Advanced Techniques in Nature Inspired Search and Optimization
Lab 4

Brendan Hart (1560038)

March 29, 2018

1 Question 4

For the following algorithms, an expression is represented by a list, where the first element is a tuple of a single item containing a string which is the function name (this can then be used as a key in the function dictionary to get the function and the number of arguments). Subsequent elements in the list are arguments to the function specified by the string. These arguments can then be further lists representing expressions. E.g. [('add'), [('data'), 0.0], [('data'), 1.0]] represents an expression consisting of the addition between data points 0 and 1 in a given data set. The fitness function used is the average squared error between the actual output and the output given by the generated expression, therefore a smaller fitness means a better performing expression on a given data set.

Throughout this pseudocode, there are functions assumed to be present. Most importantly are random_float which returns a random float between a sepcified range and get_branches which takes an expression in list form and returns a list of branches, where each branch is represented by a list of indices which specify how to access that branch from the root node.

Below is the tournament selection algorithm which was used to select members of the population for crossover.

Algorithm 1 Tournament selection

```
Require: The tournament size $k$
Require: The population $population$
 1: function TOURNAMENT_SELECTION(k, population)
 2:     $selected \leftarrow []$
 3:     for $i = 0$ to $(k - 1)$ do
 4:         chosen $\sim$ Unif(population)
 5:         selected $\leftarrow$ selected + chosen
 6:     end for
 7:     fittest $\leftarrow$ []
 8:     max_fitness $\leftarrow$ MAX_FLOAT_VALUE
 9:     for member in selected do
10:         if fittest is empty or member[1] < max_fitness then
11:             fittest $\leftarrow$ [member]
12:             max_fitness $\leftarrow$ member[1]
13:         else if member[1] equals max_fitness then
14:             fittest $\leftarrow$ fittest + member
15:         end if
16:     end for
17:     result $\sim$ Unif(fittest)
18:     return result
19: end function
```

The below algorithm creates an expression of a given depth consisting of random functions at each node. The leaf nodes are values which are randomly generated floats between -*rand_range* and *rand_range*, which is a parameter provided to the method.

Algorithm 2 Create a random tree of a given depth

```
Require: The population $population$
Require: The dictionary containing functions and their number of arguments $f\_dict$
Require: The range for random number generation $rand\_range$
 1: function FULL_METHOD(population, f_dict, rand_range)
 2:     if $tree\_depth <= 0$ then return None
 3:     end if
 4:     $random\_branch \leftarrow None$
 5:     $f \sim Unif(f\_dict.keys())$
 6:     $num\_args \leftarrow f\_dict[f][1]$
 7:     if $tree\_depth == 1$ then
 8:         $args \leftarrow [random\_float(-rand\_range, rand\_range)$ for $i = 1$ to $num\_args]$
 9:     else
10:         $args \leftarrow [full\_method(tree\_depth - 1, f\_dict, rand\_rage)$ for $i = 1$ to $num\_args]$
11:     end if
12:     return $[(f)] + args$
13: end function
```

The below algorithm chooses two points in two members of the population uniformly and swaps them over.

Algorithm 3 Crossover two population members

```
Require: A member of the population $x$
Require: A member of the population $y$
 1: function CROSSOVER(x, y)
 2:     $copy\_x \leftarrow copy(x)$
 3:     $copy\_y \leftarrow copy(y)$
 4:     $branches\_x \leftarrow get\_branches(copy\_x)$
 5:     $branches\_y \leftarrow get\_branches(copy\_y)$
 6:     if $branches\_x == []$ and $branches\_y == []$ then
 7:         return ((copy_y, None), (copy_x, None))
 8:     else if $branches\_x == []$ then
 9:         $choice\_y \sim Unif(branches\_y)$
10:         $y\_branch \leftarrow copy\_y$
11:         for $i = 0$ to $len(choice\_y) - 1$ do
12:             $y\_branch \leftarrow y\_branch[choice\_y[i]]$
13:         end for
14:         $tmp \leftarrow copy(y\_branch)$
15:         $y\_branch \leftarrow copy\_x$
16:         $copy\_x \leftarrow tmp$
17:     else if $branches\_y == []$ then
18:         $choice\_x \sim Unif(branches\_x)$
19:         $x\_branch \leftarrow copy\_x$
20:         for $i = 0$ to $len(choice\_x) - 1$ do
21:             $x\_branch \leftarrow x\_branch[choice\_x[i]]$
22:         end for
23:         $tmp \leftarrow copy(x\_branch)$
24:         $x\_branch \leftarrow copy\_y$
25:         $copy\_y \leftarrow tmp$
26:     else
27:         $choice\_x \sim Unif(branches\_x)$
28:         $x\_branch \leftarrow copy\_x$
29:         for $i = 0$ to $len(choice\_x) - 1$ do
30:             $x\_branch \leftarrow x\_branch[choice\_x[i]]$
31:         end for
32:         $choice\_y \sim Unif(branches\_y)$
33:         $y\_branch \leftarrow copy\_y$
34:         for $i = 0$ to $len(choice\_y) - 1$ do
35:             $y\_branch \leftarrow y\_branch[choice\_y[i]]$
36:         end for
37:         $tmp \leftarrow x\_branch$
38:         $x\_branch \leftarrow y\_branch$
39:         $y\_branch \leftarrow x\_branch$
40:     end if
41:     return $((copy\_x, None), (copy\_y, None))$
42: end function
```

The below algorithm takes a member of the population and chooses a random branch uniformly. It then replaces this branch with a new tree which has a random depth between 1 and the maximum depth set by the parameter *depth*.

Algorithm 4 Mutate a member of the population

```
Require: A member of the population $x$
Require: The dictionary containing functions and their number of arguments $f\_dict$
Require: The range for random number generation $rand\_range$
Require: The maximum depth to create a tree of $depth$
1: function MUTATE(x, f_dict, rand_range, depth)
2:     $copy\_x \leftarrow x$
3:     $branches \leftarrow get\_branches(copy\_x)$
4:     $rand\_tree \leftarrow full\_method(random\_int(1, depth), f\_dict, rand\_range)$
5:     if $branches == []$ then
6:         return $rand\_tree$
7:     end if
8:     $choice \sim Unif(branches)$
9:     $branch\_before \leftarrow copy\_x$
10:    for $i = 0$ to $len(choice) - 1$ do
11:        $branch\_before \leftarrow branch\_before[choice[i]]$
12:    end for
13:    $branch\_before[choice[-1]] \leftarrow rand\_tree$
14:    return $copy\_x$
15: end function
```

The below method calculates the fitness of a given member, as in part 2 of this lab. It assumes a method evaluate which reduces an expression to a value (as in part 1 of this lab).

Algorithm 5 Calculate the fitness of a given member

```
Require: The dictionary containing functions and their number of arguments $f\_dict$
Require: An expression to calculate the fitness for $expr$
Require: The data for evaluating fitness $data$
1: function CALC_FITNESS(f_dict, expr, data)
2:     $sum\_error \leftarrow 0.0$
3:     for $(X, y)$ in $data$ do
4:         $sum\_error \leftarrow sum\_error + ((y - evaluate(f\_dict, expr, X)) * *2$
5:     end for
6:     return $sum\_error/len(data)$
7: end function
```

Below is the main genetic algorithm, which makes use of the fitness function to evaluate members of the population, tournament selection to select individuals, a method to crossover two members of the population, and the mutation operator described above. The algorithm runs for *timeout* number of seconds, after which the most recent *population* is returned from which the best solution can be considered.

Algorithm 6 The main genetic algorithm

```
Require: The tournament size $k$
Require: The population size $lambda$
Require: The desired chance of mutation $mutation\_rate$
Require: The initial population tree depth $init\_depth$
Require: The max tree depth in mutation $mut\_depth$
Require: The dictionary containing functions and their number of arguments $f\_dict$
Require: The range for random number generation $rand\_range$
Require: The data for evaluating fitness $data$
Require: The number of generations until timeout $timeout$
function GENETIC_ALGORITHM(k, lambda, h, mutation_rate, init_depth, mut_depth, f_dict, rand_range, data, timeout)
    $population \leftarrow []$
    $num\_of\_gens \leftarrow 0$
    $start\_time \leftarrow time.now()$
    $current\_time \leftarrow start\_time$
    for $i = 0$ to $(lambda - 1)$ do
        $sexpr \leftarrow FULL\_METHOD(init\_depth, f\_dict, rand\_range)$
        $sexpr \leftarrow (sexpr, CALC\_FITNESS(f\_dict, sexpr, data))$
        $population \leftarrow population + sexpr$
    end for
    while $current\_time - start\_time < timeout$ do
        $new\_pop \leftarrow []$
        $EVALUATE\_POP(population, weeks)$
        while $len(new\_pop) < lambda$ do
            $x \leftarrow TOURNAMENT\_SELECTION(k, population)$
            $y \leftarrow TOURNAMENT\_SELECTION(k, population)$
            if $random\_float(0, 1) < mutation\_rate$ then
                $x \leftarrow MUTATE(x, f\_dict, rand\_range, mut\_depth)$
            end if
            if $random\_float(0, 1) < mutation\_rate$ then
                $y \leftarrow MUTATE(y, f\_dict, rand\_range, mut\_depth)$
            end if
            $(child1, child2) \leftarrow CROSSOVER(x, y)$
            $child1 \leftarrow (child1[0], CALC\_FITNESS(f\_dict, child1[0], data))$
            $child2 \leftarrow (child2[0], CALC\_FITNESS(f\_dict, child2[0], data))$
            $new\_pop \leftarrow new\_pop + child1$
            $new\_pop \leftarrow new\_pop + child2$
        end while
        $num\_of\_gens \leftarrow num\_of\_gens + 1$
        $population \leftarrow new\_pop$
        $current\_time \leftarrow time.now()$
    end while
    return (population, num_of_gens)
end function
```

2 Question 5

The following experiments were carried out using the above algorithm. Each value and parameter pair was carried out for 100 repetitions for a time of 20 seconds, at which point the fitness of the best member in the population of the most recent generation was recorded. The fittest member of the population is the one with the lowest fitness value, as we want to minimize the error. The data used in the experiments consisted of 100 examples of adding two numbers between -1.0 and

1.0. Box plots for each parameter were then plotted. The following parameters were varied:

- The mutation rate used to decide if a given member should mutate or not.
- The tournament size.
- The initial population's tree depth.
- The maximum tree depth of new branches used in mutation.
- The population size.
- The range for random values used to create leaf nodes in trees.

2.1 Mutation Rate vs Average Attendance

The mutation rate was varied with values in the list [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]. The population size was set to 100, tournament size to 2, range for random values to 50, mutation tree depth to 4 and initial population tree depth to 2. After 100 repetitions for each value, the following box plot was the result.

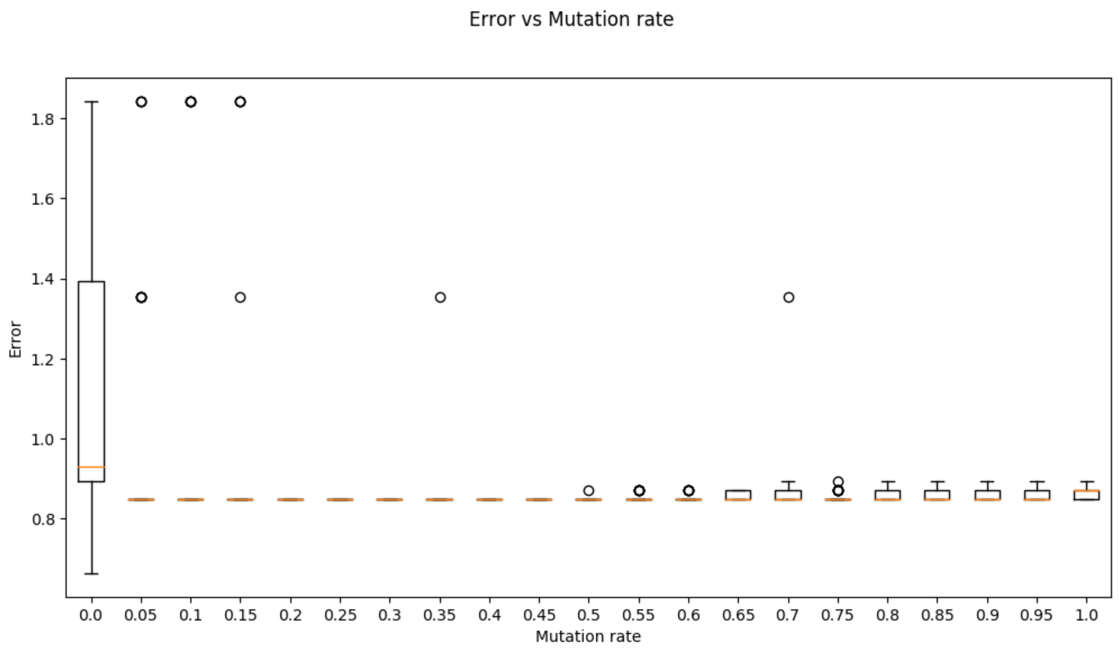

Figure 1: Error vs Mutation Rate.

Mutation rate seemed to achieve similar performance for all values except a rate of 0.0, which performed worst most of the time. A possible reason for this is that no mutation means a poor initial population is hard to recover from as the only possible change is crossover between two poor solutions.

2.2 Tournament Size vs Average Attendance

The tournament size was varied with values in the list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. The population size was set to 100, mutation rate to 0.4, range for random values to 50, mutation tree depth to 4 and initial population tree depth to 2. After 100 repetitions for each value, the following box plot was the result.

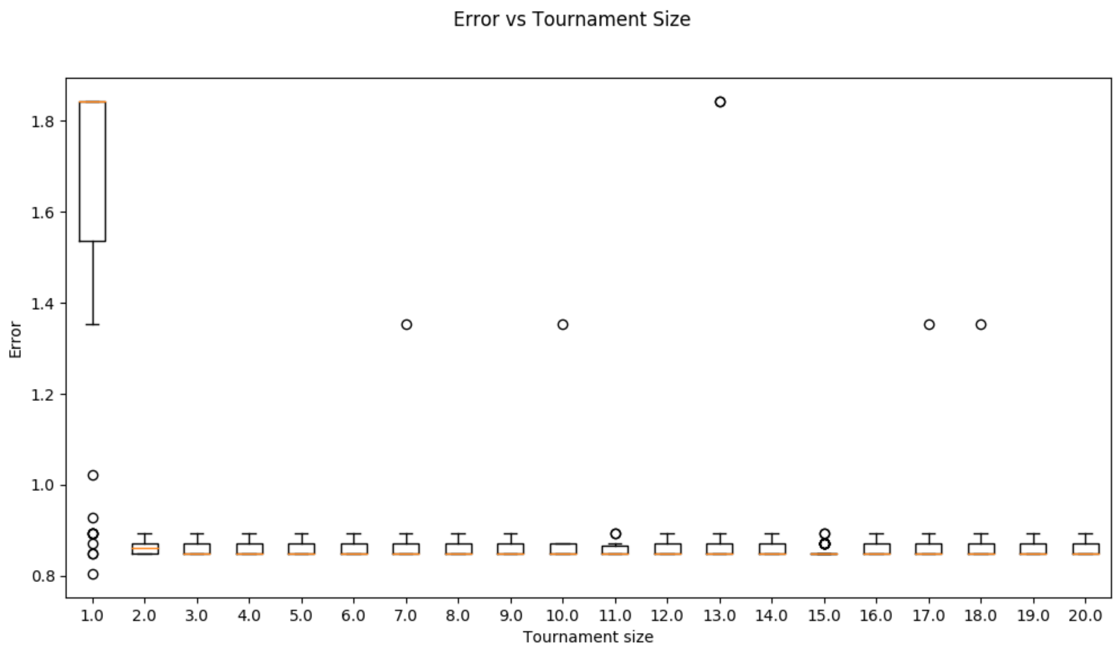

Figure 2: Error vs Tournament Size.

There only seems to be one value of interest in the graph of tournament size. When the tournament size is 1, even performance is considerably worse than the rest of the values which all achieved similar fitness. The cause of this may be a tournament size of 1 means any member of the population is selected regardless of its fitness value in relation to the others. This means the worst members of the population may be chosen too easily, resulting in the next generation also being as poor or worse. When the tournament size is k, we can be sure that the selected member is at least as good as k-1 members of the population. Therefore, the higher the tournament size, the more sure we are that we're picking a good member of the population in relation to the rest. However, a high tournament size reduces diversity.

2.3 Inital Tree Depth vs Error

The initial tree depth was varied with values in the list [1, 2, 3, 4, 5, 6, 7, 8, 9]. The population size was set to 100, tournament size to 2, mutation rate to 0.4, range for random values to 50 and mutation tree depth to 4. After 100 repetitions for each value, the following box plot was the result.

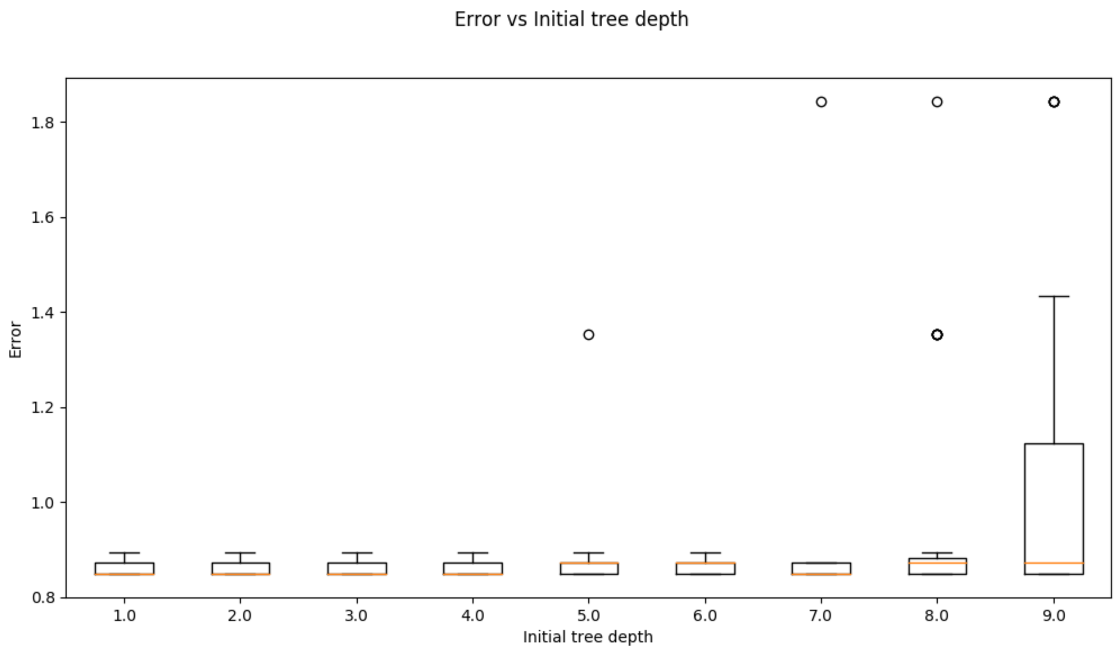

Figure 3: Error vs Initial Tree Depth.

The initial tree depth parameter did not seem to make too much of a difference to the fitness until the parameter took on the value of 9. It may be possible that from a depth of 9 onwards, the trees become too large for this data set and require far more tweaking until a good solution is found. A larger tree depth means more processing time is required, and as a result less generations are carried out in the given time budget and therefore less potential to achieve a good solution.

2.4 Mutation Tree Depth vs Error

The mutation tree depth was varied with values in the list [1, 2, 3, 4, 5, 6, 7, 8, 9]. The population size was set to 100, tournament size to 2, mutation rate to 0.4, range for random values to 50 and initial population tree depth to 2. After 100 repetitions for each value, the following box plot was the result.

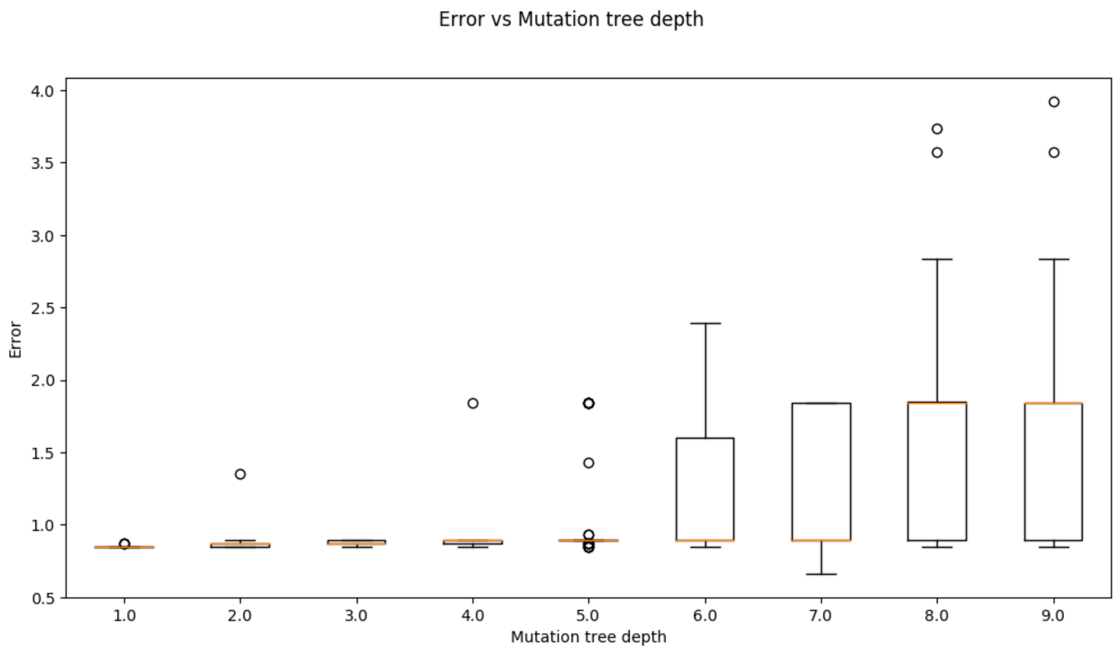

Figure 4: Error vs Mutation Tree Depth.

Mutation tree depth sees a similar trend to initial tree depth, apart from it starts earlier. We see that 1, 2, 3, 4 and 5 mutation tree depths have similar performance, and then 6 onwards performs poorer (despite 6 and 7 achieving a median vale similar to values 1-5). The cause of this may be for the same reason, that after a certain tree depth the increased time taken to process the next generation reaches a point that the population does not have enough time to evolve under the given time budget.

2.5 Population Size vs Error

The population size was varied with values in the list [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 250, 350, 500, 650, 800, 1000]. The tournament size was set to 2, mutation rate to 0.4, range for random values to 50, mutation tree depth to 4 and initial population tree depth to 2. After 100 repetitions for each value, the following box plot was the result.

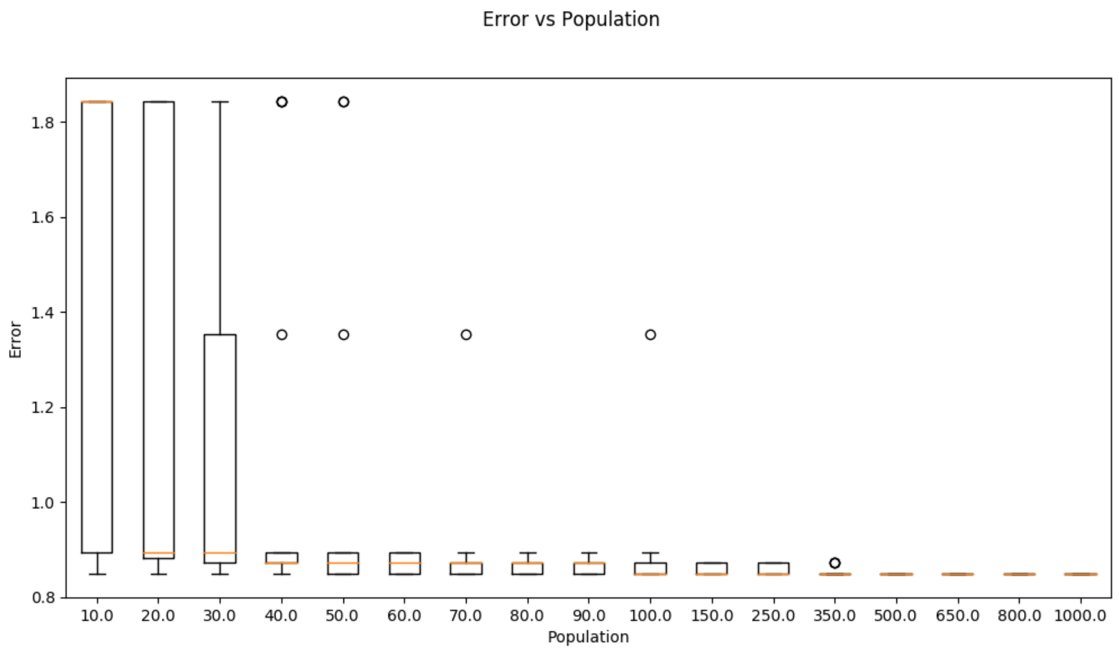

Figure 5: Error vs Population Size.

Varying the population size seems to show a trend of increased performance as population size increases. This increase in performance is most noticeable by the drop between population sizes 10 and 20, which have medians of around 1.9 and 0.9 respectively. As well as the median decreasing as population size increases, the spread of the results around the median also decreases, meaning as population size increases the results become more consistent and at a population size of 1000 almost always reach the same fitness value. The cause of this may be that as population size increases, there is greater diversity and a wider range of solutions stored in the population in each generation, and therefore more opportunity for crossover and mutation to produce a good solution.

2.6 Random Number Range vs Error

The population size was varied with values in the list [1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 125, 150, 175, 200]. The tournament size was set to 2, population size to 100 mutation rate to 0.4, mutation tree depth to 4 and initial population tree depth to 2. After 100 repetitions for each value, the following box plot was the result.

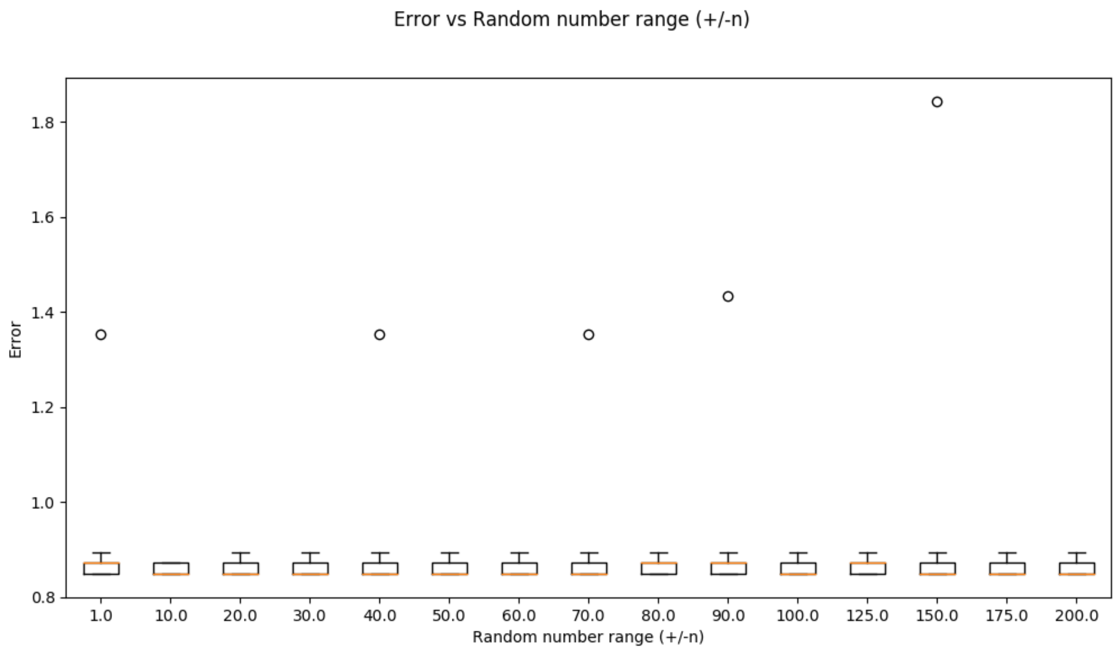

Figure 6: Error vs Random Number Range.

On the data set used, there was minimal difference in the range used for the random number generation. The cause of this may be that the values are not used in their raw form for operations such as data, and instead are using modulo some number.