CHAPTER 5

Simulated Annealing and Hill Climbing for Comparison to GP

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

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

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

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

The pseudocode for the SIHC algorithm is shown in Figure 50. At the start SIHC generates a program called `current` at random and then applies the mutation operator HVL-Mutate to it. HVL-Mutate is completely described in Section 5.3. It is sufficient for the description of SIHC to state that HVL-Mutate creates a variant (or "mutant") of `current` named `candidate` in the manner of inheritance and random variation which ensures that `candidate` is syntactically correct and has the possibility of differing from `current` in the number of its primitives (size) or its hierarchical structure. The acceptance criterion of SIHC is: if `candidate` is superior or equal in fitness to `current`, it replaces `current` and the search moves onwards from it. Otherwise another mutation on the original point, `current`, is tried. The maximum number of mutations to generate from the program `current` before abandoning it and choosing a new one at random is a parameter of the algorithm which we call `max-mutations`. Values of 50, 100, 250, 500, 2500, and 10000 for this parameter will be tried. The mutation counter for `current` is reset to zero each time `current` is replaced by `candidate`. The algorithm always keeps track of the fittest program. When a maximum number of programs (`candidates-processed` = `limit`) have been generated and each tested for fitness or a perfect solution has been found, the algorithm terminates. Because of the random restart behaviour controlled by `max-mutations`, this basic hill climbing algorithm is prefixed with "Stochastic Iterated".

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

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

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

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

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

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

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

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

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

MAIN PROGRAM

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

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

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

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

     report best, best-fitness, candidates-processed;

end Stochastic Iterated Hill Climbing Algorithm
```

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

5.2. Simulated Annealing (SA) for Program Discovery Problems

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

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

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

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

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

In the SA algorithm used in Computer Science (typically for combinatorial optimization [58, 59]) the energy state is exchanged with an objective or fitness function which measures how close a candidate solution is to the goal state, the particle arrangement is a candidate solution and the problem is reformulated into one of minimization. A mutation operator is designed to randomly "tweak" the current solution to produce a candidate that is a slight distortion of it. (For example, if a binary representation is used, a mutation operation simply randomly chooses one bit position and flips the bit at that position.) The algorithm has an annealing schedule which dictates how the temperature changes. The Boltzmann constant is incorporated into the value of temperature. Typically the algorithm is *a priori* supplied with an initial and final temperature, and the fraction of candidates to be sampled at each temperature. The temperature is discrete and changed according to a decreasing exponential rate. SA uses exactly the acceptance criterion of the Metropolis algorithm (Equation 5.1): the probability with which a mutant with lower or equal fitness than its parent is accepted decreases with the temperature of the system and depending upon the difference in fitness between them according to a Boltzmann distribution.

```
begin Simulated Annealing Algorithm

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

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

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

MAIN PROGRAM

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

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

     report best, best-fitness, candidates-processed;

end Simulated Annealing Algorithm
```

Figure 51. Pseudocode for Simulated Annealing Algorithm

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

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

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

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

Table 13. T_F and Stepsize for SA Experiments

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

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

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

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

To meet the criteria, like GP, HVL-Mutate conveniently exploits a hierarchical representation for programs. It draws its inspiration from the method for calculating the distance between trees [102]. That approach defines distance as the minimum cost sequence of editing one tree, step by step, to become the other using 3 elementary operations: substitution, insertion and deletion which are designated with costs.

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

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

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

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

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

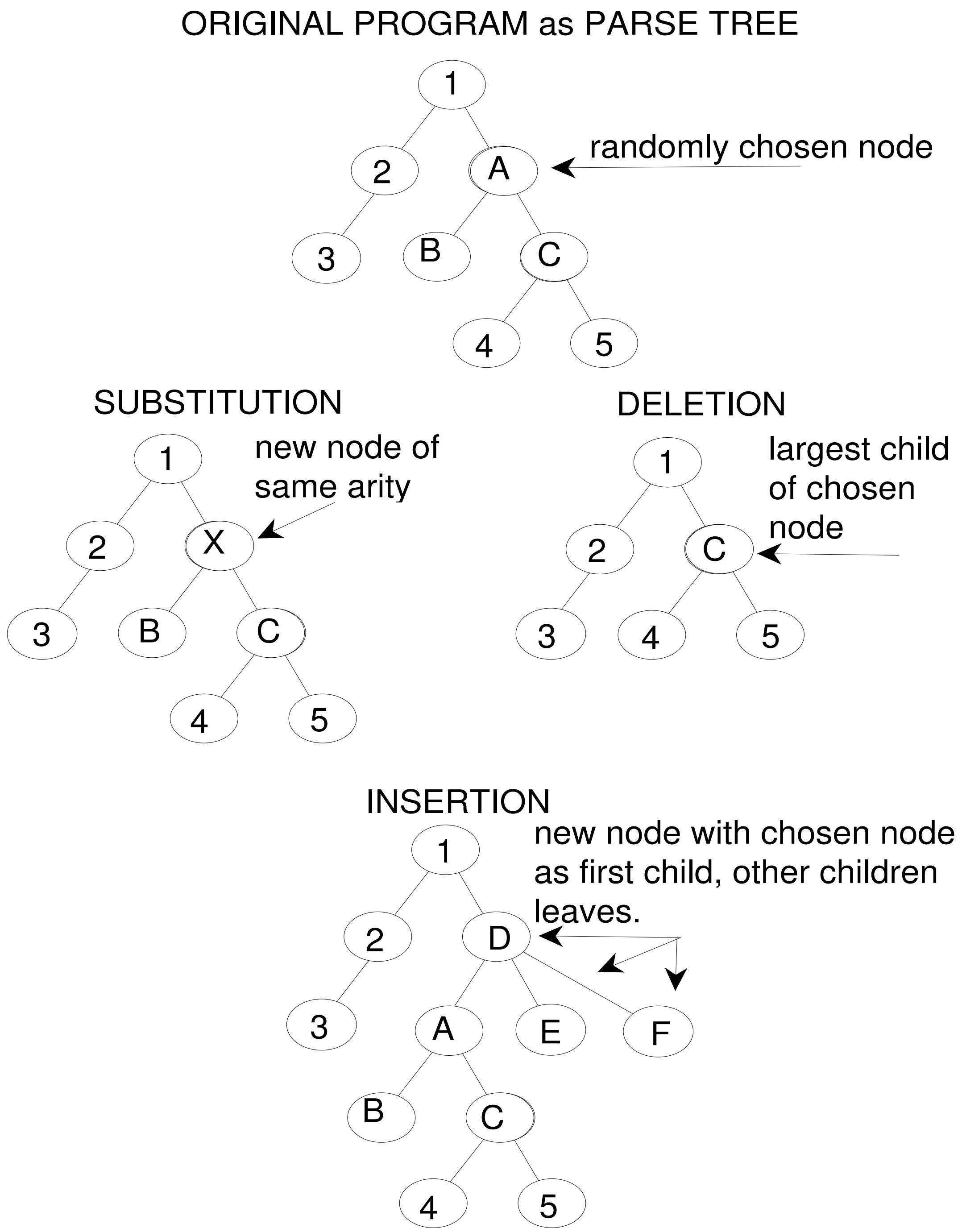

Figure 52. Demonstration of HVL-Mutate: The parse tree of the current program is used. A node is chose at random. A sub-operation (substitution, deletion or insertion) will occur at that node.

5.4. Experimental Approach

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

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

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

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

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

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

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

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

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

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

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

5.5. Experiment Results

5.5.1. 6-Mult Results

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

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

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

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

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

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

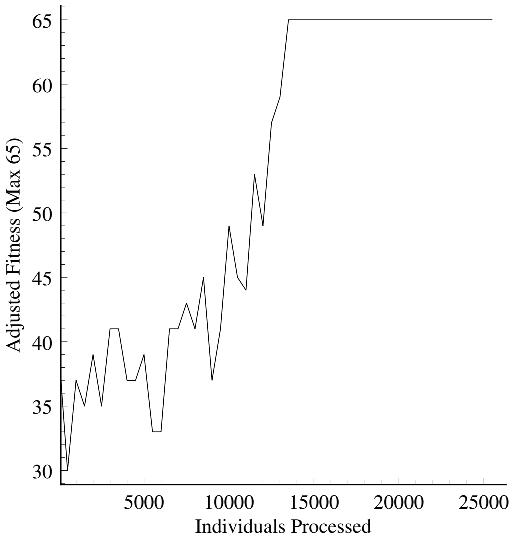

Figure 53. Plot of successful SA Run on 6-Mult, data recorded every 500 mutations.

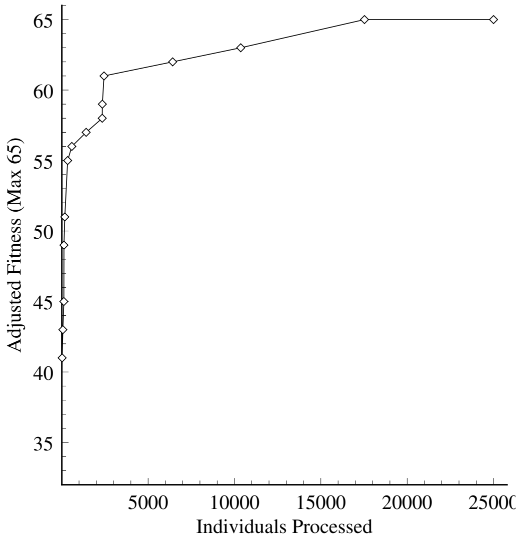

Figure 54. Plot of successful SIHC climb on 6-Mult when `max-mutations` = 5000. Data was recorded at each step, a step is denoted by a diamond.

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

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

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

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

Table 15. 6-Bit Multiplexer: SIHC Data

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

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

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

5.5.2. 11-Mult Results

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

Neither GP or SIHC could solve 11-Mult with 25500 evaluations. The fitness of the best program found by GP was 87.6% of the optimum. When the value of `max-mutations` was 5000, the fittest program SIHC could find was 88.9% of the optimum. The results for SIHC when the maximum mutations parameter was varied from 50 to 10000 were not significantly different. They are displayed in Table 18. Three SIHC climbs are plotted in Figure 56. It is interesting to note that a very simple, greedy heuristic that can be coded in less than one page outperforms a complicated and computationally expensive algorithm such as GP.

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

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

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

Table 18. SIHC and `11-Mult`

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

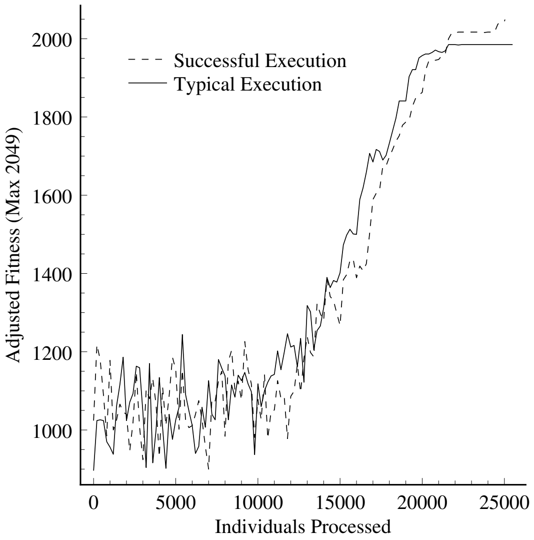

Figure 55. Plot of typical and successful SA executions on 11-Mult, data recorded every 500 mutations.

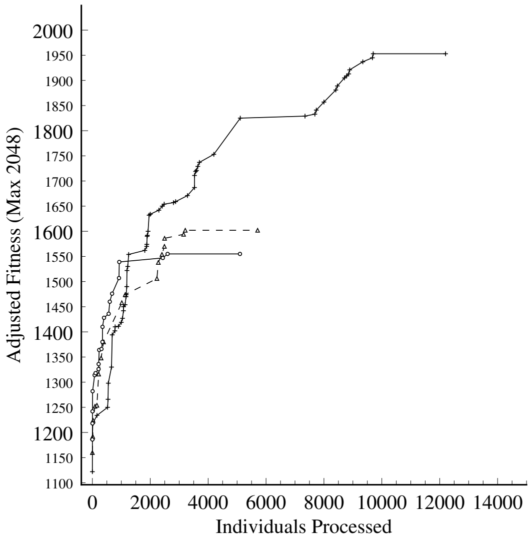

Figure 56. Plot of 3 SIHC climbs on 11-Mult. Data was recorded at each step, a step is denoted by a symbol, `max-mutations` = 2500, all climbs from same execution.

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

5.5.3. Sorting Results

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

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

On both `Sort-A` and `Sort-B` SA processed approximately 63% of the maximum allowed individuals in all executions and about 55% of maximum allowed individuals in successful executions. This is more than GP for both `Sort-A` and `Sort-B`. `Sort-A` processed 51.3% of individuals in all executions and 39.1% in successful executions. `Sort-B` processed 60.4% of individuals in all executions and 40.6% in successful executions. SIHC processed the most individuals: 72.2% of maximum, for all executions but fewer than both GP and SA on successful executions (37.5% vs. 39.1% vs 54.6%). This difference is not statistically significant however.

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

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

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

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

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

Tables 21 and 22 provide the details of SIHC and `Sort-A`. Tables 23 and 24 provide the details of SIHC and `Sort-B`. We discover that it is possible to hill climb to a solution in the fitness landscape regardless of the value of the `max-mutations` parameter. In fact, there is no significant difference in the probability of success for different values of the max-mutations parameter among the settings we tried. From the details of successful climbs one can see that, surprisingly, most climbs take between one and two steps. From this it appears that there are many randomly spaced small equally sloped peaks. This phenomenon was confirmed when the SA executions were examined in detail. Frequently there were sequences of only one or two mutations that produced a fitter program. This data shows a contrast in search space character when it is compared with that of `6-Mult` and `BS`. In `6-Mult` the steps per climb varies from 2.5 to 9.3 and in `BS`, regardless of max-mutations, there are approximately 4 steps per climb. The greedy nature of SIHC seems likely to be responsible for a significantly lower probability of success when compared to GP or SA.

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

Table 21. Sort-A: SIHC Data

Max Mutations	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	1.4 (0.5)	28.6 (18.4)	20.4	28.7 (15.3)
100	1.6 (0.5)	79.4 (32.3)	50.5	37.5 (27.6)
250	1.8 (0.4)	153.6 (91.4)	85.3	44.3 (16.7)
500	2.0 (0.5)	388.3 (255.4)	194.1	31.1 (25.9)

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

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

Table 23. Sort-B: SIHC Data

Max Mutations	Steps per Climb	Evals per Climb	Evals: Step	Evals: Exec (%)
50	1.3 (0.5)	27.2 (12.3)	20.4	37.5 (13.4)
100	1.7 (0.5)	60.7 (58.6)	36.4	51.7 (34.7)
250	1.8 (0.4)	103.6 (65.8)	57.6	43.5 (26.1)
500	1.7 (0.8)	305.2 (248.7)	174.4	52.2 (20.9)

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

5.5.4. Block Stacking Results

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

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

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

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

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

Table 26. Block Stacking: SIHC Data

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

Table 27. Block Stacking: SIHC data for successful executions

5.5.5. Results of Other Literature

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

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

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

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

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

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

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

5.6. Summary

By way of analyzing GP using comparison, we introduced program discovery approaches for two traditional single point based adaptive search algorithms: Simulated Annealing and Stochastic Iterated Hill Climbing. Despite not tuning SA, it was capable of outperforming GP in certain respects on a majority of the problems we experimented with. We observed mixed or comparable differences between operator and search techniques across different problems. This confirms the notion that the suitability of a search technique depends upon the fitness function and primitives chosen for a particular problem since these influence the nature of the search landscape.

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

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

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

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