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	48.4	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)
$XOSA-Ave$	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 e 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. Additional experimentation, 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.