CHAPTER 2

Thesis Problem Suite, Genetic Programming, and its Extensions

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

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

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

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

2.1. Thesis Problem Suite

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

2.1.1. Motivation for the Problem Suite

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

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

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

2.1.2. 6-Mult: The 6 Bit Boolean Multiplexer

Problem Definition: `6-Mult`

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

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

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

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

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

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

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

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

Test Suite: In order for a test case to be run on a program, all 6 variables are initialized to test case values. The result returned by a program is interpreted as its answer to the question: "What is the data value at the address encoded by `*A0*` and `*A1*`?" The correct answer is the *a priori* set binary value of the register at the address. A random program has a 50% probability of returning the correct answer even though it may not be functioning anything like a boolean multiplexer.

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

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

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

2.1.3. 11-Mult: The 11 Bit Boolean Multiplexer

Problem Definition: `11-Mult`

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

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

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

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

Test Suite: In order for a test case to be run on a program, all 11 variables are initialized to test case values. The result returned by a program is interpreted as its answer to the question: "What is the data value at the address encoded by `*A0*`, `*A1*` and `*A2*`?" The correct answer is the *a priori* set binary value of the register at the address. A random program has a 50% probability of returning the correct answer even though it may not be functioning anything like a boolean multiplexer.

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

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

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

2.1.4. Sort-A

Problem Definition: `Sort-A`

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

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

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

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

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

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

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

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

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

Out of Training Test Suite: Since sorting is an unbounded problem and GP is inductive, it is impossible to use GP to confirm that a perfectly general sorting solution has been found. Two tests for generalization are possible: first, a program may be functionally verified by a person, second, a different set of test cases (distinct from the test suite) can be run on a perfect program. While the second set of test cases still can not confirm the program is completely general, at least, it can be decided whether some generalization capacity exists. We coduct a study into this issue in Section 3.1.1.

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

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

2.1.5. Sort-B

Problem Definition: `Sort-B`

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

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

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

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

2.1.6. BS: Block Stacking

Problem Definition: `BS`

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

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

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

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

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

`not(parm-1)` Logical not.

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

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

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

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

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

Out of Training Test Suite: None.

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

2.1.7. Format of Experiment Description

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

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

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

3. a **fitness function**,

4. a **test suite**,

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

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

1. **problem definition**

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

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

2.2. Canonical Genetic Programming

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

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

Part 1: Types and Variables: Variables are in four groups. The first group "problem input parameters" consists of variables that link a GP application (i.e., problem) to the GP "shell". (By "shell" we mean the problem-independent part of the GP algorithm that creates the initial population of programs, and processes populations by iterating over generations.) These variables are initialized at the outset of a GP run so that the shell can generically access a set of primitives, the test suite and the fitness functions. In addition to the expected variables: `testsuite`, `primitives`, and `fitness-function`, this group includes `testsuite-size` and `max-fitness`.

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

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

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

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

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

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

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

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

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

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

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

```
PROGRAM  GENETIC-PROGRAMMING-PSEUDO-CODE

TYPE

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

```
        compares its outputs to testcase output values

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

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


VARIABLES

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

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

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

/* GENERATION 0 INITIALIZATION PARAMETERS */

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

/* RUN PARAMETERS    */

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

real:
    non-leaf-selection-bias;

/* RUN VARIABLES         */

integer:
    generation = 0;

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

```
boolean: solution-found = false;

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


/* MAIN PROGRAM */

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

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

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

    initialize-random-number-generator(seed);

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

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

    report-on-zeroth-generation;

    /* generation loop */

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

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

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

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

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

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

end; /* main program */



/*              PROCEDURES         */

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

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

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

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


procedure CREATE-ZEROTH-GENERATION
```

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

for i:=1 to population-size

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

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

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

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

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

end procedure POPULATION-FITNESS-CALCULATION

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

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

```
*/
end procedure ASSESS-FITNESS

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

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

procedure REPORT-ON-ZEROTH-GENERATION;

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

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

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

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

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

Figure 7. Genetic Programming (GP) Algorithm Pseudocode

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

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

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

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

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

2. A report is generated supplying data on the characteristics of "Generation 0".

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

- A number of programs (`parents-to-copy`) of the current generation are chosen via roulette-wheel selection (i.e., fitness proportionate probabilistic selection) to have exact copies of themselves directly propagated to the next generation.

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

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

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

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

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

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

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

Canonical GP does not use a mutation operator. One reason is that a general mutation operator i.e., one which can perform substitution of primitives and increase or decrease program length anywhere in the existing program while preserving syntactic validity, is not immediately obvious though we shall provide one later. Mutation operators that only modify entire subtrees or only substitute one primitive for another are not as general but obvious to design. Knowledge-based mutation operators that "tweak" primitives according to their meaning are also available. A variety have been used in reported experiments, e.g. [6, 57] but either they are not accompanied by a comparison (to not using mutation at all). One comparison, on `6-Mult`m does not indicate they produce significantly superior results [62].

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

2.3. Examples of Primitive Semantics

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

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

2.3.1. Directly Using Built-in Functions as Primitives

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

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

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

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

2.3.2. "Firewalling" Built-in Functions as Primitives

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

2.3.3. Arithmetic Constants as Primitives

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

2.3.4. Recursion via a Primitive

A simple way to provide recursion is to allow a program to call itself. This must be implemented a bit circuitously by using a helper primitive named, for example, `self`. `self`) must be placed in the primitive set and it must express the base case of the presumed recursive function. A program invokes itself when it contains the primitive `self` and `self is evaluated`. In order to make the evaluation of `self` the equivalent of calling the program, before the program is valued, the algorithm names it "on the fly", as a LISP function, and by using a global variable. Suppose `self-defun` is the assigned name for the program,[2]. When the program is tested and the primitive `self` is evaluated, `self`:

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

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

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

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

5. restores the calling program's execution context.

6. returns the result of the recursive call.

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

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

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

2.3.5. Iteration via a Primitive

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

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

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

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

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

The need for a "deadman's brake": It is imperative that a loop body alters the part of the execution state that eventually causes the looping condition to halt the iteration. However, programs composed out of typical primitives are vulnerable to not displaying such behaviour. To prevent a loop from iterating endlessly, all iterative primitives must include a means of cutting off iteration. Usually, they do so by updating the corresponding counters of preset variables called `local-loop-limit` and `global-loop-limit` and using the variables to terminate iteration. These variables are somewhat hidden from attention in GP but pose some problems. First, their settings must be chosen ad-hoc. Second, a program's execution may depend upon the settings. For example, a "perfect" program may only be perfect with the GP run's values of the parameters because they control the iteration as much as the iteration primitive's parameters. If they are changed, the execution changes. When a very complex progam is evolved, it may not be readily comprehensible (because GP programs are usually larger than a human-designed program using the primitives and may contain extra inconsequential primitives) and such a dependence may exist. In these cases, either tedious program simplification and explanation must be done or the program should be tested on other examples of the problem.

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

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

2.3.6. Assignment via a Primitive

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

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

2.3.7. Typing Parameters in Primitives

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

2.3.8. The advantages of using LISP

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

2.4. Crossover Operator Properties

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

2.4.1. Blind choice of crossover points

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 1. GP Crossovers and `6-Mult`

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

Table 2. GP Crossovers and `11-Mult`

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

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

Table 3. GP Crossovers and Sort-A

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

Table 4. GP Crossovers and Sort-B

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

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

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

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

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

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

2.4.2. Syntactically Correct Offspring

Because GP is supposed to be a "weak method", it is crucial that a crossover operator in GP generate programs that are syntactically correct. In other GAs, this requirement is equivalently "the crossover operator must generate encoded offspring that decode to valid candidate solutions". Should this requirement not be met, an ad-hoc, problem specific repair algorithm has to be added to either search algorithm detracting from its generality.

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

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

2.4.3. Flexible Program Length

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

2.4.4. Parent-Offspring Fitness Distribution

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

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

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

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

2.4.5. "True" Combination

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

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

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

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

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

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

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

2.5. Non-Canonical GP

The GP literature can be grouped into three categories:

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

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

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

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

- New Operators

- Alternate Selection and Generation Strategies

- Representation Enhancements

2.5.1. New Operators

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

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

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

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

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

Kinnear also introduced a "create" operator where a new program is created randomly (like in Generation 0) and inserted into the current population in lieu of a program propogated by parent selection and crossover and/or mutation. This operator can be refined to ensure that a random new program is of smaller height than the population average or shorter than a tree drawn by fitness proportionate selection [57]. No isolation study of the benefits of the hoist and create operators has been conducted.

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

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

2.5.2. Alternate Selection and Generation Strategies

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

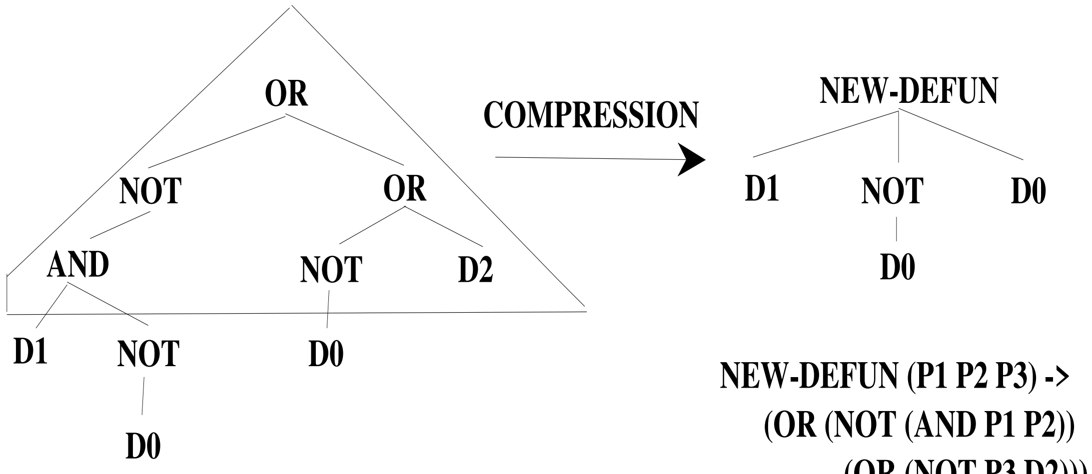

Figure 8. Angeline's compression operator encapsulates a subtree of a maximum depth, creates a function for the subtree and replaces the subtree with a call to the new functions ([5, 7]).

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

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

2.5.3. Representation Extensions

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

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

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

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

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

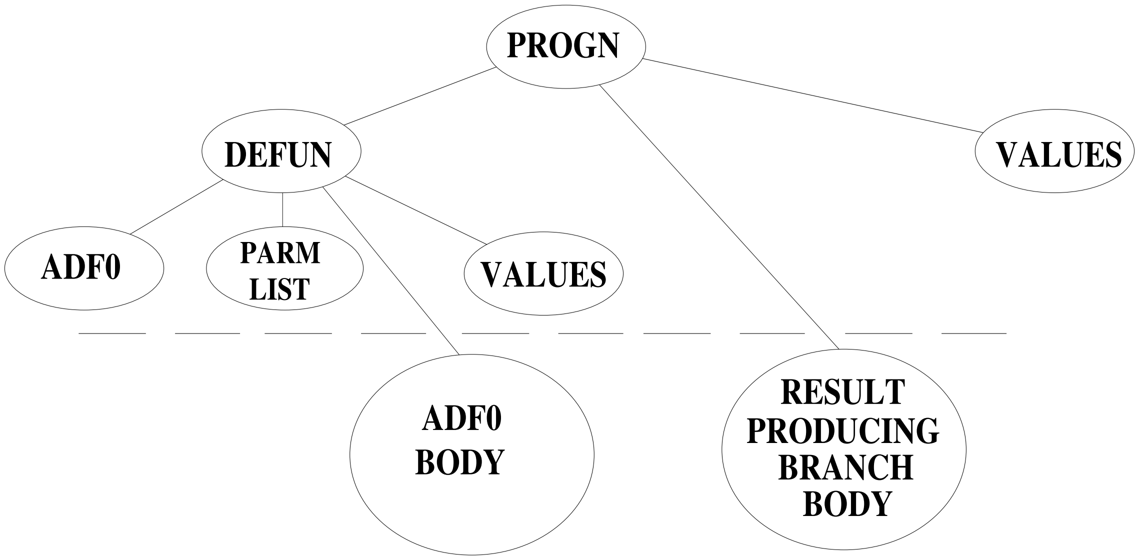

Figure 9. When using ADFs in LISP the template is a PROGN S-expression with DEFUN forms for each ADF and a one form for the result producing branch. Only the primitives below the dotted line are evolved ([70]).

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

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

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

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

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

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

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

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

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

2.6. Chapter Summary

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