one at a time, how similar its computed outputs are to the desired ones. The smaller a program's fitness, the more different its outputs are from the desired ones. The only feedback available to the program discovery process is a program's fitness value. There is no feedback which details what a program has done wrong, how a program behaved in its executions, nor what a program has done correctly. A program that meets all the functional requirements by virtue of solving all the test cases has perfect fitness.

The objective of the GP algorithm is to find a program of perfect fitness in the space of candidate solutions using only fitness values as information to adapt its search process.

Because this thesis is concerned with the analysis of GP, from this point onward we shall adopt its particular and restricted terminology and framework of inputs and task as our definition of program discovery. This also improves the clarity of the document. The reader should bear in mind that a much broader definition prevails beyond the scope of this thesis.

An Analogy with Jigsaw Puzzles

Program discovery can be loosely thought of as the assembly of a jigsaw puzzle. What makes the problem more difficult is that, except that the pieces must fit together, there are only a few stipulations on the final shape of the puzzle; it is not necessary to use all of the available pieces; and, instead of assembling a picture like the target already provided, one must create a picture from any of the available pieces that meets a set of hidden functional requirements.

In other words, think of solving a puzzle using only some or all of the pieces you are provided with. Your goal is to eventually assemble a puzzle that meets a certain set of broad conditions, for example, it is "pleasing to the eye" and "bright". The complication is: you are not told, nor do you know, what picture characteristics (e.g. patterns, colour) will accomplish this. After every assembly attempt, you can stop and ask an external viewer to take a look. Your feedback is simply a grade from the viewer that indicates how close your puzzle comes to meeting the criteria.

You must continually adapt your current puzzle by interpreting the feedback in order to eventually provide a correct one, i.e., a puzzle that is “pleasing to the eye” and “bright”.

In the case of the jigsaw puzzle, the picture on the puzzle box presents the goal, there is a box of unassembled, mixed up pieces and the completed puzzle is the solution. From one point of view, the completion is simply all of the pieces fitted together - protruding knobs fitted into sockets, and, from another, it is a picture.

For program discovery, the target could be, for example, a sorting function depicted in terms of inputs which are unsorted arrays that exist prior to running the program and outputs which are arrays of the same elements, but sorted, expected after executing the program. The box of mixed up pieces is replaced with a sample of each primitive. Like a puzzle piece, each primitive has a “knob” corresponding to its function or variable name and “sockets” corresponding to each parameter it needs when it is invoked. A program can be composed by starting from one primitive and recursively enumerating all of its parameters using any of the primitives. Any primitive can be duplicated in any amount. The recursion ends when only primitives without parameters are left. In terms of puzzle assembly, a socket is filled by a knob and assembly is finished when no sockets are left unfilled in the eventual program. In order to assemble, one has the additional liberty (granted to facilitate the puzzle analogy) that, as one piece is fit into another, the new compound piece can be shrunk to the standard size of a single piece. This enables further compound assembly under the constraints. The assembly process, conducted in this manner, will always result in a syntactically correct (and executable) program. In program discovery, only the syntactically correct programs that do not exceed a (parameterized) maximum depth (measured in the number of nested primitives) and that can be generated from the initial set of primitives are considered as candidate solutions.

The goal of the assembly process is to obtain a program that sorts correctly. A program which accomplishes this, using primitives from the set, under the constrained rules of assembly, is the solution. From one point of view, the program is simply primitives fitted together, and, from another, it is a specification of function invocations and parameters that, when executed, meets the target behaviour.

1.2. Solving The Problem of Program Discovery

The way in which program discovery is formulated as a problem seems to present major obstacles. Fitness provides the only feedback to the algorithm. There is no additional information concerning what was wrong or correct in a program's execution or output. For people, programming requires directed design effort which is acquired only after long training. Novices find it difficult to design algorithms when they are introduced to programming, and software engineering requires experience. Furthermore, programs are brittle. Programmers, regardless of their experience, recognize that small flaws (syntactic or semantic) result in incorrect programs. Often minor changes are all that is required to drastically alter the behaviour of a program and correct it from being entirely useless to being quite productive. Random changes to a program almost never help to debug it. An algorithm which does not exploit knowledge of programming semantics must use what amounts to blind changes. How is it possible to adaptively search when the search process has no knowledge of the connection between program and execution?

Despite the seemingly formidable obstacles, the Genetic Programming algorithm can accomplish program discovery. This adaptive search algorithm was designed by John R. Koza [62]. GP is an evolution-based search model that is a descendent or specialized kind of Genetic Algorithm (GA). For this reason, we immediately describe GAs and introduce GP.

1.2.1. Genetic Algorithms (GAs)

GAs were designed by John Holland [46] and were motivated by his interest in the general class of problems where "information must be exploited as acquired so that performance improved apace" [46, pg. 1]. One way to solve this class of problems is to employ a realizable general strategy that both exploits the best tested options and reduces the chance that other, as yet untested, options may be better. Holland calls such a strategy an *adaptive plan*. In an adaptive plan, structures in a search space are selected and progressively modified by operators according to the quality of their performance in past trials.

GAs are adaptive plans whose structures and operators can be interpreted by concepts borrowed from genetics and evolution. The adaptation of structures in a

GA follows a simplified version of evolutionary refinement. In a GA, a set of candidate structures is a population, each structure is an individual or genotype, and adaptive operators function like simplified computational versions of selection, crossover and mutation. A genotype represents the expressions of features in the structures of the problem domain. A commonly used encoding for a genotype in a GA is a fixed length bit string.

A GA using a fixed length bit string encoding is described in pseudocode in Figure 2. The algorithm processes a population of individuals which is usually initially generated at random. It iterates in a step called a generation. In the course of a step, pairs of individuals are chosen with fitness proportionate selection to act as parents of offspring which will form the next generation. Fitness proportionate selection is a computational abstraction of the rule of "survival of the fittest". An individual is selected from the population to be a parent with probability proportional to it fitness relative to the average fitness of the population. Each individual, once decoded, can be evaluated using an objective function which measures how well it "fits with the environment" (i.e., how well it performs the required task). This information is called "fitness".

```
PROGRAM genetic-algorithm-pseudocode

CONSTANTS
m                       :integer              /* population size        */
mutation-probability    :real in range (0,1]  /* likelihood of mutation  */
crossover-probability   :real in range (0,1]  /* likelihood of crossover */
perfect-fitness = k     :real or intger
bits-in-individual = b  :integer              /* length of bit-string    */
max-generations = n     :integer

GLOBAL VARIABLES

generation = 0          :integer

next-generation,
population              :array[1..m] of bit-string
pop-fitness             :array[1..m] of integer or real

population-summed-fitness,
population-average-fitness :real
perfect-solution = false   :boolean

SCRATCH VARIABLES
parent1, parent2,
```

```
child1, child2:         :bit-string


Procedure FITNESS (var population, pop-fitness, population-summed-fitness,
                       population-average-fitness, perfect-solution)

/* evaluates fitness of each individual in population
   calculates population average fitness
   checks for perfect solution  */

local vars:  individual, sum = 0

do individual = 1 to m
       /* fitness is a positive non-zero value */
      pop-fitness[individual] := adjusted-objective-function(population[individual])
      sum := sum + pop-fitness[individual]
      perfect-solution := pop-fitness[individual] = perfect-fitness
      population-average-fitness := sum / m
endloop
end procedure FITNESS

Procedure INITIALIZE-POPULATION (m, var population)

/* set up generation 0 */

local var:  individual

do individual = 1 to m
      population[individual] :=  /* a random bitstring */
endloop
end procedure INITIALIZE-POPULATION

Procedure CROSSOVER (parent1, parent2, var child1, child2)
/* crosses over parents to yield two children */

local vars:
      random-value = RAND(0,1)
      crossover-point = RAND(0, bits-in-individual)

if random-value <= crossover-probability
   then
      child1[1..crossover-point] := parent1[1..crossover-point]
      child1[crossover-point+1..bits-in-individual]:=
            parent2[crossover-point+1..bits-in-individual]

      child2[1..crossover-point] := parent2[1..crossover-point]
      child2[crossover-point+1..bits-in-individual]:=
            parent1[crossover-point+1..bits-in-individual]
   else
```

```
      child1 := parent1
      child2 := parent2
endif
end procedure CROSSOVER

Procedure MUTATION(child)
/* flips each bit of child with mutation-probability */

do i= 1 to bits-in-individual
      if RAND(0,1) <= mutation-probability then
         child[i] := not child[i]
      endif
enddo
end procedure MUTATION

Procedure FITNESS-PROPORTIONATE-SELECTION
      (population, population-fitness, population-average-fitness,
       population-summed-fitness,
       var parent)

local vars:
   rand-value = RAND(1, population-summed-fitness),
   sum = 0, index = 1

while index <= m
      sum := sum + population-fitness[index]
      if rand-value <= sum
         parent := copy(population[index]
         break loop
      endif
endloop
end procedure FITNESS-PROPORTIONATE-SELECTION

/* MAIN PROGRAM */
begin

initialize-population(m, population)

/* calculate the fitness of this generation */
fitness(population, pop-fitness, population-summed-fitness,
        population-average-fitness, perfect-solution)

/* loop per generation and assemble next generation */

while (NOT perfect-solution) AND (generation <= max-generations)

   generation ++

   for new-individual = 1 to (m / 2)
```

```
      /* select two parents */
      fitness-proportionate-selection(population, population-average-fitness,
                                      population-summed-fitness, parent1)
      fitness-proportionate-selection(population, population-average-fitness,
                                      population-summed-fitness, parent2)

      /* crossover over two parents to get two children with some probability */
      crossover(parent1, parent2, child1, child2)

      /* mutate each bit of each child with some probability  */
      mutate(child1)
      mutate(child2)

      /* store the children  */
      next-generation(index) := child1
      index++
      next-generation(index] := child2
      index++
   endfor

   /* calculate fitness of next generation */
   fitness(next-generation, population-average-fitness,
          population-summed-fitness, perfect-solution)

   /* update new generation to old */
   population := next-generation

endloop

report perfect solution or best of final generation

end MAIN PROGRAM
```

Figure 2. Genetic Algorithm (GA) Pseudocode

An offspring or child inherits its “genetic” specification as a combination of its two parents subject to a background influence of mutation. Two offspring typically are formed from a pair of parents using crossover. Crossover makes each offspring a combination of its parents by exchanging alleles amongst them probabilistically. See Figure 3 and procedure “crossover” in the pseudocode of Figure 2 for an example of one GA crossover operator named “single-point crossover”. The mutation operator probabilistically changes an expression of a feature randomly. (procedure mutation in Figure 2).

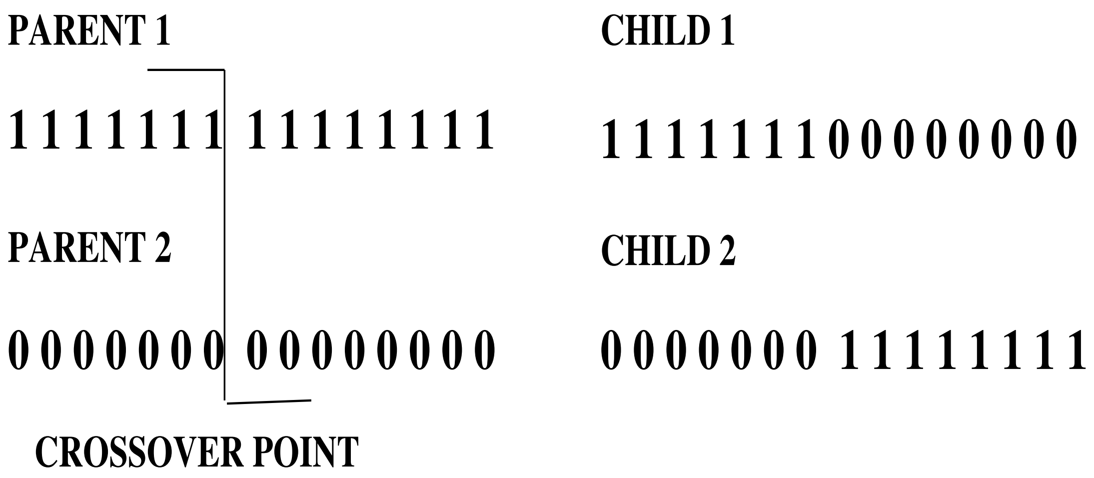

Figure 3. GA single-point crossover on a fixed length binary string

The iterative process of successive generations continues until some *a priori*, externally established criterion for termination is met. When GAs are used for optimization the criterion is either that an individual of perfect fitness has been found or that a maximum number of generations has been reached.

The major parameters of a GA are population size, probability of crossover, and the probability of mutation. The actual problem determines the features (and their expressions) that are encoded in individuals and the objective function.

The GA is a search algorithm because, as it successively forms new populations using fitness proportionate selection, crossover and mutation, it conducts a search over a space of genotypes under the guidance of an explicit optimization goal. The wide range of alternative representations supported by encoding allows a GA to be used in a variety of problem domains. De Jong was the first to demonstrate that GAs are robust adaptive plans: using the same parameter settings for population size, mutation rate and crossover rate he showed that a GA could solve a diverse range of optimization problems characterized by different search landscapes [20]. GAs have been compared to Evolution Strategies [95, 104] and Evolutionary Programs [26] in [10] on the basis of their performance on parameter optimization. GAs have been used in many different ways, e.g. [36, 37, 103, 105, 12, 78, 27, 17, 9]. De Jong and

Holland amongst others have stressed that the general adaptive capabilities of a GA make it suited for a wider variety of problems than just optimization [46, 22].

1.2.2. Genetic Programming (GP)

GP is a GA designed to solve program discovery[1]. Populations of programs are evolved according to their performance on an externally imposed fitness criterion. GP uses fitness proportionate selection, a revised genetic crossover and no mutation[2]. It is a "weak" method that can be applied a particular problem by choosing primitives, designating a test suite, and designing a fitness function. When we refer to GP in the course of this thesis we mean its canonical version. We introduce the salient features of GP immediately and shall describe it fully in Chapter 2:

- GP genotypes or individuals are programs. A program is directly manipulated by crossover and evaluated for its fitness value without encoding or decoding. Using GP with the LISP programming language [3] [112] a genotype is an S-expression. The benefit of direct evaluation is that it forestalls the representation from imposing any expressive constraints to support encoding and decoding.

  GP programs are not fixed in length or size. The maximum height of the parse tree of a program is *a priori* specified to constrain the search space but all solutions up to and including this maximum are considered. Other examples of GAs exist which also use variable length genotypes [110, 106, 34, 43, 44] so GP is not unique in this respect but this feature is useful for program discovery.

- reproduction is standard: parents are selected with a probability based on their fitness and the average fitness of the population.

- GP crossover uses a parse tree representation for a program which is as a rooted point-labeled tree with ordered branches. For example, in LISP, the name of

[1] The substitution of "Programming" in the acronym "GP" for "Algorithm" in the acronym "GA" is somewhat misleading because Programming refers to the goal of program discovery whereas "Genetic Algorithm" is a name for an algorithm. Analogously GP might be named "GA for Programming".

[2] This is true of the GP introduced by Koza in [62] but subsequent extensions vary in this respect.

[3] The choice of LISP, though not essential, is felicitous.

the S-expression is the root of the tree and the parameters of the S-expression (which may recursively be S-expressions) are the ordered children of the root (see Figure 4. In GP crossover the nodes in each of two parent programs are numbered in depth first search order and then two values, each in the node number range on a parent, are randomly selected as the crossover points with a 90% probabilistic bias towards non-leaves. The two subtrees of the programs, each rooted at the node designated by the crossover point, are swapped. Figure 5 gives an example. The GP crossover operator allows a genotype to differ from its parents in structure (size and/or shape).

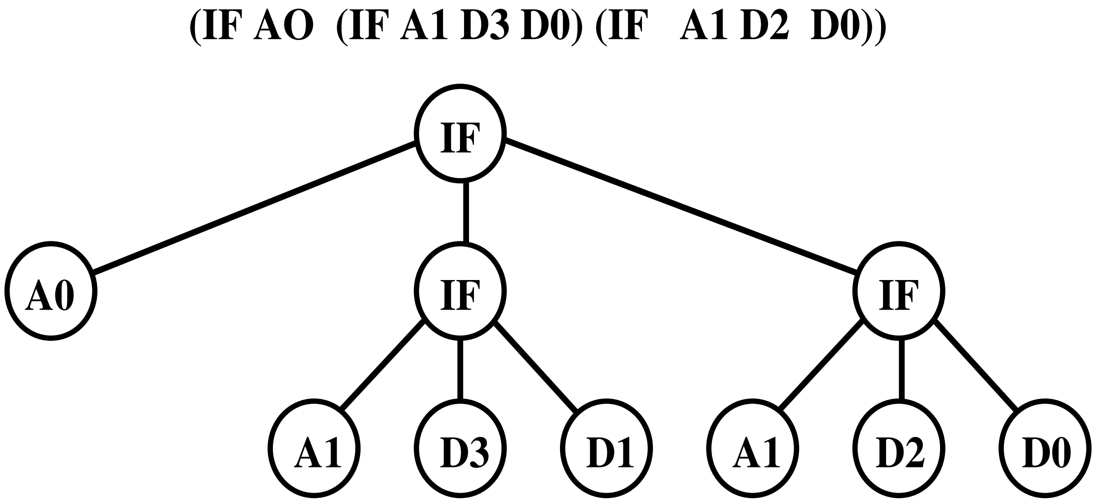

Figure 4. A LISP S-expression represented hierarchically by its parse tree

GP is not the first attempt to use abstracted mechanisms of evolution for program induction [16, 30, 29], nor is it the first non-string-based, variable length GA approach to evolve "programs" [110, 40] but, for a number of reasons, we choose to study it.

First, GP has been demonstrated to be robust. The standard GP we have described, successfully performs program discovery on a wide range of problems. This facet of robustness was established by using GP with exactly the same parameter settings on a multitude of tasks and showing that it could find solutions for each of

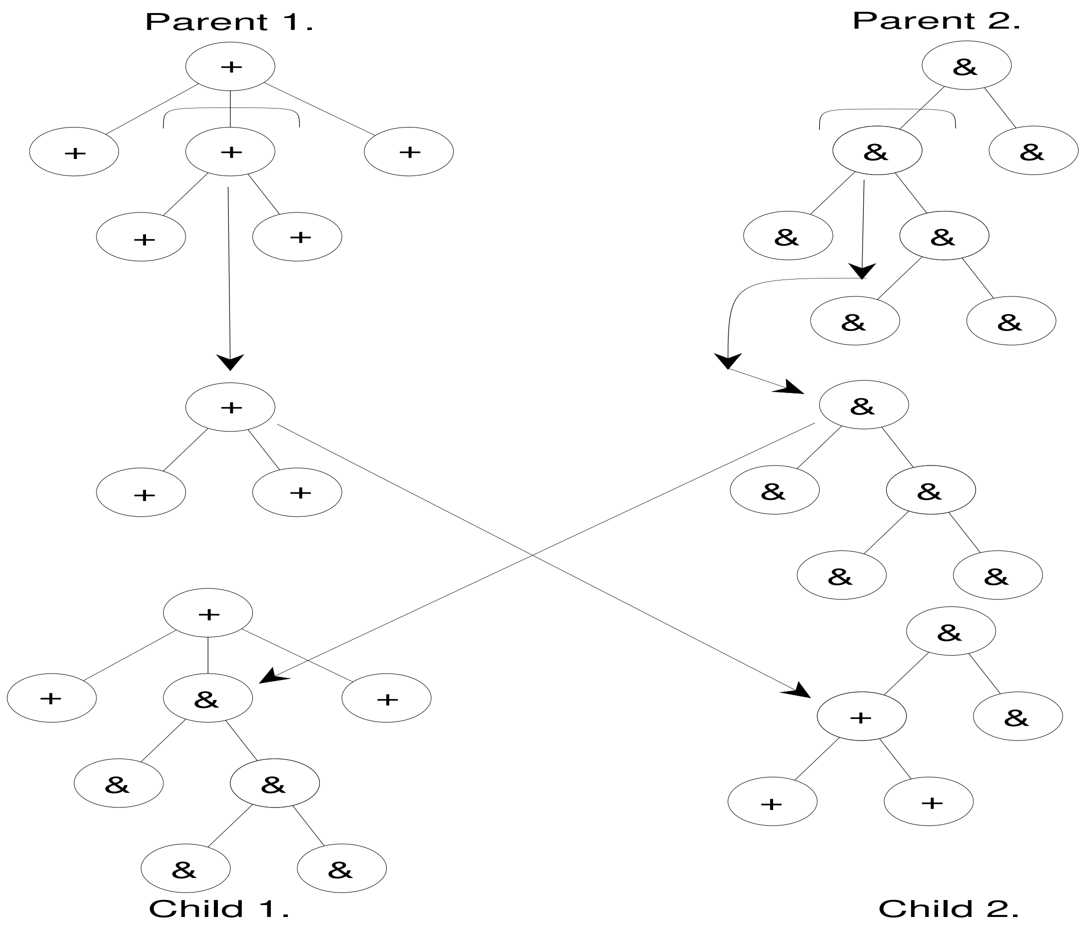

Figure 5. GP Crossover: The subtrees at the crossover points are swapped between parents.

them [69] [4].

Also, GP has yielded many results. Published accounts of GP applications include:

- a program that classifies a given protein segment as being a transmembrane domain or non-transmembrane area of the protein. [71]

- a controller for a software agent performing a corridor navigation task in a simulated two-dimensional environment where the inputs to the program are

[4]Not necessarily for all runs but at least some.

sensor readings supporting some primitives performing obstacle recognition and other primitives directing movement. [97]

- a program that specifies the node structure, interconnection and weights of a sigma-pi or minimal multilayer perceptron neural network. [126]
- an optimal annealing schedule and its parameters so that the same schedule can be used in solving the Quadratic Assignment Problem for high quality solutions [119]
- a deterministic finite automaton which functions as a language acceptor for regular languages. [24]
- a set of rules that decide on the basis of pixeled border features whether an optical character is a certain letter. [4]
- an optimized topical information query in a document retrieval system resulting in improved precision (percentage of documents retrieved that are relevant) and recall (percentage of relevant documents retrieved) [74]
- a detection algorithm for the cores of alpha-helices in protein sequences [42]
- a filter separating the noise from a signal where the filter is a final stage output filter for a rheometer (measurement of blood flow in the skin). [86]
- a program that decides whether a word in a given context has sentential or discourse meaning. [107]

It has been shown convincingly that the GP paradigm is general and provides a single, unified approach to many seemingly different problems in an astonishing variety of areas [69].

Second, while there are other non-string based, variable length GA approaches which evolve algorithms ([110, 40]), these "programs" use if-then rules, rather than the more expressive primitives used by GP. In GP, with some maneuvering, primitives can directly express recursion, assignment, iteration, conditional execution, arithmetic operations, data structure access and general symbolic manipulation.

Third, because it works with primitives and forms programs, the representation of GP is more flexible than the representations of Learning Classifier Systems (LCS) or neural networks. An LCS must represent its solutions as activation chains of fixed length classifiers (if-then rules) that operate in a message-list based paradigm. A neural network must express a solution in terms of weights and network connections. In LCSs and neural networks, the representation of solutions has no further flexibility. This means that both paradigms require a solution to "fit to" their specialized structures. If it does not, obtaining a solution will not be simple; it may be awkward or even impossible to obtain. When it is known beforehand that a neural net or LCS is particularly well suited for a problem, the appropriate system is obvious. However, when the choice is not obvious for a problem, GP is advantageous because many semantically rich programming constructs can be included in the primitive set and tried in combination.

1.3. Goal of the Thesis

The goal of this research is to provide a systematic analysis of GP that improves upon our understanding of how and why GP works. Furthermore, we wish to improve GP. In order to use comparison to extend our understanding, we design alternative program discovery algorithms. These offer insight into how to improve GP.

1.3.1. Assessing the Roles of the Designer and Hierarchy in GP

In Chapter 3 we start pursuing our goal using an experimental perspective. We first try to gauge how much GP's success depends upon designer skill in choosing the primitive set, test suite and fitness function. This is accomplished by using GP to solve a novel problem and describing the typical issues that are encountered while also performing some minor experimentation.

We then proceed to experimentally evaluate the presence of a hierarchical process in GP. There is a distinction between a **hierarchical process** and a **hierarchical solution**:

- A **Hierarchical process** identifies and promotes useful primary elements, combines them into composite, modular, reusable, and successively higher level components of a hierarchy, and the guides high level component assembly into a hierarchical solution.

- A **Hierarchical solution** has a combination of hierarchical structure and control. Hierarchical control is the execution of a task through the accomplishment of a series of subtasks. Subtasks can themselves be recursively subdivided into subtasks again and again. Sometimes, different subtasks are achieved by invoking a general purpose module which may be parameterized in order to perform the task with subtask specific data. This "true" hierarchical control should not be confused with the superficial execution branching that takes place when all programs are run.

  In programs, hierarchical structure is the existence of nested levels of procedures and functions. For example, all elements (e.g., global variables, procedure, functions) in the basic template of a PASCAL program comprise the "outermost" level and then, recursively, each procedure or function of the top level can be comprised of local variables and nested procedures as well as statements to enhance hierarchical structure. One consequence of hierarchical structure is scope. Nested variables and procedures only have scope within their enclosing procedure or function.

We suggest six reasons for conjecturing GP proceeds in the manner of a hierarchical process.

Reason 1. Hierarchical solutions are typically products of a hierarchical process. Since some of GP's solutions have subtle hierarchical strategy, a hierarchical process may be responsible for them.

Reason 2. A hierarchical process allows solutions to be found because it introduces efficiency into the search process. GP may be exploiting a hierarchical process when it manages to find solutions in very large search spaces.

It is easier and more efficient to solve a problem using hierarchical structure and control and by using a hierarchical process than doing so without. A subtask is simpler and, thus, easier to correctly complete than its more complex subsuming task. Once subtask components exist, it is only necessary to define their assembly and debug the combinative aspect of the solution. As well, if the same subtask is required more than once, the component functioning in its capacity can be copied or reused. In all, the identification of subtasks, their solution and the bottom-up combination allows structures of greater complexity to arise simply and efficiently.

Reason 3. GP crossover depends upon a hierarchical representation of a program. It swaps subtrees between parents. It is plausible to assume that a subtree commonly expresses some logical subtask. If this assumption is correct, crossover will explore new contexts for subtasks and possibly hit upon using them correctly in combination. In other words, by swapping the subtrees of two parent S-expressions to form two children, GP crossover may play an integral role in the exploration and combination of hierarchical sub-control.

Reason 4. Human computer program design requires a hierarchical process, therefore, it may also be a requirement of GP program discovery. Recall that introductory programming courses teach program design as either a top down decomposition, bottom up composition, or as both processes interleaved. Each process is a reliable means of efficiently designing programs. The hierarchical design processes of humans conducting programming suggest ways in which GP could manage to evolve hierarchical solutions.

Reason 5. Hierarchical processes are ubiquitous in evolution. Perhaps, GP's simplified model of evolution is a hierarchical process. GP abstracts four essential features of evolution: blind variation, survival of the fittest, inheritance and an extensible genotype by means of its crossover operator, roulette-wheel selection, parent selection and program representation. Could the synergy of these simple factors, which are as completely bereft of knowledge as their corresponding principles in nature, direct a hierarchical process?

Reason 6. Because GP is a specialized GA, the implicit parallel search on hyperplanes and building block behaviour that is hypothesized to occur in GAs with binary fixed length strings may occur in GP. The GA Building Block Hypothesis [32] states that a GA combines "building blocks", i.e., low order, compact, highly fit partial solutions, to compose solutions which, over generations, improve in fitness. Could it be the case that, first, this Building Block Hypothesis

> is justified for GP, and, second, that the GP "building blocks" [5] that are promoted and combined in the conjectured implicitly parallel hierarchical building block combination process are actually discernible subtask elements that are combined by a hierarchical (i.e., building block) process into a hierarchical solution?

A GP solution may be hierarchical because the primitives chosen for the problem already may implicitly encode the manner in which the task should be decomposed and how each subtask of this decomposition should function. For example, it is possible to give GP a set of primitives where each is a logical subtask of one decomposition of the problem and then simply to use GP to find the correct combination of subtasks rather than expect GP to discover subtasks on it own and then link them together. Therefore, it is important that conclusions on the presence of a hierarchical process in GP are not drawn solely from the existence of hierarchical solutions.

In order to show that GP uses an inherently hierarchical process, the correct question is: can GP evolve a program for some task using a set of primitives that requires multilevel combinations of them to accomplish subtasks and the top-down decomposition of the task into the same subtasks? To elaborate on the primitives: they must not preordain the nature of subtasks that evolve and they must be sufficiently general that GP could assemble them in any number of ways to create a hierarchical solution. We shall call such primitives "general purpose".

In Chapter 3 we detail a set of experiments that answer this question. Our method of experimentation is: for a selected program discovery problem, we start with primitives which are not specific to the problem in question and which can be composed in a variety of ways into subtasks we know can help solve the problem. We see whether GP is capable of solving a problem that would require a hierarchical process to exploit these general purpose primitives. If GP is capable, we further decompose the primitives and re-test GP. If GP can ultimately use the most general primitives to still compose solutions exhibiting hierarchical composition for the problem, it will have been shown, in at least some instances, that GP's process is inherently hierarchical.

[5] We shall define a building block rigorously in Chapter 4.

Our experiments tend to confirm that GP, in its canonical form, does not exploit a hierarchical process to obtain hierarchical solutions. It can not efficiently evolve truly hierarchical programs when it is supplied with primitives that must be hierarchically processed in order to express a solution. When started from general purpose primitives, GP is hierarchical only so far as it discovers solutions of superficial hierarchical control and it only randomly finds a solution that is truly hierarchical in control. In general, this implies that GP is strongly dependent upon primitive selection because, if the primitives are so general as to require a hierarchical process to formulate a solution, GP is generally incapable of discovering one. To ensure GP's success, primitives must be chosen that are sufficiently problem specific that a non-hierarchical process can find a successful combination of them. Furthermore, the initial human design decisions of test suite specification are crucial because propitious selection of incremental subtasks as test cases can help guide the evolution of useful subtasks in the population of programs.

Still in Chapter 3 we then proceed in three directions:

1. We explain that incorporating a hierarchical process into GP would be helpful. Since the time of our experimentation, various research motivated by this goal has been conducted. Three methods: Genetic Library Builder (GLiB) [7], Adaptive Representation GP (AR-GP) [99] and Automatically Defined Functions (ADFs) [70] have been devised. ADFs do not explicitly control a hierarchical process while both GLiB and AR-GP do. GLiB relies upon GP's existing selection and crossover mechanisms to identify and promote useful subtasks among the modules it randomly creates, but it appears these are insufficiently powerful. AR-GP bolsters the GP algorithm by maintaining centralized memory-based information to help it identify small building blocks. It shows early promise but its reliance upon centralized mechanisms conflict with GP's model of local emergent computation.

2. We expand upon the issue of incorporating domain knowledge into program discovery primitives. Using knowledge about the problem domain may be expedient but it may reduce the expression available to solutions and make it difficult to appraise the general power of GP. We examine a validation rule and process one could follow to make the claim of knowledge independence in a

program discovery problem.

We determine that, compared to other machine learning approaches, the primitives used in various GP problems are no more specialized to the specific domain of the problem. Thus the issue of knowledge-based primitives, while significant in all machine learning discussions including GP, does not strongly diminish the *comparative* quality of GP as a general framework (i.e., weak method). GP is relatively no less capable of deriving rich behaviour starting from knowledge-impoverished initial conditions than comparable machine learning paradigms.

3. Given that GP is not a true hierarchical process, it remains necessary to explain GP's success. The tempered and somewhat obvious answer is that GP is a GA where critical choices have been made to suit its goal of program discovery. We list and consider these choices with respect to their necessity, convenience and design issues.

In short, our first finding in Chapter 3 is that the quality of GP's success is significantly influenced by choices that are left up to the designer. Those choices concern the selection of the primitive set, test suite and fitness function. Regarding hierarchy, GP lacks, not entirely obvious, nor readily implementable, mechanisms or principles of evolution that, if present, would give it the power of a hierarchical process. Another reason for GP's success is due to it being a GA that is suitably specialized for program discovery.

1.3.2. A Schema-Based Theoretical Analysis of GP

Continuing with our goal of explicating GP, in Chapter 4 we fully address a conjecture proposed as a possible reason for GP searching hierarchically. Reason 6, page 20, stated that, because GP is a specialized GA, the implicit parallel search on hyperplanes and building block behaviour that is hypothesized to occur in GAs with fixed length binary strings may also occur in GP. We formulate a theory of GP using the schema-based framework suggested by its close similarity to GAs. We define schemas rigorously; derive a Schema Theorem; and examine the strength of a GP Building Block Hypothesis. There have been approximate accounts of GP theory in the literature. For example:

> The set of similar individuals sharing common features (i.e. the schemata) is the hyperspace of LISP S-expressions sharing common features. The overall effect of fitness proportionate reproduction and crossover is that subprograms (i.e. subtrees, sub-lists) from relatively high fitness individuals are used as "building blocks" for constructing new individuals and the search is concentrated for successive populations into sub- hyperspaces of S-expressions of ever decreasing dimensionality and ever increasing fitness. [64, pg. 774]

Our work fills a gap between approximate accounts and a precise schema-based analysis for GP. The analysis draws attention to the aspects in which GP differs from other GAs and thus GA theory may not be inherited exactly. While it is precise, it establishes that precise insights, based upon a Building Block Hypothesis, into how GP works or why GP may be better than other search algorithms are not forthcoming using a schema-based approach.

1.3.3. Understanding GP through Comparison and Improving Upon GP

In Chapters 5 and 6 we extend our systematic experimental analysis of GP by testing it and comparing it to program discovery versions of adaptive search algorithms. We employ a suite of program discovery problems for each of our studies. These problems are:

- `6-Mult`: a 6 bit boolean multiplexer problem
- `11-Mult`: an 11 bit boolean multiplexer problem
- `Sort-A`: sorting an array using a linear fitness function counting mismatches
- `Sort-B`: sorting an array using a fitness function based upon permutation order
- `BS`: the block stacking problem

Section 2.1 of Chapter 2 provides a full exposition of the primitive set, test suite and fitness function of each problem as well as our motivation for choosing each. Section 2.2 describes the GP algorithm we use in our experiments. The algorithms are compared in a fair manner by each being allowed to process the same maximum number of candidate solutions. Chapters 5 and 6 specify the detailed nature of the comparisons in terms of what parameter settings were used.

For each algorithm and problem, on a per run basis, we examine the probability of success, the average fitness of the best individual of each execution and the average number of individuals processed each execution. The latter two values are expressed as percentages for uniformity where 100% is perfect fitness or the maximum number of individuals allowed to be processed (25500) respectively. For successful executions we show the average number of individuals processed and the size and height of the parse trees of successful programs. We use a T-test measuring 95% confidence [91] to state that the difference between two results is statistically significant. Standard deviations for results where there is sufficient data are provided.

In Chapter 5 we present two program discovery versions of existing mutation-based, single-point, search and optimization algorithms: Simulated Annealing (SA) [1] and Stochastic Iterated Hill Climbing (SIHC) and compare them to GP.

Both SIHC and SA are adaptive search algorithms. A *single point*, adaptive search algorithm searches the space of candidate solutions a single candidate at a time starting from a random candidate. The first random candidate is named `current` and its fitness is assessed. Then a mutation of `current` is generated and called `candidate`. A mutation is a modification, typically minor, of `current` made with the overall intent that most characteristics of `current` as a solution are retained while a minor number are changed. Next, the fitness of `candidate` is assessed. Depending upon some acceptance criterion (that usually considers the fitness values of `candidate` and `current`) the search either moves to (or "accepts") `candidate` (renaming it `current`) or generates another mutation based upon `current`. This process of candidate generation, fitness evaluation then the choice of accepting `candidate` is repeated until a perfect solution is found or some maximum number of candidates have been considered. If the acceptance criterion permits the acceptance of a candidate that is inferior in fitness to `current`, the overall fittest candidate is always retained. The pseudocode corresponding to this algorithm is in Figure 6.

In basic hill climbing the acceptance criterion is whether `candidate` is at least as fit as `current`. This criterion can be optionally stricter: `candidate` must be fitter than `current` to be accepted. In Stochastic Iterated Hill Climbing, a limit on the

```
begin Adaptive-Search-Algorithm
     candidates-processed = 0;
     current = random search point;
     fitness-current = fitness(current);
     REPEAT
          candidate = mutate(current);
          fitness-candidate = fitness(candidate);
          candidates-processed ++;
          if acceptance-criterion(fitness-candidate, fitness-current)
          then current = candidate
     UNTIL
          fitness-candidate is perfect OR candidates-processed = limit
end Adaptive-Search-Algorithm
```

Figure 6. Adaptive Search Algorithm Pseudocode

number of candidates generated as mutations of `current` is set. If this limit is reached (i.e. without a candidate being accepted) `current` is randomly re-initialized.

SA ([1]) is an algorithm modeling the physical process of annealing where a solid is liquified and then cooled slowly to obtain a minimum energy state. The minimization of energy based on an arrangement of particles is driven by decreasing temperature which settles the initial random activity of the liquid state. A sequence of state transitions occurs on the way to a minimal state. Those sampled states that result in lower energy are always part of the sequence but a state with more energy only becomes part of the sequence with probability proportional to the temperature of the system and the difference in fitness between it and the state from which it was generated.

The Metropolis algorithm [80] was developed to simulate the physical annealing process. It is based upon Monte Carlo techniques and generates a sequence of states according to energy values and a temperature schedule. New sample states are generated by a slight perturbation of the current state called a "mutation". They are accepted into the minimizing sequence under two criteria. First, they are accepted if they are of lower or equal energy. Second, if they are of higher energy they are accepted according to a probability which depends on the difference in energy between them and the current state and on the system temperature. This same algorithm, in Computer Science named Simulated Annealing, can be used to solve computational optimization problems by substituting states with candidate solutions and the energy value with the value of an objective function which judges the closeness of the candidate to the goal. Problems of maximization have merely to be flipped into minimization problems to use SA.

The typical SA algorithm, one which we use, must be supplied *a priori* with its initial temperature, final temperature and the fraction of maximum candidates that should be sampled at each temperature. The temperature schedule is then calculated to decrease the temperature after each fraction of candidates is sampled according to a rate specified by $exp(\frac{T_f - T_0}{n})$ where n is the number of temperature changes. The decision criterion for acceptance of a candidate solution is to always accept the candidate if it has better (lower) or equal fitness. Or, if the candidate has worse (higher) fitness, it is accepted with probability $exp(\frac{-\Delta fitness}{T})$ where $\Delta fitness$ is the positive fitness difference between current and candidate and T is the current temperature. In Chapter 5 we shall elaborate upon the rationale for the algorithm and summarize its underlying theory.

In order to accomplish program discovery with these adaptive search algorithms, a mutation operator is required. Taking our inspiration from tree edit procedures and tree distance algorithms [102], we have designed the operator "HVL-Mutate" (HVL = Hierarchical Variable Length) which performs substitution, insertion or deletion on a program to transform it to a different program of optionally different length or structure. This operator guarantees the syntactic correctness of a candidate program by exploiting a parse tree representation in a manner similar to GP. Using HVL-Mutate we are able to modify the SIHC and SA algorithms to attempt program discovery and test them on the very same problems we use GP to solve.

We find that on our problem suite these two alternative algorithms also work. SA, on the whole, is arguable superior to GP and SIHC. Our results indicate that adaptive mutation and a degree of localized search are useful for program discovery. Furthermore, based on the success of these algorithms which do not use a population based approach, do not use crossover and do not use "survival of the fittest" selection, a more general reason for GP's (or any other adaptive search algorithm) success at program discovery is its representation rather than its unique implementation of a search strategy. GP considers a large space of candidate solutions where program length and structure can vary. Any adaptive search algorithm that is sufficiently powerful to efficiently search this space is likely to succeed at program discovery. Both GP crossover and HVL-Mutate not only explore this space, they conveniently generate candidate solutions that are automatically syntactically correct.

In Chapter 6 our first goal is to analyze GP crossover. We describe the concept of a *fitness landscape* which was first introduced by Sewall Wright in 1932 ([125]). It allows a graph theoretic formalization of evolutionary search spaces that encompasses computational search spaces including those of program discovery problems. In addition, it provides a way of studying evolutionary dynamics or single point adaptive search. We use the concepts of a fitness landscape as a framework: we analyse GP crossover with respect to its "neighbourhood" size by comparing it to GA crossover. We also analyse GP crossover with respect to a measure we introduce called "syntax correlation". Syntax correlation is defined as a measure of similarity in size, structure and syntax between a parent program and one of its offspring. The syntax correlation of a fitness landscape is the average of syntax correlation for each parent-offspring pair in a fitness landscape. By comparative analysis we estimate that GP crossover yields fitness landscapes with a lower syntax correlation than HVL-Mutate. Hence, relative to HVL-Mutate, GP crossover is a macro-mutation operator. We observe its performance when it replaces HVL-Mutate in SA and SIHC. The algorithms are called Crossover Simulated Annealing, "XOSA", and Crossover Hill Climbing, "XO-SIHC". Our conjecture that the combination of a single point search algorithm and GP crossover is well suited to solve some instances of program discovery problems is borne out. We provide detailed results based upon studying our problem suite.

Still in Chapter 6, in another set of experiments, we exploit both GP crossover and HVL-Mutate so that we can improve GP by adding a local search component. Specifically, we add a stochastic iterated hill climbing component to GP. There are two different hybrid algorithms: one uses GP crossover for hill climbing – "GP+XOHC" and the other uses HVL-Mutate – "GP+MU-HC". There are three versions of GP+XOHC that differ in terms of how a mate is supplied for the current solution. Mates are either randomly created, randomly drawn from the population at large, or drawn from a pool of fittest individuals. On all problems of our problem suite, with either operator used in the hill climbing component, the hybrid algorithm is better than GP alone with at least one of the parameter settings we tried. Prior to using this search strategy, GP had not been superior nor sometimes even on par with SA and SIHC. With hybridization it becomes comparable.

In the final section in Chapter 6 we draw upon a collective framework consisting of three key concepts of adaptive search in evolution to unite all the algorithms considered in the thesis. The concepts are: selection, inheritance and blind variation. By first showing how the GP algorithm expresses each of them, we can proceed to describe how they are actualized in GP alternatives.

In Chapter 7, the conclusion, we revisit the goal and results of Chapters 3 through 6. We list possible directions of future research in the context of the findings of this thesis. Broadening our focus, we remind ourselves that our ultimate goal lies beyond even solving program discovery. Program discovery has been chosen for study because, at this point in time, it is one relevant step on the road to a grander goal. That goal is for a computer-based software system to generate sophisticated behaviour while it operates in an environment that defies *a priori* formal acquisition and formulation of task and environment knowledge.

1.4. Ahead in the Thesis

The remainder of the thesis is as follows:

Chapter 2 describe the experimental suite of problems used in the thesis and provides a detailed description of GP and its extensions.

Chapter 3 assesses how much of GP's success can be attributed to skillful designer choice of the primitives, test suite and fitness function. It also describes the analysis and experimentation establishing whether there is true hierarchical process in GP. . It argues the need for hierarchy in solutions and process in GP and analyses some current extensions made in this regard. It provides a means of claiming that GP has been run with a general purpose set of primitives.

Chapter 4 focuses on the supposition that a GP equivalent of the GA Building Block Hypothesis applies to GP. It introduces a definition of a schema that is suited to GP's variable length, hierarchical representation and the function of GP crossover. It derives a GP Schema Theorem (GPST). Finally, it reviews various interpretations of GPST which, under scrutiny, fail to support a GP Building Block Hypothesis.

Chapter 5 describes the mutation operator HVL-Mutate. It provides adaptations of Simulated Annealing (SA) and Stochastic Iterated Hill Climbing (SIHC) for program discovery. It compares GP, SA and SIHC on the experimental suite of problems described in Chapter 2.

Chapter 6 investigates GP crossover by substituting it for mutation in SA and SIHC: Crossover Hill Climbing (XO-SIHC) and Crossover Simulated Annealing (XOSA). It then describes hybridized algorithms which combine GP and Stochastic Iterated Hill Climbing and which improve upon canonical GP. It compares all these algorithms to GP using the experimental suite of problems described in Chapter 2.

Chapter 7 is a summary, broad outlook and description of possible future directions for this research.