CHAPTER 3

An Experimental Perspective on Genetic Programming

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

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

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

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

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

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

3.1. Assessing Designer Choices and a Hierarchical Process in GP

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

3.1.1. Test suite and Fitness Function Design Issues

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

Generality and the Test Suite Sample

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

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

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

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

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

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

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

Test suite `Sort-A-Standard` consists of 48 different vectors of 5 different sizes. For the smaller sizes (2-4) every permutation of elements is a test case. Larger sizes are sampled (10 arrays of size 5 and 5 arrays of size 6). Among the test suite are 3 arrays which are initially in sorted order and 47 elements are initially in their correct positions. There are 198 elements in all.

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

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

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

Results

Decreasing the size of the test suite did not change the probability of an execution finding a solution with 100% fitness and this probability was the same for each test suite. The number of individuals processed to find a solution did not differ between the test suites. Concerning generality, as expected, the larger the testsuite, the more capable a perfect solution was in solving the out of training suite. These results are shown in Table 5. The table shows the average number of hits (i.e. arrays perfectly sorted) and average fitness (as a percentage) of the perfect solutions. While the perfect solutions obtained by the small test suite could correctly sort 60 of the arrays and place 50.7% of the elements in their correct place, the perfect solutions of the standard test suite achieved 134.2 hits and averaged a fitness of 90.4%.

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

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

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

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

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

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

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

The Coaching Aspect of Test Suite Composition

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

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

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

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

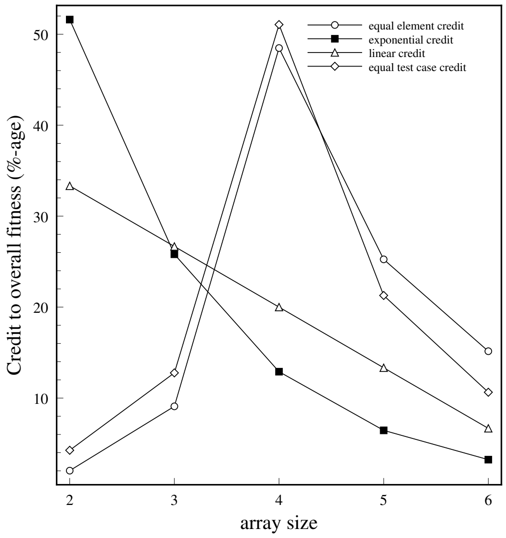

Figure 10. Different credit functions according to array size for `Sort-A`

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

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

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

Exponential Test cases with arrays of size 2 use a weight factor of 2400, size 3 use a weight factor of 800, size 4 use a weight factor of 50, size 5 use a weight factor of 60 and size 6 use a weight factor of 60. As a result, each group of test cases grouped according to array size has a credit factor in the program fitness that decreases by a half as the array size increases. Maximum program fitness is 9300.

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

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

Table 7. The Effect of Fitness Credit Schemes

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

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

3.1.2. Primitive Design Issues

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

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

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

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

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

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

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

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

A. Subtraction	B. Array Reference	C. Exchange

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

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

Table 8. Potential Sort Experiment Primitives

The same issue affects iteration primitives: The primitive

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

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

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

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

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

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

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

Any one of these arguments could be absorbed into the primitive to limit its potential behaviour. For example, `(do-until-Up-with-*j* startValue endValue true-form)` uses the variable `*j*` as implicit `loopVar`. The primitive sets `*j*` to startValue, and it always increments `*j*` at the end of each repetition, using an implicit `varDelta` of 1+. It always tests whether $*j* <=$ `endValue`, with an implicit `loop-test` of <=, to conditionally execute `true-form` .

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

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

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

Deciding upon Scope.

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

These side effects distort the structured behaviour of a program and may also have an impact on the effectiveness of crossover. A decision must be made to either "put up" with this state of affairs, use structure-preserving crossover (see Chapter 2 page 55), or enforce scope by a less transparent means by using global variables in various primitives but not including them in the primitive set. Both the latter choices require the designer to make a decision for GP that constrains expression and GP has no way of reversing the decision to gain more expressive capacity. On the other hand, both latter choices may also constrain the search space in a reasonable manner which offers a better chance for obtaining a solution.

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

3.1.3. Primitive Sets for Assessing Hierarchical Process

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

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

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

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

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

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

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

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

- another algorithm of another GP solution to Block Stacking more efficiently removes only the blocks to the point where the stack is correct and then stacks the remaining blocks correctly. [62]

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

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

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

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

There is an important distinction between hierarchical solutions and hierarchical processes: hierarchical solutions exhibit true hierarchical features (control and structure) while the notion of a hierarchical process refers to a specific manner in which hierarchical solutions arise. Expressed from the perspective of GP, hierarchical control is exemplified by strategic task decomposition and subtask composition and even subtask redundancy. A hierarchical process in GP, if it exists, is a particular capacity in GP that identifies and promotes key primitives so that they become used in higher level composite combinations that perform logical subtasks, and, ultimately, assembles layers of composite subtasks that accomplish the task.

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

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

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

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

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

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

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

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

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

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

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

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

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

Problem Definition: `Sort-TH-0`

Task: *Sorting*

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

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

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

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

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

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

Fitness Function Same as `Sort-A`.

Problem Definition: `Sort-TH-1`

Task: *Sorting*

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

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

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

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

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

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

Fitness Function Same as `Sort-A`.

Problem Definition: `Sort-TH-2`

Task: *Sorting*

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

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

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

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

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

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

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

Fitness Function Same a `Sort-A`.

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

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

Run Parameters :

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

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

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

Table 9. Hierarchical Process in GP: All Problems

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

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

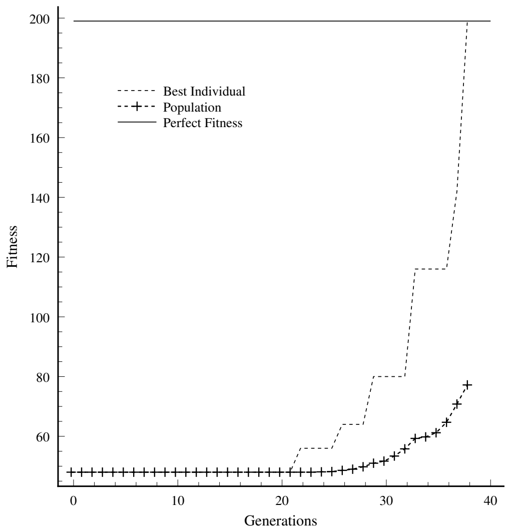

Figure 11. Sort-Th-0: Fitness of best individual and population in successful execution

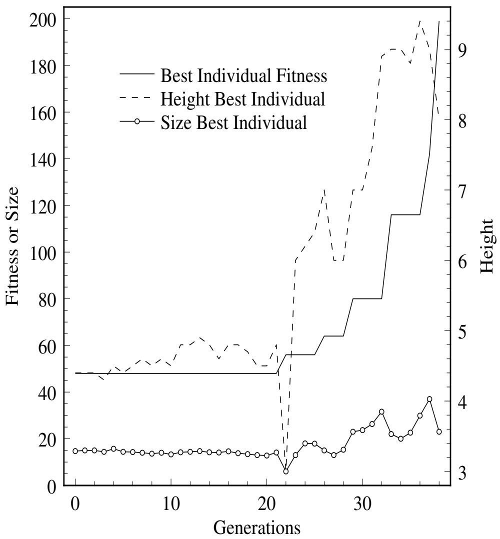

Figure 12. Sort-Th-0: Fitness, height and size of best individual in successful execution

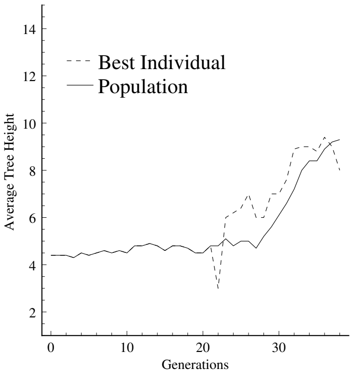

Figure 13. Sort-Th-0: Program height in successful execution

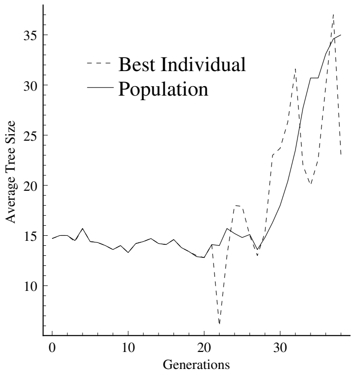

Figure 14. Sort-Th-0: Program size in successful execution

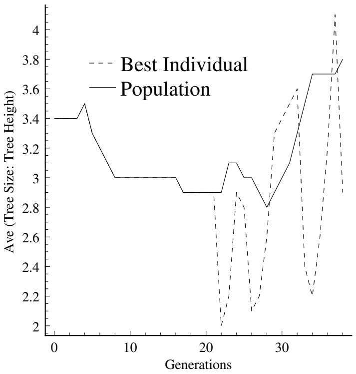

Figure 15. Sort-Th-0: Program size:height in successful execution

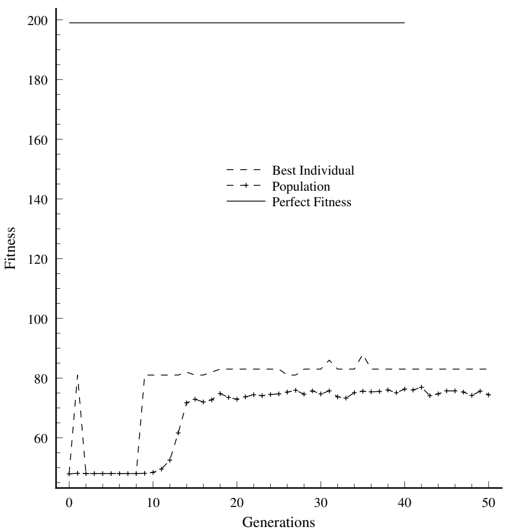

Figure 16. Sort-Th-0: Fitness of best individual and population in unsuccessful execution

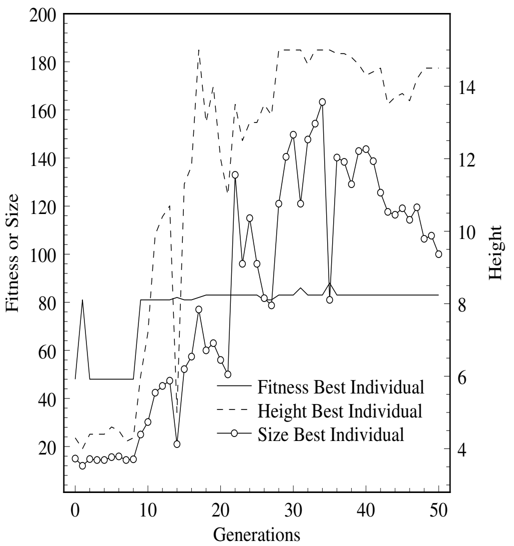

Figure 17. Sort-Th-0: Fitness, height and size of best individual in unsuccessful execution

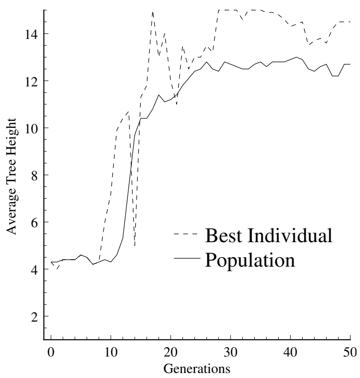

Figure 18. Sort-Th-0: Program height in unsuccessful execution

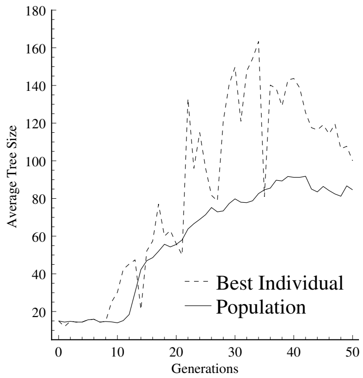

Figure 19. Sort-Th-0: Program size in unsuccessful execution

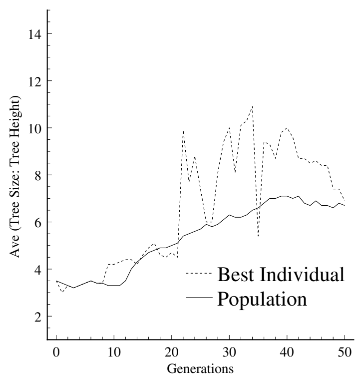

Figure 20. Sort-Th-0: Program size:height in unsuccessful execution

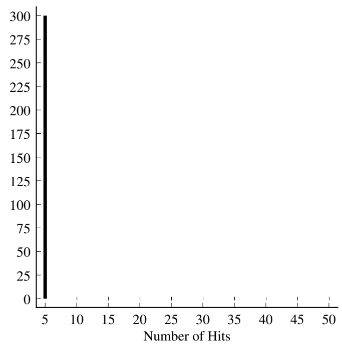

Figure 21. Sort-TH-0: Hits distribution, Generation 0

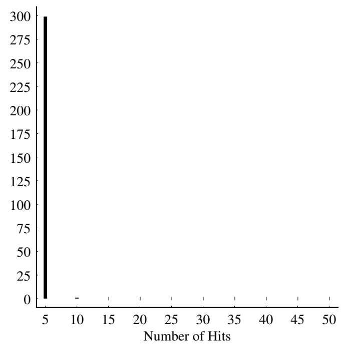

Figure 22. Sort-Th-0: Hits distribution, Generation 22

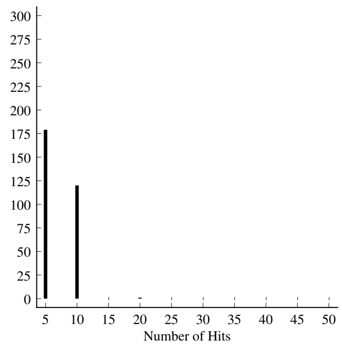

Figure 23. Sort-Th-0: Hits distribution, Generation 33

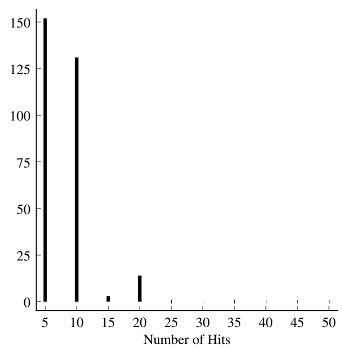

Figure 24. Sort-TH-0: Hits distribution, Generation 36

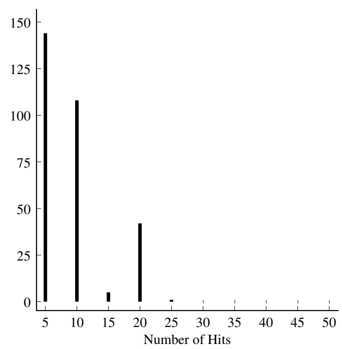

Figure 25. Sort-Th-0: Hits distribution, Generation 37

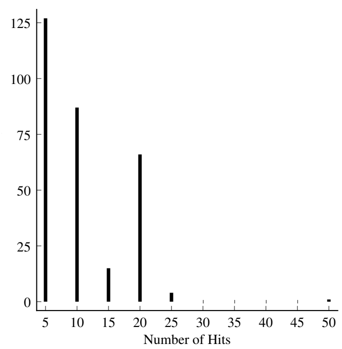

Figure 26. Sort-Th-0: Hits distribution, Generation 38

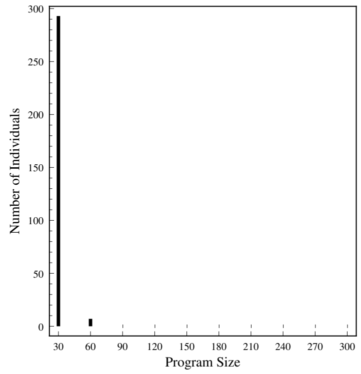

Figure 27. Sort-TH-0: Program size distribution, Generation 0

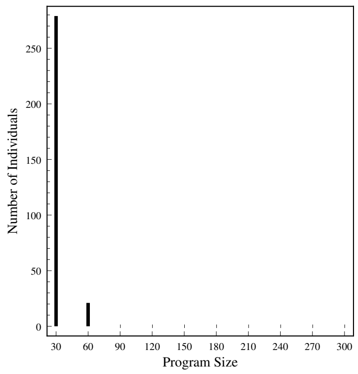

Figure 28. Sort-Th-0: Program size distribution, Generation 19

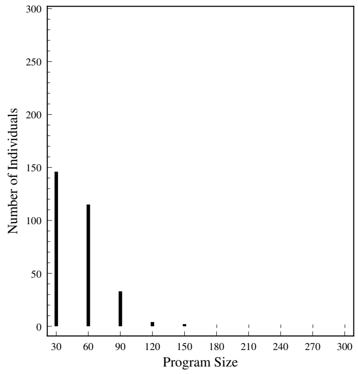

Figure 29. Sort-Th-0: Program size distribution, Generation 38

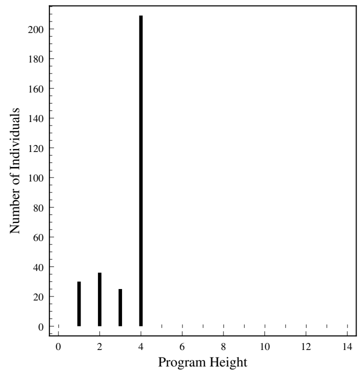

Figure 30. Sort-TH-0: Program height distribution, Generation 0

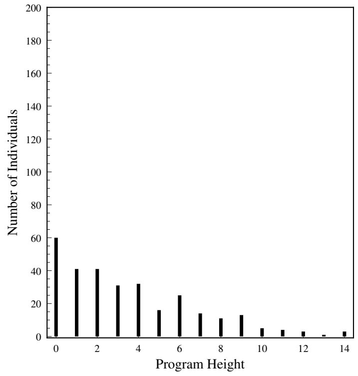

Figure 31. Sort-Th-0: Program height distribution, Generation 19

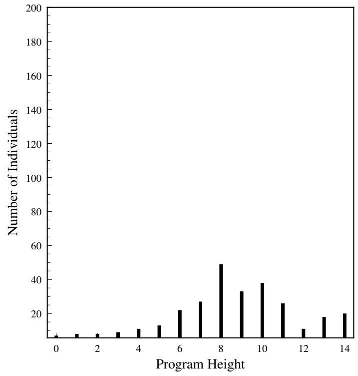

Figure 32. Sort-Th-0: Program height distribution, Generation 38

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

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

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

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

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

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

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

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

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

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

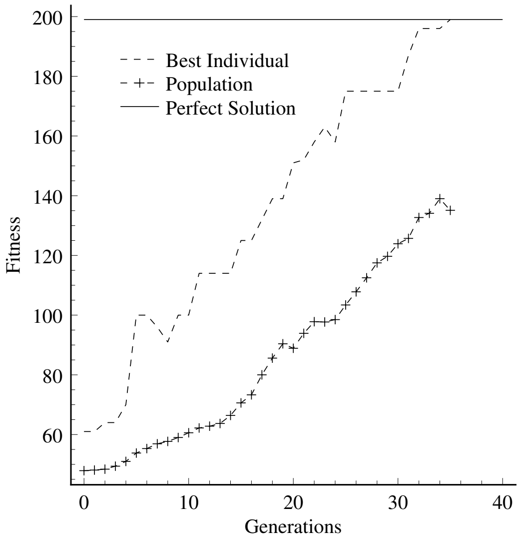

Figure 33. Sort-TH-1: Fitness of best individual and population in successful execution

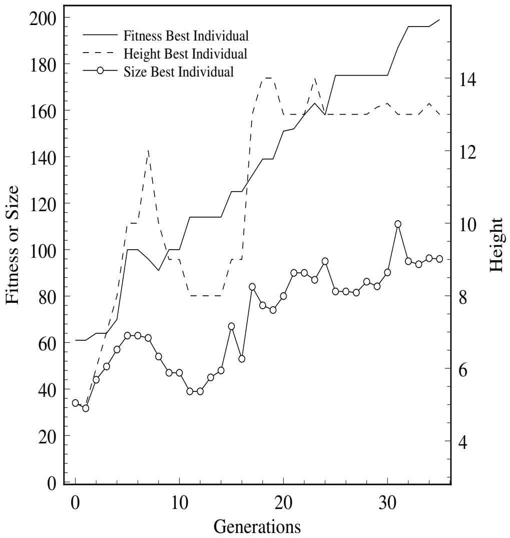

Figure 34. Sort-TH-1: Fitness, height and size of best individual in successful execution

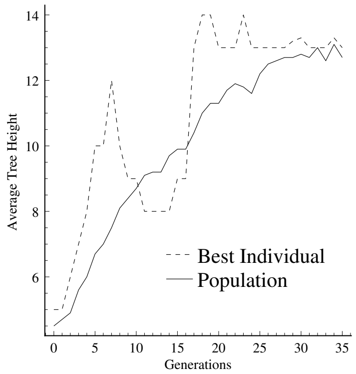

Figure 35. Sort-Th-1: Program height in successful execution

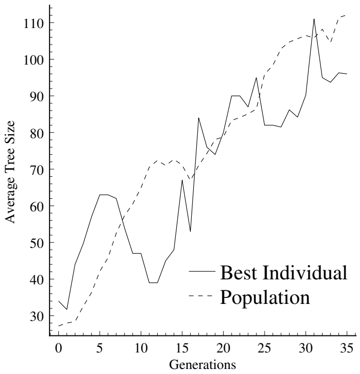

Figure 36. Sort-Th-1: Program size in successful execution

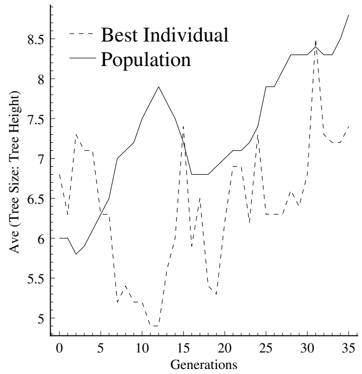

Figure 37. Sort-Th-1: Program size:height in successful execution

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

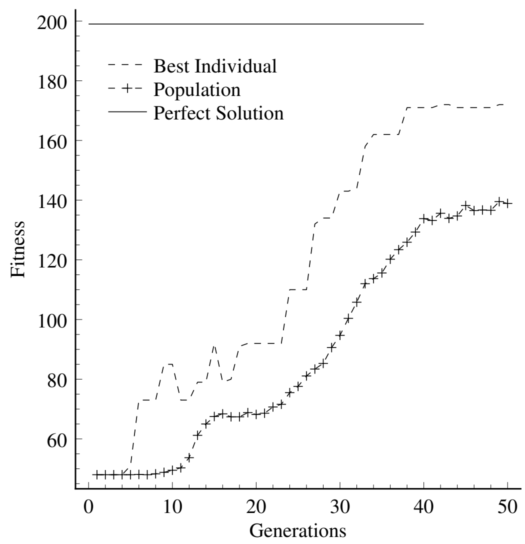

Figure 38. Sort-Th-1: Fitness of best individual and population in unsuccessful execution

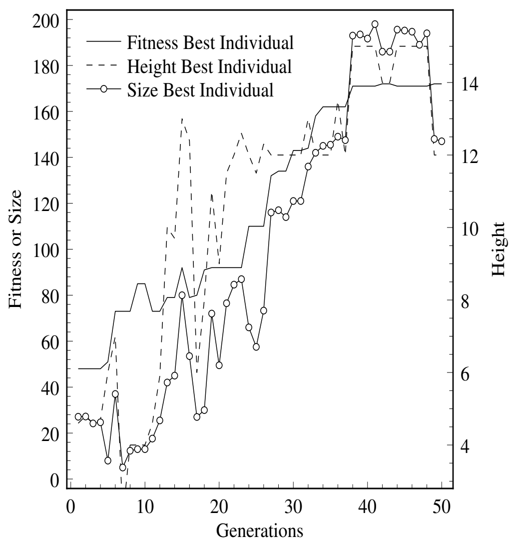

Figure 39. Sort-Th-1: Fitness, height and size of best individual in unsuccessful execution

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

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

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

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

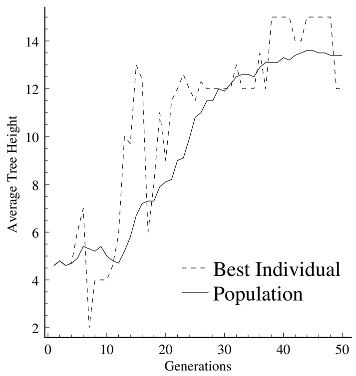

Figure 40. Sort-Th-1: Program height in unsuccessful execution

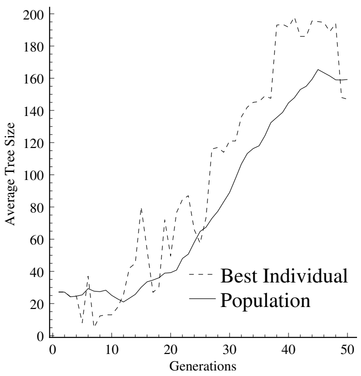

Figure 41. Sort-Th-1: Program size in unsuccessful execution

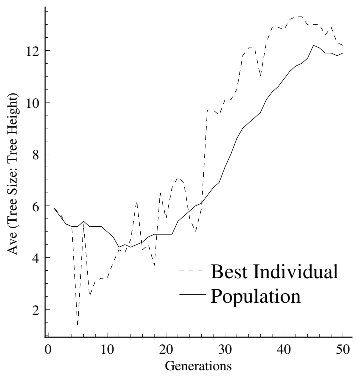

Figure 42. Sort-Th-1: Program size:height in unsuccessful execution

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

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

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

We observe that GP prematurely converges and processing more individuals does not appear to prevent this. Diversity in the population is not likely an issue because though scaling usually limits diversity in later generations of GAs, there is sufficient diversity in a GP population even among the candidates strong enough for selection. GP executions remain diverse because crossover does not take place on strictly defined structure boundaries. In fact, once the 60% neighbourhood has been reached, we see disparity in fitness, (implying diversity in programs) return to the population. On an animated histogram the "undulating slinky movement from left to right" [62] halts and a downward wave to the left occurs.

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

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

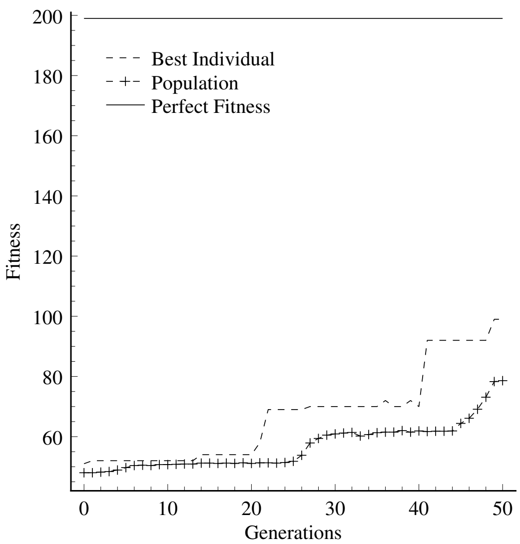

Figure 43. Sort-Th-2: Fitness of best individual and population in unsuccessful execution

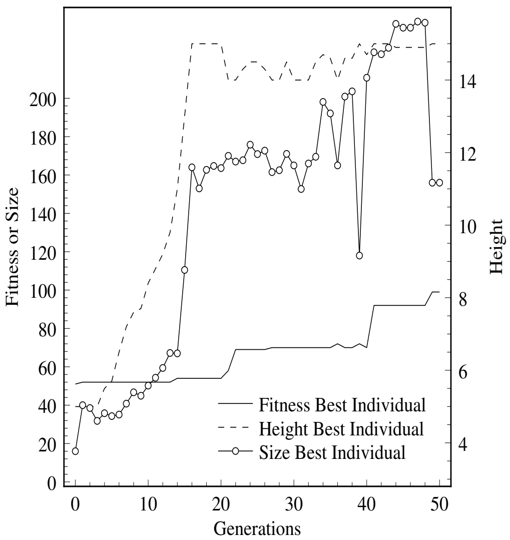

Figure 44. Sort-Th-2: Fitness, height and size of best individual in unsuccessful execution

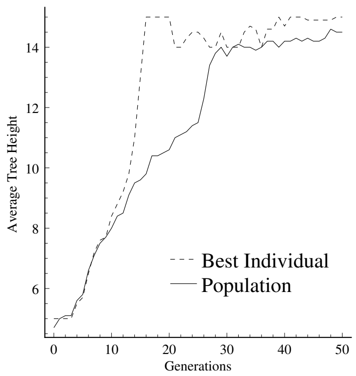

Figure 45. Sort-Th-2: Program height in unsuccessful execution

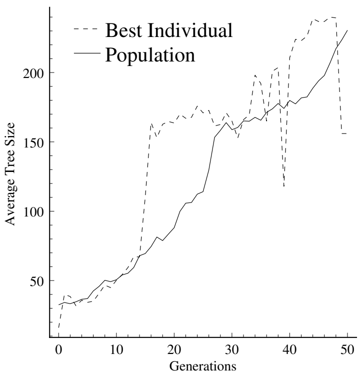

Figure 46. Sort-Th-2: Program size in unsuccessful execution

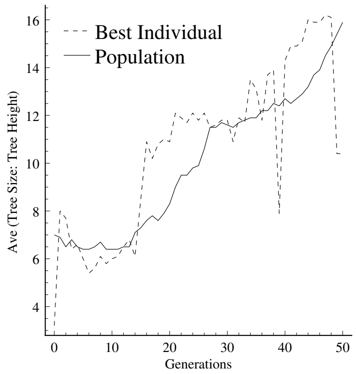

Figure 47. Sort-Th-2: Program size:height in unsuccessful execution

3.1.4. Assessing the Presence of A Hierarchical Process

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This does not appear to be true.

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

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

3.2. Improving Hierarchy in GP

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

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

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

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

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

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

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

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

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

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

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

Overall, this approach does not produce encouraging results. It would be reasonable to expect both improved computational efficiency and hierarchical solutions neither of which seem to occur. Why?

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

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

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

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

  Implementing a two level environment (by two fitness functions for example) poses problems. The second level fitness function would have to be specified *a priori* and this would seem to force the designer to identify what subtasks are desired. It also seems impossible to identify appropriate time scales for each level of evolution. Furthermore, other aspects of GP might hold this sort of approach back: GP offers no model for how subpopulations with different fitness functions and time scales should interact. All extensions collectively might swamp the basic power of the algorithm because they vastly increase the search space size and depend upon designer foresight.

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

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

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

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

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

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

3.3.1. Deriving Knowledge-Based Primitives from First Principles

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

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

From the perspective of judging the power of GP, two specific objections arise from using knowledge-based operators and operands: First, GP is using a language $L\prime$ which is a specialized subset of the programming language, L, that is used to write the primitives. $L\prime$ may not include general elements of L and this may constrain the flexibility of solution expression. Second, $L\prime$ expresses knowledge-based properties of the problem domain which GP exploits in its adaptation process. A skeptic can fairly argue that GP may not have been able to solve the problem without the guidance this knowledge provides.

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

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

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

To conform to the policy, GP would have to solve the Block Stacking problem starting with variables `*block*`, `*stack*`, `*goal-list*`, `*table*`, an assortment of list operators to iterate, add, delete and access (since the stack, goal list and table are lists), a conditional statement (e.g. if-then-else), and `NOT` and `EQ` for comparison. This seems far too demanding! No one else expects to solve the Block Stacking problem from this low a level.

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

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

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

3.3.2. Using Knowledge-Based Primitives

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

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

3.3.3. Designing a Fitness Function

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

Altenberg [2] has shown that GP is successful when the fitness distribution of the set of offspring that could possibly be produced by crossing over a pair of parents has a large upper tail. The penultimate secret of GP would indicate how to adjust a fitness function to improve the likelihood of such distributions among parent pairs. It may be too difficult however and in the immediate future it is more likely that improvement could be gained by pursuing changing a search operator given a fitness function.

3.4. GP as a GA for Program Discovery

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

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

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

3.4.1. Program-Based Encoding

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

3.4.2. GP Crossover

One obvious aspect of GP appears to be its crossover operator. However, is this choice of operator over other GA crossover operators superficial or significant to GP's success? The question was pursued by [124] and the difference was found to be superficial. The EP-I system is a GA that solves program discovery by choosing a representation that embeds a program of program discovery primitives into a fixed length linear string. Programs are able to vary in length because not all positions of the string are always interpreted as the program. The embedding works in a manner that permits linear-based GA crossover operators (e.g., one-point, uniform) to be applied to parent strings and produce directly syntactically correct offspring that could differ in size and structure from their parents. The net effect of those GA crossover operators is the same as GP crossover but the representations for that purpose differ. In terms of probability of success, on a small suite of standard GP problems (e.g., 6-Mult and Block Stacking), no statistically significant difference between EP-I and GP was found. Thus it was concluded that the use of a tree or hierarchical representation for crossover is, contrary to intuition, not essential to GP's power but merely a convenience that allows program size and structure to change.

3.4.3. Variable Length Solutions

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

3.4.4. Feature Correspondence Among Solutions

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

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

3.5. Chapter Summary

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