CS547 Assignment 2 Report

Andrew Fagan and Martin Tiggerdine

Problem Overview

Summary

The goal of this project is to implement a genetic algorithm (with no restraint on languages or libraries) which optimises ordered subsets of a large suite of tests to uncover as many faults as possible, and as early in the set as possible. The algorithm should be able to generate solutions of a fixed length, which can be changed easily. It should also be able to evaluate test suites with variable numbers of tests and faults, and should be tested on two different suites of tests. Suite 1 has 216 tests and 9 faults, and should create subsets of length 5. Suite 2 has 1000 tests and 38 faults, and by observation requires at least 7 tests to cover all faults. Performance of this algorithm should be evaluated over at least 10 runs, and should be compared to a Hill Climbing algorithm and random generation.

Fitness Function

It was recommended to use APFD (Average Percentage of Faults Detected) as the basis of the fitness function. However, uncertainty about what to do if a fault was not detected led to the development of a different fitness function:

For each test, t_p (where p is its position in the order)
Count the number of new faults it detects, n
Add n / (p + 1) to the fitness
Normalise the fitness to a value between 0 and 1 by dividing it by the number of faults

This rewards finding more faults, earlier. For example, take a random chromosome c:

$c = [t83,\ t9,\ t53,\ t194,\ t177,\ t91]$

The relevant rows of the fault matrix are:

t/f	1	2	3	4	5	6	7	8	9
83	1	1	0	1	0	1	1	1	0
9	0	0	0	0	0	0	1	0	0
53	0	0	0	0	0	0	1	0	0
194	0	0	1	0	0	0	1	0	0
177	1	0	0	0	0	0	1	0	0
91	1	0	0	0	0	0	1	0	0

The 0th test detects 6 new faults and the 3rd test discovers 1 new fault. The fitness is calculated by:

$$f = (6 / (0 + 1) + 1 / (3 + 1)) / 9 = 0.6944$$

The fitness function correlates positively with the fitness of the chromosome, so a value of 1 represents all faults found on first test, while a value of 0 means no faults were found at all.

Crossover Function

The crossover function is very simple, splitting each parent at the same random index then swapping and recombining them. The crossover function then examines the new child genes to find duplicate values, and replaces them with elements chosen from the other parent.

For example, take two parents p1 and p2, and a random index i.

$$p1 = (1, 2, 3, 4, 5, 6)$$
$$p2 = (6, 5, 4, 3, 2, 1)$$
$$i = 2$$

The parents are split at index i:

$$p11 = (1, 2, 3) \quad p12 = (4, 5, 6)$$
$$p21 = (6, 5, 4) \quad p21 = (3, 2, 1)$$

And recombined like so:

$$c1 = p11 + p21 = (1, 2, 3, 3, 2, 1)$$
$$c2 = p21 + p12 = (6, 5, 4, 4, 5, 6)$$

Finally, duplicates are replaced by drawing from the first half of the opposite parent:

$$c1 = (1, 2, 3, 6, 5, 4)$$
$$c2 = (6, 5, 4, 1, 2, 3)$$

This is a slight deviation from the crossover function described in the lecture but through experimentation it was found that this has a similar level of effectiveness.

Mutation Function

The mutation function sweeps through the population, and has a 5% chance to mutate. If a gene mutates, the function iterates over each element in the gene. Each element has a chance of mutation equal to one over the number of elements. When an element mutates, it is either swapped with its neighbour or replaced by another test from the suite at random, with equal probability.

The latter point is a deviation from the content presented in the lecture, but it was found by experimentation that the ability to bring in new genetic material was valuable to the algorithm, and also allowed the Hill Climbing test algorithm to be implemented entirely with existing code. It was decided to

substitute randomly instead of by neighbour as neighbouring tests have no particular resemblance to each other.

Implementation

Genetic Algorithm

The implementation is in python in a functional/procedural architectural style. The stages of the algorithm are as follows:

Population generation is performed by generating a fixed amount of genes. A single gene is made by randomly selecting a fixed amount of unique test labels from the input data set, formatted as a list.

The selection step is carried out in two substeps. First, the function called selection sorts the population by fitness and removes the lower 50% of the population. The remaining 50% is used both as the breeding pool, and additionally carries over to the next generation. This elitism approach was found to be very effective, as the more traditional approach of constructing an entirely new population was found to stagnate quite early on in testing. As a result of this step, there is no need for a mutation rate of less than 100%, as this would result in duplicate values entering the population.

The second substep of selection is carried out as part of the mating phase. The new population consists of the 50% of the old population (which can be referred to as the “breeding pool”) carried over combined with a new 50% created from these.

A single parent is selected for mating by choosing a pool of 10 potential parents from the breeding pool at random and finding the maximum fitness value from this small pool. For variance, the fitnesses within the pool of 10 are scaled randomly between their actual fitness and twice their fitness, which gives less fit genes a chance to be parents, but one which is proportionally smaller than fitter parents.

This process is repeated to find a second parent (which can be the same as the first, though uncommonly), these parents are crossed over (using the crossover function defined above), and the two results are added to the population. This is repeated until the population returns to its full size. A new gene created during this step is not included in the breeding pool, as it is possible to create genes which are worse than their parents.

At this stage, it would be possible to add a termination condition, perhaps checking if the maximum fitness has changed in the last several generations. This has been omitted however, as the analysis performed in the results section meant it was valuable to continue generating data for graphing.

Finally, the entire population, including those carried over from previous generations, are given the chance to mutate. Iterating over the whole population, each has a 5% chance of being allowed to mutate, by the function described above. The process is then repeated from the first selection step. On the final iteration, should it be reached, the mutation is prevented from occuring, to prevent mutating the final answer to a potentially worse one. This could be prevented as an issue by maintaining a copy of the fittest gene ever found at all times, but this creates additional processing overhead in each generation, and by experimentation has proved unnecessary.

Hill Climber

The hill climbing algorithm simply works by generating a random gene as in the genetic algorithm, then repeatedly calling the mutation function on the gene and either discarding the mutation or keeping it depending on whether its fitness increases. The function would ordinarily be terminated when some number of iterations pass without a change in fitness, but for ease of analysis this has been changed to a fixed number of total iterations.

Random solution

The random solution generates many genes each iteration, over many iterations (both of these values of “many” are variable), and outputs the highest fitness it locates. It was decided to split the random generation into iterative steps (as opposed to generating some fixed number of solutions and returning the maximum) in order to make it easier to graph the algorithm’s progress over time.

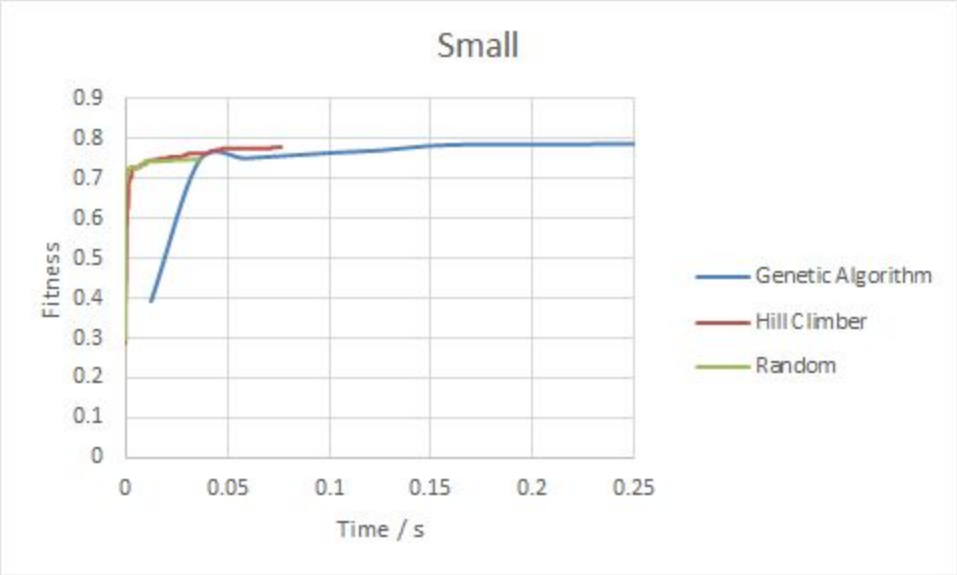
Results

To evaluate the developed algorithms, each was applied to the small and big fault matrices 10 times. Over the 10 runs. time elapsed per iteration(/generation) and maximum fitness per iteration were recorded, and the averages of these across all 10 were exported to Excel.

Small Fault Matrix

The small fault matrix had 216 tests and 9 faults. The tuple size was 6.

Genetic Algorithm: 1000 chromosomes, 100 generations.
Hill Climber: 2000 steps.
Random: 1000 random tuples.



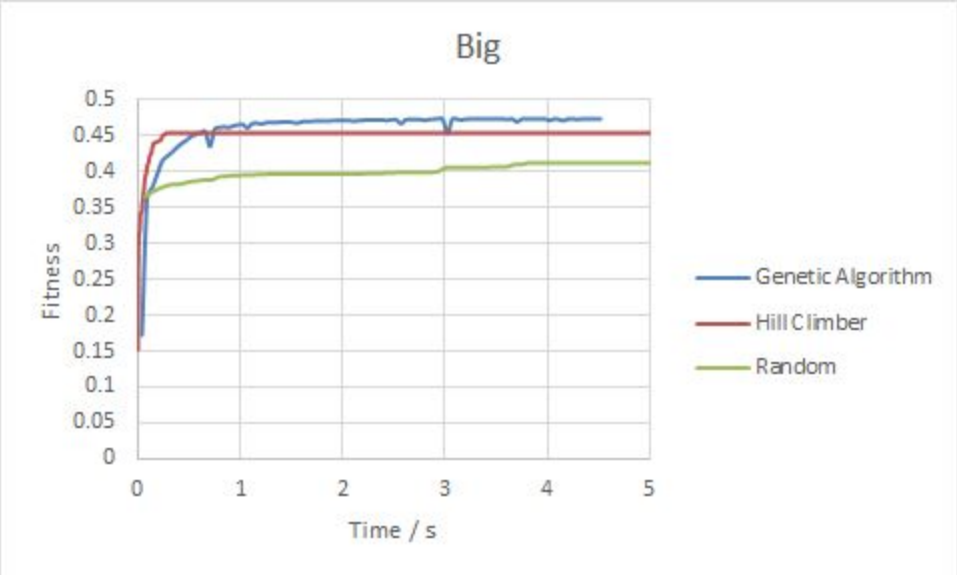
In this example, neither the genetic algorithm nor the hill climber are significantly better than random. Random breaks the 0.7 threshold in less than 1 ms, something that takes the hill climber twice as long and the genetic algorithm four times as long. Furthermore, the differences between random's fittest solution (0.75) and those of the genetic algorithm and the hill climber (both 0.78) are small. The best solution here would likely be the hill climber, as it reaches a peak solution in a fraction of the time of the genetic algorithm.

It can be inferred that the complexity of the problem is not high enough to justify the use of the more complex algorithms.

Big Fault Matrix

The big fault matrix had 1000 tests and 38 faults. The tuple size was 7.

Genetic Algorithm: 1000 chromosomes, 100 generations.
Hill Climber: ~3000 steps.
Random: 1000 random tuples.



This example was a better demonstration of the characteristics of the genetic algorithm and the hill climber and why they can be better than random. The hill climber's initial angle of ascent is the steepest but it finds a local maximum (0.45) about a quarter of a second in and stagnates. We can infer from this that the final change needed to reach the peak is a very specific one, so in this case more time would be required. Alternatively, it is possible that some runs of the algorithm reached the peak while others were caught by a local maxima.

The genetic algorithm is slower at first but overtakes the hill climber and goes on to find something fitter (0.47). There is almost no case to be made for random. It is worse than the others from start to finish and the difference between its fittest solution (0.41) and theirs is significant.

Interestingly, the genetic algorithm is the only one which can get worse rather than strictly better. There are at least five times when the blue line noticeably dips, indicating that the fittest chromosome(s) from the previous generation are lost. This is because even though elitism maintains the fittest 50% of the population, every chromosome is still subject to mutation. However, it recovers very quickly, indicating other members of the population are able to catch up efficiently.

Summary of Results

In general, while the genetic algorithm performed well, it was outperformed in terms of speed by the hill climber and random. The hill climber's performance in the small data set can likely be attributed to the fact that there were fewer local maxima in the problem space. The true strength of the genetic algorithm comes from its ability to overcome these local maxima and locate the best solutions, which is exemplified by the fact that it reached the highest fitness in the larger fault matrix, while the other two did not.

It is clear from the results of the smaller fault matrix, and the poor performance of the genetic algorithm therein that in small search spaces genetic algorithms can be unnecessary.

In time limited scenarios which are not very simplistic, both genetic algorithms and hill climbing algorithms have clear advantages over random, but given unlimited time there are scenarios where random may eventually break local maxima that the other two algorithms, especially the hill climber, may become stuck in.

Finally, on the large data set, the average value generated by the genetic algorithm of 0.47 was very slightly lower than the maximum observed value of 0.48, implying that the algorithm does not always reach the highest value. This could be as a result of the fitness function, or a result of limiting the run time for analysis. Nonetheless, it performed better on average than the hill climber and random solutions.

Lessons Learned

- The smaller the search space, the better the result of randomly searching it.
- Genetic algorithms and hill climbers, particularly the former, can be excessive if the problem space is too small.
- The weakness of hill climbers is that they can get stuck at local maxima. This creates a significant advantage for genetic algorithms, which do not have this limitation (as long as the fitness function is correct).
- Genetic algorithms can lose their fittest chromosomes but tend to find them again very quickly.
- Genetic algorithms can benefit from keeping the current fittest solution separately from the population, in order to reduce the impact of negative mutations.

Peer Assessment

Both participants contributed 50% to the project.