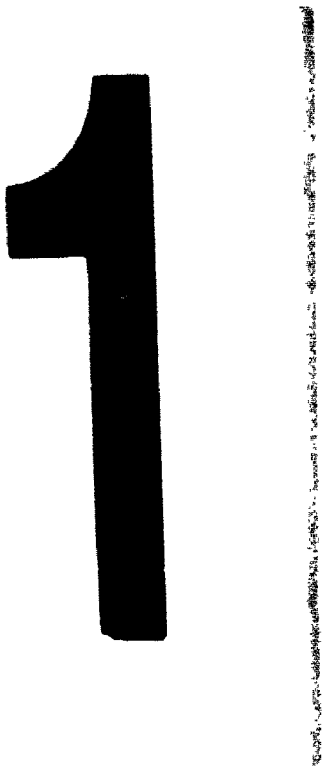
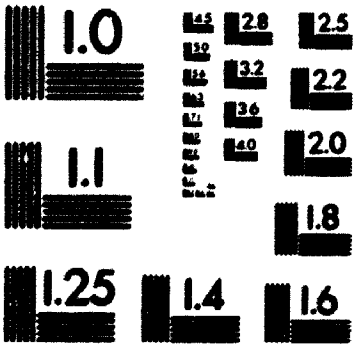


PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT



PRECISION[SM] RESOLUTION TARGETS

An Analysis of Genetic Programming

by

Una-May O'Reilly

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science

School of Computer Science

Carleton University, Ottawa, Ontario

Sept 22 , 1995

ISBN 0-612-08850-2

Canada

Name An Analysis of Genetic Programming by Una-May O'Reilly

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE — SUBJECT TERM

0987 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
- Architecture 0729
- Art History 0377
- Cinema 0900
- Dance 0378
- Fine Arts 0357
- Information Science 0723
- Journalism 0391
- Library Science 0399
- Mass Communications 0708
- Music 0413
- Speech Communication 0459
- Theater 0465

EDUCATION
- General 0515
- Administration 0514
- Adult and Continuing 0516
- Agricultural 0517
- Art 0273
- Bilingual and Multicultural 0282
- Business 0688
- Community College 0275
- Curriculum and Instruction 0727
- Early Childhood 0518
- Elementary 0524
- Finance 0277
- Guidance and Counseling 0519
- Health 0680
- Higher 0745
- History of 0520
- Home Economics 0278
- Industrial 0521
- Language and Literature 0279
- Mathematics 0280
- Music 0522
- Philosophy of 0998
- Physical 0523
- Psychology 0525
- Reading 0535
- Religious 0527
- Sciences 0714
- Secondary 0533
- Social Sciences 0534
- Sociology of 0340
- Special 0529
- Teacher Training 0530
- Technology 0710
- Tests and Measurements 0288
- Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
- Language
  - General 0679
  - Ancient 0289
  - Linguistics 0290
  - Modern 0291
- Literature
  - General 0401
  - Classical 0294
  - Comparative 0295
  - Medieval 0297
  - Modern 0298
  - African 0316
  - American 0591
  - Asian 0305
  - Canadian (English) 0352
  - Canadian (French) 0355
  - English 0593
  - Germanic 0311
  - Latin American 0312
  - Middle Eastern 0315
  - Romance 0313
  - Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
- Philosophy 0422
- Religion
  - General 0318
  - Biblical Studies 0321
  - Clergy 0319
  - History of 0320
  - Philosophy of 0322
- Theology 0469

SOCIAL SCIENCES
- American Studies 0323
- Anthropology
  - Archaeology 0324
  - Cultural 0326
  - Physical 0327
- Business Administration
  - General 0310
  - Accounting 0272
  - Banking 0770
  - Management 0454
  - Marketing 0338
- Canadian Studies 0385
- Economics
  - General 0501
  - Agricultural 0503
  - Commerce-Business 0505
  - Finance 0508
  - History 0509
  - Labor 0510
  - Theory 0511
- Folklore 0358
- Geography 0366
- Gerontology 0351
- History
  - General 0578
  - Ancient 0579
  - Medieval 0581
  - Modern 0582
  - Black 0328
  - African 0331
  - Asia, Australia and Oceania 0332
  - Canadian 0334
  - European 0335
  - Latin American 0336
  - Middle Eastern 0333
  - United States 0337
- History of Science 0585
- Law 0398
- Political Science
  - General 0615
  - International Law and Relations 0616
  - Public Administration 0617
- Recreation 0814
- Social Work 0452
- Sociology
  - General 0626
  - Criminology and Penology 0627
  - Demography 0938
  - Ethnic and Racial Studies 0631
  - Individual and Family Studies 0628
  - Industrial and Labor Relations 0629
  - Public and Social Welfare 0630
  - Social Structure and Development 0700
  - Theory and Method 0344
- Transportation 0709
- Urban and Regional Planning 0999
- Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
- Agriculture
  - General 0473
  - Agronomy 0285
  - Animal Culture and Nutrition 0475
  - Animal Pathology 0476
  - Food Science and Technology 0359
  - Forestry and Wildlife 0478
  - Plant Culture 0479
  - Plant Pathology 0480
  - Plant Physiology 0817
  - Range Management 0777
  - Wood Technology 0746
- Biology
  - General 0306
  - Anatomy 0287
  - Biostatistics 0308
  - Botany 0309
  - Cell 0379
  - Ecology 0329
  - Entomology 0353
  - Genetics 0369
  - Limnology 0793
  - Microbiology 0410
  - Molecular 0307
  - Neuroscience 0317
  - Oceanography 0416
  - Physiology 0433
  - Radiation 0821
  - Veterinary Science 0778
  - Zoology 0472
- Biophysics
  - General 0786
  - Medical 0760

EARTH SCIENCES
- Biogeochemistry 0425
- Geochemistry 0996
- Geodesy 0370
- Geology 0372
- Geophysics 0373
- Hydrology 0388
- Mineralogy 0411
- Paleobotany 0345
- Paleoecology 0426
- Paleontology 0418
- Paleozoology 0985
- Palynology 0427
- Physical Geography 0368
- Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
- Environmental Sciences 0768
- Health Sciences
  - General 0566
  - Audiology 0300
  - Chemotherapy 0992
  - Dentistry 0567
  - Education 0350
  - Hospital Management 0769
  - Human Development 0758
  - Immunology 0982
  - Medicine and Surgery 0564
  - Mental Health 0347
  - Nursing 0569
  - Nutrition 0570
  - Obstetrics and Gynecology 0380
  - Occupational Health and Therapy 0354
  - Ophthalmology 0381
  - Pathology 0571
  - Pharmacology 0419
  - Pharmacy 0572
  - Physical Therapy 0382
  - Public Health 0573
  - Radiology 0574
  - Recreation 0575
  - Speech Pathology 0460
  - Toxicology 0383
- Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
- Chemistry
  - General 0485
  - Agricultural 0749
  - Analytical 0486
  - Biochemistry 0487
  - Inorganic 0488
  - Nuclear 0738
  - Organic 0490
  - Pharmaceutical 0491
  - Physical 0494
  - Polymer 0495
  - Radiation 0754
- Mathematics 0405
- Physics
  - General 0605
  - Acoustics 0986
  - Astronomy and Astrophysics 0606
  - Atmospheric Science 0608
  - Atomic 0748
  - Electronics and Electricity 0607
  - Elementary Particles and High Energy 0798
  - Fluid and Plasma 0759
  - Molecular 0609
  - Nuclear 0610
  - Optics 0752
  - Radiation 0756
  - Solid State 0611
- Statistics 0463

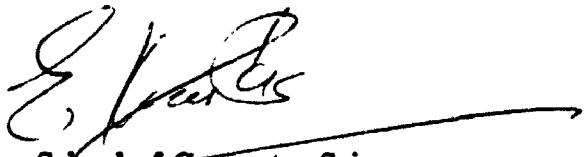
Applied Sciences
- Applied Mechanics 0346
- Computer Science 0984
- Engineering
  - General 0537
  - Aerospace 0538
  - Agricultural 0539
  - Automotive 0540
  - Biomedical 0541
  - Chemical 0542
  - Civil 0543
  - Electronics and Electrical 0544
  - Heat and Thermodynamics 0348
  - Hydraulic 0545
  - Industrial 0546
  - Marine 0547
  - Materials Science 0794
  - Mechanical 0548
  - Metallurgy 0743
  - Mining 0551
  - Nuclear 0552
  - Packaging 0549
  - Petroleum 0765
  - Sanitary and Municipal 0554
  - System Science 0790
- Geotechnology 0428
- Operations Research 0796
- Plastics Technology 0795
- Textile Technology 0994

PSYCHOLOGY
- General 0621
- Behavioral 0384
- Clinical 0622
- Developmental 0620
- Experimental 0623
- Industrial 0624
- Personality 0625
- Physiological 0989
- Psychobiology 0349
- Psychometrics 0632
- Social 0451

The undersigned hereby recommend
to the Faculty of Graduate Studies and Research
acceptance of the thesis

An Analysis of Genetic Programming

submitted by Una-May O'Reilly, B.Sc. M.C.S.
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy



Director, School of Computer Science

Thesis Supervisor

External Examiner

Carleton University

September 22, 1995

An Analysis of Genetic Programming

This thesis analyzes Koza's Genetic Programming (GP) paradigm, a genetic algorithm for program discovery. In order to improve upon our understanding of GP and to improve GP, it provides a systematic analysis of GP that is based upon experimentation and theory.

We assess the role of designer expertise in successfully using GP. Our experiments show that its performance is influenced by propitious designer choices of the test suite and primitive set. We also appraise whether GP proceeds in a hierarchical manner. In experiments with the canonical earliest version of GP, GP did not appear to exploit a hierarchical process.

The theoretical analysis develops a schema-based framework for describing GP search behaviour. We formally develop a Schema Theorem for GP, define building blocks and state a GP Building Block Hypothesis. We proceed to methodically question the plausibility that GP exploits a building block process while searching.

We conduct further experimental analysis by comparing GP to alternative algorithms. A mutation-based operator, HVL-Mutate, that generates a syntactically valid and possibly structurally different program from another is introduced. Two adaptive search algorithms, Stochastic Iterated Hill Climbing and Simulated Annealing, which use either HVL-Mutate or GP crossover are implemented to solve exactly the same class of program discovery problems as GP. The resulting algorithms are comparable to GP and sometimes even outperform it on a small suite of these problems.

Because these algorithms are relatively successful at solving the same problems GP solves, we conjecture that synthesizing a localized search strategy into GP will complement its global, population-based search and improve it. Our experiments with our problem suite confirm this insight. When we hybridize GP by adding a hill climbing component, various versions of the hybrid algorithm achieve higher likelihood of success and process less candidate programs than GP.

To my sister Glenn Crowder
for showing me courage and strength

Acknowledgments

I wish to thank Ken De Jong, Jean-Pierre Corriveau, John Goldak and Stan Matwin for agreeing to serve on my thesis committee.

I would like to thank Franz Oppacher for acting as my supervisor. Franz is an intellectual magnet. I will always be in awe of his creativity, analysis, and comprehension. For seven years he has been a great source of ideas, knowledge and feedback for me. I like to brain-storm with him, argue with him, subject my writing to his critical eye, and simply hang around with him. If I had spent more time with him on this work, it would only have improved.

I would like to thank my family for loving me through all the ups and downs of these five years. My parents, Bob and Glenda invited me into their home again and gave me tremendously valuable support while I restructured my life in 1992-93. When my courage and resolve returned, they confidently waved me off to Santa Fe. They have always encouraged me to engage in activities that test me.

Unfortunately, working at SFI has meant more time away from my brother and sister and their families. I miss them all and thank them for being supportive and patient. My sister Glenn deserves special mention. She is presently engaged in the toughest test of all - fighting for her life, and I am incredibly proud of the tenacity and courage she has shown. Her illness forced me to place my life in perspective at a time when I could have comfortably slipped into the narrow, intensive, isolating endeavors of academics. She has been an inspiration to me and I would trade my Ph.D. for a cure for her cancer.

I started my Ph.D. in Ottawa at Carleton University's School of Computer Science. Among the graduate students, I would like to especially thank Dwight Deugo and Andrew Rau-Chaplin. I don't know if I simply tried to keep up with Dwight and Andrew or whether each of us was equally influential in pushing the others. Whatever, Andrew and Dwight have been true friends. I also especially thank Mark Wineberg and wish him the best of luck with finishing his dissertation.

The School of Computer Science is staffed by excellent people to whom I am indebted for efficiently managing teaching assignments, technical reports, computer administration, etc. as well as patiently replacing my lost keys and forwarding my mail! I especially thank Marlene Wilson, Barbara Coleman, and Rosemary Carter.

Other influential Ottawa people were Nicola Santoro, Daryl Graf, Laura and Larry House, and Tony White. Thank you.

I thank Bell Northern Research for two years of financial assistance.

I spent the greater part of two years conducting my thesis research at the Santa Fe Institute in Santa Fe, New Mexico. It is an unsurmountable task to state how influential SFI has been. I would like to thank Melanie Mitchell and Stephanie Forrest for considering my first (and second) request to visit. I would like to thank Mike Simmons, Vice President of Research at SFI, for agreeing that I should stay on as a graduate fellow once I got there.

SFI has an awesome visitor's program. Crucial interactions with Peter Stadler and Richard Palmer came this way. I profited tremendously from many debates with Bill Macready. How ironic that I had to move to New Mexico to work with someone from Kanata.

Terry Jones first greeted me standing atop a concrete slab, simultaneously juggling and teasing passers by. His unicycle was parked around the corner. He has always offered his help, good cheer and attention. I value it highly. He was a crucial link to a group of Stephanie Forrest's students. I would like to thank them, in particular Ron Hightower and Derek Smith, for allowing me to occasionally join their meetings.

I felt at home at SFI almost from the moment I arrived. I would like to thank Ginger Richardson, Andi Sutherland and Deborah Smith for their warm and friendly welcome. I would also like to thank Andi Sutherland for something extra. Marita Prandoni's cheerful daily greeting was wonderful.

Among the many helpful people at SFI I especially thank Walter Fontana, Raja Das, Nick Vriend, David Wolpert, and Emily Dickinson. I owe a great deal to dear friends Mihaela Oprea (my first in Santa Fe!), Martijn Huynen, Melanie Mitchell, Ann Bell and Kai Nagel.

Last of all, I would like to thank Blake LeBaron. Blake doggedly attacked the technical issues that were obstacles to me working in Madison. He taught me that love doesn't have to always be romantically expressed. Instead, it can be communicated by technical support, objectivity, patience, and the relinquishment of a CPU!

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

The following abbreviations are used in this dissertation:

§x(y) Section x on page y.

AI Artificial Intelligence.

ADF Automatically Defined Functions. §2.5(68)

AR-GP Adaptive Representation Genetic Programming. §3.2(110)

GA Genetic Algorithm. §1.2.1(8)

GLiB Genetic Library Builder §2.5(66)

GP Genetic Programming. §2.2(40)

HVL-Mutate Hierarchical Variable Length Mutate §5.3(145)

SIHC Stochastic Iterated Hill Climbing. §5.1(138)

SA Simulated Annealing. §5.2(141)

XO-SA Crossover Simulated Annealing. §6.2.2(167)

XO-SIHC Crossover Stochastic Iterated Hill Climbing. §6.2.1(167)

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.

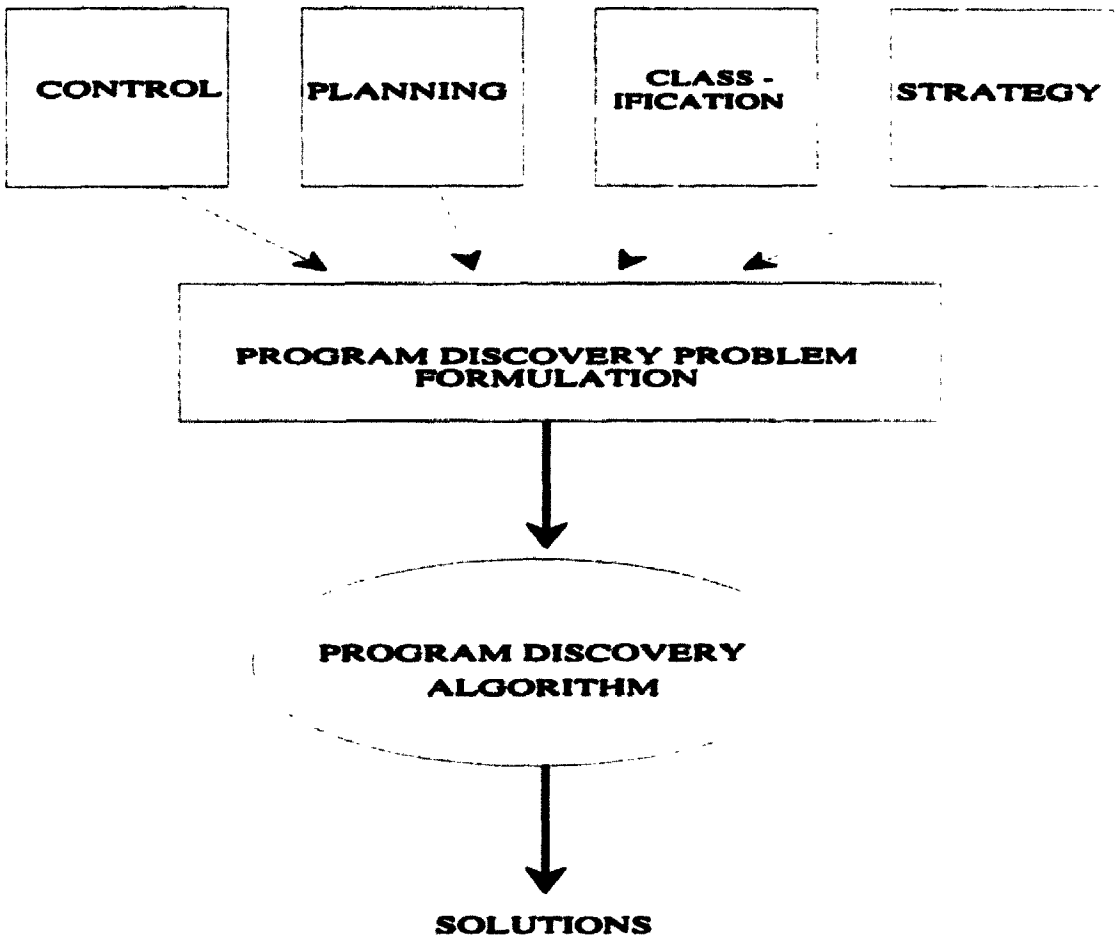


Figure 1. The value of Program Discovery: When many problems can be reformulated as program discovery, a program discovery algorithm is advantageous.

- 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).

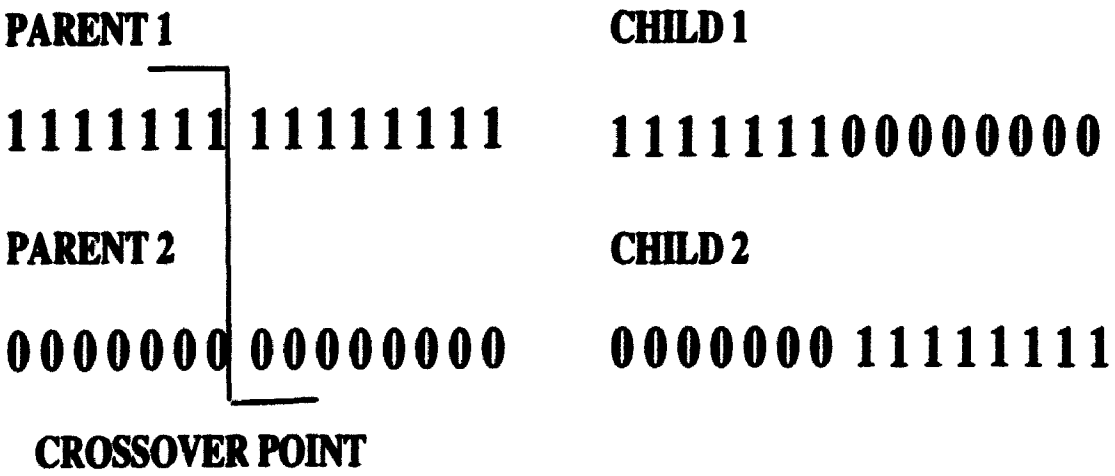


Figure 3. GA single-point crossover on a fixed length binary string

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.

---

[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.

- 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 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).

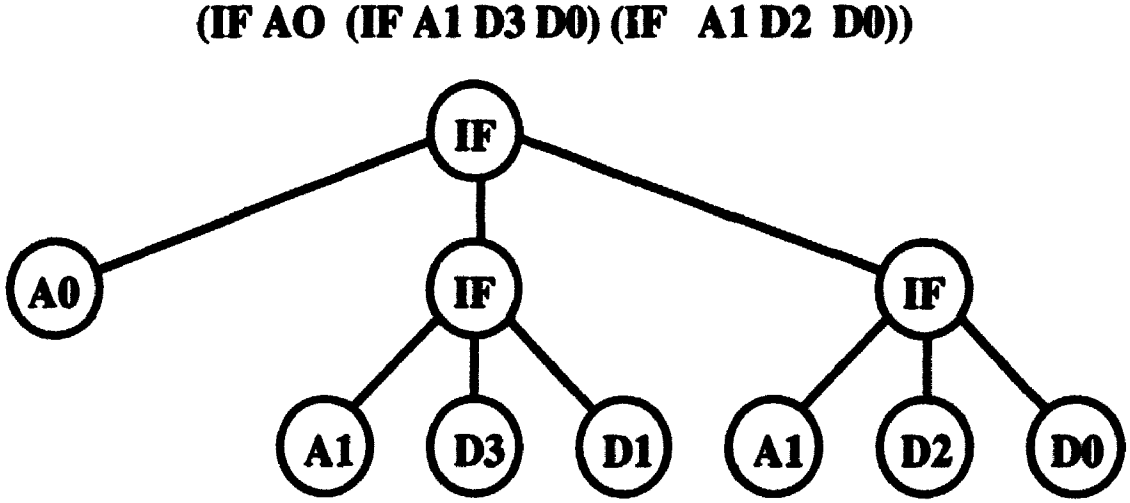


Figure 4. A LISP S-expression represented hierarchically by its parse tree

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

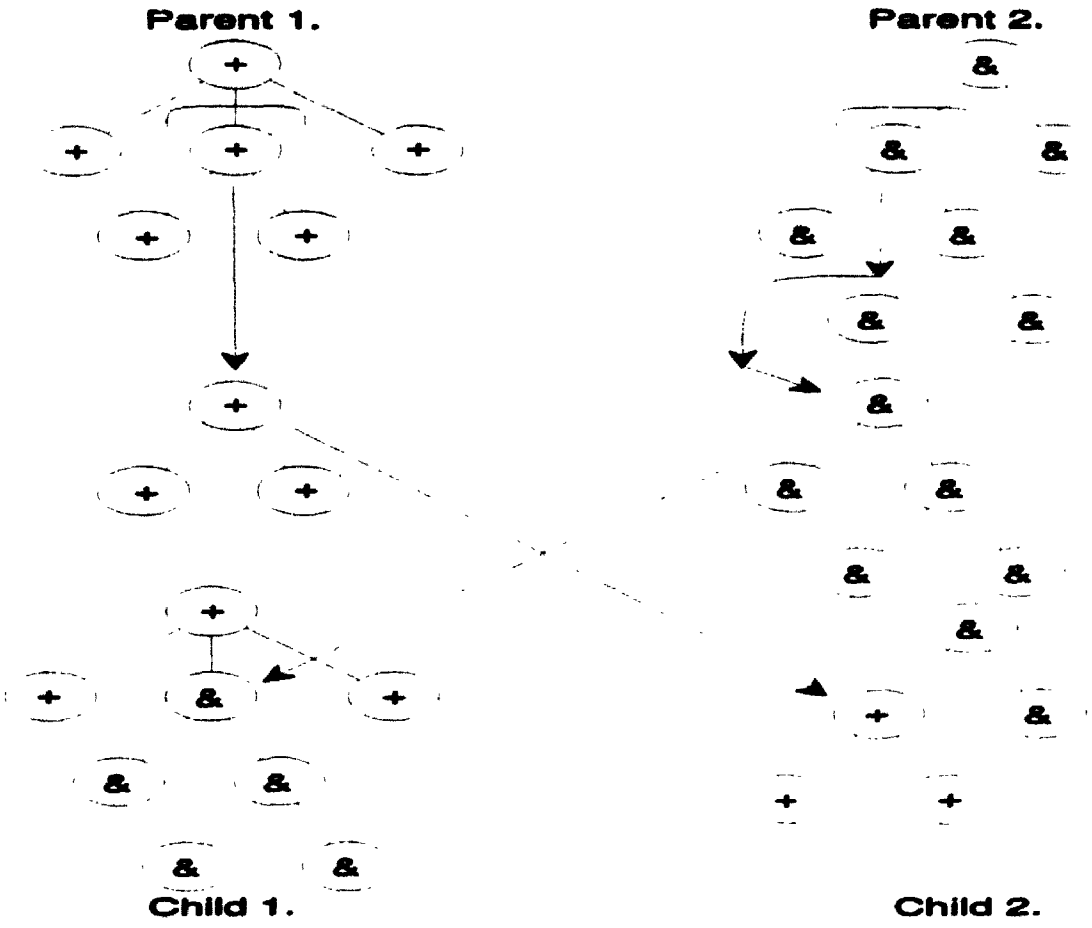


Figure 5. GP Crossover: The subtrees at the crossover points are swapped between parents.

settings on a multitude of tasks and showing that it could find solutions for each of 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]

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

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

prove 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 ruie 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 ii dividuals 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.

CHAPTER 2

Thesis Problem Suite, Genetic Programming, and its Extensions

This chapter starts by motivating and describing the five problems that we use to perform comparison of GP with other program discovery algorithms.

Next, in Section 2.2, an extensive description of Genetic Programming (GP) complementing its introduction in Chapter 1 is presented. Pseudocode of the first and simplest version of GP is presented and various aspects of the algorithm are explained. In Section 2.3 we explain how program behaviour such as iteration, recursion and conditionality can be implemented via primitives.

In Section 2.4, we discuss the salient features of crossover operators. We include a brief description of crossover operators we have experimented with and summarize their comparison. We also describe research on "context-preserving" crossover and "greedy recombination".

In Section 2.5 we describe new genetic operators, selection algorithms and representational extensions that are widely used by the GP research community.

2.1. Thesis Problem Suite

For the purposes of systematic experimental analysis, in this thesis we have selected a suite of five program discovery problems. The suite contains `6-Mult`, `11-Mult`, `Sort-A`, `Sort-B` and `BS`. Varations of these problems which focus upon specific aspects of GP are employed in Chapter 2. The problems exactly as they are stated here are used in Chapters 5 and 6 to test our conjectures concerning algorithm performance and for comparison of algorithms.

2.1.1. Motivation for the Problem Suite

We selected Boolean multiplexer problems (`6-Mult` and `11-Mult`) because they address Boolean concept learning. They also have alternative interpretation in problems of electronic circuit design. The search space of these problems is finite and well understood due to their logical nature. As well, their test suite is finite. The `6-Mult` problem is simple enough to present low computational requirements with GP or, presumably, most other algorithms. The `11-Mult` problem is a scaled up version of `6-Mult` so it (and successively larger multiplexer problems) can be used to assess scalability and efficiency. Boolean multiplexer problems are benchmark problems that have been used in Learning Classifier systems (e.g. [123]), Koza's extensive survey of GP ([69]), and among the growing GP research community (e.g. [124, 100, 47, 53]).

There are good reasons for selecting sorting. It is a highly practical problem and, while we know excellent sorting algorithms, it acts as a general indicator of learning capability. Not only is sorting easy to understand, it is a core problem in Computer Science. The upper and lower bounds on the complexity of sorting algorithms are known and it has been subjected to extensive theoretical analysis. Sorting has an infinite problem space. Using sorting allows the solutions derived by program discovery algorithms to be compared to the diverse set of algorithms devised by humans and to be inspected and analysed for comprehension. Sorting has previously been employed within the realm of evolutionary computation ([45, 13]). We introduced one particular (with respect to primitive set and fitness function) version of the sorting problem to GP literature [88] for the purposes of analysing GP and it has subsequently been picked up and investigated by others [56, 55]. Our two sorting problems, `Sort-A` and `SOrt-B`, differ in only one respect. While they use the same primitive set and test suite, they use different fitness functions. This allows one aspect of the sorting problem to be isolated and observed.

We choose block stacking (`BS`) as a problem because of the long standing consideration in the Artificial Intelligence (AI) community of its related model domain, the *blocks world*. It is a basic task in the planning domain of AI and can also be regarded as a control task. It stimulated new approaches to planning, e.g., [25, 101, 113], and appears to require sophisticated reasoning in the respect that it is *nearly decomposable*. That is, it can be divided into subproblems that only have a small amount of interaction. It is not strictly *decomposable* because the ordering of subproblems is in-

terdependent. It can also be solved more efficiently by taking advantage of primitives that express iterative control.

2.1.2. 6-Mult: The 6 Bit Boolean Multiplexer

Problem Definition: `6-Mult`

Task: *Binary Address Decoding of 2 address bits for 4 data addresses:* The task of a boolean multiplexer is to decode an address encoded in binary and return the binary data value of a register at that address. A 6 bit boolean multiplexer has 4 data registers (at decimal addresses 0 through 3) of binary values and 2 binary-valued address lines. In tandem the 2 address lines can encode binary values "00", "01", "10", and "11" which translate to decimal addresses 0 through 3.

Primitive Set: The primitives that are *a priori* selected for a 6 bit boolean multiplexer are:

`IF(cond, true-branch, false-branch)` This primitive expresses a conditional branch. It has three parameters. The primitive evaluates `cond`. If the result is `non-nil` it evaluates its `true-branch`, otherwise evaluates its `false-branch`.

`OR(parm-1, parm-2)` This primitive expressess a logical `OR` test. It returns the result of logically `OR`-ing `parm-1` and `parm-2`.

`NOT(parm-1)` This primitive expressess a unary `NOT`.

`AND(parm-1, parm-2)` This primitive expressess a logical `AND` test. It returns the result of logically `AND`-ing `parm-1` and `parm-2`.

`*A0*`, `*A1*` These two primitives are variables representing the values of two address lines.

`*D0*`, `*D1*`, `*D2*`, `*D3*` These four primitives are variables representing the values of the four data registers.

Test Suite: In order for a test case to be run on a program, all 6 variables are initialized to test case values. The result returned by a program is interpreted

as its answer to the question: "What is the data value at the address encoded by *A0* and *A1*?" The correct answer is the *a priori* set binary value of the register at the address. A random program has a 50% probability of returning the correct answer even though it may not be functioning anything like a boolean multiplexer.

All 64 possible configurations of the problem are enumerated as test cases.

Out of Training Test Suite: None. The test suite comprises all possible test cases.

Fitness Function: A program's fitness is the number of configurations for which it returns the correct data value for the given address plus one (for positive, non-zero fitness values). This results in values between 1 and 65 where a program with fitness equaling 1 is the least fit and a program with fitness equaling 65 is most fit. A program with fitness 65 is deemed a perfect solution.

2.1.3. 11-Mult: The 11 Bit Boolean Multiplexer

Problem Definition: `11-Mult`

Task: *Binary Address Decoding of 3 address bits for 8 data addresses* The task of a boolean multiplexer is to decode an address encoded in binary and return the binary data value of the register at that address. An 11 bit boolean multiplexer has 8 data registers (at decimal addresses 0 through 7) of binary values and 3 binary-valued address lines. In tandem the address lines can encode binary values "000" through "111" which translate to decimal addresses 0 through 7.

Primitive Set: The primitive set of `6-Mult` is extended to encompass the additional address line and 3 data registers. We list only new primitives here:

`*A2*` This primitive representis the value of the third address line.

`*D4*`, `*D5*`, `*D6*` These three primitives are variables representing the values of the fifth through seventh data registers.

Test Suite: In order for a test case to be run on a program, all 11 variables are initialized to test case values. The result returned by a program is interpreted as its answer to the question: "What is the data value at the address encoded

by *A0*, *A1* and *A2*?" The correct answer is the *a priori* set binary value of the register at the address. A random program has a 50% probability of returning the correct answer even though it may not be functioning anything like a boolean multiplexer.

All 2048 possible configurations of the problem are enumerated as test cases.

Out of Training Test Suite: None. The test suite comprises all possible test cases.

Fitness Function: A program's fitness is the number of configurations for which it returns the correct data value for the given address plus one (for positive, non-zero fitness values). This results in values between 1 and 2049 where a program with fitness equaling 1 is the least fit and a program with fitness equaling 2049 is most fit. A program with fitness 2049 is deemed a perfect solution.

2.1.4. Sort-A

Problem Definition: `Sort-A`

Task: *Sorting.* The task of sorting is to arrange in ascending order the elements in an array. This array shall be denoted in candidate programs by the variable `*array*`. This is the same task as `Sort-B`.

Primitive Set: The primitives that are *a priori* selected for sorting are:

`Do-Until-False` This primitive is an iterative structure. It has one parameter. The primitive continually executes its parameter until the parameter returns `nil`. To prevent the loop from iterating endlessly the primitive cuts off iteration after reaching preset limits defined by variables: `local-loop-limit` or `global-loop-limit`. It updates counters for these limits.

`Swap` This primitive uses three parameters. It returns `nil` if its first parameter is not an array. When its first parameter is an array, and the second and third parameters evaluate to integers in the range of that array's size, it exchanges the elements at the positions corresponding to the integers and returns `true`. Otherwise, it returns `nil`.

First-Wrong This primitive uses one parameter. It returns `nil` if its parameter is not an array or if the array is sorted. Otherwise it sweeps down the array from the first element and returns the index of the first element which is out of order (i.e., is less than an element before it) in the array.

Next-Lowest This primitive uses two parameters. It returns `nil` if its first parameter is not an array and its second parameter is not an integer within the index range of the array. Otherwise it returns the index of the smallest element in the array after the position indexed by the second argument. If no element is less than the element at the position indexed by its second argument, it returns `nil`.

array This primitive is a variable and thus does not have any parameters. It evaluates to the value of the variable.

Comments: In order for a test case to be run on a program, we set both a global variable and ***array*** to reference the array of the test case. When the program execution is completed, the same global variable can be checked for being perfectly ordered (because the array is "shared" by ***array*** and the global variable).

Test Suite: Each test case input is an array that is partially or fully sorted. The array will be bound to the primitive ***array*** and a global variable. The desired (or expected) output of a test case is the array completely sorted in ascending order. This can be confirmed by an *a posteriori* examination of the global variable sharing the binding of ***array***. The test suite consists of 48 different test cases. The array sizes in the test suite range from 2 to 6 (2 arrays of size 2, 6 arrays of size 3, 24 arrays of size 4, 10 arrays of size 5 and 5 arrays of size 6)and among the arrays there are 198 elements in total. Among the test suite are 3 arrays which are initially in sorted order and 47 elements initially in their correct positions.

Out of Training Test Suite: Since sorting is an unbounded problem and GP is inductive, it is impossible to use GP to confirm that a perfectly general sorting solution has been found. Two tests for generalization are possible: first, a program may be functionally verified by a person, second, a different set of test

cases (distinct from the test suite) can be run on a perfect program. While the second set of test cases still can not confirm the program is completely general, at least, it can be decided whether some generalization capacity exists. We coduct a study into this issue in Section 3.1.1.

Fitness Function: Each test case is run and, afterwards, compared to the desired perfectly sorted array. Fitness is a count of element mismatches between the program result and the perfect array. Thus a maximally unsorted array of size n with this metric has mismatch order n and the range of possible fitness values is $O(n)$. The raw fitness of a program is the sum of all the test case mismatch orders subtracted from the maximum possible test case mismatch order sum plus one. The reason for this arithmetic is that mismatch order expresses a minimization function and we prefer to view GP as a maximization process where fitness values are positive and non-zero, so we flip the extrema and add 1. This results in values between 1 and 199 where a program with fitness equaling 1 is the least fit and a program with fitness equaling 199 is most fit. This program is deemed a perfect solution because, for each different binding of `*array*` in the test suite, the original array is always perfectly sorted after the S-expression's evaluation. We refer to this fitness function as "mismatch order".

Note: `Sort-A` and `Sort-B` only differ in which fitness function each uses. `Sort-A` uses mismatch order and `Sort-B` uses "permutation order".

2.1.5. Sort-B

Problem Definition: `Sort-B`

Task: *Sorting.* The task of sorting is to arrange in ascending order the elements in an array. This array shall be denoted in candidate programs by the variable `*array*`.

Primitive Set: The primitives that are *a priori* selected for sorting are the same as `Sort-A`. see page 35.

Test Suite: `Sort-B` uses the same test suite as `Sort-A`.

Fitness Function: Each test case is run and, afterwards, the permutation order of its input array is calculated [55]. Permutation order is a count for each element of the array of the elements that follow it and are lower than it. Thus a maximally unsorted array of size n has permutation order $n*(n-1)/2$ and the range of possible fitness values is $O(n^2)$ rather than linear in n. The raw fitness of a program is the sum of all the test case permutation orders subtracted from the maximum possible test case permutation order sum plus one. The reason for this arithmetic is that permutation order expresses a minimization function and we prefer to view GP as a maximization process where fitness values are positive and non-zero, so we flip the extrema and add 1. This results in values between 1 and 340 where a program with fitness equaling 1 is the least fit and a program with fitness equaling 340 is most fit. This program is deemed a perfect solution because, for each different binding of `*array*` in the test suite, the original array is always perfectly sorted after the S-expression's evaluation.

2.1.6. BS: Block Stacking

Problem Definition: `BS`

Task: *Block Stacking.* The task of block stacking is to stack labeled blocks upon one another in correct order according to a supplied goal list. The task starts from an arbitrary configuration of a stack with other blocks individually placed on a table. Only blocks that are intended to be placed on the stack are available on the initial stack and table, that is, there are no extra blocks.

Primitive Set: `BS` uses 3 "sensors" which are primitives encoded to return state information concerning the stack and goal list. All sensor primitives have zero arguments. It also uses 5 primitives which model operators for manipulating blocks to remove from or place a particular stack on the stack (always on top). There are three global variables that are accessed by the primitives: `*goal-list*`, `*stack*` and `*table*`.

`move-to-stack(block)` Action: If `block` is on the table, places it on top of the stack. Return value: `nil` if `block` is not on table, `true` otherwise.

move-to-table(block) Action: If **block** is on top of the stack, remove it to the table. Return value: **nil** if **block** is not on top of the stack, **true** otherwise.

do-until-true(cond, body) Action: Evaluates **cond** and while it returns false, evaluates **body**. Returns **true**.

not(parm-1) Logical not.

eq(parm-1,parm-2) Logical equivalence.

top-correct-block Returns the top block on the stack such that it and the blocks beneath it are correctly stacked. When, starting from the bottom, no blocks are correctly stacked, it returns **nil**.

top-block-on-stack Returns the block on the top of the stack.

next-needed Returns the block immediately on top of the **top-correct-block**.

Test Suite: Ten test cases have an initial configuration where from 0 to 9 blocks are already correctly stacked and the remaining blocks are on the table. Nine test cases have 0 to 7 blocks already correctly stacked with exactly one incorrect block on top, while the remaining blocks are on the table. The remaining 148 test cases are a random sample of possible initial configurations.

Out of Training Test Suite: None.

Fitness Function: Each test case is run and, afterwards, the global variable ***stack*** which represents the state of the stack is compared with ***goal-list***. If the two match exactly, the test case scores one point. The fitness of a program is the sum of all the test case points plus one. This ensures all fitness values are positive and non-zero. A program with fitness equaling 1 is the least fit and a program with fitness equaling 168 is most fit. This program is deemed a perfect solution.

2.1.7. Format of Experiment Description

In this thesis a **problem definition** consists of:

1. a functional description of the **task** in terms of program inputs and outputs,

2. the **primitive set** to be used to compose candidate solutions.

3. **a fitness function,**

4. **a test suite,**

5. optionally, an additional suite of test cases called the **"out of training test suite"**. This suite can be used to evaluate the generality of a solution that performs perfectly on the test suite.

In this thesis an **experiment definition** consists of:

1. **problem definition**

2. **run parameters:** A run consists of some number of distinct executions of the algorithm, each started with a different seed for random number generation.

   - population-size
   - max-generations
   - any run parameters with non-standard values

2.2. Canonical Genetic Programming

Pseudocode for a canonical version of GP (with specialized implementation details omitted) is shown in Figure 7 which starts on page 42. The GP software system we entirely wrote and use for our experiments is heavily instrumented and includes a multitude of original extensions as well as some implementations of extensions reported in the literature. The first version was among the first systems documented. Simple and specialized versions of GP by various authors later became available on at least one ftp site.

The pseudocode is divided into three parts. Part 1 lists variable types and variables. Part 2 is the Main Program and Part 3 is the procedures and functions.

Part 1: Types and Variables: Variables are in four groups. The first group "problem input parameters" consists of variables that link a GP application (i.e., problem) to the GP "shell". (By "shell" we mean the problem-independent part of

the GP algorithm that creates the initial population of programs, and processes populations by iterating over generations.) These variables are initialized at the outset of a GP run so that the shell can generically access a set of primitives, the test suite and the fitness functions. In addition to the expected variables: **testsuite**, **primitives**, and **fitness-function**, this group includes **testsuite-size** and **max-fitness**.

The second group of variables is parameters guiding the construction of the initial population: "Generation 0". The parameters permit the composition, based upon tree height, i.e., the number (or levels) of function calls, of programs in this population to be stipulated. The variables **max-height-random-tree** (standard value 15) and **min-height-random-tree** (standard value 5) allow the user to set a minimum and maximum height for programs that are randomly created in "Generation 0". The "**initialization-method**" variable of type **initialization-methods** allows different functions to be bound into the shell in order to compose "Generation 0". Three such functions work in the following manner:

full This function ensures all programs are of tree height: **max-height-random-tree**.

grow This function ensures all programs are at least **min-height-random-tree** and up to **max-height-random-tree**. It is the standard function of our experiments.

ramped-half-and-half This function ensures an equal number of trees at each height including and between **min-height-random-tree** and the parameter **max-height-random-tree**.

Also pertaining to the initialization of "Generation 0" is the pseudocode for the random creation of programs described in Part 3 in function **random-program**. It would be invoked by an initialization method and return a program within the height tolerance stipulated by its input parameters: **min-height** and **max-height**. Because no primitive alone can solve the problem, this function starts by randomly selecting from the primitive set, with replacement, a primitive which has parameters. It then recursively draws from the primitive set, with replacement to specify the parameters each primitive needs. When the level of function nesting (via parameter specification) is one short of **max-height** only the primitives without parameters are drawn from. Typically the height (and size) of a program are recorded along with the program. This is convenient information because later the crossover operator is constrained to produce trees within a maximum height.

The third group is variables that parameterize a GP run in the following manner:

population-size The number of programs in the population each generation.

parents-to-copy Each generation, a portion of the next generation of programs consists of exact copies of parents chosen by roulette-wheel selection. This variable stipulates the size of this portion. Its standard value is 10% of **population-size**.

max-tree-height-crossover Because a subtree is removed from the copy of one parent and a subtree of the copy of the other parent is swapped into its place, the new offspring (or child) can differ in height from the original parent. As a means of making the search space finite in size, this parameter imposes an upper bound on the number of function levels in any program. Its standard value is 15.

non-leaf-selection-bias A primitive which has no parameters is called a "leaf". The GP crossover operator can be controlled by adjusting the probabilistic bias it uses to choose a non-leaf (i.e. subtree in a program represented as a tree) over a leaf as a crossover point. For example, the crossover points in a tree are grouped as leaves and non-leaves. If a random number drawn between 0 and 1 is greater than **non-leaf-selection-bias**, the crossover point is next randomly selected from the leaves versus from the non-leaves. The standard value is 90%.

max-generations The maximum number of generations to run GP. GP is usually also stopped before this number of generations if a "perfect" solution is found. Perfection is standardly defined as a program whose fitness equals **max-fitness** (see the first group of parameters) but can also be further defined by additional criteria.

```
PROGRAM  GENETIC-PROGRAMMING-PSEUDO-CODE

TYPE

    initialization-methods = {full, grow, ramped-half-half};
    testcase-input-vars = array[1...inputs];
    testcase-output-values = array[1...outputs];
    program = a composition of primitives such as list or array
    fitness-function = a function that binds program variables to
        testcase input variables of a program, executes it and
```

```
        compares its outputs to testcase output values

RECORD
   primitive-record = {
            primitive
            number-of-arguments}

    testcase-record = {
        testcase-input-vars, testcase-output-vars}


VARIABLES

/* problem input parameters     */
integer:
    testsuite-size,
    primitives-size,

    max-fitness,     /* fitness values will be between 0 and max-fitness */

array[1..testsuite-size]   of testcase-record:         testsuite;
array[1...primitives-size] of primitive-record:        primitives;
array[1...population-size] of probability:             roulette-wheel;

/* GENERATION 0 INITIALIZATION PARAMETERS */

integer:                max-tree-height-random-trees;
integer:                min-tree-height-random-trees;
initialization-methods:     initialization-method;

/* RUN PARAMETERS    */

integer:
    max-tree-height-crossover-program,
    population-size,
    parents-to-copy,
    max-generations;

real:
    non-leaf-selection-bias;

/* RUN VARIABLES         */

integer:
    generation = 0;

array[1...population-size] of program:    current-generation,
                              next-generation;
```

```
boolean: solution-found = false;

/* TRANSIENT VARIABLES */
program:  parent1, parent2, child1, child2;


/* MAIN PROGRAM */

begin
    initialize-problem(testsuite, primitives, fitness-function);

    generation-0-parameters(max-tree-height-random-program,
                            min-tree-height-random-program,
                            initialization-method);

    run-parameters(max-tree-height-crossover-program,
                   non-leaf-selection-bias,
                   population-size,
                   parents-to-copy,
                   population-size);

    initialize-random-number-generator(seed);

    create-zeroth-generation(current-generation,
                             primitives,
                             initialization-method,
                             max-tree-height-random-program);

    population-fitness-calculation(current-generation,
                                   testsuite,
                                   roulette-wheel,
                                   solution-found);

    report-on-zeroth-generation;

    /* generation loop */

    while (generation < max-generations) & NOT solution-found

        /* copy some program directly to next generation */
        for i:=1 to parents-to-copy
            next-generation[i]  := roulette-wheel-selection;
        endfor;

        /* create the remaining offspring for next generation with crossover */
        for i:= parents-to-copy + 1  to population-size by 2
            parent1:= roulette-wheel-selection;
            parent2:= roulette-wheel-selection;
            child1, child2  := gp-crossover(parent1, parent2);
```

```
            next-generation[i] := child1;
            next-generation[i+1] := child2;
        endfor;

        population-fitness-calculation(next-generation,
                                       testsuite,
                                       roulette-wheel,
                                       solution-found);
    endwhile;

    if solution-found
        report-success-and-run-statistics
    else
        report-run-statistics

end; /* main program */


/*              PROCEDURES          */

procedure INITIALIZE-PROBLEM
(testsuite, primitives, fitness-function)
/*
prompts user for testsuite, primitives and fitness-function.
These are usually supplied via functions which can be invoked
to set up the global variables testsuite and primitives.
The fitness-function function would be bound to a function variable
invoked in the calculate-fitness procedure.
*/

procedure GENERATION-0-PARAMETERS
       (max-tree-height-random-program, initialization-method)
/* prompts for and sets these values */

procedure RUN-PARAMETERS
                  (max-tree-height-crossover-program,
                  non-leaf-selection-bias,
                  population-size,
                  parents-to-copy,
                  population-size);
/* prompts for and sets these values */

procedure INITIALIZE-RANDOM-NUMBER-GENERATOR(seed);
/* a run can be recreated by starting it with the
random number generator with the same seed */


procedure CREATE-ZEROTH-GENERATION
```

```
                                (current-generation,
                                primitives,
                                initialization-method,
                                max-tree-height-random-program);
/*
Using primitives, max-tree-height-random-program and initialization method,
places population-size s-expressions into the array current-generation */

for i:=1 to population-size

/* compute the height restrictions on a random program based upon
the initialization method (e.g. ramped half-half, full, grow) */
set-max-min-height(initialization-method, i, min-height, max-height);

/* create the new program with the height restrictions */
program := new-program(primitives, max-height, min-height);
current-generation[i] := program;
endfor
end procedure CREATE-ZEROTH-GENERATION

procedure POPULATION-FITNESS-CALCULATION(
                                        current-generation,
                                        testsuite,
                                        roulette-wheel,
                                        solution-found);
local variables

array[1...population-size] : fitness-array;
pop-avg-fitness : real or integer;

/* procedure body */
assess-fitness(current-generation, testsuite,
pop-avg-fitness, solution-found, fitness-array);
form-roulette-wheel(pop-avg-fitness, current-generation, roulette-wheel,
fitness-array);

end procedure POPULATION-FITNESS-CALCULATION

procedure ASSESS-FITNESS(current-generation, testsuite,
pop-avg-fitness, solution-found, fitness-array)

/*
- Binds the test inputs to the program inputs for each member
  of current generation,
- Executes the current member of the population
- Invokes the problem fitness-function function which compares output or global
  variables affected by execution to desired values for the test case and
  assesses a fitness value put into fitness-array
- accumulates sum for average fitness of population and computes average.
```

```
*/
end procedure ASSESS-FITNESS

procedure FORM-ROULETTE-WHEEL(pop-avg-fitness, current-generation, roulette-wheel
fitness-array)

/*
- Using the population average fitness assesses each member a range of
  non-zero positive probability for being selected as a parent proportional
  to the population average fitness.  These ranges are converted to incremental
  values (by member index) to go in the roulette wheel.   When a random
   number between 0 and 1 is drawn, the roulette wheel is searched for the range
   enclosing it.  THe index corresponding to the range refers to the member of
   population that is chosen to be a parent.
*/

procedure REPORT-ON-ZEROTH-GENERATION;

/*
report average fitness of population, lowest, greatest, sizes and heights etc
*/

procedure SET-MAX-MIN-HEIGHT(initialization-method, i, min-height, max-height)

/* compute the height restrictions on a random program based upon
the initialization method (e.g. ramped half-half, full, grow)

full: all trees have max-height = min-height = max-tree-height-random-program,
grow: all trees have min-height = min-tree-height-random-program,
      and max-height = max-tree-height-random-program
ramped-half-half:  for each height from min-tree-height-random-program
       to max-tree-height-random-program a equal portion of trees are full
       and grow.  */

end procedure SET-MAX-MIN-HEIGHT
```

Figure 7. Genetic Programming (GP) Algorithm Pseudocode

The fourth group consists of variables used to control the main loop of the GP algorithm which iterates in generations (**generation**) over populations (**current-generation** and **next-generation**) that are successively created via roulette-wheel-selection and GP-crossover (**parent1**, **parent2**, **child1** and **child2**) of a certain portion of parents. The iteration stops if the variable **solution-found** ever becomes true.

Part 2: Main Program Within the Main Program there are three basic steps to the algorithm. The first step performs problem parameter, run parameter, "Generation 0" parameter, and random number generator initialization. Successively the following procedures are called:

- `initialize-problem`
- `generation-0-parameters`
- `initialize-random-number-generator`
- `run-parameters`
- `create-zeroth-generation`

The final two sub-steps of the first step are:

1. Each program in the zeroth generation is evaluated with each test case in the test suite to obtain a fitness value for it and to set up a "roulette- wheel". The "roulette-wheel" data structure is used to draw members from a population via fitness-proportionate selection.
2. A report is generated supplying data on the characteristics of "Generation 0".

The second step of the Main Program is a loop for each generation of the run. The loop is controlled by the variables `max-generations` and `solution-found`. Inside the loop the following processing is done:

- A number of programs (`parents-to-copy`) of the current generation are chosen via roulette-wheel selection (i.e., fitness proportionate probabilistic selection) to have exact copies of themselves directly propagated to the next generation.
- A number of programs (equaling `population-size - parents-to-copy`) are created for the next generation by drawing two parents at a time from the current generation via roulette-wheel selection and copying them. A pair of copies (called `parent1` and `parent2`) is then crossed over via procedure `gp-crossover` to create two new members (`child1` and `child2`) of the next generation.

- The fitness of each program in the next generation is calculated and the roulette-wheel data structure is updated to reflect the new fitness proportionate probabilities for selection.

- **next-generation**, replaces the current generation: **current-generation**.

- Generation level statistics are recorded and reported. Run level statistics are updated.

The third step of the Main Program starts after the loop terminates. If a perfect solution has been found, information concerning it is reported. As well, run statistics are reported.

Part 3: Functions and Procedures This part of the pseudocode covers the functions and procedures that supplement the Main Program. Most are self-explanatory by name, the short comments provided in the pseudocode and the descriptions just provided. The function of GP crossover is illustrated in Figure 5 on page 16.

Procedure **form-roulette-wheel** perhaps needs more description here. It must implement the GP and GA notion of "Survival of the Fittest". It does this by giving each member of the population a probability of being selected as a parent (or to have its exact copy directly propagated to the next generation) that is proportional to its fitness, relative to the average fitness in the population. The term "roulette-wheel" is used as an analogy to the process of spinning a roulette-wheel which is split into slices (one for each population member) that are sized according to the relative fitness of a member to the average fitness. The spinning is a probabilistic element and, because the slices are proportionately sized, the selection process is blind.

Canonical GP does not use a mutation operator. One reason is that a general mutation operator i.e., one which can perform substitution of primitives and increase or decrease program length anywhere in the existing program while preserving syntactic validity, is not immediately obvious though we shall provide one later. Mutation operators that only modify entire subtrees or only substitute one primitive for another are not as general but obvious to design. Knowledge-based mutation operators that "tweak" primitives according to their meaning are also available. A variety have been used in reported experiments, e.g. [6, 57] but either they are not accompanied

by a comparison (to not using mutation at all). One comparison, on `6-Multm` does not indicate they produce significantly superior results [62].

One reason why GP may not need mutation is that, unlike other GAs, a primitive is very unlikely to disappear from a GP run. In other GAs a population can become homogeneous at a feature (i.e., all the members of a population have the same bit value at a certain string position) and mutation provides the only means of allowing that feature to be re-introduced into the population. In the non-positional, variable length representation of GP, the probability of a primitive completely disappearing from the population is usually very low (given the population size and the cardinality of the primitive set), so there is no role for mutation in preventing permanent primitive disappearance.

2.3. Examples of Primitive Semantics

A degree of skillful manipulation is required to translate a problem into a GP framework. Most obviously, the primitives must be chosen with the problem in mind. In general, GP is successful because a carefully considered formulation of the problem definition and experiment, that is, one that is amenable to search, is chosen. Often the setup of the problem environment is expedient.

In this section we discuss considerations that arise once the general nature of a primitive has been decided upon (e.g., it will perform iteration, it will express a conditional branch). These issues are not specific to using LISP to implement a primitive but we illustrate them in a LISP context because we evolve and use LISP programs[1] in our GP software system. It is important to note that here we discuss how semantics can be expressed in primitives but not how the choice and design of primitives affects the difficulty of the problem or the efficacy of GP.

2.3.1. Directly Using Built-in Functions as Primitives

The boolean functions AND, OR and NOT, the arithmetic functions +, −, *, `if-then` or `if-then-else` functions, and any function which similarily is built in and can accept a fixed number of parameters, can be directly used as primitives.

[1] More correctly, "programs" in LISP are S-expressions.

For example, a **cond** construct in LISP can be used as a primitive if its number of conditional clauses is fixed.

Restricting a primitive to accepting only a fixed number of parameters is a generally acknowledged convenience of minimal cost (in terms of expressiveness and flexibility). For example, the nesting **progn**s of two parameters still allow an arbitrary number of sequential S-expressions.

2.3.2. "Firewalling" Built-in Functions as Primitives

To ensure closure or prevent the chance of an execution trap, sometimes a function taken directly from the programming language must be extended before it is used as a primitive. The "firewall" extension inspects the types of all parameters to ensure a run-time error will not occur because an operator is applied to a parameter of an unanticipated or incorrect type. For example, zero-protected integer division is commonly written as the primitive **%**. The extension may also use different versions of an operator depending upon the type of a parameter.

2.3.3. Arithmetic Constants as Primitives

Recall that the problem of symbolic regression is to find a mathematical expression of independent variables that computes a dependent variable. This problem and others require the use of arithmetic constants. When using a long range of constants as primitives (or a range of real numbers), it is not sufficient to place them among the other primitives of the set because their quantity will "swamp" the others and bias the composition of random programs (or, in the case of a range of real numbers, they can not be enumerated). Instead, so-called "ephemeral constants" [69] are used. A dummy primitive (e.g., named **int-constant**) is placed in the primitive set and each time it is drawn (in the making of a random program), it is replaced by a constant drawn randomly from the range.

2.3.4. Recursion via a Primitive

A simple way to provide recursion is to allow a program to call itself. This must be implemented a bit circuitously by using a helper primitive named, for example, **self**. **self**) must be placed in the primitive set and it must express the base case

of the presumed recursive function. A program invokes itself when it contains the primitive **self** and **self is evaluated**. In order to make the evaluation of **self** the equivalent of calling the program, before the program is valued, the algorithm names it "on the fly", as a LISP function, and by using a global variable. Suppose **self-defun** is the assigned name for the program,[2]. When the program is tested and the primitive **self** is evaluated, **self**:

1. checks whether any base case to end the recursion exists. If it does, **self** returns the result of the base case. The base case may rely upon the value of **self**'s parameters or some global state.

2. saves all input and state variables that exist in the calling program's execution context.

3. uses its parameters to alter current state, if necessary.

4. evaluates its calling program (i.e., makes the recursive call) by using **funcall** with **self-defun** as a parameter.

5. restores the calling program's execution context.

6. returns the result of the recursive call.

So, in fact, recursive calls with one program are implemented by using an intermediary function that must be included as a primitive. This style of recursive primitive is used in the problem of inducing the formula for the Fibonnacci sequence in [62] from its first 20 elements. In that example, the base case examines the value of a sequence index which is a primitive. If the index is not between 0 and the current global value of the index less 1 it returns 0. An obvious drawback to the implementation of this style of recursion is that it has to be somewhat specialized to the specific problem because the base case must be anticipated or a designer imposed recursion depth limit must be imposed.

---

[2]Actually the program does not have to be named or defined as a LISP function, it simply needs to be pointed at by a global variable. To invoke the program, an **eval** can simply be used with the global variable as its parameter.

Another form of recursion in which procedure A calls procedure B which in turn calls procedure A is available in advanced GP models. We describe Automatically Defined Functions (ADFs) which are an advanced form of primitive representation allowing procedures and a "main program" to co-evolve on page 68. They permit this form of recursion.

2.3.5. Iteration via a Primitive

There are at least four obvious ways to implement a primitive that performs iteration. For example:

Example 1: One Parameter for Loop Condition and Body: `Repeat-Until-Nil(X)`: The primitive always evaluates X (which, recall, is a primitive which is a variable or function) at least once and continues to evaluate it until it returns `nil`.

Example 2: Two Parameters for Loop Condition and Body: `While-Do (condition,body)`: Repeatedly evaluates `condition` and then `body` if the parameter `condition` returns `non-nil`.

Example 3: For Loop Parameterization: `FOR(loop-var,loop-start,loop-end, loop-incr, loop-body)`.

Example 4: Specialized to Data `For-each-element(X)`: Loop over the data structure and access the next element each time. Access is gained via binding a variable to the element or its index. Other primitives can use this variable (it must have a default value) or the variable can be a primitive itself.

The need for a "deadman's brake": It is imperative that a loop body alters the part of the execution state that eventually causes the looping condition to halt the iteration. However, programs composed out of typical primitives are vulnerable to not displaying such behaviour. To prevent a loop from iterating endlessly, all iterative primitives must include a means of cutting off iteration. Usually, they do so by updating the corresponding counters of preset variables called `local-loop-limit` and `global-loop-limit` and using the variables to terminate iteration. These variables are somewhat hidden from attention in GP but pose some problems. First, their

settings must be chosen ad-hoc. Second, a program's execution may depend upon the settings. For example, a "perfect" program may only be perfect with the GP run's values of the parameters because they control the iteration as much as the iteration primitive's parameters. If they are changed, the execution changes. When a very complex progam is evolved, it may not be readily comprehensible (because GP programs are usually larger than a human-designed program using the primitives and may contain extra inconsequential primitives) and such a dependence may exist. In these cases, either tedious program simplification and explanation must be done or the program should be tested on other examples of the problem.

Almost all primitive sets are likely to generate programs that experience endless loops. This is an inherent effect of using blind program initialization and blind crossover in program reproduction.

We shall defer specific issues concerning choosing an iterative primitive design to Chapter 3.

2.3.6. Assignment via a Primitive

LISP uses the function `SETF(*var*, *value*)` to change the value of a variable parameterized by `*var*` to `*value*`. Using a primitive that accepts any variable and any value introduces the problem that the parameter `*var*` may not actually be a variable. It may be possible to design the primitive to test this. Or, alternatively, specialized primitives changing a variable's state may be appropriate. Both or one of the parameters of the `SETF` can be explicitly included within the primitive and removed as parameters. For example, `INC-*x*` is a primitive of no parameters that always adds one to primitive `*x*` and returns the new value of `*x*`. Or, `set-*x*(*value*)` always sets the primitive `*x*` which is a variable to the return value of `*value*`.

This capacity of a primitive to alter a variable or memory location has existed since the inception of GP [62]. More recently, attention has being focused on allowing GP to generate "Turing-complete" programs by adding memory access and update capability [118].

2.3.7. Typing Parameters in Primitives

The use of type definitions in parameters and for the return value of primitives (i.e., the typing of primitives) is somewhat swept aside in the definition of Program Discovery or in canonical GP because use the context of LISP and functional programming. However, typing is required in other languages and it could be useful in the context of program discovery regardless of the programming language used to express programs. One reason is that typing constrains primitive combinations and automatically eliminates completely useless ones. A second reason is that typing may foster hierarchically controlled solutions by providing additional structure for sub-tasks to evolve and subtask assembly to take place. Various schemes of typing in GP have been investigated [69, 70, 48]. In order to use typed primitives, the GP crossover operator must be modified to only allow subtree swaps between subtrees of compatible type and the random program creation procedure must be modified to only construct programs meeting the type constraints. This type of crossover is named "*structure preserving*". The tradeoff with typing is one of search constraint versus loss of flexibility. Active investigation into it continues.

2.3.8. The advantages of using LISP

Primitives can be implemented in any programming language and GP can blindly construct programs, manipulate their parse trees in crossover, and evaluate them when they are expressed in any language. However, LISP offers some noteworthy conveniences for both purposes. First, it is interpreted so a program can be created and evaluated conveniently "on the fly". Second, it does not require any typing in function and parameter definition and therefore, types do not have to be respected when primitives act as actual parameters or when a function returns a value. It is very easy to ensure syntactic and run-time correctness using LISP S-expressions as programs. For example, in LISP, when a condition is evaluated, a strictly boolean result (i.e., `true` and `false`) is not required. Instead, any result that is not `nil` is considered the same as `true` and a `nil` is treated as `false`.

2.4. Crossover Operator Properties

In considering different crossover operators for evolutionary program discovery, the following criteria exist for a crossover operator's behaviour:

2.4.1. Blind choice of crossover points

GP is supposed to function as "weak" method and must therefore direct its search strategy without problem specific knowledge. As this dictate pertains to crossover, it implies that a crossover operator should choose crossover points randomly or blindly (i.e., without bias dependent upon the specific problem). We shall make the same distinction as [14] between random and blind behaviour. Random behaviour selects choice with equi-probability whereas blind behaviour selects choice with a bias but probabilistically. GP crossover is typically blind rather than random because it uses a probabilistic bias (see page 42) to choose between leaves and non-leaves when it selects a crossover point.

Another motivation for not incorporating any problem specific knowledge into a crossover operator (e.g. by choosing specific crossover points) is that doing so presents hazards. First, it may result in guiding the search away from potentially useful solutions. Second, it may reflect upon the resourcefulness of the task designer, rather than the power of the GP search algorithm. In an application setting, this second hazard is actually welcomed as a pragmatic method of exploiting problem knowledge. But, in a research setting, with the power of the GP model being studied, it detracts from a clear assessment.

It is debatable whether choosing crossover points blindly correctly models genetics. No crossover is strictly analogous to biological crossover because the representations are not the same as those of real genomes. But, some evidence in biology exists that certain points on a genome may be more predisposed to being crossover points than others. This evidence would lend credence to the claim that GP crossover in some slight way models biological crossover.

Concurrent with the investigation of this thesis we conducted some brief investigation into the role of `non-leaf-selection-bias` in the canonical GP crossover. This parameter is usually set to 90%. The use of a bias in crossover point selection implies that the average amount of genetic material swapped in a crossover operation

is related to the capability of the crossover. It is stated that:

> this distribution promotes the recombining of larger structures whereas a uniform probability distribution over all points would do an inordinate amount of mere swapping of terminals from tree to tree in a manner more akin to point mutation than to recombining of small substructures or building blocks. [68, pg 114].

However, this claim was not supported by any comparison or analysis that has been reported. The impact of the bias really seems less clearcut than stated because the process by which programs are initially generated, the blind aspect of crossover and the syntactic constraints of the set of primitives all interact to generate unpredictable leaf to non-leaf ratios. In fact, when leaves actually comprise less than 10% of the size of a tree, GP crossover is non-compositionally biased towards leaf selection contrary to the rationale of the bias. As well, exchanging a leaf with another leaf looks like a "tweak" or rather small change to a program. This, of course, assumes that leaf primitives are constants, variables or predicates. In fact, they could be quite complicated functions that simply have no parameters. In this case a point mutation will in fact be "sufficiently" explorative. When the leaves are constants, variables or simple predicates it is also possible that tweaks may be appropriate as localized exploration in the search process.

This relationship can be investigated by comparing crossover operators that differ solely in the bias applied to crossover point selection to GP crossover (which we abbreviate as `GP-X0`). We experimented with the following crossover operators:

Height Fair Crossover (`Height-Fair-X0`) groups subtrees of a program by height, with equal probability chooses a height-group and then randomly selects one subtree of this group as the crossover point. (A leaf has height 0 and a program has *height* + 1 levels). Thus, each height-group is chosen with probability $\frac{1}{levels}$ and choosing between leaves or the root or subtrees of the same height is equiprobable. Each crossover point is chosen with probability $\frac{1}{|height-group|} \times \frac{1}{levels}$.

Fair Crossover (`Fair-X0`) randomly selects any subtree as a crossover point. It does not use a leaf or non-leaf selection bias.

Our measures of a better crossover point choice bias (and therefore a better crossover operator) are:

- A statistically significant fewer number of individuals are processed to find a perfect solution
- A statistically significant better probability of finding a perfect solution in an execution given a maximum number of individuals processed.

We compared the crossover operators with four different problems: `6-Mult`, `11-Mult`, `Sort-A` and `Sort-B` each executed with a run of at least 30 executions. The runs used: **population size** of 500 and **max-generations** of 50. The problem definitions of `6-Mult`, `11-Mult` and `Sort-A` are provided in Section 2.1.

6-Mult and Genetic Programming (GP)	Height-Fair-XO	GP-XO	Fair-XO
Percentage of Successful Executions	86.7 (34.0)	79.5 (40.3)	60.9 (49.8)
Confidence Interval (99%, 95%, 90%)	16,12,10	17,13,11	23,18,15
Fittest Individual at End of Run (% of Opt)	98.3 (4.5)	98.0 (4.5)	96.7 (4.6)
Fitness of Population (% of Opt) at End of Exec	77.6 (4.6)	78.2 (5.2)	81.6 (3.9)
Evaluations in a Successful Exec (% of 25500)	51.4 (15.0)	48.4 (21.4)	57.4 (20.1)
Evaluations Over All Executions (% of 25500)	55.2 (21.9)	55.7 (27.8)	72.0 (26.2)
Tree Height of Successful Programs	7.9 (2.7)	6.9 (3.2)	8.4 (2.3)
Tree Size of Successful Programs	40.4 (18.4)	42.8 (32.0)	48.8 (25.1)

Table 1. GP Crossovers and `6-Mult`

11-Mult and Genetic Programming (GP)	Height-Fair-XO	GP-XO	Fair-XO
Percentage of Successful Executions	0	0	0
Best Fitness Found (%)	93.8	87.6	87.9
Fittest Individual at End of Run (% of Opt)	80.9 (5.0)	79.2 (5.5)	76.2 (4.3)
Fitness of Population (% of Opt) at End of Exec	74.0 (3.65)	74.1 (4.4)	71.2 (3.9)
Evaluations Over all Execs (% of 25500)	100	100	100

Table 2. GP Crossovers and `11-Mult`

Tables 1, 2, 3, and 4 show comparison data. Figures in parentheses are standard deviations. A t-test [91] with 95% confidence was used to judge statistically significant difference. The height and size of perfect solutions are recorded for interest in problems `6-Mult`, `Sort-A`, and `Sort-B`.

Sort-A and Genetic Programming (GP)	Height-Fair-XO	GP-XO	Fair-XO
Percentage of Successful Executions	63.3 (49.0)	80.0 (40.7)	73.3 (45.0)
Confidence Interval (99%, 95%, 90%)	20,15,12	19,14,12	21,16,13
Fittest Individual at End of Exec (% of Opt)	42.2 (21.8)	49.7 (18.1)	47.2 (19.3)
Fitness of Population (% of Opt) at End of Exec	15.8 (2.8)	18.7 (7.1)	15.1 (1.6)
Evaluations in a Successful Exec (% of 25500)	44.6 (25.9)	39.1 (26.6)	48.3(31.4)
Evaluations Over All Execs (% of 25500)	64.9 (35.0)	51.3 (35.0)	62.1 (36.2)
Tree Height of Successful Programs	6.9 (3.2)	6.8 (2.8)	5.8 (1.5)
Tree Size of Successful Programs	22.2 (21.5)	25.0 (25.3)	17.0 (8.7)

Table 3. GP Crossovers and Sort-A

Sort-B and Genetic Programming (GP)	Height-Fair-XO	GP-XO	Fair-XO
Percentage of Successful Executions	63.7 (49.9)	67.7 (47.1)	93.3 (25.4)
Confidence Interval (99%, 95%, 90%)	23,17,11	22,17,11	12,9,7
Fittest Individual at End of Exec (% of Opt)	80.6 (25.9)	85.3 (22.6)	96.5 (13.4)
Fitness of Population (% of Opt) at End of Exec	49.5 (3.5)	48.7 (2.6)	52.2 (7.7)
Evaluations in a Successful Exec (% of 25500)	44.3 (26.1)	40.6 (28.2)	33.8 (24.3)
Evaluations Over All Execs (% of 25500)	64.7 (35.2)	60.4 (37.5)	38.2 (29.3)
Tree Height of Successful Programs	6.96 (2.9)	5.7 (1.4)	6.9 (2.8)
Tree Size of Successful Programs	19.88 (15.7)	19.4 (15.0)	425.4 (19.0)

Table 4. GP Crossovers and Sort-B

For 6-Mult the results show that Height-Fair-XO and GP-XO have a significantly better expected probability of success than Fair-XO. Another significant difference was the average fitness of the population at the end of a execution (a execution was terminated as soon as a perfect individual was found) where, in contrast to the comparison in terms of probability of success, Fair-XO achieved a significantly better result and GP-XO and Height-Fair-XO were indistinguishable. The two results together might be explained by the fact that Height-Fair-XO is slower to converge because it requires a higher population fitness before it can create an individual fitter than the present fittest in the population. Both Height-Fair-XO and GP-XO needed a significantly less mean number of evaluations than Fair-XO but they did not differ with each other significantly.

GP using any of these crossovers could not solve 11-Mult with 25500 evaluations. Both Height-Fair-X0 and GP-X0 obtained significantly better best fitness and population fitness than Fair-X0 but they did not differ between themselves significantly. Height-Fair-X0 found the fittest program (93.8% of optimal) while the fitness of the best program found by Fair-X0 was 87.9% and GP-X0 was 87.6%.

In Sort-A there was no statistically significant difference in the probability of success, evaluations required, fitness, height or size of programs with any crossover operator. In Sort-B it was possible to rank the crossovers according to probability of success:

1. Fair-X0: 93.3%
2. GP-X0: 67.7%
3. Height-Fair-X0: 63.7%

GP-X0 produced programs which as trees had significantly shorter height than the other two crossovers but there was no significant difference in tree size. In terms of the percentage of evaluations required, Fair-X0 used significantly less than Height-Fair-X0 and GP-X0 but only significantly less evaluations than Height-Fair-X0 when only successful executions were considered. There was no significant difference in population fitness or the expected fitness of the best individual.

In summary, the results show that none of the crossover operators was consistently better, under both criteria, in all experiments. Nor was one crossover operator always significantly better than the others, on a single experiment. The results suggest that the capability of the crossover operator depends much more upon the specific nature of a problem (i.e., its primitives, its fitness function) than its bias in crossover point selection. Furthermore, while a crossover selection point bias may be effective, choosing an appropriate one is not simple. There is a need to further understand what information concerning the character of the search space relates to crossover selection point bias.

2.4.2. Syntactically Correct Offspring

Because GP is supposed to be a "weak method", it is crucial that a crossover operator in GP generate programs that are syntactically correct. In other GAs, this

requirement is equivalently "the crossover operator must generate encoded offspring that decode to valid candidate solutions". Should this requirement not be met, an ad-hoc, problem specific repair algorithm has to be added to either search algorithm detracting from its generality.

To avoid the ad-hoc aspect and the computational expense of repair, GA designers carefully choose an encoding that under the crossover operator is assured to yield only syntactically correct offspring. However, in cases when the design of the encoding is orthogonal to other properties of the search strategy, such as combination of solutions via crossover, designers use an encoding and crossover operator that do not yield syntactically correct offspring and resort to heuristic repairs [87, 35, 79].

In contrast to other GAs, because there is no encoding in GP (see page 14) and because GP crossover takes advantage of tree representations for a pair of parent programs, GP preserves syntactic correctness in both (or a single) offspring by swapping subtrees. Every offspring composed in this manner is directly syntactically valid.

2.4.3. Flexible Program Length

With program discovery, it is crucial that the size (number of primitives) or structure (pattern of primitive nesting) of a candidate solution need not be specified in advance of searching for a solution. GP crossover, again by taking advantage of using a tree-based representation of a program, generates offspring that, while they are syntactically valid, may differ in size and structure from their parents.

2.4.4. Parent-Offspring Fitness Distribution

Altenberg [2] has shown that the search bias in GP exploits an underlying assumption concerning the fitness distribution of the set of offspring produced by crossing over a pair of parents at all posssible points. This assumption is that a portion of the probability density covers fitness values greater than either parent (i.e., some of the possible offspring are fitter than both their parents). Most preferably the distribution of fitness among a pair of above average parents' offspring should have a large upper tail. This would indicate that many different crossover points chosen among the two parents yield offspring more fit than both parents.

The quality of a crossover operator in this respect depends upon the specific

problem because both the fitness function and primitive set influence parents-offspring fitness distribution. As yet it is not clear how to *a priori* assess this quality or how to assess it while running GP (and possibly improve it). Random sampling of the distribution may be be helpful in judging a crossover operator's quality but this is still purely speculative.

Tackett [115] uses a "Greedy Recombination" operator (also earlier referred to as "Brood Selection" [117]) instead of GP crossover as a means of estimating a pair of parents' offspring fitness distribution and improving the fitness of offspring on average. The following description is from the Conclusions section of Tackett's thesis:

> In standard GP recombination, offspring are chosen uniformly at random from among the large number possible from a single pairing. The Greedy Recombination operator samples from among these potential offspring and chooses the best from that sample, thereby substituting a greedy choice for a random choice. Arguments were presented based both upon probability and upon search, predicting that Greedy Recombination should produce fitter offspring, on average. An important result obtained empirically was that a different fitness evaluation may be used for selecting among potential offspring. In particular, that evaluation can have greatly reduced cost relative to the fitness evaluation of mature population members. The result was that Greedy Recombination consistently produced improved performance at reduced cost relative to the standard method of recombination. [116]

2.4.5. "True" Combination

One role of crossover is to "truly" combine parents' features by propagating one or the other parent's expression of a feature into an offspring. We use the adverb "truly" or adjective "true" in this context to distinguish this rich sort of combination from the superficial combination of parents' primitives using a crossover operator. Superficial combination simply exchanges "genetic material" (e.g., primitives) but does not ensure that material removed from one parent is replaced by material playing a similar role in the other parent (i.e., does not ensure feature correspondence).

"True" combinative function is simple to incorporate into a crossover operator when the feature expressions in a parent correspond (sometimes by precise placement on the bit string, other times by a feature-expression association). Many GA representations can be truly combined. Of course, the combination is complicated by feature interactions (i.e. epistasis) but, without loss of generality, "true" combination can take place via a crossover.

In GP, "true" combination is largely impossible (within the constraints of blind crossover point selection and ignorance of primitives' behaviour) because two programs do not have a strict one to one correspondence in the function and behaviour of their primitives. In other words, in GP the clearcut notion of "expression of a feature" and the direct correspondence between two parents' genetic material does not exist. Therefore, GP crossover and its variations do not "truly" combine. GP makes a tradeoff: its representation allows expressive flexibility (i.e., programs of different length and structure) but it obliterates expression correspondence that could provide a basis for "true" combination via crossover. As well, as with other GAs, in GP the strong relationship between a swapped subprogram and the program enclosing it (i.e. epistasis) make it more complicated to argue the existence of a combinative role in crossover.

The representational tradeoff in GP between flexibility of expression and expression correspondence, prompts the question of whether a crossover operator can be devised which focuses on "true" combination. This has been investigated by [23] where the author devises two new crossover operators in which only subtrees at a more or less (see strong versus weak) corresponding tree position of both parents can be swapped:

Strong Context Preserving Crossover (SCPC) A subtree of parent-1 is eligible for crossover point selection if there exists a subtree with the same node coordinate in parent-2. Node coordinate is defined by the path followed from the root to a node. Once the first crossover point is chosen, the subtree in parent-2 that has the same node coordinate is directly chosen for the swap.

Weak Context Preserving Crossover (WCPC) Once again, only subtrees which have a node coordinate common to both parents are eligible for crossover point selection. However, **any** subtree of the corresponding node coordinate subtree

is chosen for the swap.

The experimentation investigates mixing SCPC with GP-X0 or using WCPC 100% of the time. For some problems, one mix or the other (e.g., 25/75, 50/50) of SCPC with GP-X0 was better than GP-X0. However, for one problem GP-X0 was better than any mix of SCPC or WGPC. WCPC was consistently worst. The results suggest that this form of crossover may be too restrictive [23] or, alluding to epistasis and the range of effect by changing a program, that the context of expression correspondence is not well modelled by the spatial character of a parse tree. More details and explanation for the results is given in the reference [23].

2.5. Non-Canonical GP

The GP literature can be grouped into three categories:

1. Applications of GP or extensions of GP to problems translated to Program Discovery problems.

2. Extensions of GP, that is non-canonical versions of GP.

3. Investigations to improve the understanding of GP and its extensions.

Some articles, naturally, span more than one category. Extensive bibliographies are available from [69, 70, 76]. We shall not attempt to encompass the entire literature in this setting. Instead, in subsequent chapters, we discuss GP investigations and extensions relevant to our research when they are applicable. In this section, we briefly review a few GP extensions which have become either *de facto* standard options to GP software systems or contribute some interesting aspect of GP. We categorize the extensions into:

- New Operators

- Alternate Selection and Generation Strategies

- Representation Enhancements

2.5.1. New Operators

Canonical GP was introduced by a series of conference papers each demonstrating its application to a different set of problems [60, 63, 61, 65, 67, 66]. As well, [62] and subsequently [69] definitively described GP. The report and book both also include examples of "GP-Mutate", "permute", "define-building-block", and "structure-preserving crossover" (see page 55) operators which, by our chosen definition of GP, are non-canonical or extensions.

A GP operator works on either a pair of parents or a single parent. We assume that the required number of parents are drawn by fitness proportionate selection and a parent is first copied before being manipulated by an operator.

The GP-Mutate operator chooses a subtree in a single parent program at random and replaces it with a randomly created subtree. The sole parameter of the operator is a maximum tree height limit for the new subtree. Koza reports that, in his experience, when GP-Mutate replaced GP crossover, no experiment run ever produced a solution to any problem [62]. The same basic sort of mutate operator is often used alongside crossover in other researchers' experimentation without any explicit justification. The explicit role of the operation has not been experimentally nor theoretically analysed to our knowledge (for a brief discussion see page 49).

The permute operator chooses a subtree in the program at random and then permutes the order of its immediate subtrees. For example, if the subtree root was a primitive of two parameters, the subtree representing the first parameter would be swapped with the subtree representing the second parameter. When there are more than two parameters, any new permutation of the actual parameters is chosen with equal probability. Once again, Koza reports that, in his experience, in terms of the probability of success of a run or the computational effort required to find a solution, no benefits were observed with this operator.

Kinnear introduced the "hoist" operator in [57] as a special case of crossover. A subtree in a single parent program is chosen at random and elevated (i.e., hoisted up) to program status. The remainder of the parent is discarded. A "hoist" is designed to ensure that an offspring program has fewer primitives than its parent.

Kinnear also introduced a "create" operator where a new program is created randomly (like in Generation 0) and inserted into the current population in lieu of a program propogated by parent selection and crossover and/or mutation. This

operator can be refined to ensure that a random new program is of smaller height than the population average or shorter than a tree drawn by fitness proportionate selection [57]. No isolation study of the benefits of the hoist and create operators has been conducted.

Both Koza and Angeline [62, 5, 7] designed an operator that extracts and encapsulates a portion of a program (roughly a subtree) and makes it into a module. A module is essentially a new primitive. Angeline's operator is called "compression" whereas Koza's is called "define-building-block". The module is given a name "on the fly" when it is created and then substituted in the program for the subtree it encapsulates. When a program using a module is evaluated, the definition of the module in a module library is used. This operator does not change the execution of a program but it does change its specification. A name creation server and module library to manage module references are required for implementation of the operator. Angeline names his system "GLiB" for Genetic Library Builder. The "define-building- block" operator is the same as "compression" except that its modules have no arguments because the entire subtree rooted at the encapsulation point is made into a module. In contrast, "compression" modules have parameters because the portion of the program that is encapsulated is marked by both a root point and a *depth compression* level. Each subtree branch below *depth compression* level is represented by a parameter in the module definition. See Figure 8 for a pictorial description. Even though copies of the extracted subtree may exist elsewhere in the program, only the extracted subtree is substituted by the module.

To complement "compression" Angeline uses another operator named "expansion" that randomly selects a compressed module in a program and replaces it by its original definition. We shall discuss in more detail the motivation for and efficacy of these operators in Chapter 3.

2.5.2. Alternate Selection and Generation Strategies

Tournament Selection The use of tournament style fitness based selection is popular in GP research. It is used exactly as in GAs [33]. A number of programs are chosen as "tournament competitors" by being drawn with random uniform selection (without replacement) from the population. The program with the best fitness is the winner of the tournament and becomes a parent. In [57] the tournament size is seven.

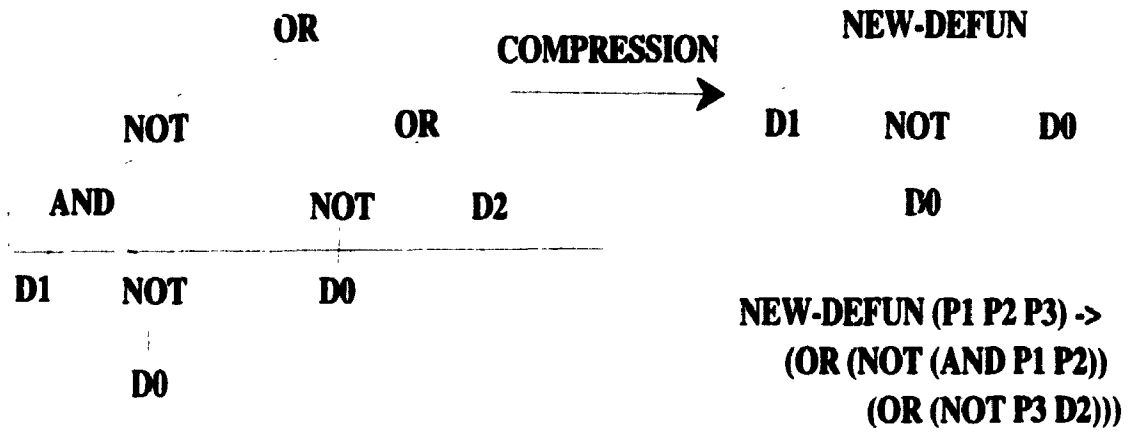


Figure 8. Angeline's compression operator encapsulates a subtree of a maximum depth, creates a function for the subtree and replaces the subtree with a call to the new functions ([5, 7]).

Other typical sizes are three and five. Typically, one speaks of a GP system using "fitness based" selection and then qualifies the selection as either fitness proportionate selection, tournament (with tournament size) or something else.

Steady State GP The use of a steady state model in GP instead of a generational model [96] is the same exchange as that made in other GAs [20, 121, 114, 19]. A steady state model is one in which programs are created one at a time, evaluated immediately for fitness, and then merged into the population in place of an existing low fitness program. In contrast, in the generational model (in canonical GP) an entirely new population is created each generation and then they are all evaluated for fitness. In Reynold's steady state GP model the uniqueness of a program is guaranteed to maximize diversity. If fitness proportionate selection is used, the difference is that the fitness proportions are readjusted with each new program in the case of steady state whereas in a generational model they are readjusted after the entire new population is created. Because there is no *generation* strictly in the steady state model, creating a population-size number of new programs constitutes the equivalent of a generation. This is helpful for comparisons. [114, 51] provide insight into the advantages and disadvantages of these methods.

2.5.3. Representation Extensions

One of the most widely recognized *de facto* standard extensions to GP is the use of Automatically Defined Functions (ADFs) which were introduced in [72, 70]. Koza argues that ADFs can be used in conjunction with GP to improve its efficacy on large problems.

> An *automatically defined function (ADF)* is a function (i.e., subroutine, procedure, module) that is dynamically evolved during a run of genetic programming and which may be called by a calling program (e.g., a main program) that is simultaneously being evolved. [70, pg. 1]

In using ADFs, one employs a template for program structure that remains fixed over the course of the search while the detailed components within the structure are permitted to co-evolve in the process of the search. By structure, we mean that a program is divided into one "result producing branch" (rp-branch) and a *a priori* selected number of "ADF" branches. GP has to find effective primitives to compose the rp-branch and each ADF branch. One type of structure is usually used for every program in the population though an assortment of structures can be *a priori* chosen with some crossover restrictions we shall mention later.

The template is that part of the program which never evolves because it is not involved in crossover. In LISP, a template consists of a PROGN at the root of the program S-expression that has nested within it: the name, parameter list, and `values` form of a `defun` form for each ADF, and another `values` form which is the last form of the PROGN. Conversely, the evolvable elements are the body of each `defun` form which are the ADF branches and the second last form of the PROGN which is the rp-branch. This distinction between what is evolved and what is fixed is illustrated in Figure 9 via a line. The portion of the program above the line is not manipulated by GP.

The rp-branch functions like a main program in the sense that it serves as a means of calling functions (i.e., ADFs) and finally returning the result(s) of the program. Each ADF branch is an *a priori* named function which can optionally use parameters. A different set of primitives is used for each ADF branch and the rp-branch. By introducing an ADF function name into the primitive set for the rp-branch one allows a possible function call of that ADF function from the main program. ADFs allow a

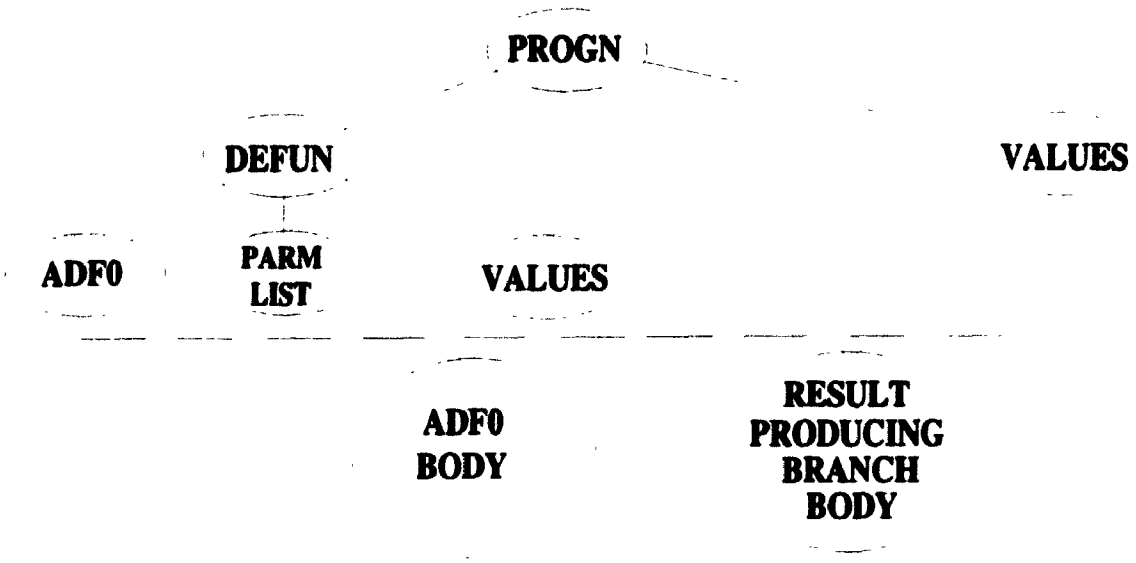


Figure 9. When using ADFs in LISP the template is a PROGN S-expression with DEFUN forms for each ADF and a one form for the result producing branch. Only the primitives below the dotted line are evolved ([70]).

hierarchy of scope to be designed prior to the search by choosing which ADF names are placed in the various primitive sets. Some primitives among the sets can be the same: e.g., primitives that function as global variables. Other primitives will be exclusive to one set because the task designer decides beforehand on a division of capability. Recursion can be facilitated by allowing a pair of ADFs to call each other (though care will have to be taken to avoid or halt endless recursive calls). Typically the different primitive sets are chosen with rough predetermined functions for an ADF branch in mind and with a rough decomposition of the task into sub-tasks in mind. This is because ADFs (because they require additional primitives) could vastly increase the size of the search space and therefore, require a gentle human hand to constrain the space to a size that can be tackled by GP and present computers.

Figure 9 illustrates the use of ADFs to solve the Two-Boxes problem in which a program should compute the difference in volume between two boxes given their heights, lengths and widths and some simple arithmetic operators. Here the program template uses one ADF branch named `Volume` of three parameters named `ARG0`, `ARG1`, and `ARG2`. This ADF is on the left hand side. The rp-branch is on the right side and it invokes the ADF `Volume` twice.

Because the primitive sets are different among ADF branches and the rp-branch, GP crossover with ADFs is specially constrained. Crossover is only allowed at points where primitives from the same primitive set will be exchanged. ADF crossover first randomly selects a crossover point anywhere within the evolvable part of the program (i.e., outside the template). Once that point has been determined, in the other parent it restricts itself to choosing among points that are in the same kind of branch. This restriction preserves the different primitive sets thus preserving the designer's rough task decomposition.

ADFs pay off because they require less computational effort and produce smaller solutions than using GP without them. On the studied problems, this payoff increases as problem size grows [70]. Main points 1 and 2 of GPII offer interpretation of why ADFs are effective and how they work:

Main Point 1. Automatically defined functions enable GP to solve a variety of problems in a way that can be interpreted as a decomposition of a problem into subproblems, a solving of the subproblems, and an assembly of the solutions to the subproblems into a solution to the overall problem (or which can alternatively be interpreted as a search for regularities in the problem environment, a change of representation, and a solving of a higher level problem).

Main Point 2. Automatically defined functions discover and exploit the regularities, symmetries, homogeneities, similarities, patterns, and modularities of the problem environment in ways that are different from the style employed by human programmers. [70]

Earlier in this section we discussed "compression" and "define-building-block" operators. They are introduced as operators but could just as easily be introduced under the subtitle of representation extensions. In contrast to ADFs which dictate a representational structure that remains static during a GP run, the "compression" and "define-building- block" operators dynamically modify a representation in the course of a run. At this point, we refer the interested reader to a comparison between ADFs and the complementary "compression" and "expansion" operators in [57]. In the context of hierarchical process we shall discuss both ADFs and "compression" in Chapter 3.

There is another extension of GP that alters representation dynamically in the general vein of "compression" and "expansion" called Adaptive Representation GP [99] but with important distinctive differences. It is neither an operator based extension nor a static representation alternative so we defer its description to Chapter 3 where it is also examined.

Many other extensions to canonical GP have been introduced but are omitted here because they are not directly relevant or are not apparently "standard".

2.6. Chapter Summary

The purpose of this chapter was to describe the problem suite of the thesis and to provide background on the GP paradigm and its extensions. We also provided insight into the nature of primitive design. With the pseudocode of GP we intended to provide the reader with a clear concept of the algorithm and its potential application. We analyzed the requisite features of a genetic crossover operator in order to emphasize their role in the search algorithm.

CHAPTER 3

An Experimental Perspective on Genetic Programming

The first section of this chapter is devoted to judging how much of GP's success arises from skillful designer choice of the primitives, test suite and fitness function and to determining whether hierarchical process is present in GP.

To assess how much of GP's success arises from skillful designer choice of the primitives, test suite and fitness function, we examine designing these three elements and describe experiments that further illustrate various issues. We report that a strong conflict existed between refraining from preordaining a solution by choosing strongly influential primitives and trying to reasonably constrain the search space and model the task environment.

To determine the presence of hierarchical process, it is not sufficient to examine GP for hierarchical solutions and from that to conclude its search process is hierarchical because it is possible to *a priori* encode subtasks as primitives and then simply use GP for subtask assembly rather than subtask identification and subtask assembly. Hierarchical process must be confirmed by testing whether GP can evolve solutions from general purpose primitives. We describe experimentation following this approach which indicates that a hierarchical process does not arise from GP's evolution-based mechanisms.

In Section 3.2, we advocate hierarchy in solutions and process in GP for the advantages of scalability and efficiency. We analyse Koza's ADFs (page 68), Angeline's GLiB (page 66) and Rosca and Ballards' Adaptive Representation GP (AR-GP). ADFs improve the hierarchical character of GP programs but do not explicitly promote a hierarchical process. Both GLiB and AR-GP direct hierarchical process but

they do not seem powerful enough or sufficiently efficient.

In Section 3.3 we define how a primitive set could be judged sufficiently general to avoid the criticism of being too specific to the problem at hand. We appraise whether primitive sets of GP problems express specialized task knowledge that helps them find a solution. We point to fitness function design as an important but difficult research issue.

GP is a GA where critical choices have been made to suit its goal of program discovery. In Section 3.4 we list and consider these choices with respect to their necessity, convenience and design issues.

3.1. Assessing Designer Choices and a Hierarchical Process in GP

For this assessment, we shall try to solve a novel problem using primitives that do not unduly *a priori* constrain the definition of subtasks and concurrently report on the choices a GP designer encounters and the extent of designer influence in GP's likelihood of success. Sorting fits this requirement because we are the first to introduce it as a program discovery problem [88]. It has other advantages which are mentioned in Section 2.1.

3.1.1. Test suite and Fitness Function Design Issues

Two closely related setup steps in GP are the composition of the test suite and the definition of a fitness function.

Generality and the Test Suite Sample

The composition of the test suite can influence how general a solution may be found. Typically the test suite must be a sample over the distribution of desired behaviours in order to enforce an acceptable degree of program generality. For sorting, a sort program which can sort an array of any size (given `*array-size*` as a constant) is clearly desirable.

The test suite can be significantly shortened by consolidating various inherently similar initial configurations and by sampling. There is a pragmatic reason for doing

this: fitness evaluation is the most computationally expensive aspect of GP. However, there is a trade-off between running the risk of over simplifying the test suite (by using a small sample) and spending a lot of time evaluating fitnesses. In GP there is no formula available that determines a sample size according to computational and generalization goals. With sorting, prior to experimental trials, similarly, it is unclear how many test cases are sufficient to evolve a general sort program.

Obviously the expressive level of primitives is a major factor in how general a program can be discovered. However, for a set of primitives, by decreasing the number of test cases progressively, it is possible to judge the relationship between test suite sample size and generality. For this purpose we used the primitives and fitness function of problem `Sort-A` and varied the size of the test suite progressively in the sorting problem. In the sorting problem the "baseline" or standard test suite (called `Sort-A-standard`) consists of 48 different vectors of 5 different sizes. For the smaller sizes (2-4) every permutation of elements is a test case. Larger sizes are sampled (10 arrays of size 5 and 5 arrays of size 6). We created two other smaller test suites called `Sort-A-small` and `Sort-A-medium`. Our experiment runs GP on `Sort-A` with them and assesses solutions scoring maximum fitness for generality by their fitness on a more extensive test suite.

Experiment Title: Effect of Test suite Sample on Generality: Problem `Sort-A`

Problem Definition: `Sort-A` with 3 different test suites and an out of training test suite.

Test suite `Sort-A-Small` consists of 8 different vectors: 2 of size 2 and 6 of size 3. Every permutation of elements with these sizes is a test case. Among the test suite are 2 arrays which are initially in sorted order and 7 elements are initially in their correct positions. There are 22 elements in all.

Test suite `Sort-A-Medium` consists of 32 different vectors of 3 different sizes (2, 3, and 4 elements). For each size every permutation of elements is a test case. Among the test suite are 3 arrays which are initially in sorted order and 24 elements are initially in their correct positions. There are 118 elements in all.

Test suite `Sort-A-Standard` consists of 48 different vectors of 5 different sizes. For the smaller sizes (2-4) every permutation of elements is a test case. Larger sizes

are sampled (10 arrays of size 5 and 5 arrays of size 6). Among the test suite are 3 arrays which are initially in sorted order and 47 elements are initially in their correct positions. There are 198 elements in all.

Out of Training Test suite This consists of 150 different arrays of random sizes (between 5 and 15) that contain random elements. No array is initially sorted. There are 514 elements in total with 43 elements in their correct positions. With 150 programs randomly generated from the `Sort-A` primitives (heights between 5 and 11), the average performance on the out of training test suite was 44 elements placed correctly and no arrays correctly sorted (i.e. 0 hits).

Generality: `Sort-A`	`Sort-A-Small`	`Sort-A-Medium`	`Sort-A-Standard`
%age Successful Executions	63.3	63.3	63.3
%age Individuals Processed	68.0 (33.7)	68.8 (33.1)	68.6 (33.1)
Hits (150 max)			
Ave	60 (68.8)	122.3 (54.3)	134.2 (46.0)
Min	0	0	0
Max	150	150	150
Fitness (%)			
Ave	50.7 (37.9)	84.4 (31.2)	90.4 (28.0)
Min	8.9	8.9	8.9
Max	100.0	100.0	100.0

Table 5. The Effect of Test Suite Sample on Generality: `Sort-A`

Results

Decreasing the size of the test suite did not change the probability of an execution finding a solution with 100% fitness and this probability was the same for each test suite. The number of individuals processed to find a solution did not differ between the test suites. Concerning generality, as expected, the larger the testsuite, the more capable a perfect solution was in solving the out of training suite. These results are shown in Table 5. The table shows the average number of hits (i.e. arrays perfectly sorted) and average fitness (as a percentage) of the perfect solutions. While the

perfect solutions obtained by the small test suite could correctly sort 60 of the arrays and place 50.7% of the elements in their correct place, the perfect solutions of the standard test suite achieved 134.2 hits and averaged a fitness of 90.4%.

We repeated the experiment using the problem `6-Mult`. Test suites `6-Mult-half` and `6-Mult-3/4` respectively use one half and three quarters of the entire standard `6-Mult` test suite. In `6-Mult-half` each address is represented equally in the test suite with half the possible register value combinations (32 test cases). In `6-Mult-3/4` there are 48 test cases where each address is represented equally in the test suite with three quarters of the possible register value combinations. We used a population size of 500 and scaled the fitnesses with linear and power scaling.

Generality `6-Mult`	`6-Mult-half`	`6-Mult-3/4`
%age Successful Executions	53.3	40.0
%age Individuals Processed	70.6 (27.8)	88.4 (21.4)
Hits (64 max)		
Ave	57.7 (6.8)	60.6 (3.3)
Min	43	52
Max	64	64

Table 6. The Effect of Test Suite Sample on Generality: `6-Mult`

The results in Table 6 show that, while an incomplete test suite can produce a 100% general solution in `6-Mult`, most often the solution it finds does not. Counter to expectations, 5 of the 16 solutions found running the `6-Mult-half` test suite scored 100% fitness (equivalent to 64 hits), while only 2 of the 12 solutions found running `6-Mult-3/4` scored 100%. However, on average the solutions from `6-Mult-3/4` performed better (60.3 compared to 57.7 hits). For a baseline comparison we ran 150 programs randomly generated from the `6-Mult` primitives with heights ranging from 5 to 11. On average a program scored 40 hits (fitness of 62.5%).

Other work examining GP's capacity to generalize is [116]. The "donut" problem provides a problem that is tunable in terms of how much noise exists in the data in addition to how large a sample of test cases is used. The conclusion was:

> For a given training set size, the classification performance as a function of noice variance was within approximately a constant amount of the

> Bayesian limit. Reasonable solutions were obtained using as few as 40 training samples. These results ... suggest that Genetic Programming derives some performance advantages through an ability to generalize. [116]

The Coaching Aspect of Test Suite Composition

A subtle aspect in the choice of a test suite is the "incremental encouragement" that can be exerted. For example, in the Block Stacking problem, solving the test case which puts the only block on the table onto the (already ordered) stack is worth as much as solving the test case that involves removing all blocks from the stack and restacking them correctly. Thus, the reward for doing a little task correctly is equal to that of doing a large task. In this manner, a "building block" subprogram for solving harder test cases would be created by solving the simple test case. This suggests that test cases whose solutions are likely to be reused could be given relatively greater weights, and that the eventual solution could be "coached" along by progressive incentives arising from an interplay between the fitness function and a cleverly chosen test suite.

We experimentally assess the influence of incremental encouragement by varying the fitness credit of test cases and checking whether this affects the number of individuals that need to be processed in order to find a solution. Once again we base our variations upon problems `Sort-A` and `Sort-B`. In both **`equal element`** and **`equal test case`**, because there are fewer smaller arrays, the relative credit of a group of test cases according to the size of the array increases in a non-linear function. The two other credit schemes, **`exponential`** and `linear` are deliberately designed to provide incremental encouragement by assigning test case groups credit so that the relative credit of a group of test cases according to the size of the array decreases with array size. All runs are executed without scaling so that effects of the credit scheme are not further amplified.

Figure 10 is a plot of array size versus the fitness credit of test cases grouped by array size. The graph shows these four test case credit functions. Note that coincidentally **`equal test case`** and **`equal element`** are very similar.

Experiment Title: Coaching Incentives: Problems `Sort-A` and `Sort-B`

2

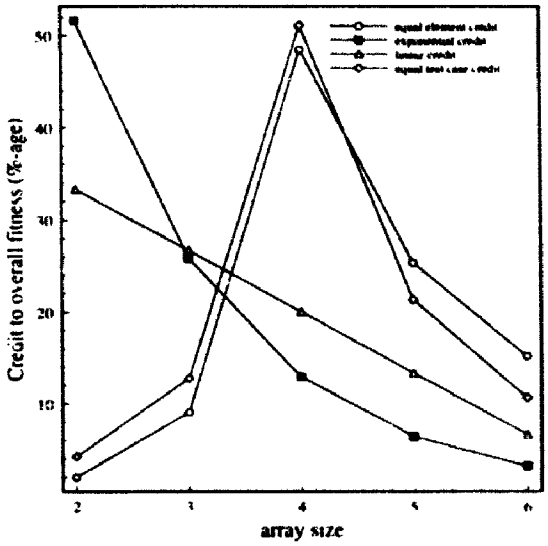


Figure 10. Different credit functions according to array size for `Sort-A`

Problem Definition: `Sort-A` and `Sort-B` with 4 different test case credit schemes. First, the "base" fitness associated with a test case is calculated using mismatch order (see page 35) for `Sort-A` and permutation order for `Sort-B`. Then the "base" fitness is weighted according to the credit scheme to yield a weighted fitness. The weighted fitness of all test cases are summed to yield the fitness of the program. The credit schemes are:

Equal test case Array size is ignored, every test case is worth 1 or 0 depending on whether its base fitness is perfect. Maximum program fitness is 48. A test case has a fitness credit of $\frac{1}{48}$.

Equal element The number of elements in a test case array determines the credit of the test case in the overall fitness of a program. For example, each array of two elements confers partial fitness equal to $\frac{2}{198}$ because two elements can be ordered correctly and the test suite consists of 198 elements in total. This is the standard `Sort-A` scheme. The weighted fitness equals the base fitness. Maximum program fitness is 199.

Exponential Test cases with arrays of size 2 use a weight factor of 2400, size 3 use a weight factor of 800, size 4 use a weight factor of 50, size 5 use a

weight factor of 60 and size 6 use a weight factor of 60. As a result, each group of test cases grouped according to array size has a credit factor in the program fitness that decreases by a half as the array size increases. Maximum program fitness is 9300.

Linear Test cases with arrays of size 2 use weight factor 2.5, size 3 use weight factor $\frac{2}{3}$, size 4 use $\frac{1}{8}$, size 5 use $\frac{1}{5}$ and size 6 use $\frac{1}{5}$. As a result, each group of test cases grouped according to array size has a credit factor in the program fitness that decreases by $\frac{1}{15}$ (i.e., linearly) as the array size increases.

Sort-A	Equal Element	Equal Test Case	Linear	Exponential
%-age Successful Executions	50.0	63.3	63.3	50.0
Avg Generations/Exec	36.7 (15.5)	33.8 (15.3)	37.3 (14.5)	37.8 (15.1)
Successful Execs Only Avg Generations	44.0 (11.3)	42.0 (11.4)	26.2 (11.5)	25.7 (12.7)
Max Generations	44	42	45	49
Min Generations	3	5	5	5
Sort-B				
%-age Successful Execs	46.7	26.7	53.3	56.7
Avg Generations/Exec	39.4 (13.3)	43.4 (14.0)	35.7 (15.7)	37.2 (14.7)
Successful Execs Only Avg Generations	27.2 (10.1)	25.1 (16.8)	23.2 (11.3)	27.5 (12.8)
Max Generations	48	49	43	49
Min Generations	9	5	4	5

Table 7. The Effect of Fitness Credit Schemes

Results As Table 7 indicates, in the case of **Sort-A**, there is no clearly superior scheme. The rough winner is **Linear** because it achieved the best probability of success and, while it did not differ in terms of average generations per execution considering all executions, it processed less individuals on average in its successful executions. While **Exponential** did not have the best probability of success, it did, on average, process the least individuals in a successful execution. With **Sort-B** the

only differentiation between the schemes concerned `Equal Test Case`: it had a much lower probability of success.

Overall, for these primitive sets, the credit assigned to test cases according to array size makes some difference in the outcome of GP. Unfortunately, that difference is neither consistent nor predictable. Thus, the GP task designer can influence the quality of GP's success by making these decisions but has little indication of how to proceed.

3.1.2. Primitive Design Issues

The design of a GP experiment also necessitates selecting a primitive set. The process can be similar to being told to pack a knapsack to travel to a known destination, without being told the means of conveyance or the mode of the voyage. Harder choices are required to select suitable primitives for our sorting than for the Boolean Multiplexer problems `6-Mult` and `11-Mult` (where primitives are either logic operators, IF, or registers). We found the process fraught with second-guessing in an effort to detect bias and somewhat frustrating because it is difficult to predict even crude behavior.

The basic tasks or utilities one could expect to be useful in sorting have to be given to GP in the direct form of primitives or via primitives that can be combined to express them. It can be quickly decided to make use of the one value we wish to generalize over: `*array-size*` (i.e., a program is not expected to induce the concept of an array size). As well, an examination of well-known sorting algorithms (such as Bubble Sort and Selection Sort) quickly suggests the need for the following program constructs:

- iteration primitives,
- a conditional primitive,
- a swap primitive, and
- array element referencing.

This basic identification does not make further choices clearer. The following issues exist:

Trading off the expressive "power" of a primitive with constraining the search space to make the search space manageable in size.

- A (relatively) constrained primitive has one or more of its operands and operators unparameterized, e.g., always uses a fixed value or global variable as an operand.

- A (relatively) expressive primitive has its behavior determined to a large extent by the type and value of its parameters.

Constrained primitives search a smaller space, and are thus more efficient, but limit the shape of the eventual solution in advance (at worst, they may preclude finding a solution). Expressive primitives do not restrict the final solution as much, but, since they have more arguments, they recursively define a larger search space (at worst, they may fail to find a solution in an acceptable time). Consider the simple examples in Table 8. Example A returns the value of a variable decremented by 1, example B accesses an array element, and example C exchanges two values.

A. Subtraction	B. Array Reference	C. Exchange

`(- var value)`
 	
Any value is subtracted from var
	`(aref array index)`	`(swap (aref array index1)`
		`(aref array index2)`
		implicit use of a temporary

`(minus1 var)` value is always 1	`(aref-*array* index)` *array* is always referenced	`(swap-*array* index1 index2)` *array* is always referenced
`(var-minus1)` value and var always referenced	`(aref-*array*-index-k)` index-k and *array* always referenced	`(swap-*array*-first-last)` first and last elements of *array* always swapped

Table 8. Potential Sort Experiment Primitives

The same issue affects iteration primitives: The primitive

```
(FOR loop-var, loop-start, loop-end, loop-incr, loop-body)
```

illustrates how every factor in a looping condition can be determined by parameterization rather than being encoded implicitly or explicitly within a primitive. Obviously, this example offers the greatest degree of flexibility for expressing solutions.

This type of primitive, however, is rarely ever used. One reason is that the size of the search space increases with the number of parameters in a primitive. At some search space size, a program discovery algorithm may simply not be powerful enough. As well, many of the primitives in the primitive set may be of a type that is incompatible with the intended function of the parameters. Thus, many junk versions of iteration will be formed in "Generation 0" and via crossover and have to be searched through. The flexibility of the **FOR** primitive may be traded off with constraining the search by choosing a more constrained iteration primitive.

Trading off exploiting problem specific knowledge (by incorporating it into the primitives) with *a priori* biasing or constraining the outcome.

We did not add the integers in the index range of the array to the terminal set because this would have greatly increased the search space and, more importantly, would have facilitated a highly specialized solution which fits the test suite precisely.

The choice of a loop primitive also encounters the specialization issue. The primitive (**do-until loop-test true-form**) iterates until **loop-test** is false but it relies upon the **true-form** performing a side-effect which changes some variable in **loop-test**. It uses no loop variable which seems to make it less suited for array sorting.

(**do-until loopVar startValue endValue varDelta loop-test true-form**) has its behaviour completely dictated by the values of its arguments. **loopVar** is set to **startValue** and each time through the loop is changed using **varDelta**. **startValue** is compared to **endValue** using the relation **loop-test** . If the result is true, **true-form** is executed. **true-form** could even be a **progn** of primitives but this creates the further unresolved issue of how to reasonably specify the number of primitives within the **progn**.

Any one of these arguments could be absorbed into the primitive to limit its potential behaviour. For example, (**do-until-Up-with-*j* startValue endValue true-form**) uses the variable ***j*** as implicit **loopVar**. The primitive sets ***j*** to startValue, and it always increments ***j*** at the end of each repetition, using an

implicit `varDelta` of 1+. It always tests whether `*j* <= endValue`, with an implicit `loop-test` of <=, to conditionally execute `true-form` .

Similarly, `(do-until-Down-with-*i* startValue endValue true-form)` loops downward using `*i*` as the loop variable. These last two primitives require that `*i*` and `*j*` be made variables for the program. Note that the loop variables must be different globals to facilitate nested loops that might sweep up and down an array. A valid question arises: does this choice of incremental and decremental loops constitute too strong an advance bias?

Besides using loops there are many ways to instill domain specific knowledge into primitives. Complex operators or operands that already exist in the problem domain can be directly translated into primitives. Or, the task designer can construct complex operators or operands for the task domain, translate them into primitives and then give them to GP. This is practical if capitalizing on existing knowledge of the problem may mean the difference between obtaining a solution or not getting one at all.

Conversely, there is cause to be cautious when specializing primitives. Specialization reflects an *a priori* rough notion in the task designer's mind of the structure of the solution in terms of subtasks or subunits of processing. A set of specialized primitives may preordain that only solutions in the form of this rough structure are successful whereas less specialized primitives could permit different program structure to successfully solve the problem. From an investigative viewpoint, designer knowledge makes it difficult to determine the power of GP.

Deciding upon Scope.

Programs in canonical GP are blindly composed (either by random creation or with blind crossover) so it is impossible to provide a local scope for a primitive: all of the parameters may be represented by any primitive in the set and all primitives can be used anywhere in a program (where syntax allows). This means that variables of a program (such as a loop variable) may be changed via side effects (e.g., in the loop body) rather than solely in a primitive intended for that purpose (e.g., via the loop increment parameter).

These side effects distort the structured behaviour of a program and may also have an impact on the effectiveness of crossover. A decision must be made to either "put up" with this state of affairs, use structure-preserving crossover (see Chapter 2

page 55), or enforce scope by a less transparent means by using global variables in various primitives but not including them in the primitive set. Both the latter choices require the designer to make a decision for GP that constrains expression and GP has no way of reversing the decision to gain more expressive capacity. On the other hand, both latter choices may also constrain the search space in a reasonable manner which offers a better chance for obtaining a solution.

To summarize the primitive set selection process, it is impossible to quantitatively determine whether a primitive is too constrained or specialized or whether scope is chosen with a minimum of bias. Choice is typically motivated by an intuitive trade-off between a feasible size of search space and a reasonable chance to achieve unanticipated eventual solutions. The GP methodology relies at various crucial junctures on designer expertise to focus its search space.

3.1.3. Primitive Sets for Assessing Hierarchical Process

A reasonable conjecture explaining GP's success is that it searches using a "hierarchical process". A "hierarchical process" is one which identifies and promotes modular reusable elements, builds composite higher level components of a hierarchy, and, guides their assembly into "hierarchical solutions".

"Hierarchical solutions" exhibit hierarchy in terms of control and structure. Hierarchical control is the execution of a task by the accomplishment of a series of subtasks. Each subtask in the series can be recursively sub-divided into a series of subtasks also. Often different subtasks are accomplished by repeated invocation of one pre-defined module that functions in a general manner but by parameterization applies its operations to different data.

Hierarchical control can be obtained by the top-down division of tasks into subtasks, the design of general purpose or specialized modules to accomplish each subtask, and then, the bottom-up composition of levels of subtasks using modules.

Programs are obviously hierarchical in control in an unimportant sense because a procedural or functional program follows a sequential execution path which branches to sub-divide area direct control of the algorithm. The control is also guided by conditional and iteration statements which cause the execution to branch or repeat.

True hierarchy in control, however, is more than the superficial execution sequence that underlies all programs. It is the observable, strategic direction of task

accomplishment. Hierarchical control is somewhat in the "eye of the beholder" yet can also be described in quantitative and qualitative terms. For example, one possible quantitative evidence of it is fewer statements in a program in comparison to another. Or, hierarchical control can be detected by qualitatively gauging the degree of statement or procedure reuse, the ease of functional description and the flexibility of the program to be extended in functionality. Not all this evidence is necessary nor does one type suffice for concluding that hierarchical control exists in a program.

For example, consider the task of writing a program to divide and multiply the pairs of integers "1 and 2", "2 and 3", "3 and 4", and "4 and 5" and display the results. One program to accomplish this task uses an arithmetic statement and an output statement for each division combination and each multiplication combination (16 statements in total). One would not claim that such a program exhibits hierarchical control. Conversely, another program that uses a loop (from index values 1 to 4) and each time through the loop multiplies the loop index value to itself plus one, divides the loop index value from itself plus one and outputs them (5 statements in total) would be considered hierarchical in control because it encapsulates one subtask (dividing, multiplying two numbers and displaying results) within a subtask of iterating and supplying two numbers at a time. If one wants to modify a program to an extension of the task (e.g., taking the power of one integer to the other in a pair or adding new pairs), the hierarchical control of the second program makes it easier to adapt. Statements can be inserted inside the loop to perform exponentiation, the loop indexing can be changed, etc. In contrast, to change the first program in the same way as its basic linear design requires inefficient effort: e.g., adding 2 statements for each new arithmetic task per pair of integers.

An S-expression evolved by GP can be truly hierarchical in control. In some examples the subtle richness of a GP solution is due to hierarchical strategy. For example,

- the algorithm of one GP solution to solve the Block Stacking problem is: 1) remove all blocks that are stacked, 2) one by one stack the needed blocks in the correct order. [62]

- another algorithm of another GP solution to Block Stacking more efficiently removes only the blocks to the point where the stack is correct and then stacks

the remaining blocks correctly. [62]

- The Boolean Multiplexer problem is solved by a GP program using a default hierarchy (i.e. a set of rules covering a variety of situations in which one subrule (called the default rule) handles a majority of the situations and one or more specific rules handle specific exceptional cases): the default rule handles the case of whether the Multiplexer address equals 3 which immediately splits the fitness cases 16:32. [69]

- An algorithm in a GP program on the task of classifying hydrophobicity in transmembrane protein sequences categorizes among factors that are similar to hand designed hydrophobicity constants.[41]

Hierarchical structure is a static characteristic of a solution rather than a dynamic characteristic like control. In programs, it is the existence of nested levels of specification. For example, the outermost level of a PASCAL program is the program template consisting of program name, constants, types, variables, functions, procedures and main program. The variables, constants and types of the outermost level are "global" in scope. Almost the entire program template can be recursively nested within any procedure or function (with the exception of a program name) to form another level. One consequence of hierarchical structure is scope. Nested variables and procedures only have scope within their enclosing procedure or function.

A superficial hierarchical structure is always present in a GP S-expression simply by virtue of the way in which GP assembles primitives. Each primitive is a primary element. Each subtree in the parse-tree representation of an S-expression is a composite element. However, we do not consider this hierarchicality sufficient to state that hierarchical structure is truly present because it is too superficial and merely a direct result of closure and how primitives are composed into GP programs. True hierarchical structure in GP S- expressions is not really possible without some extension that allows procedures to be defined.

There is an important distinction between hierarchical solutions and hierarchical processes: hierarchical solutions exhibit true hierarchical features (control and structure) while the notion of a hierarchical process refers to a specific manner in which hierarchical solutions arise. Expressed from the perspective of GP, hierarchical control is exemplified by strategic task decomposition and subtask composition and

even subtask redundancy. A hierarchical process in GP, if it exists, is a particular capacity in GP that identifies and promotes key primitives so that they become used in higher level composite combinations that perform logical subtasks, and, ultimately, assembles layers of composite subtasks that accomplish the task.

When subtasks are *a priori* defined, the important subprocess of subtask identification in a hierarchical process is not required. Because it is possible to start GP with primitives that already encode subtasks, it can not be firmly established whether GP proceeds in a hierarchical manner by simply examining its solutions for hierarchical character. Rather, in order to confirm or deny that GP proceeded hierarchically, it is necessary to ask whether GP can evolve a program starting from a set of primitives that demands their combination into subtasks plus the assembly of the subtasks. The primitives must not preordain the definitions of subtasks that could be formed. In other words, they must be "general purpose": GP could use different combinations of them to form different solutions. We use this framework in our assessment of whether GP proceeds hierarchically.

The particular versions of the sorting problem we shall use in this experiment (distinguishable by their primitive sets from **Sort-A** and **Sort-B**) are called **Sort-TH-0**, **Sort-TH-1** and **Sort-TH-2** where **''TH''** stands for "Truly Hierarchical" and the number indicates a level of primitive set generality (with 0 being the least general). The experiment uses the same test suite and fitness function as **Sort-A**.

In theory, a primitive set that is sufficient to solve sorting should contain: a general loop, (- var value), (**aref** array index) , (**if-lt** x y true-form), and (**swap** x y).[1] To simplify our task we do not experiment with this very general primitive set but use three increasingly more general primitive sets: **Sort-TH-0**, **Sort-TH-1** and **Sort-TH-2**. Should GP prove capable of discovering solutions with both primitive sets, the complexity of primitives in **Sort-TH-2** could be further decomposed and the robustness of GP retested from an even more general starting point.

In **Sort-TH-2** we make a deliberate decision NOT to use the following primitive:

```
(if-lt-swap index1 index2).
```

[1]Simplifying further, (**swap** x y) should be replaced with (**setf** x y) and a temporary variable should be made available. Thus a swap function would evolve by the following subtask: (**setf** tmp x) (**setf** x y) (**setf** y tmp).

This primitive would compare the elements of `*array*` at `index1` and `index2` and swap them if the first were less than the second. By linking comparison and exchange, this function is obviously a desirable building block in either an iterative or a recursive solution to sorting. But we want GP to discover it rather than start from it[2]. Since `if-lt-swap` does several things, for our purposes it is "too cooked". To meet the criteria of hierarchical process, GP should be capable of evolving a program which has itself discovered the conjunction of comparison and swapping in order to sort[3]. The evolving system should identify two simpler "building blocks", *compare* and *exchange*, as subprograms, and establish a larger building block by combining the subtrees through selection and crossover. We use distinct compare and swap primitives: `(if-lt value1 value2 true-form)` and `(swap*Array* index1 index2)` for this purpose. We choose `minus1` and `aref-*array*` (see Table 8, page 81) with reasonable confidence that the generality or the structure of the eventual solution will not be unduly influenced in advance. We use two iteration primitives that are not completely without challenge: they risk side effects and they are not very constrained but they meet our criterion of expressive generality (i.e., they are general purpose). They are `(do-until-up-with-*j* start-value end-value body-form)` and `(do-until-down-with-*i* start-value end-value body-form)`.

In `Sort-TH-1` we do use `if-lt-swap`. This primitive replaces `if-lt` as well as `swap-*array*` and allows `aref-*array*` to be removed.

In `Sort-TH-0`, in addition to using `if-lt-swap`, we also use more constrained and specialized versions of the loops. The starting values of the loop indices are fixed. The decrementing loop starts with the index of the last element of the array and the incrementing loop starts at 0. The primitive `*j*` is set to the value of the loop variable at the start of each iteration of the incrementing loop and the primitive `*i*` is set to the value of the loop variable of the decrementing loop at the start of each iteration. This behaviour is encoded inside the primitive and not subject to change by any parameterization of the primitive. Because the loop indices are copied to primitives,

---

[2]Using it would also allow us to decrease the search space by dropping `swap-*array*` from the primitive set.

[3]In fact, GP should also be able to discover the connection between an array and its reference function using an index. That is, `(aref*Array* index1)` is itself a building block which is implicitly included in the `if-lt-swap` primitive.

the primitives may be changed in the body of the loop but they are always reset and advanced just before the next iteration of the loop. The only parameters to each loop are the value which limits the iteration: **end-value**, and the form to be executed as the body of the loop: **body-form**.

All three sets of primitives are capable of expressing a correct sort program so they ensure another criterion of program discovery: the primitives must have the capacity to express a solution.

Problem Definition: `Sort-TH-0`

Task: *Sorting*

Primitive Set: **(minus1 var)** This primitive returns the value of one subtracted from its parameter unless the parameter is not numeric. In this case it returns **nil**.

(if-lt-swap-*array* index1 index) This primitive returns **nil** if its parameters do not evaluate to integers in the element range of variable ***array***. If they do, it exchanges the elements at the positions corresponding to the integers if the first element is less than the second and returns **true**.

(do-until-up-with-*j*-from-1 end-value loop-form) uses a private loop variable that starts at 1, increments each time through the loop and causes loop termination when it equals **end-value**. It sets ***j*** to the current value of the loop variable at the start of each iteration. The body of the loop is defined by **loop-form**.

(do-until-down-from-arraysize-less-one end-value loop-form) uses a private loop variable that starts at **(1 - *arraysize*)**, decrements each time through the loop and causes loop termination when the variable equals **end-value**. It sets ***i*** to the current value of the loop variable, at the start of each iteration. The body of the loop is defined by **body-form**.

***j*, *i*, *array-size*:** These primitives are variables and constants.

Test suite: Same suite as **Sort-A** and **Sort-B**.

Fitness Function Same as **Sort-A**.

Problem Definition: `Sort-TH-1`

Task: *Sorting*

Primitive Set: `(minus1 var)` This primitive returns the value of one subtracted from its parameter unless the parameter is not numeric. In this case it returns `nil`.

`(if-lt-swap-*array* index1 index)` This primitive returns `nil` if its parameters do not evaluate to integers in the element range of variable `*array*`. If they do, it exchanges the elements at the positions corresponding to the integers if the first element is less than the second and returns `true`.

`(do-until-up-with-*j* start-value end-value body-form)` uses the variable `*j*` as a loop variable. It sets `*j*` to `start-value`, and it always increments `*j*` at the end of each repetition. It always tests whether `*j*` $<=$ `end-value`, with an implicit `loop-test` of $<=$, to conditionally execute `body-form`.

`(do-until-down-with-*i* start-value end-value true-form)` uses the variable `*i*` as a loop variable. It sets `*i*` to `start-value`, and it always decrements `*i*` at the end of each repetition. It always tests whether `*i*` $>=$ `end-value`, with an implicit `loop-test` of $>=$, to conditionally execute `body-form`.

`*j*`, `*i*`, `1`, `*array-size*`: These primitives are variables and constants.

Test suite: Same suite as `Sort-A` and `Sort-B`.

Fitness Function Same as `Sort-A`.

Problem Definition: `Sort-TH-2`

Task: *Sorting*

Primitive Set: `(minus1 var)` This primitive returns the value of one subtracted from its parameter unless the parameter is not numeric. In this case it returns `nil`.

`(if-lt value1 value2 true-form)` This primitive returns the value `nil` if its first two parameters are not integers. If they are, it compares them and if the first is less than the second, it evaluates and returns `true-form`.

`(swap-*array* index1 index2)` This primitive returns `nil` if its paraemters do not evaluate to integers in the element range of variable `*array*`. If they do, it exchanges the elements at the positions corresponding to the integers and returns `true`.

`(aref-*array* index)` This primitive returns `nil` if its parameter is not a valid index of `*array*`. Otherwise it returns the element of `*array*` at `index`.

`(do-until-up-with-*j* start-value end-value body-form)` uses the variable `*j*` as a loop variable. It sets `*j*` to `start-value`, and it always increments `*j*` at the end of each repetition. It always tests whether `*j*` <= `end − value`, with an implicit `loop-test` of <=, to conditionally execute `body-form`.

`(do-until-down-with-*i* start-value end-value true-form)` uses the variable `*i*` as a loop variable. It sets `*i*` to `start-value`, and it always decrements `*i*` at the end of each repetition. It always tests whether `*i*` >= `end − value`, with an implicit `loop-test` of >=, to conditionally execute `body-form`.

Test suite: Same suite as `Sort-A` and `Sort-B`.

Fitness Function Same a `Sort-A`.

Experiment Title: Testing for Hierarchical Process in GP: `Sort-TH-0`, `Sort-TH-1` and `Sort-TH-2`

Problem Definition: *Sorting* with three different primitive sets named `Sort-TH-0`, `Sort-TH-1` and `Sort-TH-2`.

Run Parameters :

- population size: 300 and 500. For `Sort-TH-2` 1000 also.

- maximum generations: 50 after the creation of generation 0. For `Sort-TH-2` 100 generations also.
- maximum height of programs in generation 0: 5
- maximum height of programs generated by GP crossover: 15
- parents directly propagated each generation: 10% of population size
- conduct 30 executions with each primitive set and each population size as a run

	%-age Success	All Executions Inds Processed	Successful Executions Inds Processed
`Sort-TH-0`, pop = 300	70.0	9360 (5370)	6810 (4200)
`Sort-TH-1`, pop = 300	16.7	14640 (2130))	11400 (3000)
`Sort-TH-2`, pop = 300	0.0		
`Sort-TH-0`, pop = 500	93.3	10250 (7550)	9150 (6450)
`Sort-TH-1`, pop = 500	23.3	22750 (12980)	13700 (6750)
`Sort-TH-2`, pop = 500	0.0	25500	
`Sort-TH-2`, pop = 1000	0.0	51000	
`Sort-TH-2`, 100 gen	0.0	51000	

Table 9. Hierarchical Process in GP: All Problems

Results: Addressing the question of how well GP could perform using "general purpose" primitives, GP could solve sorting using the primitives of `Sort-TH-0` and `Sort-TH-1` but not with those of `Sort-TH-2`. For direct comparison, see Table 9 where the percentage of successful executions is listed in second column. The differences in success between primitive sets is statistically significant. With a population of 500, using `Sort-TH-0` GP found a perfect sorting program 93.3% of the time. When the number of individuals processed is examined, it is clear that, as expected, GP processes fewer individuals in order to find a solution to `Sort-TH-0` than `Sort-TH-1`.

Because the search space of `Sort-TH-2` is larger than the others (it has more primitives with more parameters), the demand for GP to use hierarchical process is

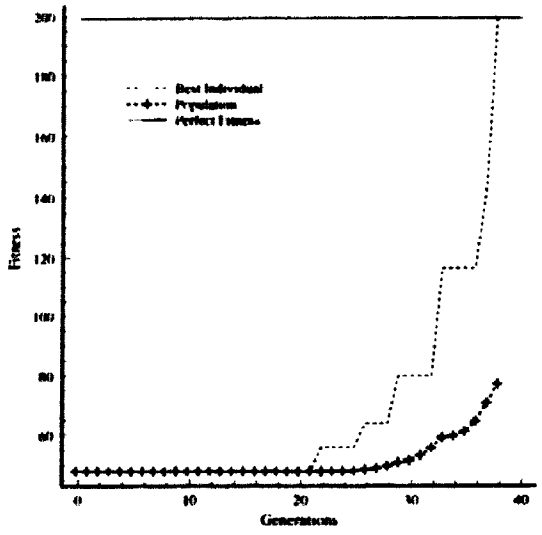


Figure 11. Sort-Th-0: Fitness of best individual and population in successful execution

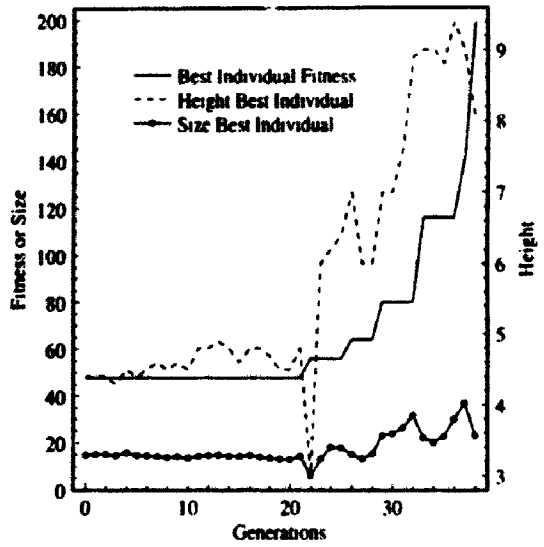


Figure 12. Sort-Th-0: Fitness, height and size of best individual in successful execution

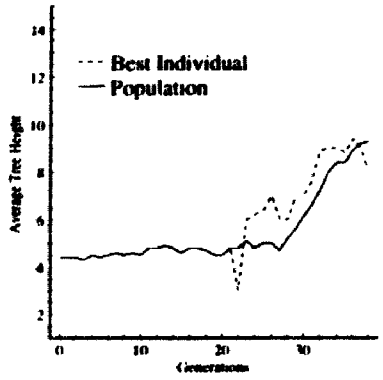


Figure 13. Sort-Th-0: Program height in successful execution

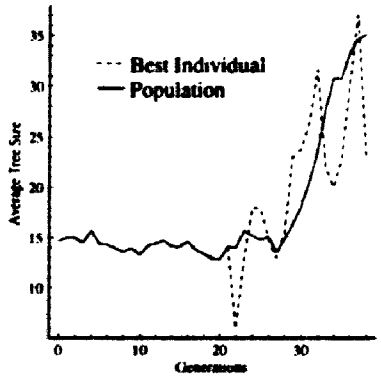


Figure 14. Sort-Th-0: Program size in successful execution

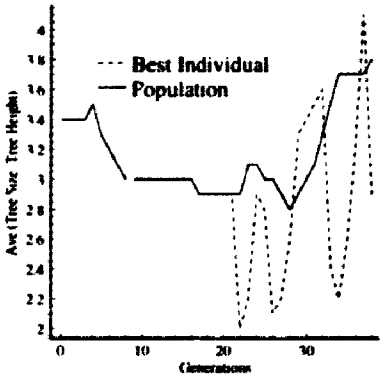


Figure 15. Sort-Th-0: Program size:height in successful execution

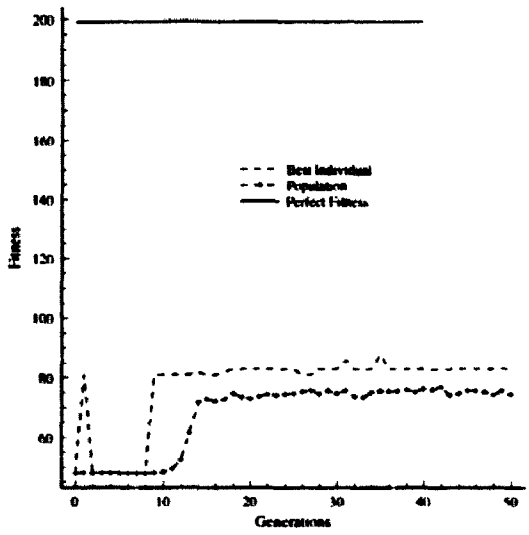


Figure 16. Sort-Th-0: Fitness of best individual and population in unsuccessful execution

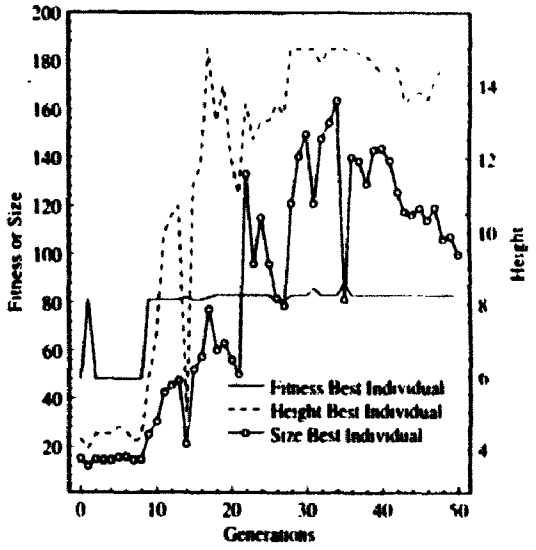


Figure 17. Sort-Th-0: Fitness, height and size of best individual in unsuccessful execution

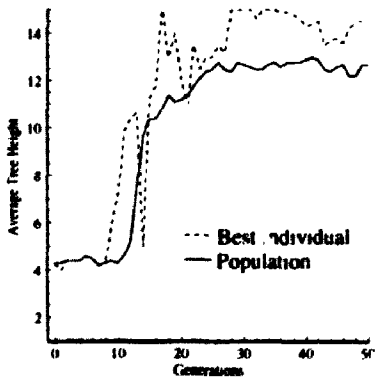


Figure 18. Sort-Th-0: Program height in unsuccessful execution

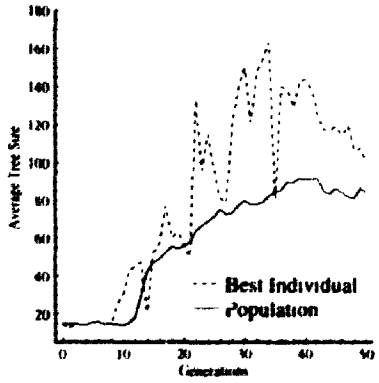


Figure 19. Sort-Th-0: Program size in unsuccessful execution

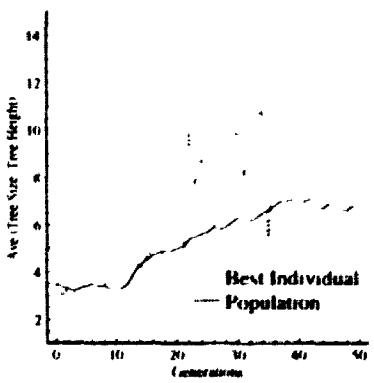


Figure 20. Sort-Th-0: Program size:height in unsuccessful execution

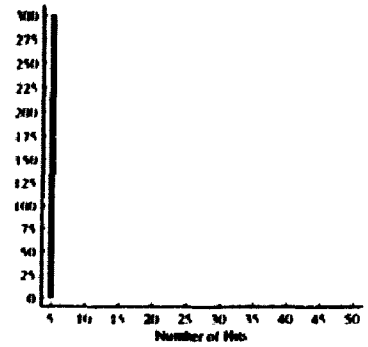


Figure 21. Sort-TH-0: Hits distribution, Generation 0

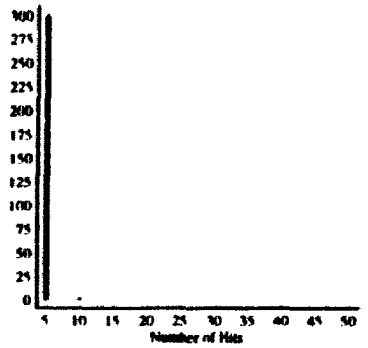


Figure 22. Sort-Th-0: Hits distribution. Generation 22

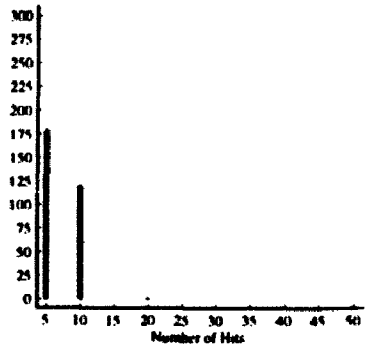


Figure 23. Sort-Th-0: Hits distribution, Generation 33

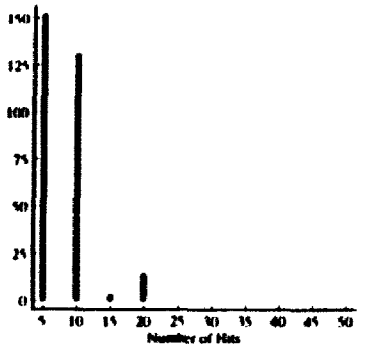


Figure 24. Sort-TH-0: Hits distribution. Generation 36

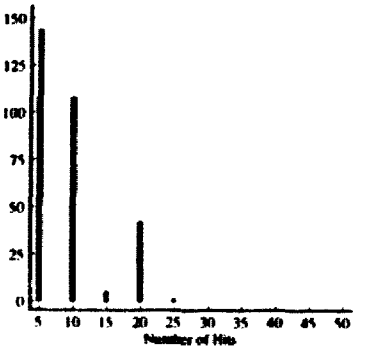


Figure 25. Sort-Th-0: Hits distribution. Generation 37

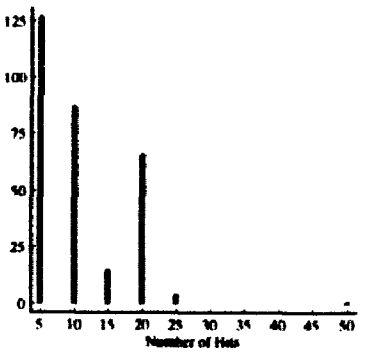


Figure 26. Sort-Th-0: Hits distribution. Generation 38

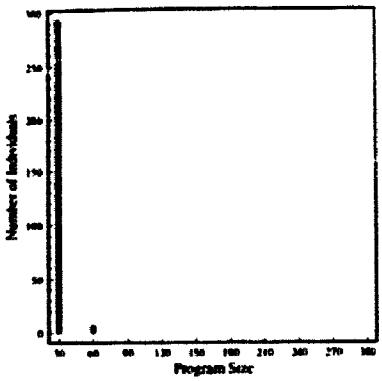


Figure 27. Sort-TH-0: Program size distribution, Generation 0

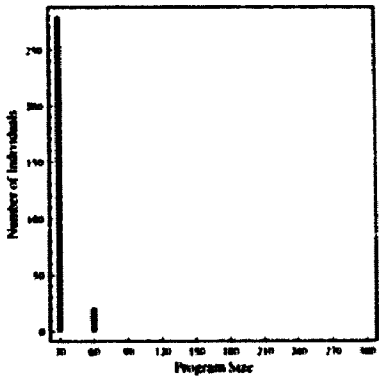


Figure 28. Sort-Th-0: Program size distribution, Generation 19

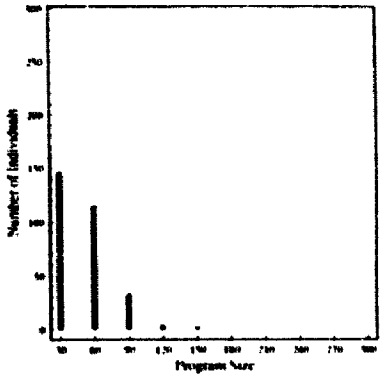


Figure 29. Sort-Th-0: Program size distribution, Generation 38

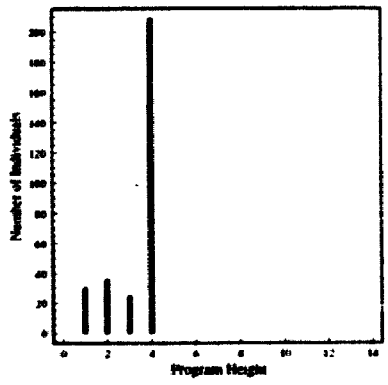


Figure 30. Sort-TH-0: Program height distribution, Generation 0

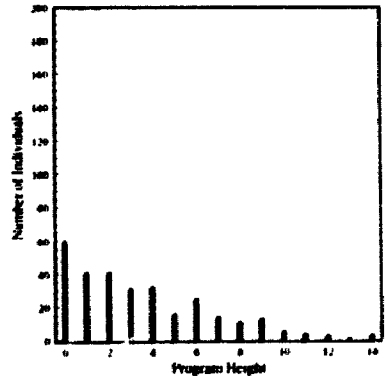


Figure 31. Sort-Th-0: Program height distribution, Generation 19

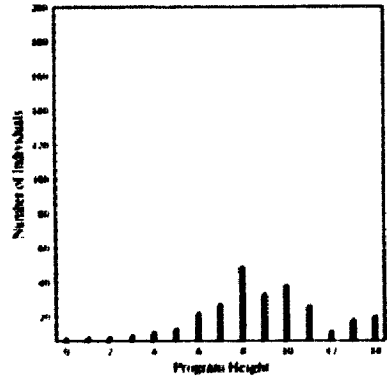


Figure 32. Sort-Th-0: Program height distribution, Generation 38

greater. However, the larger search space may, reasonably, be expected to require the sampling of more individuals. However, there are 10200 more individuals processed in executions with population of 500 compared to executions with population of 300 (when both executions proceed for 50 generations) and, consulting Table 12 which shows `Sort-TH-2` run data, there still is very little improvement from the run with the larger population. To further investigate, we also ran `Sort-TH-2` with a population of 1000 for 50 generations and with a population of 500 for 100 generations, in effect doubling the effort available to GP. The results are listed in the final two rows of Table 9. Despite doubling the effort, GP could still not find a solution.

For comparison we present a number of plots of various executions and summarize the overall executions performance with each population size in separate tables for each primitive set. Starting with the most specialized set of primitives `Sort-TH-0`:

- Table 10 provides a comparison of the runs with population size 300 and 500. The number of individuals processed did not statistically significantly differ with the two population sizes. Nor was there any appreciable difference in the program size, height and size to height ratio of the perfect programs. Standard deviations are in parentheses.

- Figure 11 and Figure 16 plot the fitness of the best individual and population by generation for, respectively, a randomly chosen successful and unsuccessful execution.

- To examine whether there is crude correlation between program growth and the success of an execution, Figures 12 and 17 plot, for the selected typical successful and unsuccessful execution respectively, the fitness of the best individual and the height and size of the best individual by generation. There is a suggestion that an increase in size and height is correlated with an increase in fitness in the successful execution. This correlation does not seem to be apparent in the unsuccessful execution.

- Figures 13 and 18 depict the height of the best individual and average height of the population per generation for the same executions (successful and unsuccessful). For these two executions there is an obvious difference in the trend.

In the successful execution the height increases slowly, whereas, in the unsuccessful execution, the height increases very quickly between generations 10 and 20 and remains close to the maximum. The height of the best individual after generation 25 stays slightly higher than that of the population.

- Figures 14 and 19 depict the size of the best individual and average size of the population per generation for the same executions (successful and unsuccessful). In the successful execution the average size of the population stays constant until generation 30 (approximately 16) after which it quickly rises to 35. In the unsuccessful execution, the size stays steady at 16 for 12 generations then steadily rises to 80 by generation 30 and stays between 80 and 100 for the rest of the execution. Smaller programs seem to be correlated with the success of an execution. In both executions the size of the best individual fluctuates greatly around the average size of the population.

- Figures 15 and 20 depict the ratio of size to height of the best individual and average ratio of the population per generation for the same executions (successful and unsuccessful). In the successful execution the population average ratio stays between 2.9 and 3.8 while, after generation 20 the ratio of the best individual fluctuates around it ranging from 2 to 4. In the unsuccessful execution, the range of the population average ratio is much wider: 3 to 7 and the range of the best individual is 3 to 10.5 with many large fluctuations. This may suggest that structural homogeneity in the population is advantageous for success.

- The distribution of hits (i.e., correctly sorted arrays), program size and program height to number of members of the population over the course of an execution can be animated in a histogram or shown in a series of plots. Figures 21 to 32 are series of hits distribution, size distribution and height distribution for the selected successful execution. With only three generations selected, it is obvious that the height distribution becomes normal whereas this does not occur with the size distribution.

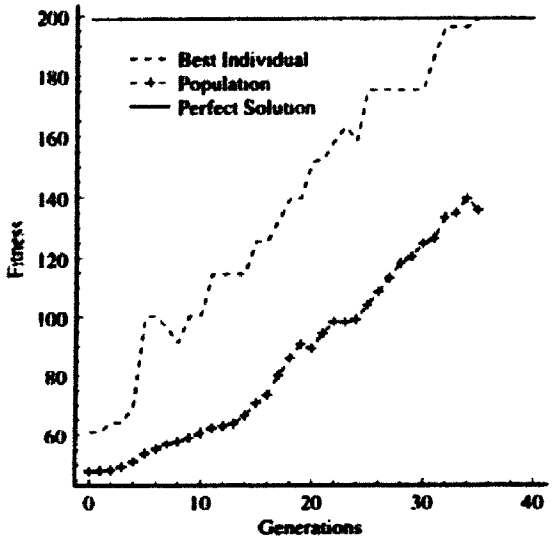


Figure 33. Sort-TH-1: Fitness of best individual and population in successful execution

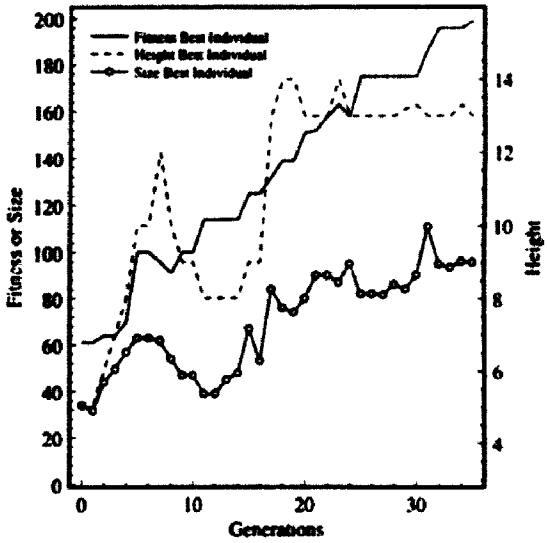


Figure 34. Sort-TH-1: Fitness, height and size of best individual in successful execution

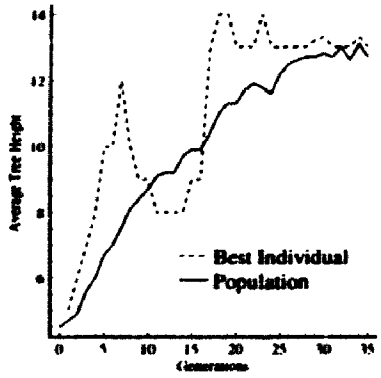


Figure 35. Sort-Th-1: Program height in successful execution

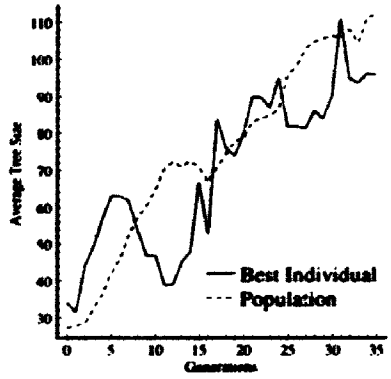


Figure 36. Sort-Th-1: Program size in successful execution

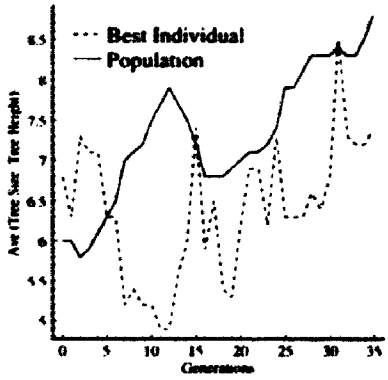


Figure 37. Sort-Th-1: Program size:height in successful execution

For **Sort-TH-1** we only show plots of a successful execution. The same type of plots except for distributions are provided. The case for correlated height, size and fitness is weaker but still suggested. The size of programs is about 5 times bigger than **Sort-TH-0** (average 46.7 compared to 252.9 for population size of 300). The height is also bigger: 10.5 versus 13.9. Table 11 provides a comparison of the executions with population size 300 and 500. The number of individuals processed

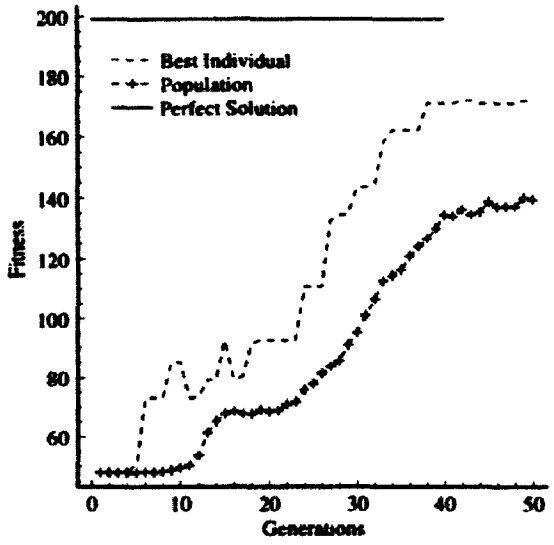


Figure 38. Sort-Th-1: Fitness of best individual and population in unsuccessful execution

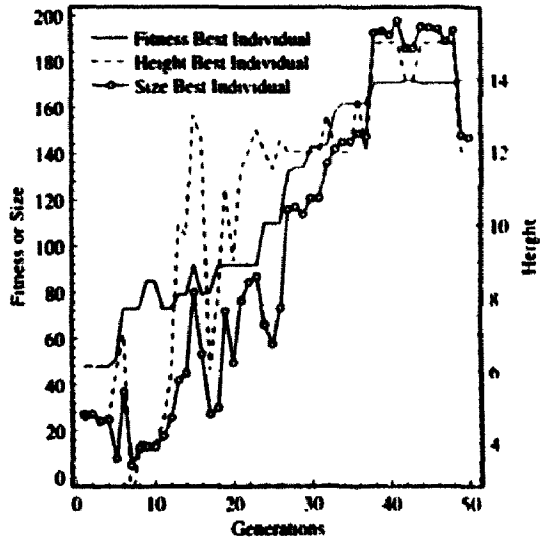


Figure 39. Sort-Th-1: Fitness, height and size of best individual in unsuccessful execution

did not statistically significantly differ with the two population sizes. Nor was there any appreciable difference in the program size, height and size to height ratio of the perfect programs.

Recall that `Sort-TH-2` was never solved by GP under our experimental setup. We show the plots of a typical execution in Figures 43 to 47 and the data of the set of executions in Table 12. Initially the test suite contains 3 correctly sorted arrays (hits) and 47 elements placed correctly (fitness 24.1%). With a population of 300,

`Sort-TH-0`	Pop = 300	Pop = 500
%-age Successful Executions	70.0 (46.0)	93.3 (25.0)
Successful Executions Individuals Processed	6810 (4200)	9150 (6450)
Best Ind: Height	10.5 (3.1)	10.5 (3.2)
Best Ind: Size	46.7 (36.0)	40.1 (23.6)
Best Ind: Size:Height Ratio	4.0 (2.1)	3.6 (1.3)
Population: Size:Height Ratio	4.0 (1.6)	3.7 (1.3)

Table 10. Hierarchical Process in GP: `Sort-TH-0`

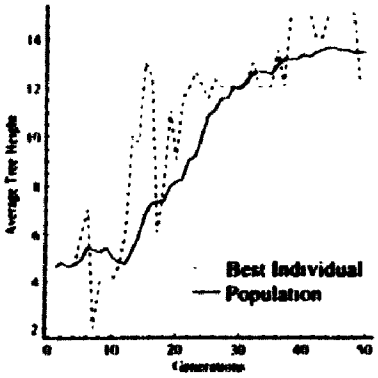


Figure 40. Sort-Th-1: Program height in unsuccessful execution

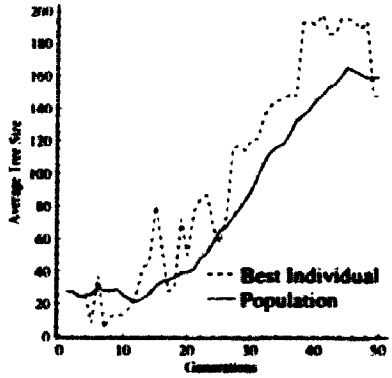


Figure 41. Sort-Th-1: Program size in unsuccessful execution

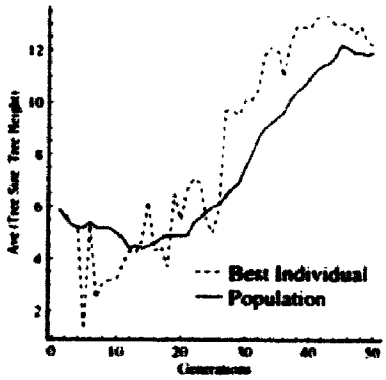


Figure 42. Sort-Th-1: Program size:height in unsuccessful execution

Sort-TH-1	Pop = 300	Pop = 500
%age Successful Executions	16.7 (37.0)	23.3 (42)
Successful Executions		
Individuals Processed	11400 (3000)	13700 (6750)
Best Ind: Height	13.9 (1.0)	12.9 (1.9)
Best Ind: Size	209.0 (73.2)	153.1 (66.8)
Best Ind: Size:Height Ratio	14.9 (4.8)	11.3 (3.9)
Population: Size:Height Ratio	14.1 (3.0)	10.8 (3.3)

Table 11. Hierarchical Process in GP: Sort-TH-1

the best execution succeeded in correctly sorting 18 arrays and placing 125 elements in the correct position yielding an (adjusted) fitness of only 62.8%. On average the best individual of an execution placed 40.5% of the elements correctly and sorted 8.3 (3.3) arrays correctly.

We observe that GP prematurely converges and processing more individuals does not appear to prevent this. Diversity in the population is not likely an issue because though scaling usually limits diversity in later generations of GAs, there is sufficient diversity in a GP population even among the candidates strong enough for selection. GP executions remain diverse because crossover does not take place on strictly denned structure boundaries. In fact, once the 60% neighbourhood has been reached, we see disparity in fitness, (implying diversity in programs) return to the population. On an

animated histogram the "undulating slinky movement from left to right" [62] halts and a downward wave to the left occurs.

Sort-TH-2	Pop = 300	Pop = 500	Pop = 1000	Pop=500 Gen=100
%age Successful Execs	0	0	0	0
Best Ind: Hits (max 47)	8.3 (3.3)	9.0 (4.0)	10.8 (5.0)	9.8 (3.3)
Best Ind: %-age fitness	40.5 (7.9)	42.0 (9.5)	46.4 (10.1)	45.0 (7.8)
Best Ind: Height	14.0 (1.5)	13.6 (1.3)	13.7 (1.4)	14.6 (0.6)
Best Ind: Size	200.6 (80.7)	199.2 (66.4)	237.0 (104.7)	311.1 (84.0)
Best Ind: Size:Height Ratio	14.2 (5.2)	14.6 (4.3)	16.9 (6.8)	21.8 (5.6)
Population: Size:Height Ratio	13.6 (3.8)	14.4 (3.5)	15.4 (3.4)	19.8 (5.1)
Best Execution %-age fitness	62.8	65.3	78.9	57.8
Hits (max 47)	18	18	31	17

Table 12. Hierarchical Process in GP: Sort-TH-2

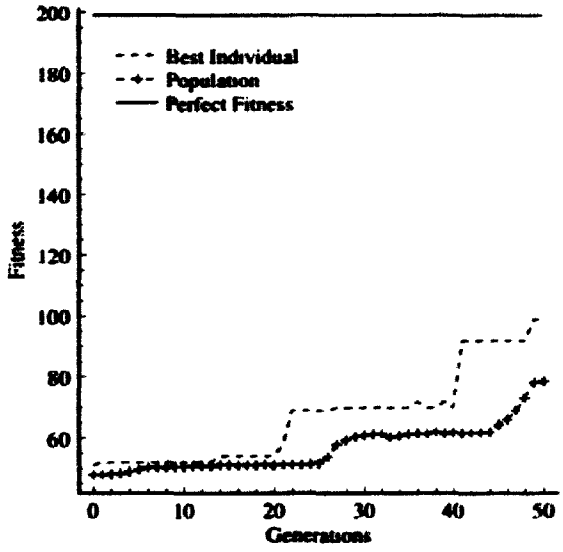


Figure 43. Sort-Th-2: Fitness of best individual and population in unsuccessful execution

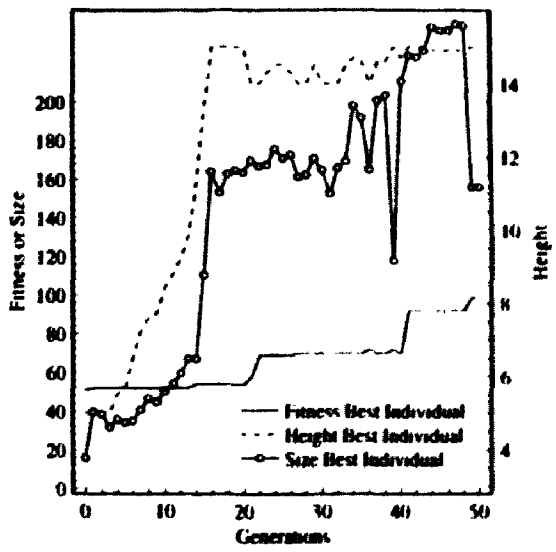


Figure 44. Sort-Th-2: Fitness, height and size of best individual in unsuccessful execution

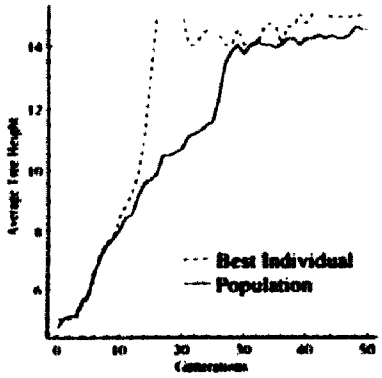


Figure 45. Sort-Th-2: Program height in unsuccessful execution

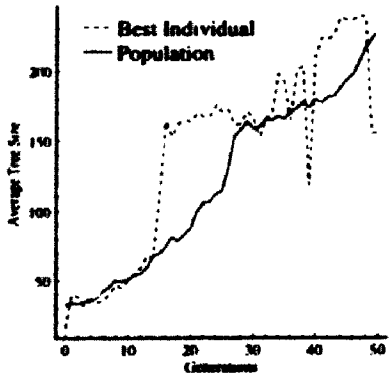


Figure 46. Sort-Th-2: Program size in unsuccessful execution

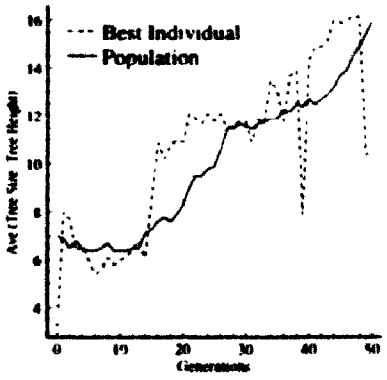


Figure 47. Sort-Th-2: Program size:height in unsuccessful execution

3.1.4. Assessing the Presence of A Hierarchical Process

These experiments reveal that canonical GP can not robustly find a solution with a truly "general purpose" primitive set. This premature convergence to a non-optimal solution could be due to a number of complicated factors. First, note that a subtask is defined collectively by a subprogram of primitives and by the behaviour of these primitives in the setting of a larger program. Plausible explanatory scenarios are:

- It may be the case that a subprogram of primitives which appear to function as a subtask only perform as such intermittently. That is, in different settings of surrounding primitives, sometimes they do perform the subtask and sometimes, they do not. The sensitivity of primitive behaviour to context because of the nature of programs lends this scenario plausibility. In one generation a subtask may be discerned as valuable and preserved and promoted by crossover (i.e., programs containing the subprogram and exhibiting the subtask behaviour are fitter and favoured by selection to propagate as "parents", the subprogram of primitives is copied from a parent into one child and "grafted" from the same parent to another child) but, in the next generation, in the context of a different program, the subtask disappears because the subprogram of primitives no longer behaves in the same manner as before. GP can not discover a subtask because the group of primitives specifying the subtask do not consistently perform the subtask function in all program contexts. GP does not have any

explicit mechanism for concurrently preserving the context of a subtask along with the subtask itself.

- Assuming that a subprogram of primitives does consistently, regardless of context, perform a discernible and useful subtask , it may be the case that the subprogram is disrupted by crossover despite selection promoting programs which contain it. Therefore a subtask is not promoted over generations and does not become more prevalent among programs in the population as generations proceed.

  To elaborate on this conjecture more precisely, it would be necessary to know, when they are of equal size and fitness, whether GP crossover promotes some subprograms faster than others on the basis of their embedding structure in programs. Unfortunately, this is not known.

  One facile but incorrect conjecture would be that GP crossover promotes a subprogram that is a subtree of primitives faster than a subprogram that is "fragmented" throughout a program (assumin they are equally fit and of equal size). The conjecture is incorrect because, in equal disruption likelihood for differently embedded subprograms is not guaranteed *even if the subprograms have the same number of primitives.* A subtree could be embedded in a small tree and a fragmented subprogram could be embedded in a larger program. In this case, a fragmented subprogram would be promoted, on average, more often than one that is a subtree.

  Consider the case when GP crossover promotes a subprogram that is a subtree of primitives faster than a subprogram that is fragments because it has a lower likelihood of disruption. The subtask that is a fragmented subprogram would not be propagated by GP crossover despite its equivalent value and GP would not discover and promote it.

- A final scenario places emphasis on the influence of the GP crossover bias in crossover point selection and on the maximum tree height parameter. These two factors introduce new non-linear complications to GP. They could be sufficiently influential to interfere with the promotion of fitter than average subtasks that

have low probability of disruption despite the behaviour of GP crossover and selection.

These scenarios, the experiment results, and the complicated nature of the GP algorithm lead us to suspect that the GP algorithm we have assessed is not as general and powerful with respect to hierarchy as it may appear from many successful examples described in the literature. GP is only hierarchical in the superficial sense that it exploits the hierarchical representation of a program as a tree and sometimes discovers programs with hierarchical control. GP is most often successful if a solution does not require hierarchical subtask process to efficiently identify and promote subtasks or if it is *a priori* given "high level" or complex primitives that encode subtasks. Complex primitives, because they are encapsulated, are protected from disruption and thus protect subtask behaviour allowing GP to simply discover a correct combination of subtasks into the problem task.

In Chapter 1 we supplied six reasons for conjecturing that a hierarchical process may proceed in GP. These reasons can be revisited:

Reason 1. Hierarchical solutions are typically products of a hierarchical process and GP sometimes produces hierarchical solutions in terms of control.

In our experiments which demand that solutions must be discovered using a hierarchical process, we have observed no evidence of hierarchy in solutions.

Reason 2. GP may derive its search efficiency from a hierarchical process.

In our hierarchy experiments GP did not exploit a hierarchical process in terms of solution design. When the search space was increased, even though a hierarchical process could have been used to search efficiently, GP could solve the sorting problem with our most general primitive set.

Reason 3. GP crossover may contribute to a hierarchical process because it is based upon tree representations of programs.

In our experiments we observe no influence by GP crossover in contributing to a hierarchical process though our methodology does not focus upon this aspect. In Section 3.3 we shall explain the research of [124] which does concentrate on the contribution of hierarchy to GP crossover.

Reason 4. GP may require a hierarchical process for program design because humans do.

GP seems to rely upon its evolution-based algorithm for power in discovering programs as opposed to a hierarchical process similar to human program design.

Reason 5. GP's simplified model of evolution may be sufficient to inherit a hierarchical process that is intrinsic in genuine evolution.

This does not appear to be true.

Reason 6. A GP equivalent of the GA Building Block Hypothesis could justifiably be assumed to operate in GP because GP is a kind of GA. If it is, a hypothetical building block search process could identify subtasks, promote them and assemble them in GP.

We shall defer the conjecture that a Building Block Hypothesis is justifiable for GP to Chapter 4. Since we do not observe a subtask level building block process in our experiments, we at least conjecture that there is no such process based upon a schema based building block hypothesis.

3.2. Improving Hierarchy in GP

Extending GP to foster hierarchical evolution will not, of course, change the basic computational power of GP's solutions or the computability of GP. However, it could lead to improved algorithms in terms of scalability, computational effort and principled organization of solutions. These issues are sufficiently valuable to pursue the idea.

On the issues of scalability and computational effort, hierarchy improves the capacity of a search algorithm to effectively search a space. Simon [108] gives an elegant example of the efficiency of hierarchy in his watchmaker parable. The identification of reusable components for solution assembly eliminates redundant work. One solution suffices for equivalent tasks. Once a component of a hierarchy has been "debugged", reuse of that component does away with repeating the effort.

On the issue of principled organization, GP does not currently produce any solutions that are realistically comparable to solutions designed by humans. Some

people would claim this is an asset by responding that a GP solution does not need to solve a problem in a way recognizable or comprehensible to humans: One should not insist that the way humans phrase solutions is the best or only way. Moreover, there may be phenomena in our environment which we will never be able to understand given the perspective of human cognition. An approach demanding comprehensibility and design principles is too restrictive because there may be principles opaque or unknown to us.

This response is debatable. If we regard GP as an artificial system of program design (i.e., automatic programming) we can legitimately desire integration of its results with "natural" (i.e., human written) programs and software engineering practices of validation, maintenance and library building. Thus we want sound algorithms which are comprehensible. The properties of a good algorithm entail a logical and efficient decomposition of problem elements complementary to an efficient control strategy which combine to formulate a coherent robust approach. For example, algorithms exploit useful functions which have been generalized, they execute with a tolerable computational complexity and they handle special cases simply.

When the goals of automatic programming are put aside and the impetus to use GP is to study and observe adaptive behaviour it is again reasonable to desire some degree of principled organization. GP models evolution and genetics because evolved genetics-based natural systems are robust, successful examples of "solutions" for difficult problem domains. Biological systems commonly demonstrate reasonably adequate genotypic and phenotypic competence in hazardous, noisy and often unpredictable environments. They are composed of integrated subsystems and it is often possible to interpret the function of each subsystem to account for the overall behaviour. Since challenging machine learning problems have the same requisite of efficiency and plasticity in noisy and novel domains, nature indicates that a solution of principled operation and principled organization may be advisable.

Canonical GP uses only fitness proportionate selection and a crossover-based form of genetic propagation as its computational version of evolution. This simplification is one reason why it falls short of achieving programs that are as well constructed as natural systems. Thus, a new question is:

How can a hierarchical process be incorporated into the GP paradigm?

The oldest approach to making GP more hierarchical focuses upon encapsulation and decomposition. In Chapter 2 we described the GLiB system of Angeline [7] which used a "compression" operator and "expansion" operator as well as the "define-building-block" operator of Koza (page 66). Encapsulation and decomposition operators are applied at random to 10% of the population after crossover. A subprogram is chosen for encapsulation by virtue of being a subtree. A new primitive (also called module) which may or may not have parameters (depending upon whether a decision is made to entirely encapsulate the subtree or "cut it off" at a certain depth) is created to replace the subtree. Afterwards, because the new primitive encapsulates a subprogram, its function will never be disrupted by crossover. It will, however, be subject to selection. The impact of these two effects will be that useful sub-tasks are identified among the new primitives and exploited for solutions.

Koza reported that his "define-building-block" operator did not improve the probability of success nor the amount of computational effort required to solve the 6-Bit Boolean Multiplexer problem. He also made a general statement that this approach was not worth pursuing [62].

Angeline used this approach to solve the classic Tower of Hanoi problem and to evolve programs to play and beat an automated Tic-Tac-Toe opponent. In the Tower of Hanoi problem he reported that modules consistently improved a variety of intermediate problem configurations. True hierarchical structure was present but discernable hierarchical control was absent. He reported:

> Unfortunately, more specific properties have been extremely difficult to identify. One observation is that the evolved modules do not employ the typical conceptual breakdown for playing Tic-Tac-Toe. ... the program modularity that results is not amenable to normal analysis techniques. As expected from an evolutionary process, the usage and semantics of the evolved language is extremely non-standard. For instance, several modules encode numerous side effects in the test position of a conditional. In short, the resulting programs, while containing multiple modules, are not examples of modular programming but share more commonality with the distributed representations of procedural knowledge found in connectionist networks.

Overall, this approach does not produce encouraging results. It would be rea-

sonable to expect both improved computational efficiency and hierarchical solutions neither of which seem to occur. Why?

The approach relies upon the existing "survival of the fittest" and reproduction mechanisms in GP to identify, select and exploit the useful sub-tasks among the randomly created new modules and use them in combination with existing primitives. It creates two levels of search: one explicitly for subtasks and, a second, for a solution using existing primitives and subtasks. Both levels of search must be supported by this single set of mechanisms.

Regarded in this perspective, several reasons can be suggested for the lack of success:

- Given that "normal" GP operators (e.g., GP crossover) are used much more often than the "hierarchy" operators (i.e., a ratio of 9 : 1) the population size is not large enough, nor is the time scale of the search sufficiently long. There is no basis to the choice of applying operators with this ratio. It is plausible that in nature, the operator ratio is equal allowing multilevel search to take place over relatively huge populations on a relatively long time scale. Unfortunately, increasing the population size, maximum number of generations or operator ratio is too simplistic a cure for GP because the algorithm is complicated by program growth. If it is run for more than 50 generations, the programs become so large and have so much epistasis and "junk" primitives that the algorithm ceases to be effective at improving population or best individual fitness. Large population sizes pose computational demands that can be unrealistic or that make it worthwhile pursuing alternative solution methods.

- Once two levels of search exist within GP it is hard to expect a single set of mechanisms functioning with only one time scale and one problem environment to aptly coordinate them. In other words, plausibly a second level of fitness function i.e., one that judges more explicitly the value of modules should coexist with the primary GP one. And, a separate selection mechanism might be necessary.

  Implementing a two level environment (by two fitness functions for example) poses problems. The second level fitness function would have to be specified *a priori* and this would seem to force the designer to identify what subtasks

are desired. It also seems impossible to identify appropriate time scales for each level of evolution. Furthermore, other aspects of GP might hold this sort of approach back: GP offers no model for how subpopulations with different fitness functions and time scales should interact. All extensions collectively might swamp the basic power of the algorithm because they vastly increase the search space size and depend upon designer foresight.

A second approach is to use Automatically Defined Functions. They are described in [70] and summarized on page 68 of Ch. 2. ADFs are an explicit mechanism for improving the hierarchical structure of solutions but they do not explicitly guide hierarchical process. Koza is careful in his claims: he states that GP's solution process can be *interpreted* (my emphasis) as hierarchical but not that it actually *is* hierarchical (see Main Point 1 quoted on page 70). ADFs do not explicitly guide logical sub-task formation but they allow GP to be scaled (see Ch 2).

A third approach is introduced by [99] and named "Adaptive Representation GP" (AR-GP). The authors suggest that the fitness of a subprogram can be estimated by

- the average fitness of the programs which use it
- some designer provided criteria
- its performance on the subset of test cases for which it has dependent variables

In their experiments they use the last criterion listed above. AR-GP tracks the fitness of small subprograms (only those that are complete subtrees of small maximum height). When a subprogram achieves perfect fitness, it is encapsulated to compose a new primitive and the algorithm enters a new epoch. At the start of an epoch, the weakest members of some portion of the population are replaced by new programs that are randomly created using the updated primitive set. It was observed that the frequency of fit subprograms increased in the population over time providing validation of the basic idea of promoting fit subprograms. The system was evaluated on the `even-n-parity` problem in which the task is to "find a logical composition of primitive Boolean functions that computes the sum of n input bits over the field of integers modulo 2". AR-GP was able to solve higher order `even-parity` problems that GP could not . It could solve all problems requiring less generations to find a solution with 99% probability and using smaller programs compared to GP with

ADFs. While the AR-GP results are encouraging, one particularly troublesome issue is that the approach relies upon a global memory based mechanism to detect building blocks which is not a plausible natural mechanism.

3.3. Knowledge-Based Primitives and Fitness Function Design as Factors in GP's success

3.3.1. Deriving Knowledge-Based Primitives from First Principles

Any claims that, when simply "left to its own devices", using obvious general purpose operators and operands of straightforward origin in the problem domain, GP robustly solves a wide spectrum of problems can be questioned in light of Section 3.1 which shows how GP demands knowledge intensive work of its task designers. The generality of GP is constrained by this demand and this constraint promotes skepticism as to the true power of the search algorithm.

The design of the primitives for a GP experiment requires modeling operators and operands from the problem domain. When the operands of the problem domain are simple objects that can be designated by constants or variables, their choice is not open to criticism. Naming them in order to abstract a solution is not problematic. However, operands which are not directly from the problem domain but which require knowledge-based definition are more contentious. For example, consider the Block Stacking problem which uses the primitives `nextNeeded`, `topCorrectBlock` and `TopBlockOnStack` as operands. These operands are sensors which are knowledge based: they require some preprocessing of knowledge concerning the problem before being defined. How operators are expressed with primitives is another source of contention. If any of the operators are expressed by combinations of primary problem domain operations, GP has again started from input which is more specialized than implied by the general claim. These primitives may be recognized by their being written as special forms rather than being built-in programming language statements.

From the perspective of judging the power of GP, two specific objections arise from using knowledge-based operators and operands: First, GP is using a language $L\prime$ which is a specialized subset of the programming language, L, that is used to write

the primitives. $L\prime$ may not include general elements of L and this may constrain the flexibility of solution expression. Second, $L\prime$ expresses knowledge-based properties of the problem domain which GP exploits in its adaptation process. A skeptic can fairly argue that GP may not have been able to solve the problem without the guidance this knowledge provides.

For program discovery, domain dependent knowledge can be claimed not to exist in programs when the primitives of $L\prime$ are applicable to many different domains. A methodological policy to ensure control over this aspect of $L\prime$ might be: Operands and operators in the problem domain must be directly represented by a primitive data type or form of L. For example, in LISP, the policy dictates that $L\prime$ can not include primitives that are user written macros or user written special forms, but it may include built-in functions such as `IF` and `cond`. When GP solves the problem with such primitives, it does so by means of its own adaptive process and without dependence on domain knowledge.

We could now revisit GP problems and clearly evaluate whether the primitives are knowledge-based by examining whether this policy is followed. GP seems to have solved the 6-Multiplexer problem without problem specific knowledge. The primitives are `AND`, `OR`, `NOT`, `IF`, `A0`, `A1`, and `D0` ... `D3`. Both the operands (registers) and the operators (built-in Boolean operators) come directly from LISP.

Conversely, referring to the policy, it allows us to recognize that the primitives of the Block Stacking problem are knowledge-based. The special forms or macros `DU` (do-until), `NOT`, and `EQ` are built-in forms of LISP so they conform to the policy. However, blocks are not encoded using variables. They are expressed by special forms which process sensor information and return a block. Two sensors, `TB` (top correct block) and `NN` (next needed), base their respective responses on specific assessments of the stack. The `CS` (block on top of stack) sensor could be coded using the LISP `car` function but it is hard coded to access the stack. The operators `MS` (move-to-stack) and `MT` (move-to-table) reference the stack and table but not as parameterized operands. Neither the stack or the goal list are operands in the Block Stacking primitive set. This absence glaringly points out that the designer has shown GP a problem specific way of using them.

To conform to the policy, GP would have to solve the Block Stacking problem starting with variables `*block*`, `*stack*`, `*goal-list*`, `*table*`, an assortment of

list operators to iterate, add, delete and access (since the stack, goal list and table are lists), a conditional statement (e.g. if-then-else), and NOT and EQ for comparison. This seems far too demanding! No one else expects to solve the Block Stacking problem from this low a level.

At this point it is fair to backtrack and say there is nothing wrong with using an $L\prime$ that does not conform to our policy, *except* that it implies that claims of GP's power have to be qualified: $L\prime$ should be recognized as the language used by GP, not L. This ensures a more accurate assessment of GP's true power. It is always reasonable to scrutinize $L\prime$ to see if it conforms to an adequate level of general functionality as typically seen in programming languages. Does it contain primitives which are applicable to many different domains and which are sufficient for many different compositions? Does it contain epistemologically valid classes of data (i.e. types) and operators which are not particular to a narrow domain of problems and do not employ knowledge of a specific problem domain? If so, accurate claims concerning the true power of GP can be made without it being argued that GP is assisted by domain dependent knowledge.

Alternatively, it is also sufficient to establish that the knowledge-based operators and operands may themselves have evolved. One would then be showing that an "evolutionary pathway" exists from the general purpose statements of a programming language to the primitives. It should be remembered that such a pathway does not have to solely arise from selective pressure to solve the particular problem at hand. It may be the case that existing helpful small sub-tasks arising for other reasons could be exploited rather than be built once again from scratch. In GP it would be sufficient to "reverse engineer" any primitive in $L\prime$, i.e., show that it could arise from a more general set of primitives using GP.

However, while reverse engineering produces an account of how a particular primitive could have arisen, it would be more interesting to discover successively more complex knowledge-based primitives. In other words, one could ask what series of languages would evolve by starting from "scratch" and building new languages using primitives taken from solutions in the previous language. This would most likely involve hand selecting the elements of the new language and the fitness functions because, at present, it appears that there is no way GP could perform this processing automatically.

3.3.2. Using Knowledge-Based Primitives

If the primitives used in representative GP problems are more specialized to the specific domain of the problem than the knowledge incorporated and used by other machine learning approaches, GP would not be as powerful a "weak" method as these other methods. It would support a claim that GP is less capable of deriving rich behaviour starting from knowledge impoverished conditions than them.

Upon careful examination most primitive set choices are naturally directly derived from the domain or they express domain knowledge that is not substantially different from the knowledge supplied to other paradigms. For example, in the GP experiment to learn wall-following behaviour, the same primary motor functions and sensor data were used as in the original subsumption approach [68]. The Block Stacking problem does use specialized primitives but these primitives precisely model the same operators and operands used to solve the problem with the original planning system of [85]. It is only because GP is able to solve a broad array of problems that it has the appearance of requiring substantial domain knowledge. This impression simply arises from observing all of the problems together. Other paradigms do not give the impression of dependence upon domain knowledge because they are typically demonstrated with only a few examples and this prevents it from being noticed. If another existing paradigm could solve all the problems GP can (which none can), the same dependence upon domain knowledge would be apparent.

3.3.3. Designing a Fitness Function

The fitness function clearly has a large impact on the success of GP but to date the research community has figured out little useful information concerning its design. The answer lies in a better understanding of how different fitness functions interact with GP by changing the character of the search space in terms of factors such as: the number of global optima, the number of local optima, the size of "basins" (a basin is defined by the solutions from which a "hill climb" will result in finding one global optimum), and the fitness values of solutions within a neighbourhood defined by a search operator.

Altenberg [2] has shown that GP is successful when the fitness distribution of the set of offspring that could possibly be produced by crossing over a pair of parents

has a large upper tail. The penultimate secret of GP would indicate how to adjust a fitness function to improve the likelihood of such distributions among parent pairs. It may be too difficult however and in the immediate future it is more likely that improvement could be gained by pursuing changing a search operator given a fitness function.

3.4. GP as a GA for Program Discovery

Given that GP is not a true hierarchical process, it remains necessary to explain GP's success aside from designer oriented factors such as primitive selection, test suite design and fitness function definition. GP is a GA with choices made for the sake of accomplishing program discovery. These choices are:

- GP uses a program-based encoding for solutions
- GP crossover
- GP manipulates variable length solutions
- GP solutions do not have a direct feature correspondence with each other

It is useful to consider which of these design choices are necessary for GP to work, which are simply convenient, and which are simply good tradeoffs.

3.4.1. Program-Based Encoding

GP directly manipulates programs composed of primitives. This is a favourable design decision. A different choice would result in substantial inconvenience and require design deliberation. There is no way to conclusively argue that changing this decision would significantly alter GP's success though it may impact its efficiency.

3.4.2. GP Crossover

One obvious aspect of GP appears to be its crossover operator. However, is this choice of operator over other GA crossover operators superficial or significant to GP's success? The question was pursued by [124] and the difference was found to be superficial. The EP-I system is a GA that solves program discovery by choosing a

representation that embeds a program of program discovery primitives into a fixed length linear string. Programs are able to vary in length because not all positions of the string are always interpreted as the program. The embedding works in a manner that permits linear-based GA crossover operators (e.g., one-point, uniform) to be applied to parent strings and produce directly syntactically correct offspring that could differ in size and structure from their parents. The net effect of those GA crossover operators is the same as GP crossover but the representations for that purpose differ. In terms of probability of success, on a small suite of standard GP problems (e.g., 6-Mult and Block Stacking), no statistically significant difference between EP-I and GP was found. Thus it was concluded that the use of a tree or hierarchical representation for crossover is, contrary to intuition, not essential to GP's power but merely a convenience that allows program size and structure to change.

3.4.3. Variable Length Solutions

Because GP processes programs directly, it can explore solutions that differ in size and structure. Program discovery needs this flexibility because many different programs can be written to all accomplish the same task and it is impossible to know in advance the minimal or optimal size of a program. Any restriction of a search space in size or structure may actually make the problem more difficult to solve. We conjecture the choice to use variable length solutions in GP is a crucial feature in its success. We shall exploit variable length solution encoding when we try different search algorithms for program discovery because it is arguably a necessary property.

3.4.4. Feature Correspondence Among Solutions

As observed in Chapter 2 (page 62), in GP two programs do not have a strict one to one correspondence in syntactic structure or behavioural structure. By using primitives and its particular means of assembling them into a program or creating programs with GP crossover, GP makes a tradeoff: it allows expressive flexibility (i.e., programs differ in syntactical and behavioural structure) but foregoes correspondences among programs that could provide a basis for "true" combination of them via crossover. It seems clear that this expressive flexibility is essential to accomplish program discovery because the structural nature of the desired program is not known in advance.

Therefore, this tradeoff seems key to fostering GP's success.

3.5. Chapter Summary

GP's success is strongly dependent upon expedient primitive and test suite design and fortuitous fitness function design. Based upon the experiments detailed in this chapter, we observe that GP does not exploit a true hierarchical process. GP is a GA outfitted for program discovery. Several design decisions in equipping the GA play a critical role in its ability to solve program discovery: GP manipulates programs directly, searches a space of solutions which vary in length and structure, and trades off syntactic and behavioural structural correspondence among programs for expressive flexibility.

CHAPTER 4

The Troubling Aspects of a Building Block Hypothesis for Genetic Programming

In Chapter 3 we pursued experimental evidence that GP conducted a hierarchical search process. One of the reasons for our conjecture was that a GP equivalent of the GA Building Block Hypothesis perhaps could be assumed to operate in GP because GP is a specialization of a GA.

We precisely define a schema in GP and derive a lower bound on the growth of the expected number of instances of a GP-schema from one generation to the next. Following precedent of the literature for GAs which use fixed length binary strings, we refer to this as the GP Schema Theorem (GPST). We also wish to show that although a notion of building blocks arises from an interpretation of the GPST, it is questionable whether such building blocks reliably exist throughout the course of a GP run. Finally, we emphasize that, as with GAs that use fixed length binary strings, hypothesizing building block combination requires greater liberty with the interpretation of the Schema Theorem than is justifiable.

Our investigation is motivated by the historical precedent of the Schema Theorem and BBH as an explanation of the search power of GAs [46, 32]. Holland's analyses have been the foundation for more precise explanations (some diverging from a schema-based approach) of GA search behaviour. GA theory for fixed length binary strings promises to be a useful source of analogy for GP because, just like other GAs it uses the same central algorithm loop which applies the basic evolution-based genetic operators and both act as a "shell" which accepts fitness function and problem

encoding as parameters. Other GAs and GP use genetic exchange within the population (crossover) and fitness-based selection (and both have been widely employed with fitness proportional selection). The similarity of operators and central algorithm make it worthwhile to formulate a GP theory along the lines of GA theory for fixed length binary strings.

Some recent experimental [82. 28] and theoretical [32. 92. 38. 120. 2. 3] research has questioned the value of the Schema Theorem and BBH as a description of how the GA searches or as the source of the GA's power. In this chapter we confirm that the GPST and a GP BBH similarly fail to provide an adequate account of GP's search behaviour and that various plausible interpretations of the GPST fail to support a GP BBH. Some reasons for the inadequacy of the interpretations are the same as for GAs which use fixed length binary strings. others pertain more directly to GP. and are due to its representation and crossover operator. We hope. however. that is investigation of how interpretations of the GPST fall short of supporting a GP BBH will provide insights for subsequent improved accounts of GP's search behaviour.

The chapter is organized as follows:

Section 4.1 discusses various options for a schema definition. and a more general definition than the one given in [69] is chosen. We also give formal definitions of GP-schemas and of schema order and defining length.

Section 4.2 presents the GPST as a recurrence relation that expresses the lower bound on expected instances of GP-schemas from one generation to the next.

In Section 4.3 we discuss the approximations and questionable assumptions involved in interpreting the GPST to hypothesize that a building block process characterizes GP search.

Section 4.4 concludes the chapter.

4.1. Schema Definition and Related Concepts

The first question to be considered is: what schema definition in GP is useful in formulating a description of GP search? Schemas. or similarity templates. are simply one way of defining subsets of the search space. There are obviously many ways in which the GP search space could be partitioned (e.g. according to function. fitness. number of nodes in tree. height of tree) but it is logical to stay close to the spirit of

the GA schema definition because it permits a description of the crossover operator's behaviour to be incorporated into the recurrence relation that counts the schema instances each generation. For example, if we were to instead choose to define subsets of the space according to fitness, we would not be able to explicitly formulate how many instances within a partition of a given fitness would propagate to the next generation since it is not known how crossover affects the samples of this partition.

The first schema definition we will consider is from [69, p. 117 118]. According to Koza,

> a *schema* in GP 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 (i.e., a set of rooted, point-labeled trees with ordered branches) sharing common features.

The distinctive aspect of Koza's schema definition is that a schema is a number of S-expressions each of which is isomorphic to a tree. See Tree A of Figure 48. It can be parsed inorder to form the S- Expression (IF (< 3 4) (+ 1 2) (dec x)) where IF, <, +, and dec are names of S-expressions with 3, 2, 2 and 1 argument, respectively.

Koza's definition implies that no schema is defined by an incompletely specified S-expression such as (+ # 2) where "#" is a wildcard denoting the substitution of any S-expression. There are wildcards implicit in the Koza definition but they are restricted to S-expressions which enclose the schema rather than lie within it. Thus, the schema can also be written correctly as (# (IF (< 3 4) (+ 1 2) (dec x))), with the interpretation that the wildcard can be matched by any S-expression which has at least one argument matching the schema. In other words, the schema can be embedded as a subtree anywhere in a larger tree. In consideration of the variable length representation in GP a wildcard can also be null (i.e. represent nothing and the parentheses which match it are eliminated). In this case the schema defines a partition of one instance.

There are less restrictive schema definitions which are worthy of consideration. First, while this schema definition seems intuitive because subtrees or syntactically complete S-expressions are swapped by GP crossover, it ignores the possibility of an incompletely specified S-expression such as (+ # b) or (+ (# 3 4) b). This sort of S-expression has a specific name (or root in the corresponding tree, + in this example)

but some parts within it are not specified because of the presence of internal wildcards. One obvious example of such an incompletely specified S-expression is the part of the parent tree "left behind" to be joined with a subtree taken from the other parent. We call the hierarchical structure (which is not strictly a tree) corresponding to what is left intact by repeated crossovers a **tree fragment** or simply a **fragment**. An example is Fragment B of Figure 48 which corresponds to a schema (# (IF (< # #) (+ 1 #) (dec #))).

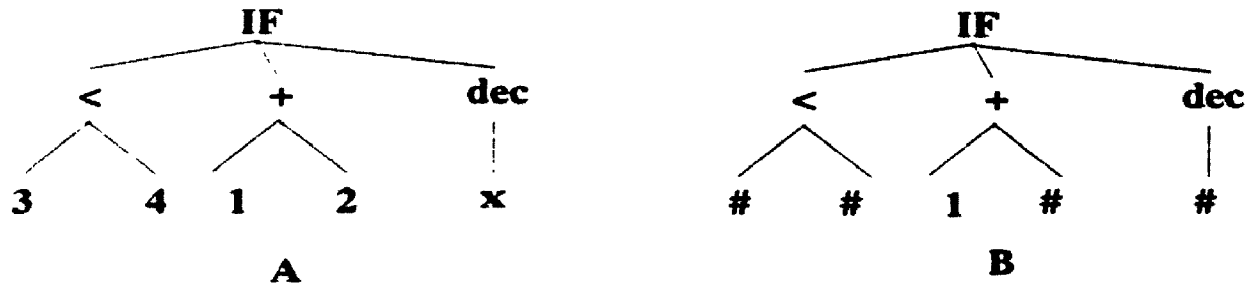


Figure 48. GP-schemas: Tree *(A)* versus Fragment *(B)*. A is a tree because it can be parsed inorder to form a syntactically complete S-expression. All of A's leaves are variables, constants or primitives that do not require arguments. B is a fragment, not a tree, because it has wildcards as leaves.

A fragment is essentially a tree that has at least one leaf that is a wildcard. It corresponds to an incomplete S-expression with wildcards inside it. There is always a wildcard at the root of a fragment to denote that it can be fully embedded in a tree.

It should be noted that the root wildcard (implicit in Koza's schema definition) can be matched more freely than a fragment's leaf wildcard. Although both kinds of wildcard eventually match with a primitive, a primitive can match a leaf wildcard only if it is in a specified position of an argument list. A primitive can match the root wildcard each time one of its arguments matches the specified part of the schema. This is because the schema definition does not state what position the specified part has in the root primitive's argument list. For example, both (− (+ 3 4) 5) and (− 5 (+ 3 4)) are instances of the schema (# (+ 3 4)) because the definition does not state which argument (+ 3 4) must match. The schema definition is not restricted so as to require a specific argument position, such as the first, or the n-th, to match because, in order to designate the match, wildcards of different arity would have to be introduced. This, in turn, would defeat the generality a wildcard is supposed to provide. Instead

this ambiguity is accepted as a natural consequence of a representation which does not use fixed positioning.

Considering fragments, an even more general schema definition is possible. A schema may be defined as an unordered collection of both completely defined S-expressions (as in Koza's definition) and incompletely defined S-expressions (i.e., fragments). The schema definition does not specify exactly how the fragments or S-expressions are linked within an instance but it requires that they must all be matched.

For example, consider the unordered 3 element collection of (+ 3 4), (+ 3 4) and (− # #) and three individuals:

1. (IF (+ 3 4) (+ 3 4) (− x 2)),
2. (IF (− x 2) (+ 3 4) (+ 3 4)) and
3. (AND (+ 3 4) (+ 3 4) (+ 3 4) (− x y))

Individuals 1 and 2 instantiate the schema once and individual 3 actually instantiates it three times because there are three combinations of two (+ 3 4) subtrees and one (− # #) fragment. Figure 49 shows a different example in terms of trees.

This general schema can be written more simply as a set (i.e. an unordered collection without duplicates) of pairs by pairing each fragment or completely defined S-expression with the number of its occurrences that must be matched by an instance, and the root wildcard is implicitly assumed. In the above example, the schema would be represented as {((+ 3 4).2), (− # #).1)}.

The more general schema definition is relevant because it allows the description of partial solutions in GP, i.e., of combinations that persist for more than one generation as crossover dissects and substitutes the parts of a tree corresponding to wildcards. Another reason is this: Consider that GP estimates the fitness of a schema by sampling the fitness of its instantiations (i.e. by sampling the fitness of a program each time the schema is found in it). The accuracy of the estimate depends on how many different samples GP processes. Other considerations being equal, the accuracy of the estimate is not related to whether the schema is a complete S-expression tree or multiple fragments and trees.

We therefore choose to define GP-schemas as follows:

> A **GP-schema** H is a set of pairs. Each pair is a unique S-expression tree or fragment (i.e., incomplete S-expression tree with some leaves as wildcards) and a corresponding integer that specifies how many instances of the S-expression tree or fragment comprise H.

An individual program in the population **instantiates** a GP-schema once **for each way** it matches the number of occurrences of trees and fragments in the GP-schema. For example, in Program ii of Figure 49 schema $H = \{((+\ 3\ 4).\ 2)\}$ is matched by six different combinations of two subtrees and, thus, Program ii instantiates H six times. While argument position is not considered when placing a fragment into the parse tree, the order of the arguments found inside a fragment must match the order of the arguments in the target subtree. Thus the program (+ 4 3) does not match with the schema $\{((+\ 3\ 4).\ 1)\}$.

We stress that a program instantiates a GP-schema once **for each way** it matches the GP-schema because we are ultimately interested in counting the expected occurences of a program pattern. Our GP-Schema definition captures the notion of a program pattern and, due to GP's representation, a program pattern (i.e., GP-schema) can occur more than once in a program. Consider a program h. An **instantiation** of a schema H by a program h, $inst(h, H)$, is an element of $Inst(h, H)$, which, in turn, is a function that produces a set of all matches of H by the target program h, based on the matching procedure described above. Figure 49 shows 6 instantiations of $H = \{((+\ 3\ 4).\ 2)\}$ by Program ii. Though an instantiation designates one specific match of the GP- schema, the notation $inst(h, H)$ does not specify the designation. The designation is, however, important and should be implicitly noted because it is later required in counting the edges connecting the GP-schema instantiation to determine its *defining length*.

Once a GP-schema definition has been adopted, the next task - in analogy to GA theory for fixed length binary strings - is to define measures of GP-schema specificity or order and of GP-schema defining length. These two concepts are used together - again in analogy to GA theory for fixed length binary strings - to determine the likelihood that a GP-schema will be disrupted by crossover. The notion of order makes it possible to compare the relative sample sizes of schemas, and can be transferred directly from GA theory for fixed length binary strings. The notion of defining length, however, can not be directly lifted from the GA domain because of the variable

structure of trees and fragments.

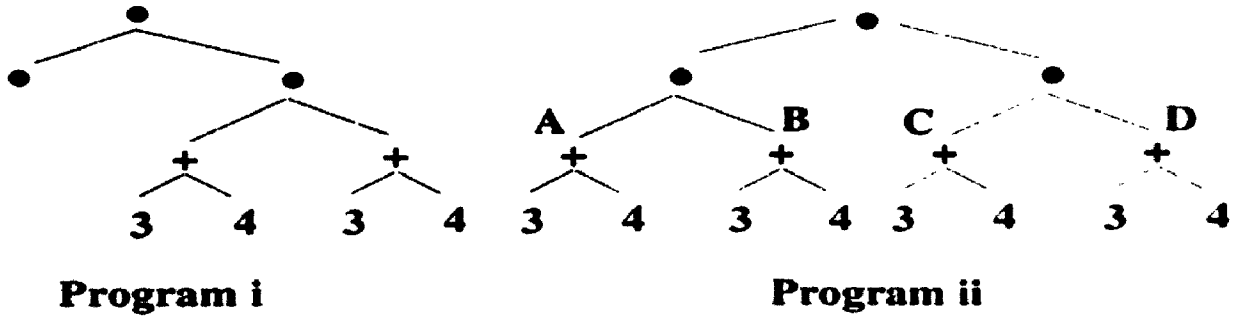


Figure 49. The GP-schema $H = \{((+34).2)\}$ has one instance in Program i and six instances in Program ii (AB. AC. AD. BC. BD. CD). The order of H is 6 and $D_{fixed}(H) = 4$. In Program i $D_{var}(h) = 2$. The instances in Program ii (see previous list) have $D_{var}(h) = 2, 1, 1, 1, 1, 2$ respectively.

The **order** of a GP-schema is the number of nodes in the graphs corresponding to its S-expressions and fragments. For example, in Figure 49 the schema $H = \{((+ 3 4). 2)\}$ has order 6, the schema $\{((\text{IF } \# \# \#). 1)\}$ has order 1, and the schema $\{((\text{IF a } \# \#). 1)\}$ has order 2.

Order is a straight-forward concept for GP-schemas. Schemas of higher order or greater specificity, other considerations being equal, will have fewer instances in a population than those of lower order or lesser specificity.

The **defining length D** of a GP-schema instantiation is the sum of its variable and fixed defining lengths:

$$D(inst(h.H).H) = D_{fixed}(H) + D_{var}(inst(h.H).H)$$

Below, we let $D(h.H)$ be short for $D(inst(h.H).H)$, and $D_{var}(h.H)$ be short for $D_{var}(inst(h.H).H)$.

The **fixed defining length** of a GP-schema H, $D_{fixed}(H)$, is the number of edges within each S-expression or fragment of H, not including edges connected to a wildcard. It is derivable from the GP-schema alone, independently of any program. For example, in schema $H = \{((+ 3 4). 2)\}$ of Figure 49, $D_{fixed}(H)$ equals 4.

The **variable defining length** of a GP-schema instantiation, $D_{var}(h.H)$, is the number of edges which connect together the S-expressions or fragments in H, i.e., the

sum of the lengths of the shortest paths between a schema fragment and the deepest common ancestor of all the schema fragments in the instantiation. $D_{var}(h.H)$ must be calculated for each instantiation and depends upon how the schema instance is embedded in the program. For example, in Program ii in Figure 49, instantiations AB and CD have a variable defining length of 2 and the others have a variable defining length of 4.

Disruption of a GP-schema instantiation $inst(h.H)$ occurs when a node in h is selected as a crossover point and the swapping of the subtree rooted at the crossover point with a subtree from another program changes h sufficiently so that it no longer instantiates H.

Let the sample space of disruption of a GP-schema instantiation be the number of nodes in program h, $Size(h)$, and let $P_d(h.H)$ be short for $P_d(inst(h.H).H)$. The upper bound on the probability of disruption under crossover of a GP-schema instantiation is its defining length divided by the number of nodes in h:

$$P_d(h.H) = \frac{D_{fixed}(H) + D_{var}(h.H)}{Size(h)}$$

Proof: The defining length of a GP-schema instantiation equals the number of crossover events that can destroy it. $Size(h)$ is the number of available crossover locations. In the worst case, no subtree swapped in will re-create the instantiation. □

For example, when h = Program i in Figure 49 and $H = \{((+\ 3\ 4).2)\}$, $P_d(h.H) = \frac{2}{3}$ since $D(h.H) = 6$ ($D_{var}(h.H) = 2$, $D_{fixed}(H) = 4$) and the total program consists of 9 nodes.

Most reported experiments of GP are run with a probabilistic bias in the crossover point selection. Leaf crossover points are probabilistically selected with a leaf bias $L_b = 0.1$ and interior points are probabilistically selected with bias $1 - L_b = 0.9$. Let the number of leaf nodes in a schema H be $V(H)$. Since the changing of a leaf node by crossover is accounted for by $D_{fixed}(H)$ and not $D_{var}(h.H)$, the probability of disruption can now be biased accordingly to yield the more precise formulation of $P_d(h.H)$:

$$P_d(h.H) = \frac{L_b V(H) + (1 - L_b)(D(h.H) - V(H))}{Size(h)}$$

Proof: The probability that a leaf in schema H will be chosen as a crossover point is $L_b V(H)$. The probability that an interior point in schema H will be chosen is $(1 - L_b)(D(h, H) - V(H))$. The space of all crossover events is $Size(h)$. □

For example, using again Program i in Figure 49 as h, $P_d(h, H) = 0.24$ since $L_b V(H) = 0.4$ ($L_b = 0.1$ and $V(H) = 4$) and the program consists of 9 nodes.

We define the **compactness** of a GP-schema instantiation as the converse of its probability of disruption: $c(h)$ equals $1 - P_d(h, H)$. Thus, when the probability of schema disruption of an instantiation is high, its compactness is low, and, when the probability of disruption of an instantiation is low, its compactness is high. Since $P_d(h, H)$ is a upper bound on the probability of disruption, $c(h, H)$ is the lower bound on the compactness.

In summary, in this section we have motivated and introduced a general GP-schema definition and defined concepts of specificity, disruption and compactness relating to this definition.

4.2. A GP Schema Theorem

This section formulates a GP Schema Theorem (GPST) that expresses the lower bound of the growth of the expected number of instances of a GP-Schema.

Recall that the GA Schema Theorem for fixed length binary strings expresses the lower bound on growth in expected membership of a schema in the population from time t to $t + 1$. It has three factors:

1. the expected membership of the schema at time t in a population of n strings,
2. the reproductive factor of the expected membership of a schema contributed by fitness proportional selection, and
3. the lower bound estimate of whether a schema member will survive crossover and mutation.

To formulate a similar lower bound for GP the following adjustments are required.

1. The GA Schema Theorem refers to expected **members** of a schema. Membership is appropriate in GAs that use fixed length binary strings because a string instantiates a given schema at most once. The GPST needs to count

the expected **instances** of a GP-schema because a program may instantiate a given schema more than once (see Section 4.1). Let $i(H.t)$ be the number of instances (by virtue of instantiation) of schema H at time t in the population of programs. $E[i(H.t)]$ denotes the expected number of instances of schema H.

2. The expected number of instances of a schema H that are reproduced via fitness proportional selection of programs must be calculated. Let $\hat{f}(H.t)$ denote the estimated average fitness of schema H at time t.

Lemma 1 $E[i(H,t+1)] = i(H,t)\frac{\hat{f}(H.t)}{\bar{f}(t)}$.

Proof: Let n denote the population size and $\bar{f}(t)$ the average fitness of all programs in the population. Let $f(j)$ denote the fitness of program j. Since the number of times a program h instantiates schema H can be calculated as $|Inst(h.H)|$. the number of instantiations of H in the population at generation t. $i(H.t)$. is given by $i(H.t) = \sum_k |Inst(k.H)|$. Fitness proportional selection reproduces a program k with probability:

$$\frac{f(k)}{\sum_{j=1}^{n} f(j)}$$

The expected number of times that program k will be copied into the next generation is:

$$n\frac{f(k)}{\sum_{j=1}^{n} f(j)}$$

and the number of instantiations of H that will be in the next generation because of program k is:

$$n\frac{f(k)}{\sum_{j=1}^{n} f(j)}|Inst(k.H)|$$

Thus. the expected total number. $E[i(H.t+1)]$. of instantiations of H in the next generation. is:

$$E[i(H.t+1)] = \sum_{k=1}^{n} n\frac{f(k)}{\sum_{j=1}^{n} f(j)}|Inst(k.H)|$$

Since $\bar{f}(t) = \frac{\sum_{j=1}^{n} f(j)}{n}$.

$$E[i(H.t+1)] = \frac{1}{f(t)} \sum_{k=1}^{n} f(k)|Inst(k.H)|$$

$$E[i(H.t+1)] = \frac{i(H.t)}{f(t)} \sum_{k=1}^{n} \frac{f(k)|Inst(k.H)|}{I(H.t)}$$

Since the estimated average fitness is:

$$\hat{f}(H.t) = \sum_{k=1}^{n} \frac{f(k)|Inst(k.H)|}{I(H.t)}$$

Therefore, $E[i(H,t+1)] = i(H,t)\frac{\hat{f}(H.t)}{f(t)}$ □

To see how a schema that appears several times in a program can be expected to reproduce in proportion to its fitness via fitness proportional selection on programs, it is important to realize that the average fitness of a schema H is the sum of the partial contributions of the fitnesses of programs that instantiate H. The fitness contribution of a program is proportional to the number of instantiations the program adds to the total instantiations of H in the population. If the above divisor were the number of programs that are members of the schema, rather than $i(H.t)$, fitness proportional selection would not guarantee expected reproduction relative to estimated average fitness.

3. Because the standard GP [69] does not use mutation, we do not account for it. We define an estimate of the probability of disruption of any instantiation of H because $P_d(h.H)$ is not the same for every GP-instantiation involving H. The upper bound probability of disruption of any instantiation of H is defined as follows:

$$P_d(H.t) = \sup P_d(h.H) \text{ at time t.}$$

$P_d(H.t)$ is a very conservative upper bound. The GPST would be more precisely bounded if the probability of disruption of schema instances were more accurately represented.

The modifications result in the following GP Schema Theorem, where $E[i(H.t)]$ denotes the expected number of instances, not members, of H at t. $\hat{f}(H.t)$ is the

observed average fitness of the instances of the GP-schema H. $\bar{f}(t)$ is the average fitness of the population. $P_d(H.t)$ is the upper bound on the probability of schema disruption. and P_{xo} is the probability of using crossover:

GP SCHEMA THEOREM: $E[i(H,t+1)] \geq i(H,t)\frac{\hat{f}(H)}{\bar{f}(t)}(1-P_{xo}P_d(H,t))$

Proof: By Lemma 1. without crossover. the expected reproduction of schema H is $i(H,t)\frac{\hat{f}(H.t)}{\bar{f}(t)}$. With crossover. the lower bound probability that schema H survives intact is $(1-P_{xo}P_d(H.t))$ because $P_d(H.t)$ is an upper bound on the probability that a schema H will be disrupted. □

On first glance. the above recurrence is the same as the GA Schema Theorem but an interesting difference is due to the "**crossover survival term**" $(1-P_{xo}P_d(H.t))$. This is an **estimate** of the minimal number of schema instances that survive crossover. As in GAs that use fixed length binary strings. the **actual** minimum likelihood of survival may be greater than the estimate and may change each generation. Whether this is indeed the case depends upon the composition of the population from which mates are drawn. Given an appropriate subtree from its mate. a schema disrupted by removing a subtree at the crossover point can be "repaired" and reinstantiated. The estimate of crossover survival in both accounts is inaccurate because it does not account for such an event.

However. in GP. the inaccuracy of the crossover survival term is further exacerbated by the fact that both the size and shape of a program containing a schema instance can change in a generation even when the schema instance is not disrupted. As a consequence of the variable length representation and the behaviour of the crossover operator. the defining length of a schema and the size of the program in which it is embedded both can change. These changes will also impact the observed probability of disruption.

Not only is crossover probabilistic. but fitness proportional selection is not explicitly correlated to program size and height[1]. These facts. together with the facts stated in the previous paragraph. imply that the probability of disruption of a schema

[1] Fitness proportional selection is probably implicitly correlated to program size and height but the relationship is not known and it may differ for each GP problem.

- while an upper bound can be formulated - changes so drastically from generation to generation, and in such an unpredictable fashion, that it can best be represented as a random variable. Since the term $P_d(H,t)$ in the GPST can be considered a random variable, we refer to it as $\tilde{D}_t$.

$\tilde{D}_t$ represents the observed probability of disruption of schema H in the population at time t. We use $\tilde{D}_t$ to formulate definitions of disruption and compactness. This allows us to interpret the GPST with respect to the allocation of trials among schemas.

Schema Disruption: Let R be the event that, at time t, $\tilde{D}_t$ is less than J, a constant. The disruption likelihood of a schema H is defined as the probability of event R, P_R. For $P_R < \alpha$, a constant, a schema is disruption prone.

Schema Compactness: Compactness is defined as $1-P_R$. If, at time t, $P_R > \alpha$, a constant, a schema is compact. Intuitively a schema is compact if its maximum probability of disruption is low regardless of the size and structure of the programs which contain it.

4.3. Building Block Definition and Building Block Hypothesis

In this section we propose and critically examine a definition of GP building blocks and a GP Building Block Hypothesis (BBH). Both the definition and the hypothesis result from an interpretation of the GPST and are intended to be fully analogous to the definition of building blocks for GAs that use fixed length binary strings and to the BBH for GAs that use fixed length binary strings. However, as will be pointed out below, our seemingly straightforward interpretation of the GPST rests on several questionable assumptions. Without these assumptions, no GP BBH can be formulated in analogy with the GA BBH.

GP building blocks: GP building blocks are low order, consistently compact GP schemas with consistently above average observed performance that are expected to be sampled at increasing or exponential rates in future generations.

GP Building Block Hypothesis (BBH): The GP BBH states that GP combines building blocks, the low order, compact highly fit partial solutions of past samplings, to compose individuals which, over generations, improve in fitness.

Thus, the source of GP's power, (i.e., when it works), lies in the fact that selection and crossover guide GP towards the evolution of improved solutions by discovering, promoting and combining building blocks.

Let us now review the assumptions presupposed by the GP BBH.

1. The GP BBH refers to the combining of schemas yet the GPST, by referring to the expected instances of only one schema, fails to describe the interactions of schemas. In this respect, the GP BBH is not supported by any interpretation of the GPST.

   Previous GA work in the context of fixed length binary strings [39, 77, 122] has made this point in much more detail. Many complicated interactions between competing schemas and hyperplanes take place in the course of a GA run. None of this activity can be described by a Schema Theorem because the latter simply considers one schema in isolation. Since the GPST does not differ from the GA Schema Theorem in this respect, the above argument applies with equal strength to GP.

   Vose has pointed out that, without knowing the composition of the population in a GA, it is impossible to precisely state how schemas combine and how many schemas can be expected [120]. Again, this point applies equally to GP.

2. The GPST also fails to lend support to the GP BBH because hyperplane competition in GP is not well defined. In GAs, trial allocation competition takes place among hyperplanes which have common features but where each "competitor" differs in the expression of that feature. The lack of a feature-expression orientation in the GP representation (i.e., GP's non-homologous nature) results in an unclear notion of which hyperplanes compete for trial allocation. This inherent lack of clarity concerning hyperplane competition seems to indicate that schema processing may not be the best abstraction with which to analyze GP behaviour.

3. Grefenstette [38] has called the classic GA BBH a "Static Building Block Hypothesis". This he states as

> Given any short, low order hyperplane partition, a GA is expected to converge to the hyperplane with the best static fitness (the "expected winner"). [38, p. 78]

Static fitness is defined as the average of every schema instance in the entire search space to distinguish it from the observed fitnesses the GA uses as an estimate of static schema fitness. He argues that "the dynamic behavior of a GA cannot in general be predicted on the basis of static analysis of hyperplanes" [38, p. 76]. Two of the reasons that the true dynamics of a GA is not estimated by the static fitness of schemas are "collateral convergence" and high fitness variance within schemas. The first reason is that, once the population begins to converge even a little, it becomes impossible to estimate static fitness using the information present in the current population. The second reason is that, in populations of realistic size, high fitness variance within schemas, even in the initial generation, can cause the estimate and static fitness to become uncorrelated.

This argument applies to GP. Furthermore, the issue of high fitness variance within a schema may be especially important in GP. As far as we know, the amount of fitness variance for GP schemas has not been empirically sampled. To discuss the issue, one must consider that schemas in GP are "pieces of program". A schema instance acquires the fitness of the program which embeds it. If the primitives are functionally relatively insensitive to context, there may be schemas in the search space that are also relatively insensitive to program embedding and thus have low fitness variance among their instances. GP is also known to evolve large programs full of functionally inert material. This material may act to shield partial solutions from interference with each other and prevent their fitness from changing when surrounding code is sampled [2]. In contrast, it intuitively seems that rearranging code or simply inserting a new statement into a program can lead to drastic changes in its fitness. This argues that the fitness variance of a schema's instances may be high.

---

[2] Admittedly this is simplistic; in programs it is difficult to ever clearly state that pieces of code do not interact.

4. The assumption of expected increasing or exponential trials for building blocks requires certain behaviour to be constant over more than one time step. The GPST does not describe behaviour for more than one time step and it is not the case that the required behaviour is constant.

   The inaccuracy in the assumption arises from estimating the long term behaviour of the reproduction and crossover survival terms in the GPST. In fact, the GPST describes behaviour for only one step and this hides important dependencies in the iteration [3]. The GPST states that **in the next generation** schema H grows or decays depending upon a multiplication factor that is the product of two terms: the probability of the schema being reproduced (i.e. the schema's fitness relative to the population average) and the probability that the schema is not disrupted.

$$\textbf{MultiplicationFactor} : \frac{\hat{f}(H.t)}{\bar{f}(t)}(1 - P_{xo}P_d(H.t)) \tag{4.1}$$

   Clearly if the Multiplication Factor (4.1) is greater or equal to 1, the expected trials of a schema will increase **in the next generation.**

   Interpreting the GPST to describe the expected allocation of trials to a schema **asymptotically** or **over more than one generation** relies on interpreting the Multiplication Factor in the GPST for more than one time step. If the time dependence of the terms were ignored by assuming that the margin by which a schema's estimated average fitness is better than the population average is constant and that $P_d(H.t)$ never changes, the claim of expected **exponentially** increasing trial allocation would be justified.

   If we resist ignoring the time dependence of the two terms (because they are a reality!) to avoid misleading over-simplification, the assumption that the expected number of trials will grow exponentially is weakened by the qualification that the Multiplication Factor must be stationary:

$$\frac{\hat{f}(H.t)}{\bar{f}(t)}(1 - P_{xo}P_d(H.t)) = \frac{\hat{f}(H.t+1)}{\bar{f}(t+1)}(1 - P_{xo}P_d(H.t+1))$$

[3] See [3] for a different and crucial Schema Theorem dependency

and the crucial time dependent relationship is the logarithmic growth of the reproductive term relative to the negative logarithmic growth of the crossover survival term:

$$\Delta \log \frac{\hat{f}(H.t)}{\bar{f}(t)} = -\Delta \log (1 - P_{xo} P_d(H.t))$$

5. As a schema starts dominating, the margin by which it is fitter than the average fitness of the population decreases. The only thing that enables it to continue growing is a decrease in its probability of disruption. The problem is that there is no guarantee that $\tilde{D}_t$ decreases at a rate ensuring positive growth of expected allocation of trials over the same interval.

   We can only consider the plausibility of this decrease in the upper bound on disruption likelihood in this situation. An exact answer is not possible[1]. Consider the growth of programs in GP runs: In GP the maximum height or size of a program is set to a lower value for the initial population than the value crossover is constrained to use in creating trees in subsequent generations. This allows programs to grow larger each generation (up to the maximum). We cannot make precise statements about the size and height distribution of a schema's instances but if one assumes they are uniformly distributed within the population, program growth in each generation may indeed cause the upper bound on the probability of disruption to decrease. Whatever the circumstances of program growth, in GP any decrease in the likelihood of disruption is fortuitous or roundabout rather than explicit in the algorithm. That is, the crossover operator and selection process do not explicitly control the size and shape of programs in a correlation with fit schemas. It should be noted that the decrease in probability of disruption caused by program growth also works in favor of unfit program fragments. Because of the variable length representation of GP, the 'cushioning' of unfit programs due to program growth is more of a problem than it would be in GAs that use fixed length binary strings.

6. Building blocks may only exist for a time interval of the GP run because the

[1] Because it requires an account of population composition that is lacking in the GPST.

estimated fitness relative to the population fitness and upper bound probability of disruption of a schema vary with time.

Consider an interpretation of the GPST that is intended to explain why a particular GP run did not find an optimal solution. An explanatory hypothesis might be that consistently highly fit partial solutions are not consistently compact. Or, that consistently compact partial solutions are not highly fit. These hypotheses reveal a caveat of the BBH : a partial solution is a building block only if its sample is consistently **both** above the population average in fitness and compact (i.e., consistently has a low maximum probability of disruption). When the Multiplication Factor is not greater than one, **despite fit partial solutions or compact partial solutions**, building blocks **do not exist**.

To elaborate further, consider an interval when a schema's margin of fitness above the population stays constant. This could happen when the observed average fitness of the schema increases (due to updated sampling) at the same rate as the population fitness. In this interval, the upper bound probability of the schema's disruption becomes the crucial factor in determining whether it will be a building block. Relative to its fitness, the schema could be allocated fewer trials for one interval than a comparable schema because the tree sizes and shapes of its instances have changed. This implies that, in some sense, **partial solutions can be inert as building blocks at some generations and active at others**. It is not a conceded fact that a building block persists in the subsequent course of the run. Indeed, in the described circumstance highly fit partial solutions may never be building blocks because, despite reproduction, they could be too prone to disruption.

7. The BBH assumes that solutions can be arrived at through linear combination of highly fit partial solutions. This is a statement about the problem of program induction rather than GP. There is no basis for assuming that a solution's subcomponents are independent. The BBH is a statement about how GP works only if there is linearity in the solution.

The basic lesson is that the GPST (and any similar schema theorem) omits important dependencies from the recurrence and is, thus, bound to oversimplify GP

dynamics. In particular, the dynamics of crossover and selection that are of interest last longer than one time step. The BBH also assumes the existence of the same building blocks throughout a run and is not specific about the dynamics of building block discovery, promotion and combination in the course of a run.

In summary of Section 4.3, we presented a definition of GP building blocks and a GP Building Block Hypothesis. We then discussed crucial issues in their usefulness and credibility. The most serious issue concerns the time dependent behaviour of schema disruption and observed average fitness relative to the population fitness. We also have cautioned that there are times when the BBH will not hold because the BBH presupposes the existence of building blocks despite the fact that compactness and consistent fitness are not guaranteed.

4.4. Conclusion

We conclude that the GP BBH is not forthcoming without untenable assumptions. Our critical discussion has led us to identify what we take to be perhaps the major problem of GP: it exerts no control over program size but program size directly affects disruption probability. Furthermore, how the probability of disruption of a schema changes over time, even from one generation to the next, is unpredictable. This time dependent behaviour is almost certainly a stochastic process (i.e., it may have underlying structure but is primarily driven by randomness): while selection and crossover determine the structure of individuals for the next generation, they control program size - which affects disruption probability - in only a roundabout way. A more useful and precise GP building block definition should state something about the time dependent behaviour of the probability of disruption but this is not quantifiable without some empirical data or simulation.

4.5. Summary

In this chapter we carefully formulated a Schema Theorem for GP using a schema definition that accounts for the variable length and the non-homologous nature of GP's representation. In a manner similar to early GA research, we used interpretations of our GP Schema Theorem to obtain a GP Building Block definition and to state

a "classical" Building Block Hypothesis (BBH): that GP searches by hierarchically combining building blocks. We report that this approach is not convincing for several reasons: it is difficult to find support for the promotion and combination of building blocks solely by rigourous interpretation of a GP Schema Theorem; even if there were such support for a BBH, it is empirically questionable whether building blocks always exist because partial solutions of consistently above average fitness and resilience to disruption are not assured; also, a BBH constitutes a narrow and imprecise account of GP search behaviour.

Other related work conducted into formulating an understanding of GP based upon schema or building block processing has been proposed ([90]) and conducted ([115]).

The doubts raised in this chapter about the GP BBH strengthen our conclusion in Chapter 3, that was based upon experimental investigation. It is unlikely that GP conducts its search exploiting true hierarchical process.

Furthermore, the formal investigation of this chapter fails to support any claim that the evolution-based search process of GP plays an exclusive role in the success of program discovery. It suggests that other search algorithms may also succeed in discovering problems described in the program discovery framework. It is in this direction we shall now proceed. Rather than assuming that GP is better than other techniques on the general grounds that it accumulates good partial solutions in parallel and hierarchically combines them, we shall empirically investigate other search algorithms and compare them to GP.

CHAPTER 5

Simulated Annealing and Hill Climbing for Comparison to GP

This chapter asks how GP compares to other, existing adaptive search algorithms when they are outfitted to solve program discovery. Will an adaptive search algorithm that is not based upon processing a population with fitness proportionate selection or using crossover be as powerful as GP?

We modify Simulated Annealing (SA) and Stochastic Iterated Hill Climbing (SIHC) so that they can be used to solve program discovery problems. In Sections 5.1 and 5.2 we describe the adjusted versions of these algorithms. The adjustment is the introduction of a novel mutation operator. We have named the operator "HVL-Mutate". Its design and function are described in Section 5.3. If these algorithms can successfully solve program discovery problems, they will provide insights into the nature of program discovery approaches.

We test our versions of SA and SIHC with the 5 problems (`6-Mult`, `11-Mult`, `Sort-A`, `Sort-B`, and `Block Stacking (BS)`) of the thesis suite (see Section 2.1). The bases of comparison are described in Section 5.4 followed by the results and analysis in Section 5.5.

5.1. Stochastic Iterated Hill Climbing (SIHC) for Program Discovery Problems

The pseudocode for the SIHC algorithm is shown in Figure 50. At the start SIHC generates a program called `current` at random and then applies the mutation operator HVL-Mutate to it. HVL-Mutate is completely described in Section 5.3. It is

sufficient for the description of SIHC to state that HVL-Mutate creates a variant (or "mutant") of **current** named **candidate** in the manner of inheritance and random variation which ensures that **candidate** is syntactically correct and has the possibility of differing from **current** in the number of its primitives (size) or its hierarchical structure. The acceptance criterion of SIHC is: if **candidate** is superior or equal in fitness to **current**, it replaces **current** and the search moves onwards from it. Otherwise another mutation on the original point, **current**, is tried. The maximum number of mutations to generate from the program **current** before abandoning it and choosing a new one at random is a parameter of the algorithm which we call **max-mutations**. Values of 50, 100, 250, 500, 2500, and 10000 for this parameter will be tried. The mutation counter for **current** is reset to zero each time **current** is replaced by **candidate**. The algorithm always keeps track of the fittest program. When a maximum number of programs (**candidates-processed** = **limit**) have been generated and each tested for fitness or a perfect solution has been found, the algorithm terminates. Because of the random restart behaviour controlled by **max-mutations**, this basic hill climbing algorithm is prefixed with "Stochastic Iterated".

One possible variation of the acceptance criterion of SIHC accepts **candidate** only when it is *better* in fitness than **current**, i.e., it accepts only strictly better moves. Without **max-mutations** as a control, SIHC with this variant will get "stuck" at a program whose one-mutation offspring ("neighbours") are all of equal fitness even though one of those neighbours may itself have a neighbour which is more fit. On search spaces which have many local minima of this nature, processing is wasted once such a local minimum is encountered. Therefore, we elected as an initial choice to proceed with the acceptance criterion that accepts equal or more fit candidates.

We use the following terminology for describing sequences recognized in the running of SIHC.

step: The acceptance of a mutant because it is an improvement. A step is not to be confused with an acceptance. Acceptance includes both equal and improved mutants whereas steps count moves to strictly fitter programs.

climb: A succession of steps ending with the random creation of a new **current** because either **max-mutations** is reached or ending when a perfect program is found.

evaluations: This defines the number of mutations performed in a climb. The term "evaluations" is not the same as climb because it accounts for mutations regardless of whether they resulted in accepted programs. In tables this value is abbreviated as "evals".

```
begin Stochastic-Iterated-Hill-Climbing Algorithm

INPUT PARAMETERS
limit           :integer        /* maximum candidates searched  */
max-mutation    :integer        /* maximum mutations of current */

GLOBAL VARIABLES
current, candidate,best :program
current-fitness,
candidate-fitness,
best-fitness            :integer or real
candidates-processed=0  :integer
mutation-count=0        :integer

Sub-Routine acceptance-criteria
     if candidate-fitness > current-fitness
     then
          best := candidate;
          best-fitness := candidate-fitness;
     endif
     return (candidate-fitness >= current-fitness);
end sub-routine

MAIN PROGRAM

     current := random search point;
     fitness-current := fitness(current);
     REPEAT
          candidate = HVL-Mutate(current);
          fitness-candidate = fitness(candidate);
          candidates-processed ++; mutation-count++;

          if acceptance-criteria
          then current = candidate;
               current-fitness := candidate-fitness;
               mutation-count := 0;
          endif;

        if mutation-count > max-mutation
        then candidate := random search point;
             fitness-candidate := fitness(candidate);
             candidates-processed++; mutation-count := 0;
             if candidate-fitness > current-fitness
```

```
            then best := candidate;
                 best-fitness := candidate-fitness;
            endif
            current := candidate;
            current-fitness := candidate-fitness;
       endif
     UNTIL
          fitness-candidate is perfect
                       OR
          candidates-processed == limit

     report best, best-fitness, candidates-processed;

end Stochastic Iterated Hill Climbing Algorithm
```

Figure 50. Pseudocode for Stochastic Iterated Hill Climbing Algorithm (cont'd from previous page)

5.2. Simulated Annealing (SA) for Program Discovery Problems

Simulated Annealing ([1]) is an algorithm modelling the physical process of annealing used in condensed matter physics. The goal of physical annealing is to obtain the minimal energy state of a solid placed in a heat bath. The solid (e.g. a metal) is heated to liquid so that all its particles are randomly arranged and randomly interacting. It is then gradually cooled to a solid state when the particle arrangement becomes a highly structured lattice and the energy of the arrangement is minimized. Annealing works because, as a physical substance is cooled, it naturally moves towards a minimum energy state. The rearrangement of particles is driven by temperature and random activity. An arrangement movement that decreases energy always falls into the sequence of particle arrangements leading to minimal energy configuration. An arrangement change which increases the energy of the system will fall into the sequence with a probability $p = exp(\frac{-\Delta E}{k_b T})$ where ΔE is the positive change in energy, k_b is Boltzmann's constant and T is the temperature.

In 1953 Metropolis et al [80] developed the Metropolis algorithm that allowed the annealing process to be simulated as a system reaching thermal equilibrium at each of a series of discrete, decreasing temperatures. The algorithm is based upon Monte Carlo techniques. It generates a sequence of states of the solid by, first, generating one state from the current via a slight distortion (simulating a perturbation) of the particle arrangement. Let the current state be denoted by i and the new state by j and their respective energy by E_i and E_j. If the energy difference of the two states, $(E_i - E_j)$, is greater than or equal to zero, j is accepted as the current state. Otherwise, j is accepted with probability

$$exp(\frac{E_i - E_j}{k_b T}) \tag{5.1}$$

This acceptance rule is known as the Metropolis criterion. If the temperature in the Metropolis algorithm is lowered slowly enough, the solid can reach thermal equilibrium at each temperature. This requires a large number of transitions to be generated at each temperature. Thermal equilibrium is characterized by the Boltzmann distribution: $P_T\{X = i\} = \frac{1}{Z(T)} exp(\frac{-E_i}{k_b})$ where:

- X is a stochastic variable denoting the current state of the solid.
- $Z(T)$ is the partition function defined as $Z(T) = \sum_j exp(\frac{-E_j}{k_b})$ where the summation extends over all possible states. [1]

In the SA algorithm used in Computer Science (typically for combinatorial optimization [58, 59]) the energy state is exchanged with an objective or fitness function which measures how close a candidate solution is to the goal state, the particle arrangement is a candidate solution and the problem is reformulated into one of minimization. A mutation operator is designed to randomly "tweak" the current solution to produce a candidate that is a slight distortion of it. (For example, if a binary representation is used, a mutation operation simply randomly chooses one bit position and flips the bit at that position.) The algorithm has an annealing schedule which dictates how the temperature changes. The Boltzmann constant is incorporated into the value of temperature. Typically the algorithm is *a priori* supplied with an initial and final temperature, and the fraction of candidates to be sampled at each temperature. The temperature is discrete and changed according to a decreasing exponential

rate. SA uses exactly the acceptance criterion of the Metropolis algorithm (Equation 5.1): the probability with which a mutant with lower or equal fitness than its parent is accepted decreases with the temperature of the system and depending upon the difference in fitness between them according to a Boltzmann distribution.

```
begin Simulated Annealing Algorithm

INPUT PARAMETERS
T-start, T-end  :real
num-steps       :integer
limit           :integer      /* maximum candidates searched  */

GLOBAL VARIABLES
current, candidate,best     :program
current-fitness,
candidate-fitness,
best-fitness                :integer or real
candidates-processed=0      :integer
cooling-rate,
current-temp                :real

Sub-Routine acceptance-criteria /* for minimization */
     fitness-delta :=  current-fitness - candidate-fitness;
     if delta >= 0.0
     then  if current-fitness > candidate-fitness then
              best := candidate;
              best-fitness := candidate-fitness;
           endif
           return true
     else  return ( exp(fitness-delta / current-temp) >  random[0,1));
     endif
end sub-routine

MAIN PROGRAM

     current := random search point;
     fitness-current := fitness(current);
     current-temp := T-start;
     rate := pow(T-end / T-start, 1.0 / num-steps);
     REPEAT
          candidate = HVL-Mutate(current);
          fitness-candidate = fitness(candidate);
          candidates-processed ++;
          if acceptance-criteria
          then current = candidate;
               current-fitness := candidate-fitness;
          endif;
          /* adjust the temperature if necessary */
```

```
        if mod(candidates-processed, num-steps) == 0
        then current-temp := current-temp * cooling-rate;
    UNTIL
        fitness-candidate is perfect OR
        candidates-processed == limit

    report best, best-fitness, candidates-proce[illegible];

end Simulated Annealing Algorithm
```

Figure 51. Pseudocode for Simulated Annealing Algorithm

The SA algorithm is guaranteed to asymptotically converge to the set of globally optimal solutions under the condition that the stationary or equilibrium distribution of the system is attained at each temperature in the annealing schedule. This demands that the cooling be done sufficiently slowly and an infinite number of candidates are processed at each temperature. Clearly the latter condition is impractical. Even achieving a quasi-equilibrium where the actual distribution is ϵ close to the stationary distribution and ϵ is small requires at each temperature the sampling of candidates that number in the quadratic order of the solution space. This in turn leads to an exponential time execution of the algorithm. In practise, the designer resorts to practically adequate schedules where typically the trade off between sampling size and temperature decrement favors small temperature decrements and small samples. The schedule is often experimentally tuned. The reference [1] should be consulted for a complete exposition of theoretical analysis of SA.

The pseudocode for the SA algorithm we use is shown in Figure 51. It is a standard version where HVL-Mutate is used as the mutation operator. Fitnesses are normalized to lie in [0,1]. The temperature decreases according to the exponential cooling schedule set up by the algorithm. In the pseudocode the variable **cooling-rate** is calculated in this manner.

In our experiments a starting temperature of 1.5 is always decreased to a user specified final temperature. The variable **current-temp** decreases after each of a user supplied fraction of candidates are processed. Table 13 lists the final temperature (T_f) and stepsize for each problem.

Problem	T_F	Stepsize
6-Mult	0.0001	500
11-Mult	0.0001	500
Sort-A	0.0025	500
Sort-B	0.001	500
BS	0.003	500

Table 13. T_F and Stepsize for SA Experiments

5.3. A Hierarchical Variable Length Mutate Operator: HVL-Mutate

In Chapter 3 it was stated that, for program discovery, it appears crucial that programs that vary in length and structure are considered as candidate solutions. Furthermore, it is very convenient when attempting program discovery to generate candidate programs which are syntactically correct without requiring any ad-hoc repair.

On this basis it is straightforward to decide that the HVL-Mutate operator should meet the following criteria:

- In the spirit of the physical annealing process, the operator must create a new program from **current** that is a "small distortion" of it. Some of the structure and character of **current** must be retained and novelty must be introduced through randomly guided decisions.
- The operator must produce a directly evaluable program (i.e. syntactically correct) without any ad-hoc repair functions being required.
- The operator must be capable of generating a mutant that differs in number of primitives (size) and/or hierarchical structure from **current**.

To meet the criteria, like GP, HVL-Mutate conveniently exploits a hierarchical representation for programs. It draws its inspiration from the method for calculating

the distance between trees [102]. That approach defines distance as the minimum cost sequence of editing one tree, step by step, to become the other using 3 elementary operations: substitution, insertion and deletion which are designated with costs.

HVL-Mutate first randomly selects a node in a copy of **current**'s parse tree. This node is called **chosen-node**. HVL-Mutate then flips a fairly biased three-sided coin to decide upon one of three sub-operations to perform: insertion, deletion, or substitution.

- If the sub-operation is substitution, **chosen-node** is directly replaced with another primitive of the primitive set which has an equal number of parameters. This is demonstrated in Figure 52.

- If the sub-operation is deletion, the largest subtree of the tree rooted at **chosen-node** is promoted to replace it. This is demonstrated in Figure 52. When more than one subtree of the tree rooted at **chosen-node** is largest, one of the largest is chosen randomly. When a leaf is chosen for deletion, it is replaced by a different randomly chosen leaf.

- If the sub-operation is insertion, a primitive is chosen from the primitive set at random. It will replace **chosen-node** but if it requires primitives the subtree rooted at **chosen-node** will act as its first parameter. All remaining children of the newly inserted node will be randomly drawn from the subset of primitives that require no parameters (i.e. are leaves).

All these sub-operations preserve the syntactic correctness of the mutant and try to minimize the change to a program within the constraints of supporting a non-binary variable length hierarchical representation. HVL-Mutate does not guarantee that mutation will not drastically change the size and shape of a tree, but does reduce the probability of that event. All three sub-operations are random perturbations of **current**.

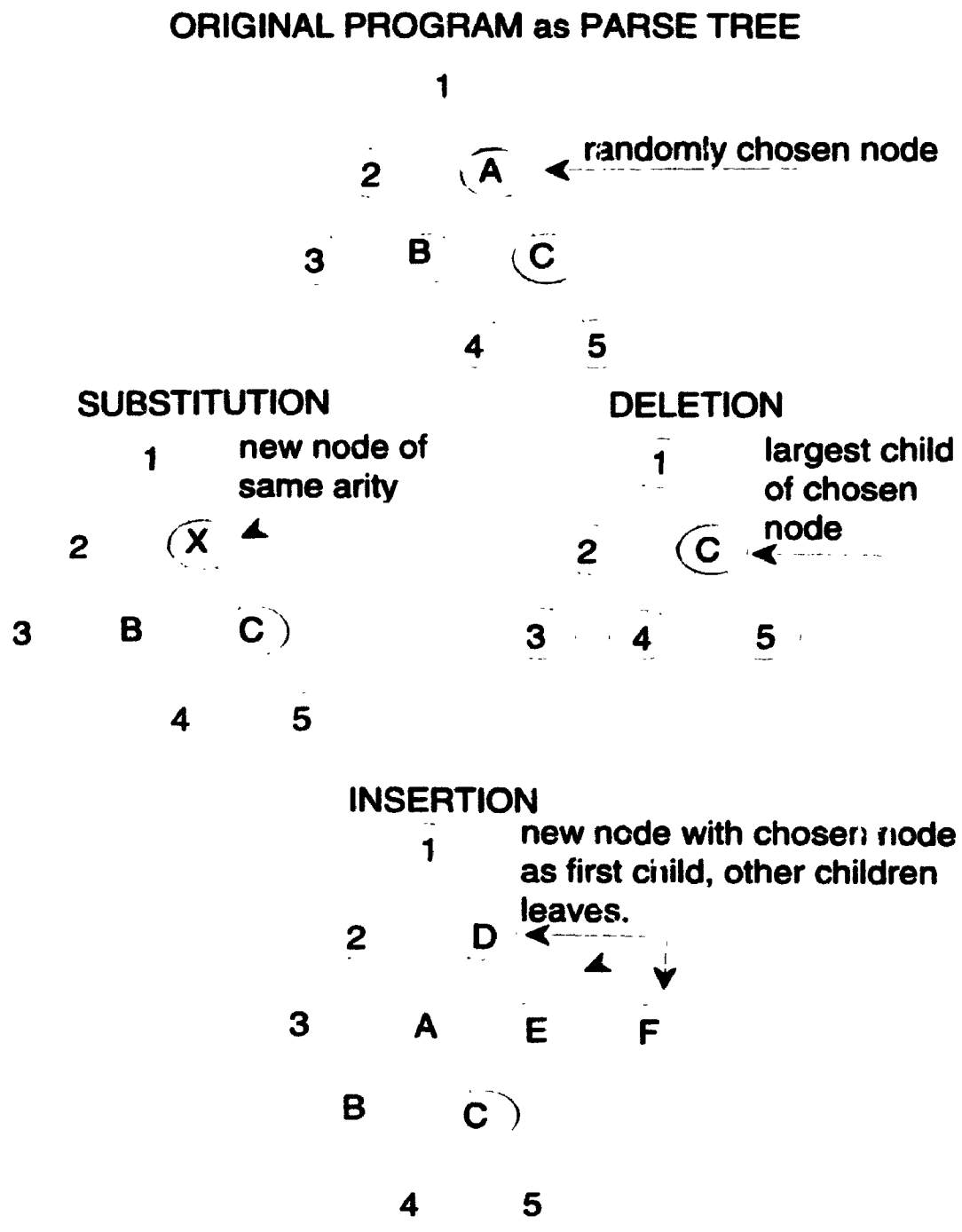


Figure 52. Demonstration of HVL-Mutate: The parse tree of the current program is used. A node is chose at random. A sub-operation (substitution, deletion or insertion) will occur at that node.

5.4. Experimental Approach

We compare the performance of SA, SIHC and GP using the five problems in the thesis problem suite. These problems are described in Section 2.1. For recall, they are:

- `6-Mult`
- `11-Mult`
- `Sort-A`
- `Sort-B`
- `Block Stacking (BS)`

The combinations of algorithms (3) and problems (5) comprise 15 experiments. An experiment (or run) consists of 30 executions (with an exception for SIHC) of the algorithm using the same parameter settings but each started with a different seed for the random number generator. In SIHC experiments approximately 10 executions are run when variations on **`max-mutations`** are tried. Then one or two settings of **`max-mutations`** are chosen and executed for a total of 30 times.

The three algorithms are compared by run when each execution of a run is allowed to process a maximum of 25500 candidate solutions or individuals.

Thus, each GP execution is permitted to process for a maximum of 50 generations beyond the initial population because the population size is 500. Processing implies a candidate solution being selected and then crossed over and evaluated for fitness, if necessary. It also includes the random generation and fitness evaluation of generation 0. This yields a maximum of 25500 individuals processed. The fitness values in the GP runs for all 5 problems were scaled by linear and exponential factors of 2. Each GP execution uses a "generation gap" of 0.9 which means that the crossover operator is applied 90% of the time with the remaining 10% of individuals chosen by selection being directly copied into the next generation. The latter individuals are not re-evaluated for fitness, therefore GP performs a maximum of 23000 fitness evaluations [1]. Because the latter individuals are copied, a definite lower bound on the number of

[1] $500 + (10\% \times (50))$

duplicate candidate solutions GP considers is 10%. GP does not check for duplication generated by crossover or selection.

In an execution of SA or SIHC processing implies an individual (or candidate) being generated by mutation and evaluated for fitness. SA and SIHC will evaluate the fitness of every candidate solution because they do not check for duplicates.

Fitness evaluation is the most computationally expensive aspect of all three algorithms. Therefore, another basis of equality would be fitness evaluations rather than how many individuals are processed. This is not unreasonable but our choice allows the generation gap to be changed as a parameter of a GP run without affecting requiring the other two algorithms to be re-evaluated with different maximums on processing. Furthermore, it accounts for the selection process in GP: while some individuals are copied directly, they first are selected. Selection is a crucial element of the algorithm because GP relies upon a population-based mechanism. As well, an unresolved issue arising when using fitness evaluations is how to account for an implementation of the algorithm that considers duplicates but, by recording fitnesses, does not have to re-evaluate them for fitness.

The comparison uses a consistent set of parameter settings for each algorithm across the problems of the problem suite. There is no "tuning" of a technique to its best potential on a particular problem. For example, in GP, for a given problem and primitive set, while it is known that population size can affect performance, only a one population size was used in this experimentation. In SA, for example, the cooling schedule could also have been adjusted to improve effectiveness but was not.

For each experiment, on a per run basis, we show 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 indicated in parentheses.

5.5. Experiment Results

5.5.1. 6-Mult Results

Table 14 summarizes the results obtained for the 6-Mult problem. Figures 53 and 54 plot typical executions for a typical SA experiment and one climb of a successful SIHC experiment respectively.

Both SA and SIHC are capable of solving this particular program discovery problem. In fact, the probability of SA succeeding was 100%. This is statistically significantly better than GP which was successful 79.5% (40.3) of the time. We had no intuition of a sensible value for **max-mutations** in the SIHC experiment so we tried 5 different runs with the parameter ranging from 10 to ten thousand. The results for all executions and just those that were successful are displayed in Tables 15 and 16. When **max-mutations** was ten thousand a solution was found in 77% (50.4) of the run's executions. This value does not differ significantly from the GP likelihood of success but it does differ from that of SA.

On average SA processed 54.2% (7.7) of the maximum individuals allowed. This is approximately 13770 (1836) out of 25500 individuals. GP processed only slightly more – 55.7% (27.8) of the maximum individuals allowed, over all executions but on successful executions processed only 48.4% (21.4). There are large standard deviations in the GP data compared to the standard deviations of the SA data. SIHC (**max-mutations** = 10000) processed the most individuals: 61.5% of maximum, over all executions and 57.7% (30.2) over just successful ones. Statistically, there was no distinction on this criterion among the three algorithms.

Because SA solved all executions of its run, it achieved a 100% fitness on average. Despite SIHC and GP solving less executions, the average fitness of the best individual was 98.8% (2.8) and 98.0% (4.5) respectively. SA is statistically significantly better than SIHC and GP in this respect. SIHC and GP do not significantly differ.

Table 28 displays the height and size of the parse tree of successful programs for all problems and experiments. Interestingly, on **6-Mult** SIHC did find successful programs which were significantly shorter (i.e. had less primitives or fewer nodes) and of less depth (i.e. the height of the program trees was less).

With SIHC the best setting for **max-mutations** was ten thousand. With it, on average, a successful climb was 9.4 steps and the number of evaluations per step in a

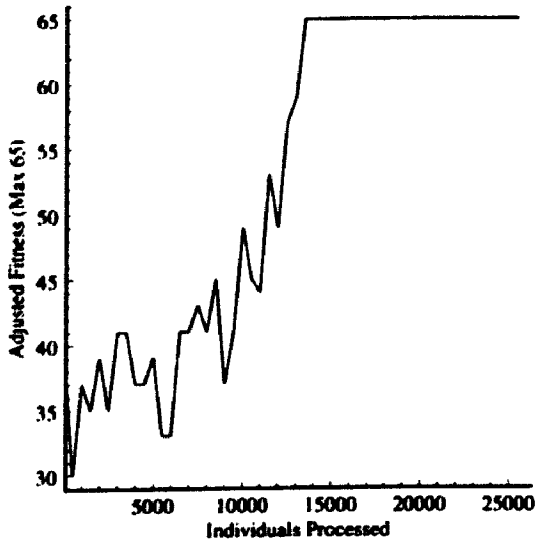


Figure 53. Plot of successful SA Run on 6-Mult, data recorded every 500 mutations.

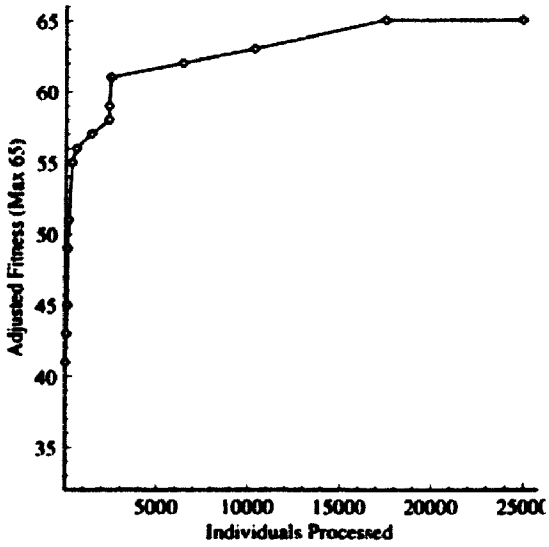


Figure 54. Plot of successful SIHC climb on 6-Mult when `max-mutations` = 5000. Data was recorded at each step, a step is denoted by a diamond.

successful climb was 7336 resulting in an average of 777.1 evaluations per step. One way of interpreting this value is that it quantifies the local nature of the landscape, i.e., with the given operator, how much effort is required to find a higher point than the present one. The fact that SIHC was successful implies that there may not be many local optima to stymie the search or that there may be many peaks of optimal height. Notice that the steps per climb and evaluations per step increase as `max-mutations` is increased but not at a linear pace. It appears that it becomes harder to find a

`6-Mult`	Rate of Success (%)	Fittest Individual (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	79.5 (40.3)	98.0 (4.5)	55.7 (27.8)	48.4
SIHC, max-mu = 500	40.0 (49.0)	96.6 (2.3)	81.8	61.3
SIHC, max-mu = 10K	76.7 (50.4)	98.8 (2.8)	61.5	57.7
SA	100.0	100.0	54.0	54.0 (7.2)

Table 14. Comparison of GP, SA and SIHC on `6-Mult`

Max Mutations	Rate of Success (%)	Best Fitness (%)	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	10	89.8 (3.1)	2.6	80	30.8	94.0
100	20	91.4 (3.6)	3.5	165	47	81.7
250	20	93.5 (3.5)	4.4	425	97	90.0
500	40	96.6 (2.3)	5.4	855	159	81.8
10000	77	98.8 (2.8)	9.3	9787	1052	61.5

Table 15. 6-Bit Multiplexer: SIHC Data

Max Mutations	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	2.6	79.8	31.1	59.0 (25.9)
100	3.1	166.4	51.2	17.8
250	4.0	394.1	99.7	58.7 (28.2)
500	5.0	833.4	165.6	61.3 (22.0)
10000	9.3	11271.4	1207.6	57.7 (30.2)

Table 16. 6-Bit Multiplexer: SIHC data for successful executions

better candidate as the number of overall steps in a climb increases. Regardless of **max-mutations**, a successful execution requires approximately the same number of evaluations.

5.5.2. 11-Mult Results

Results for the 11-Mult problem are summarized in Table 17. Only SA among the 3 algorithms was able to obtain a perfect solution. This it accomplished on 5 out of 30 runs. One solution was a program with tree height of 14 and 1891 nodes. On average SA found a solution which was 93.0% of the optimum (adjusted fitness of 1905.6 out of 2049). Figure 55 shows a typical and successful SA execution.

Neither GP or SIHC could solve 11-Mult with 25500 evaluations. The fitness of the best program found by GP was 87.6% of the optimum. When the value of **max-mutations** was 5000, the fittest program SIHC could find was 88.9% of the

optimum. The results for SIHC when the maximum mutations parameter was varied from 50 to 10000 were not significantly different. They are displayed in Table 18. Three SIHC climbs are plotted in Figure 56. It is interesting to note that a very simple, greedy heuristic that can be coded in less than one page outperforms a complicated and computationally expensive algorithm such as GP.

11-Mult	Rate of Success (%)	Best of All Execs (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Execs (%)
GP	0	87.6	79.2	100	
SIHC					
max-mu = 500	0	84.4	81.6	100.0	
max-mu = 5000	0	95.31	88.9	100.0	
SA	16.7	100.0	93.0	99.5	98.2

Table 17. Comparison of GP, SA and SIHC on 11-mult

SIHC					Best Climbs			
Max Mu	Steps per Climb	Evals: Climb	Evals: Step	Best Fitness (%)	Steps: Climb	Evals: Climb	Evals: Step	Best Fitness (%)
50	5.0	107.6	21.5	73.6	20	476	23.8	77.35
100	6.6	229.0	34.5	75.5	32	525	16.4	76.62
250	10.5	637.2	60.8	81.2	60	2273	37.9	87.95
500	15.0	1466.0	97.0	81.6	32.5	2436	74.4	84.38
2500	13.2	3939.0	296.6	74.1	57	12203	214.1	95.31
5000	5.9	2518.9	429.3	88.9 (34.2)	8.4	22093	2630.1	95.31
10000	25.6	21059.0	822.6	88.4 (39.0)	40	25000	625	96.88

Table 18. SIHC and 11-Mult

The 11-Mult problem experiments were also interesting because the degree of success of the three search techniques varied. SA is statistically superior to GP or SIHC based upon rate of success. When the expected best fitness is compared, GP and SIHC when max-mutations = 5000 do not differ statistically significantly. GP is slightly out-performed by hill climbing when the best values are compared: the best

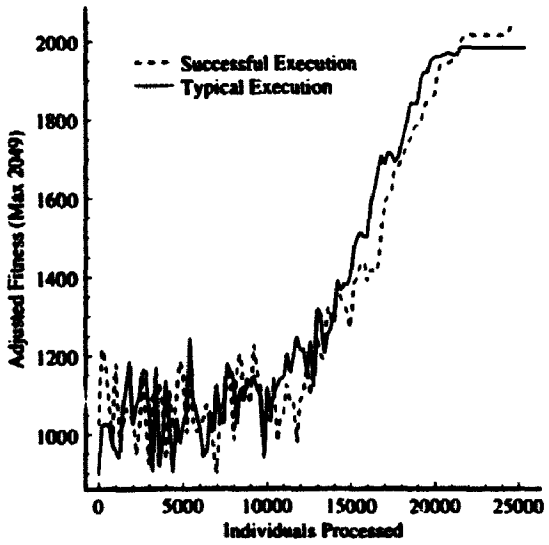


Figure 55. Plot of typical and successful SA executions on 11-Mult, data recorded every 500 mutations.

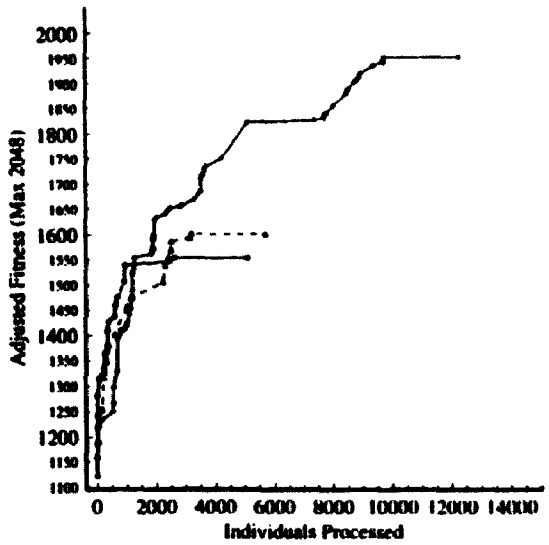


Figure 56. Plot of 3 SIHC climbs on 11-Mult. Data was recorded at each step, a step is denoted by a symbol, **max-mutations** = 2500, all climbs from same execution.

fitness ever obtained by GP was 87.6% of the optimum and with SIHC it was 96.88% of the optimum (max-mutations = 10000).

5.5.3. Sorting Results

Recall that we have two different versions of the problem called `Sort-A` and `Sort-B`. Each uses the same set of primitives and test suite but employs a different fitness function. See Chapter 2 for details.

On both `Sort-A` and `Sort-B` SA had approximately an 88% probability of success and the expected fitness at 25500 evaluations was very close to optimal (87.4% and 91.2% respectively). This is comparable to GP for `Sort-A` which solved 80% of the executions and superior to GP for `Sort-B` which solved 66.7% of the executions.

On both `Sort-A` and `Sort-B` SA processed approximately 63% of the maximum allowed individuals in all executions and about 55% of maximum allowed individuals in successful executions. This is more than GP for both `Sort-A` and `Sort-B`. `Sort-A` processed 51.3% of individuals in all executions and 39.1% in successful executions. `Sort-B` processed 60.4% of individuals in all executions and 40.6% in successful executions. SIHC processed the most individuals: 72.2% of maximum, for all executions

but fewer than both GP and SA on successful executions (37.5% vs. 39.1% vs 54.6%). This difference is not statistically significant however.

The size and height of the trees of successful programs in Sort-A with SA were less than those obtained with the GP. We could not test for statistical significance because the samples were too small.

Sort-A	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	80.0 (40.0)	49.7 (18.1)	51.3 (35.0)	39.1 (26.6)
SIHC, max-mu = 100	46.7 (49.9)	72.6 (40.7)	72.2	37.5 (27.6)
SA	83.3 (37.9)	87.4 (28.3)	62.2 (31.0)	54.6 (28.5)

Table 19. Sort-A Comparison of GP, SA and SIHC

Sort-B	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	67.7 (47.1)	85.3 (22.6)	60.4 (37.5)	40.6 (28.2)
SIHC, max-mu = 50	60.0	79.1 (31.3)	62.5	37.5 (13.4)
SA	88.3 (37.3)	91.2 (19.7)	64.0 (30.5)	56.6 (28.5)

Table 20. Sort-B Comparison of GP, SA and SIHC

Tables 21 and 22 provide the details of SIHC and `Sort-A`. Tables 23 and 24 provide the details of SIHC and `Sort-B`. We discover that it is possible to hill climb to a solution in the fitness landscape regardless of the value of the **`max-mutations`** parameter. In fact, there is no significant difference in the probability of success for different values of the max-mutations parameter among the settings we tried. From the details of successful climbs one can see that, surprisingly, most climbs take between one and two steps. From this it appears that there are many randomly spaced small equally sloped peaks. This phenomenon was confirmed when the SA executions were examined in detail. Frequently there were sequences of only one or two mutations that produced a fitter program. This data shows a contrast in search space character

when it is compared with that of **6-Mult** and **BS**. In **6-Mult** the steps per climb varies from 2.5 to 9.3 and in **BS**, regardless of max-mutations, there are approximately 4 steps per climb. The greedy nature of SIHC seems likely to be responsible for a significantly lower probability of success when compared to GP or SA.

Max Mutations	Rate of Success (%)	Best Fitness (%)	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	50	67.6 (47.7)	0.003	49.98	16406.4	64.3
100	46.7	72.6 (40.7)	0.008	100.2	12302	72.2
250	50	62.1 (55.5)	0.0135	249.7	18400.4	72.2
500	46.7	63.2 (50.0)	0.0326	499.7	15343.2	68.2

Table 21. Sort-A: SIHC Data

Max Mutations	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	1.4 (0.5)	28.6 (18.4)	20.4	78.7 (15.3)
100	1.6 (0.5)	79.4 (32.3)	50.5	37.5 (27.6)
250	1.8 (0.4)	153.6 (91.4)	85.3	44.3 (16.7)
500	2.0 (0.5)	388.3 (255.4)	194.1	31.1 (25.9)

Table 22. Sort-A: SIHC data for successful executions

Max Mutations	Rate of Success (%)	Best Fitness (%)	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	60.0	81.8 (43.8)	0.0034	49.98	14489.3	62.5
100	30.0	79.1 (31.3)	0.006	100.1	16399.8	83.6
250	50.0	79.4 (42.4)	0.018	249.5	14050.5	71.6
500	40.0	74.1 (43.7)	0.02	498.9	22837.1	80.6

Table 23. Sort-B: SIHC Data

Max Mutations	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	1.3 (0.5)	27.2 (12.3)	20.4	37.5 (13.4)
100	1.7 (0.5)	60.7 (58.6)	36.4	51.7 (34.7)
250	1.8 (0.4)	103.6 (65.8)	57.6	43.3 (26.1)
500	1.7 (0.8)	305.2 (248.7)	174.4	52.2 (20.9)

Table 24. Sort-B: SIHC data for successful executions

5.5.4. Block Stacking Results

Tables 25 and 26 show the comparative results and details of the different SIHC runs where `max-mutations` is varied for BS. GP, with a success rate of 76.7% (16.3), had a significantly lower rate of success than either SA or SIHC. SA had a 100% rate of success and SIHC had a 94.3% rate of success. The only significant difference in the number of individuals processed was between GP and SA (43.7% to 28.5%). There was no significant difference in individuals processed when only successful executions were considered.

The ranking for tree heights was statistically significant: SIHC had the shortest trees (7.7), GP followed (10.1) and SA was 13. SIHC also produced significantly shorter programs (in terms on numbers of nodes) when compared to GP and SA (25.7, 36.9 and 45.1 respectively). Thus there is a trade off between probability of success and compactness of solution. SA seems to guarantee a solution but, perhaps, at the expense of readability and efficiency of solution.

Block Stacking	Prob of Success (%)	Fittest Individual (%)	Evals Used (%)	Evals Used in Succ Run (%)
GP	76.7 (16.3)	87.7 (31.9)	43.7 (32.1)	35.1 (26.6)
SIHC, max-mu=100	94.3 (23.2)	94.9	34.5	30.5 (24.0)
SA	100.0	100.0	28.5 (22.2)	28.5 (22.2)

Table 25. Block Stacking Comparison of GP, SA, SIHC

Max Mutations	Rate of Success (%)	Best Fitness (%)	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	40.0 (24.0)	70.0 (50.7)	0.96	61.8	64.6	74.3
100	94.3 (5.4)	94.9 (26.6)	1.2	121.0	104.6	34.5
250	80.0 (4.5)	84.8 (43.0)	1.3	285.6	224.3	43.8
500	70.0 (21.0)	72.5 (54.4)	1.3	527.8	406.5	45.7
1000	66.7 22.3)	72.2 (54.5)	1.4	1028.2	749.9	60.6

Table 26. Block Stacking: SIHC Data

Max Mutations	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	4.0	57.3	14.3	36.0 (28.2)
100	3.5	94.4	26.9	30.5 (24.0)
250	4.4	173.7	39.7	29.6 (27.4)
500	4.1	181.3	43.6	22.8 (13.7)
1000	3.9	507.6	131.8	41.7 (26.5)

Table 27. Block Stacking: SIHC data for successful executions

5.5.5. Results of Other Literature

Subsequent to [89] another comparison of hill climbing and GP was conducted. Juels and Wattenberg ([52]) compared GP to a hill climbing algorithm on one problem, `11-Mult`. Their hill climbing algorithm is equivalent to ours if `max-mutations` is set to the maximum number of individuals processed.

The authors only describe a specialized mutation operator in the context of `11-Mult`. It first chooses a node of the program parse tree at random. Another primitive which is drawn with 50% probability from the set of AND, OR, NOT and IF and 50% probability from the set of address and data values is substituted for the chosen node. If the new primitive has too many parameters, an appropriate number are randomly deleted. If it has too few, random primitives drawn from the set of address and data values are added.

Obviously this operator has a generalization though the authors did not state

Problem		SA	GP	SIHC
6-Mult	Tree Height	8.6 (2.8)	6.9 (3.2)	5.0 (1.9)
	Tree Size	58.0 (49.4)	42.8 (32.0)	23.9 (0.2)
Sort-A	Tree Height	5.64 (2.0)	6.8 (2.8)	6.7 (1.7)
	Tree Size	12.9 (6.0)	25.0 (25.3)	48.3 (7.8)
Sort-B	Tree Height	5.6 (2.0)	5.7 (1.4)	7.3 (1.5)
	Tree Size	12.6 (5.6)	19.4 (15.0)	44.3 (5.9)
BS	Tree Height	13 (1.2)	10.1 (3.0)	7.7 (1.8)
	Tree Size	45.1 (16.3)	36.9 (16.3)	25.7 (1.0)

Table 28. Successful Program Size and Structure Data: SIHC, SA, GP

or test one. Qualitatively it does not differ from HVL-Mutate because it generates syntactically correct programs and allows a mutant to differ in size and structure from its parent. Quantitatively one would expect small differences between it and HVL-Mutate that will depend upon the primitive set which is used. The distribution of leafs to nodes in a primitive set influence the detailed specific behaviour of each operator.

Similar to our findings the authors reported that their algorithm could solve 11-Mult. They provided a comparison to a version of GP that used a population size of 4000. It was based upon 20000, 40000, 60000, and 80000 fitness evaluations. They reported that, while GP had a 28% likelihood of solving 11-Mult after 40000 evaluations and a 75% likelihood after 60000 evaluations, the hill climbing algorithm's probability of success was 61% and 98% respectively. More details are available from the paper ([52]).

5.6. Summary

By way of analyzing GP using comparison, we introduced program discovery approaches for two traditional single point based adaptive search algorithms: Simulated Annealing and Stochastic Iterated Hill Climbing. Despite not tuning SA, it was capable of outperforming GP in certain respects on a majority of the problems we experimented with. We observed mixed or comparable differences between operator and

search techniques across different problems. This confirms the notion that the suitability of a search technique depends upon the fitness function and primitives chosen for a particular problem since these influence the nature of the search landscape.

It seems critical that program discovery algorithms are carefully compared and well understood to avoid premature conclusions regarding superiority. Our finding that GP is not always better than other program discovery algorithms on the problem suite is not surprising. Altenberg [3] has shown the particular search bias GP exploits for power and the search landscape character it depends upon in order to succeed. While the restricted definition of program discovery that we use in this thesis originates from the GP paradigm, quite obviously nothing within it stipulates that GP will always be best.

The experimental results of our comparison clearly suggest something that is may initially strike one as counter-intuitive: that adaptive mutation and a degree of localized search are useful for program discovery. Typically one thinks that programs are so sensitive to context that tweaks are too radical. Since GP swaps subtrees which are actually sub-programs there is some semantic level exchange of encapsulated function which makes crossover seem more intuitive than mutation. Yet, HVL-Mutate seems to indicate that any intuition that this sort of exchange is necessary is incorrect: the hierarchical representation seems to allow tweaks to explore the search space in an efficient manner.

Many of the experiments in this chapter prompt further investigation. Related work that could be pursued includes:

- searching for statistical measures which would indicate the superior technique for a particular problem or the superior crossover operator to choose in GP
- finding out how these search algorithms compare on larger program discovery problems, i.e. how well do they scale? Automatic definition of functions (ADF) [70] is claimed to improve performance on big problems. ADF is a technique of representation rather than an operator so it seems likely that HVL-mutate can be modified to support it. It would be interesting to find out whether SA or SIHC with the extended HVL-Mutate are sufficiently powerful to handle the same scale of problems as GP using ADFs.

CHAPTER 6

Crossover Hill Climbing and Crossover Simulated Annealing for Comparison to GP

The crossover operator is a crucial factor in the power of GAs because of its combinative nature. It is one reason GAs may be superior to other search techniques for certain problems (or certain fitness landscapes). Recent GA research has focused upon finding out for what class of problems GAs are more effective than other algorithms (e.g. [83]). In the context of program discovery, this implies that role of GP crossover needs to be understood. The first goal of this chapter is to investigate the basic nature of the GP crossover operator by observing its performance when it replaces HVL-Mutate in Simulated Annealing and Stochastic Iterated Hill Climbing.

In Section 6.1, we describe the concept of a *fitness landscape* and introduce a measure we call "syntax correlation". We estimate that the parent-offspring syntax correlation of GP crossover is lower than that of HVL-Mutate. Consequently, among the class of fitness landscapes created by GP crossover, there are plausibly some landscapes that are amenable to search by SA or SIHC. In Section 6.2 we describe crossover based SA and SIHC algorithms. In Section 6.3 we compare the performance of these algorithms to GP using the suite of problems of the thesis.

A second goal is to improve GP. In Section 6.4, based on the reasonable success of the alternatives we compared to GP, we suggest that integrating a local search into GP will improve it by complementing its global search power. In Section 6.5 we provide a variety of hybrid schemes combining GP and stochastic iterated hill climbing. The hybrid algorithms are tested for viability and compared in Section 6.6.

They indeed result in better performance than GP.

Our final goal is to broaden our attention from individual algorithms to the collective group of algorithms this thesis has either introduced or described. In Section 6.8 we reconcile their important differences at one level, by describing their similarities at a more general, qualitative level. Succinctly, the algorithms are united by a framework of evolution-based concepts: selection, inheritance and blind variation. Each is a unique design and implementation of the collective concepts. Recognizing this alliance sets the stage for future directions in research into computation-based behaviour.

6.1. Combining GP Crossover with SA or SIHC

The notion of a fitness landscape was introduced by Sewall Wright in [125]. It is a concept in evolutionary theory that supports study of the dynamics of evolutionary optimization by providing a formal definition of the underlying structure of the search space. It is used to study the dynamics of any adaptive search ([81, 31]), even those based upon single point dynamics rather than population dynamics. It can obviously be extended to help analyze the dynamics of GAs [50]. Stadler [111] describes a fitness landscape as:

> a collection of genotypes arranged in an abstract metric space, with each genotype next to those other genotypes which can be reached by a single mutation, as well as a value assigned to each genotype [111]

A clear correspondence between fitness landscapes concerned with program discovery and those concerned with biological evolution exists:

- A candidate solution (i.e. a program) corresponds to a genotype.
- An application of a genetic operator corresponds to mutation.
- The fitness of a candidate solution in program discovery is the value assigned to each genotype.

A graph theory perspective and vocabulary is often used to discuss fitness landscapes. A fitness landscape corresponds to a graph of vertices and edges. A vertex represents a candidate solution (or more). Two vertices in the graph are connected

if an application of the search operator to the corresponding program (or programs) generates the other[1]. A search is a graph traversal along edges that occurs when a search operator is applied and the search moves from one candidate solution to another.

A fitness landscape has an associated notion called *neighbourhood*. The neighbourhood of two mates involved in GP crossover is the set of all offspring that can possibly be generated from crossing over the two mates and that are different from them. The neighbourhood of a program involved in HVL-Mutate is the set of all mutants that can possibly be generated by applying HVL-Mutate to it and that are different from it. The term *neighbour* is synonymous with offspring, child or mutant.

Concerning neighbourhoods, it is interesting to contrast the size of those of GP crossover to those of single-point crossover combined with a fixed length bit string. In GP crossover, the maximum size of the neighbourhood is the product of the number of crossover points in each parent. The actual size depends upon the sizes of the specific parents involved in the crossover and upon the redundancy of offspring (duplicate offspring are not counted in the size of a neighbourhood).

In GAs that use a fixed length representation, the maximum size of the single-point crossover neighbourhood is $2l$ where l is the length of the representation. The actual size of the neighbourhood is $2l$ less the number of duplicates. The fixed position representation implies that, if each mate has the same value at a given position, all offspring will only have that value at that position. Let us term this incident "allele redundancy". In a binary representation, the probability of allele redundancy at a bit position in two independent strings is 50% and this considerably constrains the neighbourhood size. With an alphabet of higher cardinality the probability of allele redundancy is less but nonetheless when the two mates are equal, the number of duplicates equals $2l$ and the neighbourhood size equals zero.

In GP crossover, there are two reasons to expect less redundancy among the neighbours of two mates. First, GP has a non-binary alphabet which reduces the probability of the recipient and donor containing an identical subtree. Second, because there is no fixed positioning in the representation, any primitive(s) in the donor

[1]We assume the parent-mutant or parents-children relationship is symmetric. That is, the operator is capable of generating vertex A from B as well as vertex B from A

can be placed anywhere in the recipient and thus provide another offspring. For example, consider two duplicate 2 node S-expressions with a distinct root and child. This is akin to the case of a binary alphabet and identical mates. In GP there are 4 possible crossovers and while two of these produce duplicates, the remaining 2 produce original trees. As another example, consider two duplicate 3 node S-expressions where the root has 2 children and each node is a distinct primitive. There are nine offspring in the crossover of the two mates but only three of the nine produce a duplicate.

Succinctly the contrast is: with GP crossover, a crossover neighbourhood is likely to be larger than that resulting from a single-point crossover applied to binary strings because its maximum size is the product of both mates' sizes rather than $2l$ and because GP crossover is likely to generate fewer duplicates.

We can compare GP crossover to HVL-Mutate in a different respect. When considered independent of a particular program discovery problem, GP crossover landscapes and HVL-Mutate landscapes generally differ in *syntax correlation*. Syntax correlation is a measure of similarity in structure, size and primitives between one parent and one offspring. The greater the similarity between a parent and offspring in these respects, the greater the syntax correlation. The syntax correlation of a landscape is the estimated average of all individual syntax correlations.

We have not ever precisely calculated this measure on landscapes of GP crossover or HVL-Mutate but propose a way of doing so: use a tree distance metric [102] where 1) the cost difference between primitives of equal number of parameters is one less than the cost difference between primitives of unequal numbers of parameters and 2) the basic cost of insertion, deletion and substitution is equal. Without precisely calculating syntactic correlation, is nonetheless possible to make an approximate relative comparison by comparing each sub-operation of HVL-Mutate to one application of GP crossover (the generation of one offspring from two parents). For reference the sub-operations of HVL-Mutate are described on page 52 and the GP crossover operator is shown in Figure 5 and described in Section 1.2.2:

- **Substitution** HVL-Mutate simply exchanges `chosen-node` for another primitive which has the same number of parameters as it.

GP crossover can only substitute a leaf node. For one leaf to be replaced with another requires a leaf node to be chosen as the crossover point in both parents. The likelihood of leaf substitution in GP crossover is low among the likelihood of other changes.

This implies that, when one compares the one third of all independent instances where substitution is HVL-Mutate's chosen sub-operation to an equivalent number of independent GP crossover operations, on average, GP crossover results in an offspring that is more different than its parent compared to HVL-Mutate.

- **Deletion** If **chosen-node** is a leaf, it is replaced by another leaf. This is essentially a substitution. Otherwise, HVL-Mutates removes the subtree rooted at **chosen-node** and replaces it with that subtree's largest child. Two syntactic differences result:

  1. The child has fewer nodes than its parent.
  2. All the nodes of the child are a subset of its parent.

  To determine the closest equivalent to deletion in GP crossover, consider a pair of parents and the generation of one offspring. One parent can be designated the "donor": it gives a subtree to the other, and the other can be designated the "recipient": it has a subtree removed and replaced with one of "donor's". A deletion occurs on the donor.

  Now consider syntax correlation between either donor or recipient and the offspring. The size and structure of the donated material is completely random **with respect to** the recipient. It could be smaller or larger than the removed subtree. The odds that it is different in primitive composition is very high. Furthermore, all the nodes of a child do not come solely from one of its parents. This leads one to conjecture that, when one compares the one third of all instances where deletion is HVL-Mutate's chosen sub-operation to an equivalent number of GP crossover operations, on average, GP crossover results in more material being different between the parent and offspring.

- **Insertion** If the new primitive is a leaf, **chosen-node** is replaced by it. This is equivalent to substitution. Otherwise, a new primitive replaces **chosen-node**

and, if the new primitive requires parameters, the entire subtree rooted at **chosen-node** is used as one child and primitives without parameters are used as the remaining others.

The closest interpretation of an insertion with GP crossover is the same as that given for deletion but from the perspective of the recipient. The relative syntax correlation is the same also: an HVL-Mutate insertion preserves more material when generating an offspring from its parent.

This estimated, qualitative comparison allows us to argue that

- GP crossover usually generates an offspring which is more different in size, structure and primitives to its parents than the offspring generated by HVL-Mutate
- fitness landscapes created by HVL-Mutate have a higher syntax correlation than those created by GP crossover

In fact, GP crossover is a *macro mutation* operator that performs larger *tweaks* of a candidate solution than HVL-Mutate. Therefore, GP crossover may improve the performance of SIHC or SA in circumstances where the GP crossover landscape has fewer local optima or where the algorithm is able to exploit the existence of a shorter path to the global optima.

It also seems unlikely that macro-mutation will always work well because it may not be sufficiently "fine grained" in some circumstances. That is, when it is appropriate to "fine tune" a solution that is almost optimal by modifying it only slightly, GP crossover is unable or has very low likelihood of generating a small change.

We now proceed to test our conjectures.

6.2. Crossover Based, Single Point Algorithms

GP crossover based versions of SA and SIHC are modifications accommodating the fact that GP crossover requires two parents. The search traverses two points at a time and an adjustment is made to the acceptance criteria.

6.2.1. Crossover Hill Climbing: XO-SIHC

Crossover hill climbing was first described by Terry Jones [50, 49]. The basic design of crossover hill climbing still uses better or equal fitness as the move acceptance criterion of the search but substitutes GP crossover as the move-operator. The candidate solution is, therefore, generated from one "current" solution and a random mate using a version of GP crossover that is a simple two-parent to one-child function.

The algorithm always maintains the fittest-overall-solution (i.e. the fittest point of all points examined) and a current-solution. At the outset, a mate for the current-solution is randomly generated. For some number of attempts, `mate-xo-tries-limit`, offspring are generated via crossover from the pair of current solution and mate. If an attempt yields an offspring that is accepted, the offspring replaces the current solution and the process repeats with the number of crossover attempts reset to zero and the same mate. If the number of crossover attempts reaches `mate-xo-tries-limit` without an offspring being accepted, a new mate is chosen for the current-solution. The number of times the current-solution is used, `xo-tries-limit`, is also a parameter of the algorithm. After `xo-tries-limit` crossovers, the current-solution is discarded and a new one randomly generated. After a fixed number of fitness evaluations or when a perfect solution is found the algorithm terminates and returns the fitness of the fittest-overall-solution.

We experimented with values for both parameters of this algorithm. Since a mate is randomly generated, the algorithm was not very sensitive to the value of `mate-xo-tries-limit`. However, `xo-tries-limit` is integral to the algorithm because it sets a limit for crossover attempts after which the search moves *randomly* elsewhere. If its value is set too low, the search may not find a fitter candidate even though one exists in the neighbourhood. If it is set too high, the search may be trapped in a local optimum. We used values equal to the most successful `max-mutation` settings in the SIHC runs.

6.2.2. Crossover Simulated Annealing: XOSA

The XOSA algorithm uses GP crossover (to generate candidate points from two "current" solutions) and the SA acceptance criterion. It has one new parameter: `xo-tries-limit`. Every `xo-tries-limit` crossovers, the weakest current solution is

replaced with a random program. This ensures sufficient novelty.

The acceptance criterion is revised to use 2 parameters instead of global variables. The formal parameters are: `current-fitness` and `candidate-fitness`. The predicate uses the current temperature, a calculated fitness differential and a random number generator to indicate whether the candidate state of the system should be accepted.

There are three different versions of XOSA: XOSA-Average, XOSA-One, and XOSA-Each, which differ in terms of what values are passed as arguments to the acceptance criterion predicate and in terms of which current solution is replaced if acceptance is indicated.

In XOSA-Average, the actual parameter for `current-fitness` is the average fitness of the two current solutions. The actual parameter for `candidate-fitness` is the average fitness of two candidate solutions derived from twice crossing over the current solutions. If acceptance is indicated, both the candidate solutions replace the current solutions. Since XOSA-Average uses the SA component for each pair of fitness evaluations, the SA component is adjusted to use half as many steps in the cooling schedule.

In XOSA-One, only one child is generated, via crossover, from the current solutions. The fitness of the weaker parent is the value for `current-fitness` and the fitness of the child is the value for `candidate-fitness`. The weakest parent is replaced by the child, if acceptance is indicated.

XOSA-Each uses a 2-parent to 2-children crossover function with the option of accepting one child or both. If the weakest current solution can be replaced by the fittest child, this is done and then an attempt is made to exchange the fitter current solution for the weakest child. Otherwise, an attempt is made to exchange the weakest parent for the weakest child.

The only parameter of the crossover component of the algorithm is `xo-tries-limit`. Once this many crossover attempts have been made with the same pair of parents, the weaker parent is replaced by a random program. Like `xo-tries-limit` in XO-SIHC this parameter is a sort of patience threshold. We used the same values for it in XOSA as in XO-SIHC.

6.3. Crossover Based Experiments

We experiment with XO-SIHC and XOSA and compare them to GP. We use the problem suite of the thesis (described Section 2.1) just as we did in Chapter 5. The problems are:

- `6-Mult`
- `11-Mult`
- `Sort-A`
- `Sort-B`
- `Block Stacking (BS)`

The three algorithms are also compared by the same standards as Chapter 5. To reiterate, each execution of a run is allowed to process a maximum of 25500 candidate solutions or individuals. A GP execution is permitted to process for a maximum of 50 generations beyond the initial population because the population size is 500. Processing implies a candidate solution being selected and then crossed over and evaluated for fitness, if necessary. It also includes the random generation and fitness evaluation of generation 0. This yields a maximum of 25500 individuals processed. The fitness values in the GP runs for all five problems were scaled by linear and exponential factors of 2. Each GP execution uses a "generation gap" of 0.9 which means that the crossover operator is applied 90% of the time with the remaining 10% of individuals chosen by selection being directly copied into the next generation. The latter individuals are not re-evaluated for fitness, therefore GP performs a maximum of 23000 fitness evaluations[2].

In an execution of XOSA or XO-SIHC processing implies an individual (or candidate) being generated by mutation and evaluated for fitness. We use the same initial temperature (1.5) for all XOSA runs and the final temperatures listed in Table 13 on page 145.

For each experiment, on a per run basis, we show the probability of success the average fitness of the best individual of each execution and the average num-

[2] $500 + (10\% \times (50))$

ber 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 indicated in parentheses.

6.3.1. Results of XOSA, XO-SIHC and GP: 6-Mult

6 Bit Boolean Multiplexer	Rate of Success (%)	Fittest Individual (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	79.5 (40.3)	98.0 (4.5)	55.7 (27.8)	48.4
$XO-SIHC$, xo-tries-limit = 100	100.0	100.0	20.56	20.56 (14.2)
$XO-SIHC$, xo-tries-limit = 1000	96.7 (17.8)	99.5	23.2	23.2
$XOSA-Each$	36.7 (48.2)	91.6	92.0	80.5
$XOSA-One$	16.7 (37.3)	64.6	94.8	70.0
$XOSA-Ave$	3.3 (17.9)	65.7	99.2	87.0

Table 29. GP, XOSA and XO-SIHC Results for `6-Mult`

Clearly `6-Mult` is an easy program discovery problem for all the algorithms considered in this thesis because each of them found solutions. Of the three algorithms (GP, XOSA, XO-SIHC) shown in Table 29 on page 170, XO-SIHC stood out from GP and XOSA by solving the problem 100% of the time or 96.7% (17.8) of the time depending on the value of `xo-tries-limit`. XOSA performed much worse than the other two algorithms. Its most successful version, XOSA-Each only solved `6-Mult` with a rate of 36.7%. This equates to 11 executions out of 30. All the rates of success for XOHC are significant despite the large standard deviations.

6.3.2. Results of XOSA, XO-SIHC and GP: 11-Mult

The results for `11-Mult` are shown in Table 30 on page 171. We remarked in Chapter 5 that processing a maximum of 25500 individuals was too great a constraint for GP in attempting to solve `11-Mult`. The same result held for XOSA regardless of whether the version was XOSA-each, XOSA-one, XOSA-average. However, like the SA result of Chapter 5, XO-SIHC was able to solve this problem. Its rate of success was 16.7% (5 executions out of 30) which was the same as SA. XO-SIHC with `xo-tries-limit` set to 5000 only needed to process an average of 60% of the maximum allowed individuals on its successful executions. While XOSA did not find a solution, the best solution it found over all executions and the average fitness of the best solution of each execution was not significantly different from GP.

11 Bit Boolean Multiplexer	Rate of Success (%)	Best of All Runs (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Execs (%)
GP	0	87.6	79.2	100	
XO – SIHC, xo-tries-limit=100	16.7	100.0	93.6	95.4	72.4
XO – SIHC, xo-tries-limit=5000	16.7	100.0	94.4	93.3	60.0
XOSA – Each	0	75.0	66.5	100.0	
XOSA – One	0	78.9	68.5	100.0	
XOSA – Ave	0	81.3	71.7	100.0	

Table 30. GP, XOSA and XO-SIHC Results of `11-Mult`

6.3.3. Results of XOSA, XO-SIHC and GP: Sorting

The results for Sorting are a contrast to those discussed already and `Block Stacking`. XO-SIHC performed the worst of the three algorithms rather than best. In Tables 31 and 32 we show the best result of XO-SIHC given the different values we tried for `xo-tries-limit`. It only solved `Sort-A` with a 10% rate of success and `Sort-B` with a 40% rate of success. This was significantly lower than either GP or XOSA.

Furthermore, whereas XOSA did not perform well on `6-Mult`, `11-Mult` or `BS`, on Sorting at least one version matched or equalled the strong success of GP. For `sort-A`

GP had a success rate of 80% (40.0) and XOSA-one had a success rate of 93.3% (25.0). While this difference is not statistically significant, in terms of individuals processed for all executions and only successful executions, XOSA required significantly less. For `Sort-B` GP had a success rate of 67.7% (44.1) and XOSA-average had a rate of 90.0% (30.0). This difference is significant.

`Sort-A`	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	80.0 (40.0)	49.7 (18.1)	51.3 (35.0)	39.1 (26.6)
XO – SIHC, xo-tries-limit = 50	10.0 (30.0)	57.8	92.3	22.7 (22.5)
XOSA – Each	70.0 (45.8)	80.0	56.8	55.3 (25.5)
XOSA – One	93.3 (25.0)	98.0	25.1	22.5 (22.5)
XOSA – Ave	66.7 (47.1)	85.2	67.8	52.0 (31.5)

Table 31. GP, XOSA and XO-SIHC Results for `Sort-A`

`Sort-B`	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	67.7 (47.1)	85.3 (22.6)	60.4 (37.5)	40.6 (28.2)
XO – SIHC, xo-tries-limit = 50	40.0 (49.0)	82.0 (17.5)	81.4	53.6 (31.1)
XOSA – Each	80.0	89.4	58.9	48.75 (25.1)
XOSA – One	83.3	91.2	48.[illegible]	38.3 (30.9)
XOSA – Ave	90.0 (30.0)	97.6	65.3	61.5 (17.9)

Table 32. GP, XOSA and XO-SIHC Results for `Sort-B`

6.3.4. Results of XOSA, XO-SIHC and GP: Block Stacking

The results of `Block Stacking` confirm that it seems to be a simple program discovery problem despite the apparent sophistication of the task. They are displayed in Table 33. Once again, XO-SIHC was tremendously successful. It solved the problem

with a 100% rate of success and only processed an astonishing 2.6% of the maximum allowed individuals. XOSA-One was close to XO-SIHC in rate of success but on successful runs it required, on average, processing 2.58% (27.1) of the maximum individuals. This value is significant despite its large standard deviation. GP ranked third on this problem. Its rate of success was 76.7% (16.3) and it processed an average of 25.1% of the maximum allowed individuals on successful executions.

Block Stacking	Rate of Success (%)	Fittest Individual (%)	Inds Proc'd (%)	Inds Proc'd in Succ Execs (%)
GP	76.7 (16.3)	87.7	43.7	35.1 (26.6)
XO – SIHC, xo-tries-limit=250	100.0	100.0	2.6	2.6 (2.3)
XOSA – Each	70.0 (45.8)	87.7	75.1	64.7 (26.5)
XOSA – One	96.7 (17.8)	98.4	28.3	25.8 (27.1)
XOSA – Are	60.0 (49.0)	84.5	88.4	80.8 (18.96)

Table 33. GP, XOSA and XO-SIHC Results for `Block Stacking`

6.3.5. Summary of XO-SIHC and XOSA Results

Among these XO-SIHC and SA algorithms, it is puzzling that, on any given problem, neither both algorithms with the same operator, nor, both operators with the same algorithm had correlated performance. Future work investigating precise reasons for this is worth pursuing.

XO-SIHC is definitely a search algorithm worth consideration for program discovery problems, even given the small number of problems in our test suite. It outperforms or equals the best of the other algorithms on `6-Mult`, `11-Mult` and `Block Stacking`. However, its poor performance on Sorting, relative to the other algorithms, is a reminder that, because there is a sensitive relationship between a problem, search operator and fitness landscape, it can not be expected to always be superior.

In contrast to the strong performance of XO-SIHC, the XOSA results are not as uniformly good. This concurs with our earlier conjecture that XOSA may not work because the GP crossover operator generates offspring that are too different from their parents when the temperature is quite low. We observe that there was a

consistent relationship between XOSA and XO-SIHC in one respect: whenever XO-SIHC performed well, XOSA did not, and, whenever XOSA performed well, XO-SIHC did not.

From this suite it was impossible to claim that one version of XOSA was superior overall to the others. On each problem, except Sort-B, there was significant difference among the three versions (Average, Each, One). While Average was never solely best, Best and Each exchanged rankings on different problems.

Our goal at the outset was to determine if GP crossover would prove useful when coupled with SA or SIHC. The answer is affirmative but it demands qualification. That is, XOSA and XO-SIHC can solve program discovery problems, *but*, their efficiency depends upon how suited the algorithm is to exploit the fitness landscape created by the GP crossover operator.

Given this qualification (which holds more generally for any algorithm and the fitness landscapes of any search operator), one avenue of future work concerns determining how an algorithm (complete with search operator) should be chosen. Does there exist a statistical test that, given the computational overhead of executing it, can indicate what algorithm is best to use? Should this test be centered around search operator behaviour? Is it best to run this test before selecting an algorithm or, can one algorithm combining all operators and search strategies dynamically evaluate its performance and adaptively select the method it uses?

For more serious problems, (i.e., those that are not benchmarks), a practical issue to investigate is how much parameter tweaking is required to "squeeze out" the best result? Does the difficulty in parameter selection, imply a ranking in the usefulness of the algorithms?

6.4. Hybridization of GP and Local Search

The results of the alternative algorithms we compared to GP are sufficiently encouraging to suggest adding a local search component to GP towards the goal of improving it. GAs are a population-based technique that provides a means of quickly focusing search on a fit area of the search space. This is the recognized effect of the algorithm as, over time, the population becomes more homogeneous due to selection based upon fitness and the combinative effect of crossover. Mutation is usually set

to a background rate and plays only a minor role in terms of exploitative search (it ensures no premature allele loss). However, once a GA has found fit areas of the search space, it searches over only a small fraction of the neighbourhood around each search point. It must derive its power from integrating multiple single neighbourhood explorations in parallel over successive generations of a population. This "many points, few neighbours" global strategy is in direct contrast to a hill climber or simulated annealer (at low temperature) which potentially focuses effort on a greater fraction of the search neighbourhood of one point but only around one point at a time. This local strategy might be called "few points, many neighbours". The two strategies have been complementary [15, 18, 54, 21] in other problem domains with the GA component serving to "zero in" on regions of high fitness and the local search component serving to thoroughly explore the regions the GA has found. Therefore, our conjecture is that GP plus a hill climbing component might be profitable on our suite of problems and, more generally, program discovery.

We implement two major types of hybridized algorithm: one uses the stochastic iterated mutation-based hill climbing algorithm of Chapter 5, "GP+MU-HC" and the other uses the GP crossover stochastic iterated hill climbing algorithm of Section 6.2.1. This permits the comparison of the hybrid with a mutation based hill climber (using HVL-Mutate) to a crossover-based one. The algorithms are described in Section 6.5.

The obvious qualitative difference worth analyzing is that mutation introduces totally unselected genetic material while the crossover operator (if it draws a mate from the population at large or from the pool of fittest individuals) replaces swapped out genetic material with material that has undergone selection by surviving through GP's simulated process of evolution.

6.5. Hybridized GP and Hill Climbing Algorithms

The algorithms designed for hybridized GP and hill climbing, GP + XOHC and GP + MU-HC are simple: they are controlled by user specified parameters and they do not adapt their behaviour based upon performance.

Every g generations, the f fittest individuals in the population are used as the starting points of a hill climbing search. Each hill climb processes c candidate solutions. For convenience, in this section we shall call each candidate solution processed

a "step". The best individual from each hill climb is placed in the next generation and then the remaining individuals of the population for that generation are generated standardly (i.e. with crossover or direct reproduction). For an execution the move operator of the hill climb is either entirely GP crossover or entirely HVL-Mutate.

The interesting design issue is how to obtain mates for the crossover hill climb. Should one exploit the knowledge embodied by the current population by using its membership as a source of mates? The answer provides insight into the role of GP crossover. We experiment with 3 versions of GP + XOHC:

Random: Mates are not drawn from the population at all, but are randomly created. This is similar to using the implemented version of XO-SIHC. It allows non-selected material to be combined with selected material but seems to run contrary to exploiting the fact that a population of GP programs has undergone selection.

Best: Mates are drawn from the group of individuals with the highest fitness. This group will vary in size from one to population size (early in an execution). **Best** potentially increases the chances that a crossover recombination will generate a very fit offspring under the assumption that genetic material from the best members is a source of building blocks because it has undergone selection.

Population: Mates are randomly drawn from the population at large. This version is "middle of the road" between **best** and **random**. The population is a more diverse source of genetic material compared to the **best** pool, but, unlike the random option, it has undergone selection.

We decided to check for the presence of a perfect individual only at the end of a generation. With this decision it does not matter whether hill climbing is done before or after the normal GP crossover of a generation. One consequence is that the actual number of fitness evaluations reported for successful runs is slightly over-estimated but no more than if a standard GP run were executed and the same check done at the end of each generation.

6.6. Hybrid Algorithm Experiments

We compare this selection of hybrids to GP. We use the same test suite and standard GP experimental parameters previously used in Chapter 5 and Section 6.3. For brevity the reader is referred to the description of Section 6.3 starting on page 169.

The parameters g, f, and e are supplied before execution and are consistent for a run. We run three parameter settings. All runs hill climb from the five fittest programs: $f = 5$. When the climb is 500 evaluations, the interval g is either 2 or 5 generations. When the climb is 100 evaluations, the interval g is 3 generations.

The maximum number of generations of a hybrid algorithm execution is adjusted to take into account the additional processing of individuals by the hill climbing element. A maximum which gives a close approximation to the *a priori* given maximum number of individuals processed and population size is calculated. In the case of 25500 evaluations and a population of 500 (which is used in every GP+HC run), for a run with $f = 5$, $g = 3$ and $e = 100$ the maximum generations is 39 and the maximum individuals processed is 26435. For $f = 5$, $g = 2$, $e = 500$, the maximum generations is 15 and the maximum individuals processed is 25465. Finally, for $f = 5$, $g = 5$, $e = 500$, the maximum generations is 26 and the maximum individuals processed is 25975.

6.6.1. Results of Hybrids: 6-Mult

The GP+XOHC-population hybrid solved `6-Mult` 100% of the time when the five best programs were selected for hill climbs of 500 steps every two generations. This was statistically superior to GP by 20 percentage points. In the GP runs the successful executions with higher probability of success unfortunately processed more individuals than the successful runs with lower probability of success. The individuals processed by *GP+XOHC-Population* were (significantly) 10 - 20% lower than GP. Thus, GP+XOHC-Population was not only more reliable but also less computationally expensive.

The parameters favouring more frequent and longer hill climbs proved better than the others. Any version of GP+XOHC i.e., Population, Best or Random, which, every second generation, executed five 500 step crossover hill climbs ($g = 2$ and $e = 500$) was comparable or better than GP. However, when the crossover hill climbing element

6 Bit Boolean Multiplexer	Rate of Success (%)	Fittest Individual (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	79.5 (40.3)	98.0 (4.5)	55.7	48.4
$GP + MU - HC,\ f = 5$				
$e = 500, g = 5$	93.3 (25.0)	99.5	45.2	43.3 (22.5)
$e = 500, g = 2$	90.0 (30.0)	99.5	32.5	30.6 (5.1)
$e = 100, g = 3$	80.0 (40.0)	99.8	57.0	47.7 (21.3)
$GP + XOHC_{f=5,\ e=100,\ g=3}$				
Best	50.0 (50.0)	96.8 (3.7)	74.0 (29.8)	50.0 (24.9)
Random	60.0 (14.7)	96.9 (4.2)	72.1 (29.6)	32.4 (17.9)
Pop	63.3 (14.5)	95.7 (4.2)	65.3 (32.5)	44.9 (20.8)
$GP + XOHC_{f=5,\ e=500,\ g=2}$				
Best	86.7 (10.2)	99.4 (1.7)	44.7	32.4 (17.9)
Random	93.3 (7.5)	99.7 (1.2)	45.9	44.9 (17.7)
Pop	100.0	100.0	31.3	31.3 (17.4)

Table 34. GP and Hybrid Results for `6-Mult`

was instead changed to execute 100 steps every three generations ($g = 3,\ e = 100$), the corresponding rate of success fell quite below that of GP alone.

GP+MU-HC also had a significantly better rate of success than GP. As with the crossover hill climbing hybrids, the executions with hill climbs of 500 steps were more successful than the hill climbs of 100 steps. There was no statistical difference in rate of success between executing the 500 step hill climb every five or two generations. The choice of any e or g was comparable or better than GP. The best result had a rate of success equaling 93.3% (25.0).

6.6.2. Results of Hybrids: 11-Mult

Once hill climbing was combined with GP, a GP-based algorithm finally existed that could find a perfect solution to `11-Mult` while processing less than 25500 individuals. GP+XOHC-Random and GP+MU-HC, both executing with five hill climbs of 500 steps every second generation, each managed to solve `11-Mult` once in 30 executions.

This had not ever been done with GP alone. While these two GP hybrids did yield an improvement over GP, they did not better XO-SIHC and SA which were significantly the best. Both XO-SIHC and SA had a rate of success equal to 16.7% versus the 3.3% rate of the former two hybrids. It comes as little surprise that the Random version of GP+XOHC would fare as well as GP+MU-HC because using a random partner is similar to HVL-Mutate's use of random material. The fact that using selected material did not improve the crossover hill climbing versions best and population did not find a solution) concurs with results of Section 6.3.2 and Chapter 5: both XOHC and SA achieved success rates of 16.7% and they do not exploit selected material either.

11 Bit Boolean Multiplexer	Rate of Success (%)	Best of All Execs (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	0	87.6	79.2	100	
$GP + MU - HC, f = 5$					
$e = 500, g = 5$	0	91.9	89.6	100.0	
$e = 500, g = 2$	3.3	100.0	91.0	99.5	48.6
$GP + XOHC, f = 5, e = 500, g = 2$					
Best	0	91.9	89.6	100.0	
Random	3.3	100.0	90.2	99.8	97.9
Population	0	91.4	85.0	100.0	

Table 35. GP and Hybrid Results for `11-Mult`

6.6.3. Results of Hybrids: Sorting

Once again there are indications that the fitness landscapes of both sorting problems differ from the rest in our test suite. Recall that only sorting did not respond to XO-SIHC and did respond to XOSA. In the case of hybrid experimentation, crossover hill climbing did not yield significant improvement over GP but mutation hill climbing did. On both `Sort-A` and `Sort-B`, GP+MU-HC runs, with a 100 step hill climb every three generations ($e = 100, g = 3$) were 100% successful. When more effort was devoted to hill climbing more frequently and longer, (500 steps every two generations),

the rate of success was still good: 93.3%, but not as high. These two results concur with the low number of improvement steps per climb (between 1 and 2) observed in the successful SIHC executions that were reported in Chapter 5.

Sort-A	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	80.0 (40.0)	49.7 (18.1)	51.3 (35.0)	39.1 (26.6)
$GP + MU - HC$, $f = 5$				
$e = 500, g = 2$	93.3 (25.0)	94.9	32.5 (29.8)	11.3 (20.0)
$e = 100, g = 3$	100.0	100.0	42.0	42.0 (26.9)
$GP + XOHC$, Pop, $f = 5$				
$e = 500, g = 2$	70.0 (45.8)	88.5 (20.3)	58.4 (31.2)	43.4 (18.2)
$e = 100, g = 3$	80.0 (40.0)	89.4 (21.0)	50.6 (34.3)	39.6 (26.6)
$GP + XOHC$, Random, $f = 5$				
$e = 100, g = 3$	90.0 (30.0)	58.3 (9.6)	42.5 (29.8)	35.7 (25.5)

Table 36. GP and Hybrid Results for Sort-A

Sort-B	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	80.0 (40.0)	49.7 (18.1)	51.3 (35.0)	39.1 (26.6)
$GP + MU - HC$, $f = 5$				
$e = 500, g = 2$	93.3 (25.0)	96.5	32.3 (30.0)	30.9 (30.0)
$e = 100, g = 3$	100.0	100.0	40.3	40.3 (27.2)
$GP + XOHC$, Pop, $f = 5$				
$e = 500, g = 2$	66.7 (47.1)	88.5 (20.3)	58.8 (31.0)	44.3 (18.4)
$e = 100, g = 3$	80.0 (40.0)	89.4 (21.0)	50.8 (34.0)	40.0 (26.2))
$GP + XOHC$, Random, $f = 5$				
$e = 100, g = 3$	90.0 (30.0)	97.1 (10.5)	43.2 (29.3)	36.8 (25.1)

Table 37. GP and Hybrid Results for Sort-B

6.6.4. Results of Hybrids: Block Stacking

Block Stacking	Rate of Success (%)	Fittest Individual (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	76.7 (16.3)	87.7	43.7	35.1 (26.6)
$GP + MU - HC$, $f = 5$				
$e = 500, g = 2$	100.0	100.0	5.3	5.3 (3.3)
$e = 100, g = 3$	100.0	100.0	12.5	12.5 (10.9)
$GP + XOHC$, Pop, $f = 5$				
$e = 500, g = 2$	100.0	100.0	5.3	5.3 (3.3)
$e = 100, g = 3$	100.0	100.0	13.1	13.1 (11.8)
$GP + XOHC$, Random, $f = 5$				
$e = 100, g = 3$	100.0	100.0	10.6	10.6 (6.3)

Table 38. GP and Hybrid Results for **Block Stacking**

In this problem hybrids produced a difference that boosted the rate of success up to 100% compared to 76.7% with GP. However, as specified, **Block Stacking** seems an easy program discovery problem. While every version and parameter combination of GP plus hill climbing hybrid solved **BS** 100% of the time, so did XO-SIHC and SA. Among the hybrids there is differentiation based upon how many individuals were processed. The hybrids which hill climbed for 500 steps every two generations (e = 500, g=2) were better regardless of whether crossover or mutation hill climbing was used. These processed approximately 5.3% of available individuals. This result also concurs with the fact that the algorithms which strictly hill climbed (XO-SIHC and SIHC) were very successful.

6.7. Summary of Hybrid Results

Hybridization of GP and local search via hill climbing improved GP. At least one version of a hybrid of GP plus Hill Climbing was better than GP alone **on all problems, in both forms - GP+XOHC, GP+MU-HC,** with at least one setting of the parameters. We observed no consistent significant ordering between the mutation hill

climbing option or crossover hill climbing option across the test suite. Prior to using this search strategy, GP had not been superior nor sometimes even on par with either crossover-based or mutation-based versions of SA and SIHC. With hybridization the evolution inspired search model is, at the least, comparable. It is more encouraging in view of the simple parameterized style of the algorithms. Future work could pursue adaptive hybrid versions which adjust hill climb duration or the timing of a hill climbing according to the rate of fitness change of the population or best individual. Also, a meta-level parameter selection scheme might relieve the designer of parameter choices and improve a hybrid algorithm by improving its adaptability.

Regarding the relative merits of the Random, Best, and Population versions of GP+XOHC, current data is not decisive. Additiona .perimentation, beyond the present scope, seems called for. Qualitatively, Best may not be explorative enough because it is limited to a mate pool that may be very small. Preliminarily, Best was outperformed on 6-Mult but comparable on 11-Mult. Population worked better than Random on 6-Mult but the results were reversed on 11-Mult. On Block Stacking and both sorting problems Random and Population were equal. Another direction future work might follow would be to understand what better performance of a hybrid version implies about the efficacy of non hybrid algorithms. If, on a given problem, random material proves as useful as duly evolved and selected material, single-point search algorithms such as SA and SIHC may perform superior to GP.

6.7.1. Results of Other Literature

Lang in [75] also investigated the role of GP crossover and asked whether

> the idea that the pool of highly fit individuals selected from earlier generations constitutes a valuable genetic resource that facilitates the creation of even more fit individuals in the future [75, p. 340]

Lang used a version of crossover hill climbing that starts with a single random program and always keeps track of the best program seen. At each step, one candidate new program is probabilistically created and compared with the current single best. If the new candidate is at least as fit as the current best program, it becomes the new best program.

The candidate is created differently on alternate steps. On the even-numbered steps (starting with time step 0), the new candidate is a randomly created entirely new program. On the odd-numbered steps, the new candidate is the offspring produced by crossing over the current best program with a randomly created new program using GP crossover.

Lang's algorithm over even-numbered steps is simply random search. Such steps have little effect on the success of a climb with large problem spaces. They are inefficient. Lang's algorithm over odd-numbered steps only and our own relate in the following way: If Y is the number of times Lang's algorithm is iterated and X is the maximum number of odd-numbered steps in one iteration (e.g., in Lang's experiments, Y was 100 and X was 625 because he examined 100 runs with a maximum of 1250 individuals processed in each hill climb), then Y executions of our crossover hill climb algorithm with mate-xo-tries-limit set to one, xo-tries-limit set to X and the maximum number of individuals processed set to X is equivalent. Given the random behaviour of Lang's algorithm over even-numbered steps, one expects it to perform, overall, less efficiently than ours.

Lang based his comparison on 100 runs of all 80 distinct 3-paramater Boolean functions. He compared his algorithm to GP run with a population of 50 for a maximum of 25 generations yielding a maximum of 1250 individuals processed. He considered both the probability of success and the number of candidate solutions processed. The latter value he measured using the ratio of candidates to solutions during the complete set of runs for a given target function. He found that on any given run the hill climbing algorithm was four times less likely that GP to find a solution. However, when it found a solution it did so about fifty times faster. Given this, because hill climbing can be used in an iterated manner, he states, "hill climbing is more efficient than genetic programming on this task" ([75], p. 341).

He concluded

> Since we found solutions much faster that GP did, apparently the high-fitness population maintained by GP was a worse than random source of components for building improved individuals. [75, p. 342]

Because Lang had results for only one small problem, his conclusions appear over-extended. First, his conclusion begs the question of choosing between the length of a hill climb and how many hill climbs to perform when a maximum number of fitness

evaluations can be devoted to iterated hill climbing. As our analysis has shown this is an important yet opaque decision that may influence performance.

Second, Koza in [73] responded to Lang's conclusions with an explanation of why they will not extend to boolean problems that are only slightly larger. He points out that Lang's hill climbing algorithm will not scale to 4 and 5 bit parity problems when judged by the same maximum number of fitness evaluations (1250). As the hill climb continues upward, its even-numbered steps become practically irrelevant because the likelihood of finding a better solution in the space is less than a million. This leaves the onus of the algorithm on the odd-numbered steps and Koza surmises these steps improve fitness because Lang's hill climbs continue upwards even as the even-numbered steps become useless. Our crossover hill climbing results can validate what Koza surmises because it has success. Our results both support and qualify it because, while crossover hill climbing was successful with Boolean multiplexer problems that are similar to parity functions and Block Stacking, it did not perform well on the Sorting problems. Thus, it is plausible to expect Lang's even-number steps to perform well on the parity problem but not, in general, on all problems.

By his analysis Koza finds a measure of justification for stating that, contrary to Lang's claim, the population pool is an important element in GP. In crossover hill climbing the current best solution is itself selected and it is (the sole member of the) population pool However, to us, it appears that crossover hill climbing states little about the role of GP crossover in GP because the population of GP is much larger and undergoes selection differently. The important insight we derive from crossover hill climbing is that, in this algorithm, GP crossover functions as a macro-mutation operator.

Our hybrid versions of GP provide a better means of gaining insight about the utility of the population pool than a crossover hill climbing algorithm. As we state in Section 5.6, our hybrid comparisons, despite using a larger suite and one with one problems that increases in scale from another, are not extensive enough to shed definitive light on the issue. It appears that the issue may be quite complicated and not sharply divided. The issue is important for GAs in general and recent work by Terry Jones ([49] using his "Headless Chicken" test is quite extensive. The Headless Chicken test compares a GA with crossover to one which uses crossover that only selects one parent from the population and uses a random solution for its mate. Jones'

test suite does not include any program discovery problems or GP but a Headless Chicken test on a GP with a suite of program discovery problems would complement the crossover analysis of this chapter.

6.8. Program Discovery Algorithms Reviewed

Each algorithm investigated in this thesis: GP, SA, SIHC, XOSA, XO-SIHC, GP+XOHC, and GP+MU-HC, obviously has one or more salient features that distinguish it from the others. These particular differences earn it a special niche among the others and afford it some degree of merit. They can be found by examining which search operator an algorithm uses and how the algorithm controls traversal of the search space. They also indicate the importance of exploiting knowledge of the character of the search space and understanding the behavioural interactions of search operators and search strategies on a search space.

A stronger, general observation about the collection of algorithms is that none of them is *always* superior to the rest. Thus, it is as important to understand their commonalities as it is to contrast them.

Why do all these different algorithms work? They all face the problem of searching the large space of programs that vary in length and structure by assuming the framework of program composition from primitives. Both search operators: GP crossover and HVL-Mutate generate programs that differ in size and structure from their parent(s). They both also, generate programs that, while capturing some inherited features and introducing other random ones, are always syntactically correct. A valuable insight is that a variable length hierarchical representation may be a more fundamental asset to program discovery than any particular search technique.

Another crucial commonality among the algorithms is that each implements a unique version of three principles which are characteristic of the adaptive search process of evolution. These principles are: selection, inheritance and blind variation. While there is little argument that this contention holds for GP, it is not immediately obvious how it holds for SA, XOSA, SIHC and XO-SIHC. By explaining how the principles are expressed in GP, it becomes clearer how they are realized in SA, XOSA, SIHC and XO-SIHC:

- **Selection** "Survival of the fittest" in biology is an evolutionary selection process

in which superior individuals survive long enough to reproduce and propagate in greater numbers than inferior ones. In GP, roulette-wheel selection of parents is probabilistically biased towards programs with greater than average fitness. When a program is chosen by roulette-wheel selection, it is given the opportunity to propagate itself or its "features", in effect, following the principle of "survival of the fittest". All offspring survive in the sense that they form the next generation. This survival is completely deterministic. In SA or XOSA the search always moves when a candidate program (generated from the current point) is equal or superior in fitness to the current point. In this respect, deterministically the fittest survive. Because SA accepts a candidate program of inferior fitness with a probability dependent upon the system temperature and the different in the two programs' fitnesses, it has a similarity to roulette-wheel selection which always accords each individual in the population, regardless of fitness, a non-zero chance of being selected as a parent. SIHC is the simplest implementation of selection: the search only moves to candidate programs of equal or superior fitness.

- **Inheritance** In the sexual reproduction of biology, an offspring inherits its genetic material from its parents. The GP crossover operator implements sexually derived inheritance. Whether the crossover creates two "child" programs from a pair of parents or simply one, an offspring inherits from its parent(s) in so far as before any crossover, it is a copy of one parent and, after crossover, it is a combination of both. The HVL-Mutate operator implements asexual inheritance in the respect that it generates a candidate program that retains some of its parent program contents. Therefore, any algorithm making use of either GP crossover or HVL-Mutate, implements inheritance.

- **Blind Variation** In biology, mutations play the role of blindly changing a genome and allowing an offspring to express a genetic feature that is not common to its parents. The GP crossover operator does not have a direct correspondence for mutation in this context. However, it has a "blind" aspect which is the manner in which a crossover point is chosen. Blind crossover point selection provides blind variation in the sense that the place of combination of "genetic information" from the parent programs is not deterministically chosen.

HVL-Mutate obviously expresses blind variation.

One issue concerning the appropriateness of an evolution-based, common descriptive framework arises because, while GP is population-based, SA, XOSA, SIHC and XO-SIHC do not appear to be. A coarse argument is that the GP alternatives all consider and process a population of size one. In fact, it is possible to be more careful about this response. The evolution of a population results in a set of parallel ancestry lines. In the case of GP these ancestry lines are interconnected, concurrent and synchronously processed. In GP alternatives, they simply are neither interconnected nor concurrent. If each line of ancestry is placed beside each other, the programs at corresponding time points in the parallel sequences represent a population. The members of each ancestry line can be seen as evolving independently and asynchronously.

A final question arises: why is this common framework useful? Admittedly, it does not help the task of selecting an algorithm to accomplish a task efficiently. Differences are arguably more useful to study than similarities for this purpose. However, a common, evolution-based framework indicates:

- that new implementations of selection, inheritance and blind variation are only limited by the scope of our imagination
- that future effort should be devoted to studying the "real" process of evolution and using it as a model for extensions to all adaptive search algorithms. For example, when SA is viewed as an annealing process it appears to have limited extension. However, parallelizations of it and the substitution of new search operators, based on ideas borrowed from evolution, provide it with new potential.
- that it is important to ask how a sub-process or characteristic of any model can be incorporated into each of the algorithms rather than just one of them. Any proposed extensions to one algorithm in the form of an operator or selection strategy should be generalizable to the rest.

6.9. Chapter Summary

In this chapter we demonstrated that GP crossover, when integrated into SA and SIHC as a substitution for HVL-Mutate has some success with program discovery

problems. We conjectured this much based upon comparing the syntax correlation of GP crossover landscapes to those of HVL-Mutate and our recognition of it as a macro-mutation operator.

We introduced hybridized algorithms that combined GP and stochastic iterated hill climbing. The hill climbing components of the algorithms differed in terms of whether they used HVL-Mutate or GP crossover. Three different versions of GP+XOHC were proposed and tested: Population, Best, and Random. Our testing did not reveal enough about the versions to provide a adequate comparison but we compared them on a qualitative basis.

In Chapter 7 we shall review the goal and results of the thesis, expand to a broader focus in order to place them in perspective and suggest possible new related agendas.

CHAPTER 7

Conclusions and Future Work

7.1. Summary of Thesis Results

The overall goal of this thesis was to analyze GP so we could extend our understanding of GP. We also desire to improve GP.

Essentially, Chapters 3 and 4 clarified how GP works. In Chapter 3 our specific goals were to judge how much of GP's success arises from skillful designer choice of the primitives, fitness function and test suite and to verify a conjecture that GP was hierarchical in solutions and process. They were accomplished by considering a novel program discovery problem called Sorting. We analysed the choices encountered considering primitive selection and test suite choice while concurrently we tried to solve the problem using primitives that did not unduly *a priori* constrain the definition of subtasks. Independently we considered fitness function design.

Concerning the first goal, we report a strong dependence of success upon the skillful choice of designer-supplied primitives and the design of a test suite. Fitness function design, at present, is fortuitous rather than deliberately controlled.

- regarding test suite design, we considered the influence of test suite size on the degree of solution generality. We found that for a problem with an unbounded number of possible test cases – Sort-A, as we enlarged the sample we improved generality. A test suite of approximately 200 array elements in 48 different arrays was sufficient to produce solutions that could correctly solve an out of training test suite approximately three times larger approximately 90% of the time. For the problem 6-Mult which had a finite number of possible test cases, the test suite with half the possible number generalized better than the test

suite with three quarters. As well, the sample test suites only found a 100% correct solution 25% of the time.

- regarding test suite design, we experimentally assessed the influence of expressing incremental encouragement, "coaching", via test case weighting. We weighted the same suite of **Sorting** test cases using four different fitness credit schemes: **linear**, **exponential**, **equal test case** and **equal element**. The credit assigned made some difference in performance but that performance was unpredictable. This suggests a need for designer expertise.

- regarding fitness function design, we conclude that it has a large influence in determining the character of the search space in terms of optima and basins. Ideally, a fitness function should be chosen because it helps define a search space with characteristics amenable to GP search. However, despite its large impact, little information is yet available to designers to guide their choices. Good fitness function choices appear to be fortuitous rather than truly deliberate.

- regarding primitive design, we related the design decision process when solving **Sorting** with GP. We reported that the designer makes decisions on the following issues:

  - trading off the expressive power of a primitive with constraining the search space to make the search space manageable in size.
  - trading off exploiting problem specific knowledge (by incorporating it into the primitives) with *a priori* biasing or constraining the outcome.
  - deciding upon the scope of variables for control of program side effects

  In addition, the graduated primitive sets used in the exercise of verifying hierarchical process showed that GP's success depends upon its being supplied with primitives of a sufficiently high level of functionality. As primitive generality increased, GP's ability to solve the **Sorting** decreased.

Regarding the second goal, the process of confirming a hierarchical process demands that GP be tested using primitives that do not preordain the definitions of subtasks that could be formed. We devised a detailed set of experiments that used

progressively more generalized primitive sets. We compared runs using two population sizes - 300 and 500, executed for a maximum of 50 generations. We examined the percentage of successful executions, individuals processed on average in an execution, and individuals processed on average in successful executions. As the primitive sets became increasingly general, GP became less able to use them to compose a solution - even when it was permitted to search through twice as many candidate solutions. On the most specialized primitive set its rate of success was 93.3%. On the next two graduated primitives sets, its rate of success fell to 23.3% and 0%. When the population size or maximum generations were doubled, GP could still not find a solution among the most general primitive set. The evidence indicates that GP is not a hierarchical search process. It was unable to consistently identify and promote critical building blocks that we knew existed and that could be exploited to construct efficient solutions.

We suggested that a number of complicated factors may underlie GP's inability to proceed in the manner of a hierarchical process.

- GP may not discover a subtask because the group of primitives specifying it do not consistently perform the subtask function in all program contexts.

- a subprogram that does consistently accomplish a subtask may be disrupted by GP crossover despite selection promoting programs which contain it. In GP this is possible because a subprogram's disruption likelihood depends not just upon its structure (a subprogram may not be a subtree of a GP program's parse tree) but also upon its embedding in a program.

- the bias of non-leaf selection over leaf selection and the maximum tree height parameter may introduce non-linear complications to GP that interfere with the promotion of fitter than average subtasks that have a low probability of disruption despite favourable treatment by GP crossover and selection.

GP is only hierarchical in the superficial sense that it exploits the hierarchical representation of a program as a tree and sometimes discovers hierarchical programs. GP is most often successful if a solution does not require hierarchical subtask identification and assembly or if it uses "high level" or complex primitives that directly encode subtasks. Complex primitives, because they are encapsulated, are protected

from disruption and thus protect subtask behaviour allowing GP to simply discover a correct combination of subtasks into the problem task.

We argued for improving hierarchy in GP because it offers efficient computational effort and principled organization in solutions. We traced the difficulties of current approaches to a number of sources:

- the need for a long time scale and large population can not be directly met because of the rate of program growth and computational expense
- multiple levels in a hierarchy most likely require distinct time scales and selection forces which may be difficult to concurrently evolve and integrate
- possible dependence upon a centralized, external memory is implausible for an evolution based model

We stated that claims of GP's power and success must be qualified in terms of the programming language subset it uses. GP's success on one problem should be assessed according to the level of general functionality its primitive set meets. If an "evolutionary pathway" explaining the derivation of specialized primitives can be demonstrated, the importance of general functionality is justifiably reduced.

When GP is compared to other weak methods, it fares well because a fair comparison shows that, for a given problem, usually the same amount of domain knowledge is exploited by all paradigms.

Given that GP is not a true hierarchical process, it remains necessary to explain GP's success aside from designer oriented factors such as primitive selection, test suite design and fitness function definition. Several design decisions in making GP a GA specialized to solve program discovery play a critical role in its success. GP manipulates programs directly, searches a space of solutions which vary in length and structure, and trades off syntactic and behavioural structural correspondence among programs for expressive flexibility.

In Chapter 4 we focused on the supposition that a GP equivalent of the GA Building Block Hypothesis could be assumed to operate in GP because of GP is a specialized GA. Because it is a GA, GP uses genetic exchange within the population (crossover) and fitness-based selection but its operators and representation choice make it worthwhile to formulate a specific GP theory along the lines of existing GA theory.

We formulated a precise schema definition in GP that would be useful in formalizing a description of GP search. We stayed close to the spirit of a GA schema definition for fixed length bit strings to permit a description of the crossover operator's behaviour to be incorporated into the recurrence relation that counts the schema instances each generation.

A general definition that does not restrict schema to subtrees of GP programs (it includes incompletely specified S-expressions with wildcards) was chosen. Informally, a GP-schema is an unordered collection of both completely defined S-expressions and incompletely defined S-expressions (i.e., fragments). A fragment is the hierarchical structure (which is not strictly a tree) corresponding to what is left intact by repeated crossovers removing subtrees of a program. The schema definition does not specify exactly how the fragments or S-expressions are linked within an instance but it requires that they must all be matched. We are ultimately interested in counting the expected occurrences of a program pattern. Our GP-Schema definition captures the notion of a program pattern and, due to GP's representation, a program pattern (i.e., GP-schema) can occur more than once in a program.

We defined measures of GP-schema specificity or order and of GP-schema defining length. The notion of order makes it possible to compare the relative sample sizes of schemas, and can be transferred directly from GA theory.

Disruption of a GP-schema instantiation $inst(h, H)$ occurs when a node in h is selected as a crossover point and the swapping of the subtree rooted at the crossover point with a subtree from another program changes h sufficiently so that it no longer instantiates H. We used the upper bound on the probability of disruption under crossover of a GP-schema instantiation to define an estimate of the probability of disruption of a schema.

With these definitions in hand, we derived a lower bound on the growth of the expected number of instances of a GP-schema from one generation to the next. Following GA precedent, we referred to this as the GP Schema Theorem (GPST). The GP Schema Theorem expresses the lower bound on growth in expected instances of a schema in the population from time t to $t + 1$. It has three factors:

1. the expected instances of the schema at time t in a population of n strings,

2. the reproductive factor of the expected instances of a schema contributed by

fitness proportional selection, and

3. the lower bound estimate of whether a schema instance will survive crossover and mutation.

We next proposed a definition of GP building blocks and a GP Building Block Hypothesis (BBH). The definition and hypothesis are interpretations of the GPST and are intended to be fully analogous to the definition of GA building blocks and to the GA BBH.

GP building blocks: GP building blocks are low order, consistently compact GP schemas with consistently above average observed performance that are expected to be sampled at increasing or exponential rates in future generations.

GP Building Block Hypothesis (BBH): The GP BBH states that GP combines building blocks, the low order, compact highly fit partial solutions of past samplings, to compose individuals which, over generations, improve in fitness.

Thus, the source of GP's power, (i.e., when it works), lies in the fact that selection and crossover guide GP towards the evolution of improved solutions by discovering, promoting and combining building blocks.

We proceeded to review the assumptions presupposed by the GP BBH. We summarize them here and refer the reader to Chapter 4 for complete consideration.

1. The GP BBH refers to the combining of schemas yet the GPST, by referring to the expected instances of only one schema, fails to describe the interactions of schemas. In this respect, the GP BBH is not supported by any interpretation of the GPST.

2. The GPST also fails to lend support to the GP BBH because hyperplane competition in GP is not well defined. The lack of a feature-expression orientation in the GP representation (i.e., GP's non-homologous nature) results in an unclear notion of which hyperplanes compete for trial allocation. This inherent lack of clarity concerning hyperplane competition seems to indicate that schema processing may not be the best abstraction with which to analyze GP behaviour.

3. The true dynamics of GP is not estimated by the static fitness of schemas because:

- once the population begins to converge even a little, it becomes impossible to estimate static fitness using the information present in the current population.
- high fitness variance within schemas, even in the initial generation, can cause the estimate and static fitness to become uncorrelated. It intuitively seems that rearranging code or simply inserting a new statement into a program can lead to drastic changes in its fitness. This argues that the fitness variance of a GP-schema's instances may be high.

4. The assumption of expected increasing or exponential trials for building blocks requires certain behaviour to be constant over more than one time step. The GPST does not describe behaviour for more than one time step and it is not the case that the required behaviour is constant.

5. As a schema starts dominating, the margin by which it is fitter than the average fitness of the population decreases. The only thing that enables it to continue growing in fitness is a decrease in its probability of disruption. The problem is that there is no guarantee that this decreases at a rate ensuring positive growth of expected allocation of trials over the same interval.

6. Building blocks may only exist for a time interval of the GP run because the estimated fitness relative to the population fitness and upper bound probability of disruption of a schema vary with time.

7. The BBH assumes that solutions can be arrived at through linear combination of highly fit partial solutions. This is a statement about the problem of program induction rather than GP. There is no basis for assuming that a solution's subcomponents are independent. The BBH is a statement about how GP works only if there is linearity in the solution.

Basically, the GPST (and any similar schema theorem) omits important dependencies from the recurrence and is, thus, bound to oversimplify GP dynamics. In particular, the dynamics of crossover and selection that are of interest last longer than one time step. The BBH also assumes the existence of the same building blocks throughout a run and is not specific about the dynamics of building block discovery,

promotion and combination in the course of a run. Therefore, as with GAs, hypothesizing building block combination requires greater liberty with the interpretation of the Schema Theorem than is justifiable.

After Chapters 3 and 4, we turned our focus to comparing alternative algorithms for program discovery to GP and to improving upon GP.

In Chapter 5, by way of comparing GP, we modified Simulated Annealing (SA) and Stochastic Iterated Hill Climbing (SIHC) by introducing a novel mutation operator called HVL-Mutate. HVL-Mutate meets three criteria:

1. it creates a new program from **current** that is a "small distortion" of it. Some of the structure and character of **current** must be retained and novelty must be introduced through randomly influenced decisions.

2. it produces a directly evaluable program (i.e. syntactically correct) without any ad-hoc repair functions being required.

3. it is capable of generating a mutant that differs in number of primitives (size) and/or hierarchical structure from **current**.

It does so by conveniently exploiting a hierarchical representation for programs as does GP crossover. HVL-Mutate first randomly selects a node in a copy of **current**'s parse tree. This node is called **chosen-node**. HVL-Mutate then flips a fairly biased three-sided coin to decide upon one of three sub-operations to perform: insertion, deletion, or substitution.

- If the sub-operation is substitution, **chosen-node** is directly replaced with another primitive of the primitive set which has an equal number of parameters.

- If the sub-operation is deletion, the largest subtree of the tree rooted at **chosen-node** is promoted to replace it. When more than one subtree of the tree rooted at **chosen- node** is largest, one of the largest is chosen randomly. When a leaf is chosen for deletion, it is replaced by a different randomly chosen leaf.

- If the sub-operation is insertion, a primitive is chosen from the primitive set at random. It will replace **chosen-node** but if it requires primitives the subtree rooted at **chosen-node** will act as its first parameter. All remaining children of the newly inserted node will be randomly drawn from the subset of primitives that require no parameters (i.e. are leaves).

HVL-Mutate's suboperations are demonstrated in Figure 52 on page 147.

The pseudocode for the SIHC algorithm is shown in Figure 50 on page 141. At the start SIHC generates a program called **current** at random and then applies the mutation operator HVL-Mutate to it. HVL-Mutate creates a variant (or "mutant") of **current** named **candidate**. The acceptance criterion of SIHC is: if **candidate** is superior or equal in fitness to **current**, it replaces **current** and the search moves onwards from it. Otherwise another mutation on the original point, **current**, is tried. The maximum number of mutations to generate from the program **current** before abandoning it and choosing a new one at random is a parameter of the algorithm called **max-mutations**. The mutation counter for **current** is reset to zero each time **current** is replaced by **candidate**. The algorithm always keeps track of the fittest program. When a maximum number of programs (**candidates-processed** = **limit**) have been generated and each tested for fitness or a perfect solution has been found, the algorithm terminates.

The pseudocode for the SA algorithm is shown in Figure 51 on page 144. The SA algorithm must be supplied *a priori* with its initial temperature T_0, final temperature T_f, 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.

We tested the new versions of SA and SIHC with the thesis suite (**6-Mult**, **11-Mult**, **Sort-A**, **Sort-B**, and **Block Stacking** (**BS**)). The three algorithms were compared by run when each execution of a run was allowed to process a maximum of 25500 candidate solutions or individuals. The bases of comparison were:

- rate of success (successful executions per run)
- average number of individuals processed in an execution
- average number of individuals processed in a successful execution
- average size and height of parse trees of successful programs

All problems could be solved by SA. All problems, except `11-Mult` could be solved by SIHC. Tables 39 to 43 show complete data of all comparisons. Overall, the experimental results of our comparison indicated that adaptive mutation or varying length programs and a degree of localized search are useful for program discovery.

`6-Mult`	Rate of Success (%)	Fittest Individual (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	79.5 (40.3)	98.0 (4.5)	55.7 (27.8)	48.4
SIHC, max-mu = 500	40.0 (49.0)	96.6 (2.3)	81.8	61.3
SIHC, max-mu = 10K	76.7 (50.4)	98.8 (2.8)	61.5	57.7
SA	100.0	100.0	54.0	54.0 (7.2)
XO − SIHC, xo-tries-limit = 100	100.0	100.0	20.56	20.56 (14.2)
XO − SIHC, xo-tries-limit = 1000	96.7 (17.8)	99.5	23.2	23.2
XOSA − Each	36.7 (48.2)	91.6	92.0	80.5
XOSA − One	16.7 (37.3)	64.6	94.8	70.0
XOSA − Ave	3.3 (17.9)	65.7	99.2	87.0
GP + MU − HC, $f = 5$				
$e = 500, g = 5$	93.3 (25.0)	99.5	45.2	43.3 (22.5)
$e = 500, g = 2$	90.0 (30.0)	99.5	32.5	30.6 (5.1)
$e = 100, g = 3$	80.0 (40.0)	99.8	57.0	47.7 (21.3)
GP + XOHC, $f = 5, e = 100, g = 3$				
Best	50.0 (50.0)	96.8 (3.7)	74.0 (29.8)	50.0 (24.9)
Random	60.0 (14.7)	96.9 (4.2)	72.1 (29.6)	32.4 (17.9)
Pop	63.3 (14.5)	95.7 (4.2)	65.3 (32.5)	44.9 (20.8)
GP + XOHC, $f = 5, e = 500, g = 2$				
Best	86.7 (10.2)	99.4 (1.7)	44.7	32.4 (17.9)
Random	93.3 (7.5)	99.7 (1.2)	45.9	44.9 (17.7)
Pop	100.0	100.0	31.3	31.3 (17.4)

Table 39. Comparison of all algorithms on `6-Mult`

The first goal of Chapter 6 was to investigate the basic nature of the GP crossover operator and observe its performance when it replaced HVL-Mutate in SA and SIHC. Relying upon the concept of a *fitness landscape* and a measure we introduced called

11-Mult	Rate of Success (%)	Best of All Execs (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Execs (%)
GP	0	87.6	79.2	100	
SIHC					
max-mu = 500	0	84.4	81.6	100.0	
max-mu = 5000	0	95.31	88.9	100.0	
SA	16.7	100.0	93.0	99.5	98.2
XO − SIHC, xo-tries-limit=100	16.7	100.0	93.6	95.4	72.4
XO − SIHC, xo-tries-limit=5000	16.7	100.0	94.4	93.3	60.0
XOSA − Each	0	75.0	66.5	100.0	
XOSA − One	0	78.9	68.5	100.0	
XOSA − Ave	0	81.3	71.7	100.0	
GP + MU − HC, $f = 5$					
$e = 500, g = 5$	0	91.9	89.6	100.0	
$e = 500, g = 2$	3.3	100.0	91.0	99.5	48.6
GP + XOHC, $f = 5, e = 500, g = 2$					
Best	0	91.9	89.6	100.0	
Random	3.3	100.0	90.2	99.8	97.9
Population	0	91.4	85.0	100.0	

Table 40. Comparison of all algorithms on 11-mult

"syntax correlation", we reviewed GP crossover. Syntax correlation is a measure of similarity in structure, size and primitives between one parent and one offspring. The greater the similarity between a parent and offspring in these respects, the greater the syntax correlation. The syntax correlation of a landscape is the estimated average of all individual syntax correlations. We estimated that the parent-offspring syntax correlation of GP crossover is lower than that of HVL-Mutate. Consequently, among the class of fitness landscapes created by GP crossover, there are plausibly some landscapes that are amenable to search by SA or SIHC. Therefore, we designed two novel versions of SIHC and SA - Crossover Hill Climbing (XO-SIHC) and Crossover Simulated Annealing (XOSA) which took advantage of GP crossover by substituting it for HVL-Mutate.

Sort-A	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	80.0 (40.0)	49.7 (18.1)	51.3 (35.0)	39.1 (26.6)
SIHC, max-mu = 100	46.7 (49.9)	72.6 (40.7)	72.2	37.5 (27.6)
SA	83.3 (37.9)	87.4 (28.3)	62.2 (31.0)	54.6 (28.5)
XO − SIHC, xo-tries-limit= 50	10.0 (30.0)	57.8	92.3	22.7 (22.5)
XOSA − Each	70.0 (45.8)	80.0	56.8	55.3 (25.5)
XOSA − One	93.3 (25.0)	98.0	25.1	22.5 (22.5)
XOSA − Ave	66.7 (47.1)	85.2	67.8	52.0 (31.5)
GP + MU − HC, $f = 5$				
$e = 500, g = 2$	93.3 (25.0)	94.9	32.5 (29.8)	11.3 (20.0)
$e = 100, g = 3$	100.0	100.0	42.0	42.0 (26.9)
GP + XOHC, Pop, $f = 5$				
$e = 500, g = 2$	70.0 (45.8)	88.5 (20.3)	58.4 (31.2)	43.4 (18.2)
$e = 100, g = 3$	80.0 (40.0)	89.4 (21.0)	50.6 (34.3)	39.6 (26.6)
GP + XOHC,Random, $f = 5$				
$e = 100, g = 3$	90.0 (30.0)	58.3 (9.6)	42.5 (29.8)	35.7 (25.5)

Table 41. Comparison of all algorithms on Sort-A

Crossover hill climbing was first described to us by Terry Jones [50, 49]. The basic design of crossover hill climbing still uses better or equal fitness as the move acceptance criterion of the search but substitutes GP crossover as the move operator. The candidate solution is, therefore, generated from one `current` solution and a random mate using a version of GP crossover that is a simple two-parent to one-child function. The algorithm always maintains the `fittest-overall-solution` (i.e. the fittest point of all points examined) and a `current-solution`. At the outset, a mate for `current-solution` is randomly generated. For some number of attempts, `mate-xo-tries-limit`, offspring are generated via crossover from the pair of `current-solution` and mate. If an attempt yields an offspring that is accepted, the offspring replaces `current-solution` and the process repeats with the number of crossover attempts reset to zero and the same mate. If the number of crossover

attempts reaches `mate-xo-tries-limit` without an offspring being accepted, a new mate is chosen for `current-solution`. The number of times the `current-solution` is used, `xo-tries-limit`, is also a parameter of the algorithm. After `xo-tries-limit` crossovers, `current-solution` is discarded and a new one randomly generated. After a fixed number of fitness evaluations or when a perfect solution is found the algorithm terminates and returns the fitness of `fittest-overall-solution`.

Sort-B	Rate of Success (%)	Fittest Ind (%)	Inds Proc'd (%)	Inds Proc'd in Succ Exec (%)
GP	67.7 (47.1)	85.3 (22.6)	60.4 (37.5)	40.6 (28.2)
SIHC, max-mu = 50	60.0	79.1 (31.3)	62.5	37.5 (13.4)
SA	88.3 (37.3)	91.2 (19.7)	64.0 (30.5)	56.6 (28.5)
XO − SIHC, xo-tries-limit= 50	40.0 (49.0)	82.0 (17.5)	81.4	53.6 (31.1)
XOSA − Each	80.0	89.4	58.9	48.75 (25.1)
XOSA − One	83.3	91.2	48.4	38.3 (30.9)
XOSA − Ave	90.0 (30.0)	97.6	65.3	61.5 (17.9)
GP + MU − HC, $f = 5$				
$e = 500, g = 2$	93.3 (25.0)	96.6	32.3 (30.0)	30.9 (30.0)
$e = 100, g = 3$	100.0	100.0	40.3	40.3 (27.2)
GP + XOHC, Pop, $f = 5$				
$e = 500, g = 2$	66.7 (47.1)	88.5 (20.3)	58.8 (31.0)	44.3 (18.4)
$e = 100, g = 3$	80.0 (40.0)	89.4 (21.0)	50.8 (34.0)	40.0 (26.2))
GP + XOHC,Random, $f = 5$				
$e = 100, g = 3$	90.0 (30.0)	97.1 (10.5)	43.2 (29.3)	36.8 (25.1)

Table 42. Comparison of all algorithms on `Sort-B`

The XOSA algorithm uses GP crossover to generate candidate points from two `current` solutions and the SA acceptance criterion. It has one new parameter: `xo-tries-limit`. Every `xo-tries-limit` crossovers, the weakest current solution is replaced with a random program. The acceptance criterion is revised to use 2 parameters instead of global variables. The formal parameters are: `current-fitness` and `candidate-fitness`. The predicate uses the current temperature, a calculated

fitness differential and a random number generator to indicate whether the candidate state of the system should be accepted.

There are three different versions of XOSA: XOSA-Average, XOSA-One, and XOSA-Each, which differ in terms of what values are passed as arguments to the acceptance criterion predicate and in terms of which current solution is replaced if acceptance is indicated.

XOSA-Average: the actual parameter for `current-fitness` is the average fitness of the two current solutions. The actual parameter for `candidate-fitness` is the average fitness of two candidate solutions derived from twice crossing over the current solutions. If acceptance is indicated, both the candidate solutions replace the current solutions. Since XOSA-Average uses the SA component for each pair of fitness evaluations, the SA component is adjusted to use half as many steps in the cooling schedule.

XOSA-One: only one child is generated, via crossover, from the current solutions. The fitness of the weaker parent is the value for `current-fitness` and the fitness of the child is the value for `candidate-fitness`. The weakest parent is replaced by the child, if acceptance is indicated.

XOSA-Each: a 2-parent to 2-children crossover function is used with the option of accepting one child or both. If the weakest current solution can be replaced by the fittest child, this is done and then an attempt is made to exchange the fitter current solution for the weakest child. Otherwise, an attempt is made to exchange the weakest parent for the weakest child.

We assessed the performance of these algorithms on the thesis problem suite. The results are included in the data of Tables 39 to 43.

XOHC is definitely a search algorithm worth considering for program discovery problems, even given the small number of problems in our test suite. It outperforms or equals the best of the other algorithms on `6-Mult`, `11-Mult` and `Block Stacking`. Its performance on Sorting was, however, poor relative to the other algorithms.

The XOSA results were not as uniformly good. This concurred with our earlier conjecture that XOSA may not work because, at low temperatures, the GP crossover operator still generates offspring that are quite syntactically different from their parents.

Block Stacking	Prob of Success (%)	Fittest Individual (%)	Evals Used (%)	Evals Used in Succ Run (%)
GP	76.7 (16.3)	87.7 (31.9)	43.7 (32.1)	35.1 (26.6)
SIHC, max-mu=100	94.3 (23.2)	94.9	34.5	30.5 (24.0)
SA	100.0	100.0	28.5 (22.2)	28.5 (22.2)
XO − SIHC, xo-tries-limit=250	100.0	100.0	2.6	2.6 (2.3)
XOSA − Each	70.0 (45.8)	87.7	75.1	64.7 (26.5)
XOSA − One	96.7 (17.8)	98.4	28.3	25.8 (27.1)
XOSA − Ave	60.0 (49.0)	84.5	88.4	80.8 (18.96)
GP + MU − HC, $f = 5$				
$e = 500, g = 2$	100.0	100.0	5.3	5.3 (3.3)
$e = 100, g = 3$	100.0	100.0	12.5	12.5 (10.9)
GP + XOHC, Pop, $f = 5$				
$e = 500, g = 2$	100.0	100.0	5.3	5.3 (3.3)
$e = 100, g = 3$	100.0	100.0	13.1	13.1 (11.8)
GP + XOHC, Random, $f = 5$				
$e = 100, g = 3$	100.0	100.0	10.6	10.6 (6.3)

Table 43. Comparison of all algorithms on Block Stacking

On this problem suite it was impossible to claim that one version of XOSA was superior overall to the others. On each problem, except Sort-B, there was significant difference among the three versions (Average, Each, One). While Average was never solely best, Best and Each exchanged rankings on different problems.

Overall, among these XO-SIHC and SA algorithms on any given problem, neither both algorithms with the same operator, nor, both operators with the same algorithm had correlated performance. We observed that there was a consistent relationship between XOSA and XO-SIHC in one respect: whenever XO-SIHC performed well, XOSA did not, and, whenever XOSA performed well, XO-SIHC did not.

A second goal of Chapter 6 was to improve GP. To do so, we integrated a local search component into GP to complement its global search power. We implemented two major types of hybridized algorithm: one uses mutation-based stochastic iterated hill climbing, "GP+MU-HC" and the other uses GP crossover stochastic iterated hill

climbing.

Every g generations, the f fittest individuals in the population are used as the starting points of a hill climbing search. Each hill climb processes e candidate solutions. The best individual from each hill climb is placed in the next generation and then the remaining individuals of the population for that generation are generated standardly (i.e. with crossover or direct reproduction). For an execution the move operator of the hill climb is either entirely GP crossover or entirely HVL-Mutate.

The obvious qualitative difference between mutation based hill climbing and crossover hill climbing is that mutation introduces totally unselected genetic material while the crossover operator (if it draws a mate from the population at large or from the pool of fittest individuals) replaces swapped out genetic material with material that has undergone selection by surviving through GP's simulated process of evolution. We experimented with 3 versions of GP + XOHC:

Random: Mates are not drawn from the population at all, but are randomly created. This is similar to using the implemented version of XOHC. It allows non-selected material to be combined with selected material but seems to run contrary to exploiting the fact that a population of GP programs has undergone selection.

Best: Mates are drawn from the group of individuals with the highest fitness. This group will vary in size from one to population size (early in an execution). **Best** potentially increases the chances that a crossover recombination will generate a very fit offspring under the assumption that genetic material from the best members is a source of building blocks because it has undergone selection.

Population: Mates are randomly drawn from the population at large. This version is "middle of the road" between **best** and **random**. The population is a more diverse source of genetic material compared to the **best** pool, but, unlike the random option, it has undergone selection.

The hybrid algorithms were tested for viability and their results are summarized in the data of Tables 39 to 43. They indeed result in better performance than GP. At least one version of a hybrid of GP plus Hill Climbing was better than the GP algorithm we compared them to on all problems, in both forms - GP+XOHC, GP+MU-HC, with at least one setting of the parameters. We observed no consistent

significant ordering between the mutation hill climbing option or crossover hill climbing option across the test suite. Prior to using this search strategy, GP had not been superior nor sometimes even on par with either crossover-based or mutation-based versions of SA and SIHC. With hybridization the evolution inspired search model is, at the least, comparable.

Regarding the relative merits of the Random, Best, and Population versions of GP+XOHC, current data is not decisive. Qualitatively, Best may not be explorative enough because it is limited to a mate pool that may be very small. Preliminarily, Best was outperformed on `6-Mult` but comparable on `11-Mult`. Population worked better than Random on `6-Mult` but the results were reversed on `11-Mult`. On Block Stacking and both sorting problems Random and Population were equal. Additional experimentation, beyond the present scope, seems called for.

The final goal of Chapter 6 was to reflect upon the general qualitative similarities in the entire group of algorithms this thesis described. This is important because no algorithm is always superior to the rest.

We analysed the common approach of all algorithms: GP, SA, XOSA, SIHC, XO-SIHC, GP+XOHC, and GP+MU-HC. A general reason for any adaptive search algorithm's success at program discovery is its representation rather than its unique implementation of a search strategy. A successful program discovery algorithm must consider a large space of candidate solutions where program length and structure can vary. Both search operators: GP crossover and HVL-Mutate generate programs that differ in size and structure from their parent(s). They both also, conveniently, generate programs that, while capturing some inherited features and introducing other random ones, are always syntactically correct. A variable length hierarchical representation appears more fundamental to program discovery than any particular search technique. Any adaptive search algorithm that is sufficiently powerful to efficiently search this space is likely to succeed at program discovery.

Furthermore, the algorithms are united by a framework of evolution-based concepts: selection, inheritance and blind variation. This framework supports their success. Each algorithm is a unique design and implementation of the three concepts. While it is obvious how GP fulfills this framework, with XOSA, SA, XO-SIHC and SIHC some discussion is useful:

- **Selection** "Survival of the fittest" in biology is an evolutionary selection process

in which superior individuals survive long enough to reproduce and propagate in greater numbers than inferior ones. In SA or XOSA the search always moves when a candidate program (generated from the current point) is equal or superior in fitness to the current point. In this respect, deterministically the fittest survive. Because SA accepts a candidate program of inferior fitness with a probability dependent upon the system temperature and the difference in the two programs' fitnesses, it has a similarity to GP's roulette-wheel selection which always accords each individual in the population, regardless of fitness, a non-zero chance of being selected as a parent. SIHC is the simplest implementation of selection: the search only moves to candidate programs of equal or superior fitness.

- **Inheritance** In the sexual reproduction of biology, an offspring inherits its genetic material from its parents. The GP crossover operator implements sexually derived inheritance. Whether the crossover creates two "child" programs from a pair of parents or simply one, an offspring inherits from its parent(s) in so far as before any crossover, it is a copy of one parent and, after crossover, it is a combination of both. The HVL-Mutate operator implements asexual inheritance in the respect that it generates a candidate program that retains some of its parent program contents. Therefore, any algorithm making use of either GP crossover or HVL-Mutate, implements inheritance.

- **Blind Variation** In biology, mutations play the role of blindly changing a genome and allowing an offspring to express a genetic feature that is not common to its parents. The GP crossover operator does not have a direct correspondence for mutation in this context. However, it has a "blind" aspect which is the manner in which a crossover point is chosen. Blind crossover point selection provides blind variation in the sense that the place of combination of "genetic information" from the parent programs is not deterministically chosen. HVL-Mutate obviously expresses blind variation.

An evolution-based framework for interpretation is also appropriate for GP alternatives despite the lack of a formalized population concept. The candidate programs at corresponding time points in the different ancestry sequences, represent a population. The members of each ancestry line evolve independently and asynchronously.

This common framework is useful because it indicates:

- the importance of investigating a variety of selection techniques, population sizes and search operators. Such investigation generates useful comparison, prevents general properties from being overlooked and eliminates the risk of prematurely abandoning plausibly superior techniques. It forestalls any premature claims that one program discovery algorithm will always be superior to others.

- that future effort should be devoted to studying the "real" process of evolution and using it as a model for extensions to all adaptive search algorithms. For example, when SA is viewed as an annealing process it appears to have limited extension. However, parallelizations of it and the substitution of new search operators, based on ideas borrowed from evolution, provide it with new potential.

- that it is important to ask how a sub-process or characteristic of any model can be incorporated into each of the algorithms rather than just one of them. Any proposed extensions to one algorithm in the form of an operator or selection strategy should be generalizable to the rest.

7.2. Future Work

Throughout the thesis, we have mentioned potential avenues of future research. We provide a concise summary here:

- Improving hierarchy in GP could pay off in terms of yielding more efficient search that will be sufficiently powerful to use more general level primitive sets. It would also improve the principled organization of solutions. Investigating a GP extended with module encapsulation for longer time scales and with larger populations is possible if parallel computation is exploited. Longer times scales and larger populations need to be supplemented with general controls or selection pressures towards program parsimony but they potentially may assist hierarchy in arising. Or, a new extended GP model which would have co-evolving but distinct levels of selection and crossover for different levels of program hierarchy might improve hierarchy. As well, one could pursue a decen-

tralized memory based mechanism that tracks fitness of small subprograms so that fitter ones can be encapsulated and promoted as building blocks.

- One could approach the problem of choosing an efficient search algorithm by pursuing statistical measures on search operator and fitness function induced fitness landscapes with a focus on program discovery problems. The goal would be to derive a measure that yields an indication of amenability to algorithms while factoring in its own computational expense.

- The ability of any of the alternative program discovery algorithms to scale is as yet uninvestigated. Because HVL-Mutate can easily be extended to work with an ADF representation, SA (and XOSA) can be tested with ADF benchmark problems. One could assess whether, with ADFs, the computational efficiency , and rate of success, as compared to GP or non-ADF SA and non-ADF XOSA are better than that of GP as the problems scale up.

- More extensive investigation of hybridization is in order. It clearly improved upon GP and, perhaps, on larger problems, it may prove better than SA alone. A superior algorithm would adaptively choose the interval, number and length of the local search component by exploiting knowledge about the progress of the search. Furthermore, it would be nice to further resolve the relative value among the three crossover hill climbing GP hybrids we designed.

- In terms of theory, many difficult questions remain open. A schema-based approach did not prove very useful in formalizing GP search. Perhaps approaches based upon Statistical Mechanics or Markov models that account for population distribution will be fruitful.

- As we stated previously, many new extensions for these algorithms should be tried using the process of evolution as an inspirational model. The extensions should be assessed as representational or search oriented in nature and this will serve as a useful guide in figuring out how they can be generalized across the algorithms.

7.3. Final Remarks

In closing, it is appropriate to consider program discovery from a broader perspective where it becomes one specific means of investigating rich behaviour expressed in terms of computation.

Our ultimate goal is to design a computer-based software system that generates sophisticated behaviour while it operates in an environment that defies *a priori* formal acquisition and formulation of task and environment knowledge. As such, the system can only be initially provided with simple fundamental operators and operands that it must creatively combine to make it capable of interacting with its task environment. It is required to learn and to be flexible enough to react robustly and correctly. It must rely upon a robust, knowledge-starved mechanism that promotes complex structures composed of primitive elements without preordained, centralized guidance.

Using computer programs as the testbed for such behaviour is ideal because programming languages have simple fundamental elements (e.g. in the form of statements, built-in functions and data structures) that can be combined to exhibit sophisticated behaviour. The amount of knowledge of the environment that is provided to the system can be controlled and issues of representing that knowledge can be examined. Using computation as a behavioural medium makes it possible to examine the artifacts of a system, evaluate them and even compare them to aspects of human reasoning.

Evolution is clearly an ideal process to model given the long range goal we have stated. It is the most successful example of a process in which sophistication arises from simple elements with decentralized guidance. Evolution provides an abundant source of proliferating ideas for how to extend the power of adaptive algorithms. These ideas are supported by biological justification. For example, junk genes, complex genes, gene duplication, coevolving populations, coevolving environment and population, etc. are biological phenomena which may be fruitful to model in computation.

To us, this indicates that future research should continue to explore evolution-based learning models that work with computation as a medium. However, it may prove more insightful to diverge from the strict framework of program discovery. It is necessary to remember that program discovery is simply one example of a compu-

tation problem that could be pursued. In fact, it has limited suitability for pursuing longer range goals because it sets up an optimization problem framework. It defines success as reaching an end-point as efficiently as possible with a strict, stationary set of test cases for a fitness criterion. Alternative goals are as worthy. For example, it would be valuable to study a system where the fitness criteria of programs were incrementally made more demanding as the system behaviour improved. This might push the system to greater levels of performance. The goal of the behaviour would focus upon the progress of successive generations rather than whether an ultimate capability is achieved. As another example, the goal of a system might be to discover programs that complement other programs in the population so that collectively the population can always supply one useful program even when the environment is non-stationary.

The point is that the broader goal of computational discovery will become more important than program discovery. Future research into computational discovery based upon evolution inspired algorithms promises profitable insights that will assist us in achieving our grand goal.

REFERENCES

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*.. Wiley, New York, NY, 1989.

[2] L. Altenberg. The evolution of evolvability in genetic programming. In K. E. Kinnear Jr., editor, *Advances in Genetic Programming*. MIT Press, 1994.

[3] L. Altenberg. The schema theorem and Price's theorem. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms*, volume 3, San Mateo, CA, 1995. Morgan Kaufmann.

[4] D. Andre. Learning and upgrading rules for an OCR system using genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, 1994.

[5] P. J. Angeline. *Evolutionary Algorithms and Emergent Intelligence*. PhD thesis, Ohio State University, 1993.

[6] P. J. Angeline. Genetic programming and emergent intelligence. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 4. MIT Press, 1994.

[7] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. In C. G. Langton, editor, *Artificial Life III, SFI Studies in the Sciences of Complexity*, volume XVII. Addison-Wesley, 1991.

[8] G. Guiho A.W. Biermann and Y. Kodratoff. An overview of automatic program construction techniques. In G. Guiho A.W. Biermann and Y. Kodratoff, editors, *Advances in Automatic Programming*, chapter 1, pages 3–30. Macmillan, New York, NY, 1988.

[9] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9, San Mateo, CA, 1991. Morgan Kaufmann.

[10] T. Bäck and H.-P. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–24, 1993.

[11] A. Barr and E. Feigenbaum. *Handbook of Artificial Intelligence. Ch. 10: Automatic Programming.* William Kauffman, Los Altos, 1982.

[12] R. K. Belew and L. B. Booker, editors. *Proceedings of the Fourth International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA, 1991.

[13] R. K. Belew, J. McInery, and N. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. In C. G. Langton, editor, *Artificial Life II, SFI Studies in the Sciences of Complexity*, volume X. Addison-Wesley, 1991.

[14] D. T. Campbell. On the evolutionary theory of knowledge. In Paul Levinson, editor, *In Pursuit of Truth: essays on the philosophy of Karl Popper on the occasion of his 80th birthday*, chapter 23. Humanities Press, 1982.

[15] H. Chen and N. Flann. Parallel simulated annealing and genetic algorithms: A space of hybrid methods. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving From Nature - PPSN III, volume 866 of Lecture Notes in Computer Science*, Berlin, 1994. Springer-Verlag.

[16] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 183–187, Hillsdale, NJ, 24–26 July 1985. Carnegie Mellon University, Lawrence Erlbaum.

[17] Y. Davidor, H.-P. Schwefel, and R. Männer, editors. *Parallel Problem Solving from Nature — Proceedings 3rd Workshop PPSN II.* Springer Verlag, Berlin, Germany, 1994.

[18] L. D. Davis, editor. *Genetic Algorithms and Simulated Annealing.* Research Notes in Artificial Intelligence. Pitman Publishing and Morgan Kauffmann, Los Altos, CA, 1987.

[19] L. D. Davis, editor. *The Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[20] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, 1975. Dissertation Abstracts International 36(10), 5410B. (University Microfilms No. 76-9381).

[21] P. de Souza and S. Talukdar. Genetic algorithms in asynchronous teams. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann.

[22] K. DeJong. Genetic algorithms are not function optimizers. In L. D. Whitley, editor, *Foundations of Genetic Algorithms*, San Mateo, CA, 1993. Morgan Kaufmann.

[23] P. D'haeseleer. Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261. IEEE Press, 1994.

[24] B. D. Dunay, F. E. Petry, and W. P Buckles. Regular language induction with genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*. IEEE Press, 1994.

[25] R.E. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3):189–208, 1971.

[26] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence Through Simulated Evolution.* John Wiley, New York, 1966.

[27] S. Forrest, editor. *Proceedings of the Fifth International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA, 1993.

[28] S. Forrest and M. Mitchell. The performance of genetic algorithms on Walsh polynomials: Some anomalous results and their explanation. In R. K. Belew and L. B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann.

[29] G. J. Friedman. Digital simulation of an evolutionary process. *General Systems Yearbook*, 4:171–184, 1959.

[30] C. Fujiki and J. Dickinson. Using the genetic algorithm to generate Lisp source code to solve the Prisoner's dilemma. In J. J. Grefenstette, editor, *Genetic*

Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms, Hillsdale, NJ, 1987. Lawrence Erlbaum Associates.

[31] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[32] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[33] D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, volume 1, pages 69–93, San Mateo, CA, 1991. Morgan Kaufmann.

[34] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis and first results. *Complex Systems*, 4:415–444, 1989.

[35] D. E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 154–159. Lawrence Erlbaum, Hillsdale, NJ, 24–26 July 1985.

[36] J. J. Grefenstette, editor. *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Carnegie-Mellon University, Pittsburgh, PA, 1985.

[37] J. J. Grefenstette, editor. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[38] J. J. Grefenstette. Deception considered harmful. In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, San Mateo, CA, 1993. Morgan Kauffman.

[39] J. J. Grefenstette and J. E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989. Morgan Kaufmann.

[40] J.J. Grefenstette. A system for learning control strategies with genetic algorithms. In J. David Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 183-190, San Mateo, CA, June 4-7 1989. Morgan Kaufmann.

[41] S. Handley. Automatic learning of a detector for alpha-helices in protein sequences via genetic programming. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 271-278. Morgan Kaufmann, 1993.

[42] S. Handley. Automated learning of a detector for the cores of a-helices in protein sequences via genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 474-479. IEEE Press, 1994.

[43] S. A. Harp and T. Samad. Genetic synthesis of neural network architecture. In L. D. Davis, editor, *Handbook of Genetic Algorithms*, pages 202-221. Van Nostrand Reinhold, 1991.

[44] I. Harvey. The puzzle of the persistent question marks: A case study of genetic drift. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 15–22, San Mateo, CA, 1993. Morgan Kaufmann.

[45] W. D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42:228–234, 1990.

[46] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, 1992. Second edition (First edition, 1975).

[47] H. Iba, H. de Garis, and T. Sato. Genetic programming using a minimum description length principle. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 12, pages 265-284. MIT Press, 1994.

[48] H. Iba, T. Karita, H. de Garis, and T. Sato. System identification using structured genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 279–286. Morgan Kaufmann, 1993.

[49] T. C. Jones. Crossover, macromutation, and population-based search. In L. J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, 1995. (to appear).

[50] T. C. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*. PhD thesis, University of New Mexico, Albuquerque, NM, March 1995.

[51] K. A. De Jong. Generation gaps revisited. In L. D. Whitley, editor, *Foundations of Genetic Algorithms*, San Mateo, CA, 1993. Morgan Kaufmann.

[52] A. Juels and M. Wattenberg. Stochastic hill climbing as baseline methods for evaluating genetic algorithms. Technical report, University of California, Berkely, September 1994.

[53] M. J. Keith and M. C. Martin. Genetic programming in C++: Implementation issues. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 13. MIT Press, 1994.

[54] T. Kido, H. Kitano, and M. Nakashani. A hybrid search for genetic algorithms: Combining genetic algorithms, tabu search, and simulated annealing. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, CA, 1993. Morgan Kaufmann.

[55] K. E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2. IEEE Press, 1993.

[56] K. E. Kinnear Jr. Generality and difficulty in genetic programming: Evolving a sort. In *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 287–294. Morgan Kaufmann, 1993.

[57] K. E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6. MIT Press, 1994.

[58] S. Kirkpatrick. Optimization by simulated annealing: quantitative studies. *Journal of Statistical Physics*, 234:975–986, 1984.

[59] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, May 1983.

[60] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89,Detroit, MI*, volume 1, pages 768–774, Palo Alto, CA, 20-25 August 1989. Morgan Kaufmann.

[61] J. R. Koza. A genetic approach to econometric modeling. In *Sixth World Congress of the Econometric Society*, Barcelona, Spain, 1990. 27 August.

[62] J. R. Koza. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical Report STAN-CS-90-1314, Department of Computer Science, Stanford University, Stanford, CA, 1990.

[63] J. R. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI, Herndon, Virginia, USA*, volume 1, pages 819 827. IEEE Computer Society Press, Los Alamitos, CA, USA, 6-9 November 1990.

[64] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Conference on Machine Learning*, pages 768–774, Cambridge, MA, 1990.

[65] J. R. Koza. Evolution and co-evolution of computer programs to control independent-acting agents. In Jean-Arcady Meyer and Stewart W. Wilson, editors, *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, Paris, 24-28, September 1990*, pages 366–375, Cambridge, MA, 1991. The MIT Press.

[66] J. R. Koza. Evolving a computer program to generate random numbers using the genetic programming paradigm. In Rik Belew and Lashon Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algo-*

rithms, pages 37–44, San Mateo, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[67] J. R. Koza. Genetic evolution and co-evolution of computer programs. In Christopher, Taylor, Charles, Langton, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume X of *SFI Studies in the Sciences of Complexity*, pages 603–629. Addison-Wesley, Redwood City, CA, USA, 1991.

[68] J. R. Koza. Genetic evolution and co-evolution of computer programs. In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 603–629, Reading, MA, 1992. Addison-Wesley.

[69] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[70] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, 1994.

[71] J. R. Koza. Recognizing patterns in protein sequences using iteration-performing calculations in genetic programming. In *1994 IEEE World Congress on Computational Intelligence*. IEEE Press, 1994.

[72] J. R. Koza. Scalable learning in genetic programming using automatic function definition. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*. The MIT Press, Cambridge, MA, USA, 1994.

[73] J. R. Koza. Discussion of hill climbing and genetic search. Internet Machine Learning Digest, v7n14, August 1995.

[74] D. H. Kraft, F. E. Petry, W. P. Buckles, and T. Sadasivan. The use of genetic programming to build queries for information retrieval. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 468–473. IEEE Press, 1994.

[75] K. J. Lang. Hill climbing beats genetic search on a boolean circuit synthesis problem of koza's. In A. Prieditis and S. Russell, editors, *Proceedings of the Twelth International Conference on Machine Learning*, pages 340–343, San Mateo, CA, July 9-12 1995. Morgan Kaufmann.

[76] W.B. Langdon. Genetic programming bibliography. Technical report, University College London, 1995.

[77] G. Liepins and M. D. Vose. Deceptiveness and genetic algorithm dynamics. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, volume 1, pages 36–50, San Mateo, CA, 1991. Morgan Kaufmann.

[78] R. Männer and B. Manderick, editors. *Parallel Problem Solving from Nature — Proceedings 2nd Workshop PPSN II.* Elsevier Science, Brussels, Belgium, 1992.

[79] K. Mathias and L. D. Whitley. Genetic operators, the fitness landscape and the traveling salesman problem. In R. Männer and B. Manderick, editors, *Parallel Problem Solving From Nature*, volume 2, pages 219–228, Amsterdam, The Netherlands, 1992. Elsevier Science Publishers B.V.

[80] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[81] M. Mezard, G. Parisi, and M. Virasoro. *Spin Glass Theory and Beyond.* World Scientific, Singapore, 1987.

[82] M. Mitchell, S. Forrest, and J. H. Holland. The royal road for genetic algorithms: Fitness landscapes and GA performance. In F. J. Varela and P. Bourgine, editors, *Proceedings of the First European Conference on Artificial Life. Toward a Practice of Autonomous Systems*, pages 245–254, Cambridge, MA, 11–13 Dec 1992. MIT Press.

[83] M. Mitchell, J. H. Holland, and S. Forrest. When will a genetic algorithm outperform hill climbing? In J. D. Cowan, G. Tesauro, and J. Alspector (editors), *Advances in Neural Information Processing Systems 6.* San Mateo, CA: Morgan Kaufmann.

[84] S. Muggleton. *Inductive Logic Programming.* Academic Press, San Diego, CA, 1992.

[85] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, CA, 1980.

[86] H. Oakley. Two scientific applications of genetic programming: Stack filters and non-linear equation fitting to chaotic data. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 17. MIT Press, 1994.

[87] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In J. J. Grefenstette, editor, *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230, Hillsdale, NJ, 1987. Lawrence Erlbaum.

[88] U.-M. O'Reilly and F. Oppacher. An experimental perspective on genetic programming. In R Manner and B Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 331–340, Brussels, Belgium, 1992. Elsevier Science.

[89] U.-M. O'Reilly and F. Oppacher. Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving From Nature - PPSN III. Volume 866 of Lecture Notes in Computer Science*, pages 397–406, Berlin, 1994. Springer-Verlag.

[90] U.-M. O'Reilly and F. Oppacher. Building block functions to confirm a building block hypothesis for genetic programming. Technical Report 95–02–007, Santa Fe Institute, Santa Fe, NM, February 1995.

[91] W. H. Press. *Numerical Recipes in C.* Cambridge Press, Cambridge, MA, 1992.

[92] N.J. Radcliffe. Formal analysis and random respectful recombination. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann.

[93] L. De Raedt. *Interactive Theory Revision: An Inductive Logic Programming Approach.* Academic Press, San Diego, CA, 1992.

[94] L. De Raedt and M. Bruynooghe. Interactive theory revision. In R. Mihalski and G. Tecuci, editors, *Machine Learning. A Multistrategy Approach*, chapter 9. Morgan Kaufmann, San Mateo, CA, 1994.

[95] I. Rechenberg. *Evolutionsstrategie: Optimierung Techniquer Systeme nach Prinzipien der Biologischen Evolution.* Frommann-Holzboog Verlag, Stuttgart, 1973.

[96] Craig W. Reynolds. An evolved, vision-based behavioral model of coordinated group motion. In Meyer and Wilson, editors, *From Animals to Animats (the proceedings of Simulation of Adaptive Behaviour.* MIT Press, 1992.

[97] Craig W. Reynolds. The difficulty of roving eyes. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 262–267. IEEE Press, June 1994.

[98] C. Rich and R. C. Waters. The programmer's apprentice. In P.H. Winston, editor, *Artificial Intelligence at MIT, Expanding Frontiers*, pages 166–195. Cambridge, MA, 1990. MIT Press.

[99] J. P. Rosca and D. H. Ballard. Learning by adapting representations in genetic programming. in *Proceedings of the 1994 IEEE World Congress on Computational Intelligence.* IEEE Press, 1994.

[100] J. P. Rosca and Dana H. Ballard. Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning.* Morgan Kaufmann, 1994.

[101] E.D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135, 1974.

[102] S. Sankoff and J. B. Kruskal. *Time Warps, String Edits and Macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, Reading,MA, 1983.

[103] J. D. Schaffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms.* Morgan Kaufmann, San Mateo, CA, 1989.

[104] H.-P. Schwefel. *Evolutionsstrategie und numerische Optimierung.* PhD thesis, Technische Universität Berlin, Berlin, 1975.

[105] H.-P. Schwefel and R. Männer, editors. *Parallel Problem Solving from Nature — Proceedings 1st Workshop PPSN I*, volume 496 of *Lecture Notes in Computer Science*. Springer Berlin, Dortmund, Germany, 1990.

[106] C. G. Shaefer. The ARGOT strategy: Adaptive representation genetic optimizer technique. In J. J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 50–55, Hillsdale NJ, 1987. Lawrence Erlbaum Associates.

[107] E. V. Siegel. Competitively evolving decision trees against fixed training cases for natural language processing. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 19. MIT Press, 1994.

[108] H. A. Simon. *Sciences of the Artificial.* MIT Press, Cambridge, MA, 1969.

[109] R. E. Smith. *Special Issue on Classifier Systems in Journal of Evolutionary Computation, volume 2, no. 2.* MIT Press, Cambridge, MA, 1994.

[110] S.J. Smith. Flexible learning of problem solving heuristics through adaptive search. In *Proceedings of 8th International Joint Conference on Artificical Intelligence*, San Mateo, CA, 1983. Morgan Kaufmann.

[111] P. Stadler. Landscapes and their correlation functions. Unpublished manuscript., July 1995.

[112] G. Steele. *Common LISP: the language.* Digital Press, Bedford, MA, 1990.

[113] G.J. Sussman. *A Computer Model of Skill Acquisition.* MIT Press, Cambridge, MA, 1975.

[114] G. Syswerda. A study of reproduction in generational and steady state algorithms. In G.E. Rawlins, editor, *Foundations of Genetic Algorithms*, San Mateo, CA, 1991. Morgan Kaufmann.

[115] W. A. Tackett. Greedy recombination and genetic search on the space of computer programs. In D. Whitley and M. Vose, editors, *Foundations of Genetic Algorithms 3.* Morgan Kaufmann, 1994.

[116] W. A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs.* PhD thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.

[117] W. A. Tackett and A. Carmi. The unique implications of brood selection for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence.* IEEE Press, 1994.

[118] A. Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9. MIT Press, 1994.

[119] U. W. Thonemann. Finding improved simulated annealing schedules with genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence.* IEEE Press, 1994.

[120] M. D. Vose. Modeling simple genetic algorithms. In L. D. Whitley, editor, *Foundations of Genetic Algorithms*, volume 2, pages 63–73, San Mateo, CA, 1993. Morgan Kaufmann.

[121] L. D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116–121, San Mateo, CA, June 4–7 1989. Morgan Kaufmann.

[122] L. D. Whitley. Fundamental principles of deception in genetic search. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, volume 1, pages 221–241, San Mateo, CA, 1991. Morgan Kaufmann.

[123] S. W. Wilson. Classifier systems and the animat problem. *Machine Learning*, 2:199–228, 1987.

[124] M. Wineberg and F. Oppacher. A representation scheme to perform program induction in a canonical genetic algorithm. In Y. Davidor, H.-P. Schwefel, and R. Manner, editors, *Parallel Problem Solving From Nature – PPSN III, volume 866 of Lecture Notes in Computer Science*, pages 292–301, Berlin, 1994. Springer-Verlag.

[125] S. Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. In D.F. Jones, editor, *Proceedings of the Sixth International Congress on Genetics, Vol 1*, pages 356–366, San Mateo, CA, June 4–7 1932. Morgan Kaufmann.

[126] B. T. Zhang and H. Mühlenbein. Synthesis of sigma-pi neural networks by the breeder genetic programming. In *Proceedings of IEEE International Conference on Evolutionary Computation (ICEC-94), World Congress on Computational Intelligence*, pages 318–323. IEEE Computer Society Press, New York, 1994.

END

31-05-96

FIN