# 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) | 96.5 | 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 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 decentralized 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 computation 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.