Regression Testing - Liam Devlin & David Smith

Regression testing is a method of ensuring that individual changes made to applications are not disrupting its overall performance. The large scope of regression testing means that it can often take extensive periods of time to be carried out. Therefore it is essential that the order of testing is optimized to ensure maximum efficiency in fault detection.

Chapter 1: Overview of problem

1.1 Theory

Our project was concerned with an evolutionary algorithm and a path-based hill climber search method to analyse two data sets. These data sets consisted of small and large series of tests and the respective faults the tests detected. The task was to identify the optimal order in which the tests could be run so the maximum number of faults could be detected in the quickest time.

To prioritize test cases, we had to have a way of quantifying individual test suite performance. There exists several relevant performance indicators, however we were primarily concerned with the rate of fault detection. The way we evaluate this is by considering the average percentage fault detection, or APFD, of an individual test suite. This metric measures the weighted average of the percentage of faults detected by each test, over the life of the test suite. We can see the formula for this below:

$$APFD = 1 - \frac{TF_1 + TF_2 + TF_m}{nm} + \frac{1}{2n}$$

Here *TFi* represents the first individual test case which reveals fault *i*, for *n* test cases covering *m* faults. Essentially, the APFD gives us an idea of how many faults are being picked up in the quickest time. With the APFD function performed on high numbers of test suites, we had to account for the possibility of zero faults being detected. If this was the case, the APFD would have to return zero. The reworking of the formula can be seen below:

$$APFD = 0 = 1 - \frac{TF_1 + TF_2 + TF_m}{nm} + \frac{1}{2n}$$

$$0 = 1 - \frac{x + x + x}{nm} + \frac{1}{2n}$$

$$0 = 1 - \frac{x(m)}{nm} + \frac{1}{2n}$$

$$\frac{x}{n} = 1 + \frac{1}{2n}$$

$$x = n + \frac{1}{2}$$

From this restructuring we can deduce that when an error does not occur in a given test suite, we must substitute the total number of tests in the suite plus a half. This also accounts for the possibility that a whole test suite will cover no faults at all, as when this occurs, the return of the formula will be zero.

1.2 Structure of approach

1.2.1 Genetic Algorithm

Considering the smaller test set initially, we used a predetermined test suite size to generate a random population of five test suites. After inputting the above APFD formulae we could rank and sort the population based on respective APFD scores. The population was sorted in a descending manner with the highest APFD first which would aid selection.

An elitist selection method was then used, favouring the top 20% of the population at each subsequent generation. Using a weighting of 90%, the top performing parent pool was favoured for parent selection. However to maintain diversity, parents of a lower standard were occasionally chosen. The decision to crossover was also based on a 75% probability. Parent candidates were mated to produce, ideally, better children candidates. Using a 2-way tournament-based approach, the children produced from the parents are compared with one another and the child with the higher APFD is chosen. The new children are then passed through a mutation function, employed to deter stagnation in the population's diversity. Like the crossover rate, this was also based on a probability rate but was considerably lower at 5%.

1.2.2 Hill Climber

As an alternative approach we ran a basic hill climbing method. The hill climber does not use evolutionary concepts and is concerned with finding locally-best solutions instead of globally. Therefore the method has a tendency to languish upon local maxima instead of continually searching for the overall best solution.

Instead of generating a population, a hill climber considers one arbitrary solution and then a set amount of neighbours which are similar to the initial solution, having only incremental differences. Beginning with the arbitrary solution of a test suite, a single change is made (a random test case is chosen and substituted for a random test from the test pool). If the new test suite is of a higher APFD, it will replace the previous as the next "current" solution.

As previously mentioned, getting "stuck" on a local maxima is a common issue for hill climbers. In order to prevent this, we implement a continual self-assessment method. The most recent "best" solutions are collected in an array and monitored: if the score becomes constant across the array then convergence has been reached and a new, randomly selected solution is chosen. The size of this array is 50% of the total iteration count. However, this can result in drastic decreases in APFD rates, resulting in lower-performing solutions, as demonstrated in our results.

1.2.3 Random Search

We also implemented a random search to characterise the previous two algorithms and see how they performed against a fully random, uninformed search. Details on the random search can be found in the following section.

Chapter 2: Implementation

2.1 Genetic Algorithm

A variety of collection databases were used throughout our code. The initial use of dictionaries was a practical decision, with the nature of the refined test data (test numbers, faults detected) intuitively aligning with the ordering of a dictionary (key, value). We could then easily sort the dictionary into an ordered population. Once sorting by APFD had been completed, the dictionary was then transformed into an array of tuples.

The rationale behind our specific crossover and mutation rates can also be explained here. A crossover probability of 75% was prevalent in relevant literature and was an appropriate choice for this case. Concerning the mutation rate, there was slightly more variation in recommended levels, with most sources quoting rates between 1 and 10%. Choosing an average of 5% was a reasonable decision and produces a desirable convergence speed and greater genetic diversity.

An important factor to consider is the possibility of duplicates occuring during the crossover or mutation stage, and thus a "duplicateCheck" method was created. The method works by converting an existing testSuite into a dictionary, which only allows new entries to be appended. Once this check has been complete, if the length of the "check dictionary" is not the same as that of the original test suite, we pick a new test at random until one that does not exist in the original test suite has been selected. We then add this to the check dictionary and convert it back to a list of tuples.

To visualise the results produced by the algorithm we used the *matplotlib* package. This allowed us to make plots of the varying methods against a comparative random search on a single chart, as well as allowing us to visualize the spread of each algorithm

2.2 Hill climber

For the hill climber method, we share some key aspects with the genetic algorithm (GA). The evaluation of test suite APFDs is performed in the same way as in the GA. For continuity, we also generate a number of neighbouring solutions equal to that of the population generated in the GA.

The remainder of the approach is novel. We can see how our solution is changed, carried out by randomly altering a single test case in the suite. In the subsequent main function we determine whether or not to adopt the new test suite or to keep searching. We can also see our failsafe for

remaining trapped on a local maxima with an automatic restart if the solution APFD score plateaus. This is achieved by setting a maximum number of iterations which the solution

The same “duplicateCheck” as used in the GA is used to ensure no duplicates are found in a given testSuite generated by the hill climbing algorithm.

2.3 Random search

Our comparative randomized search is relatively simplistic, with the outline essentially mirroring that seen in the GA except all evolutionary steps (sorted population, selective mating, crossover, mutation) are omitted. The random search generates a population of test suites and selects a solution at random each generation.

Chapter 3: Results

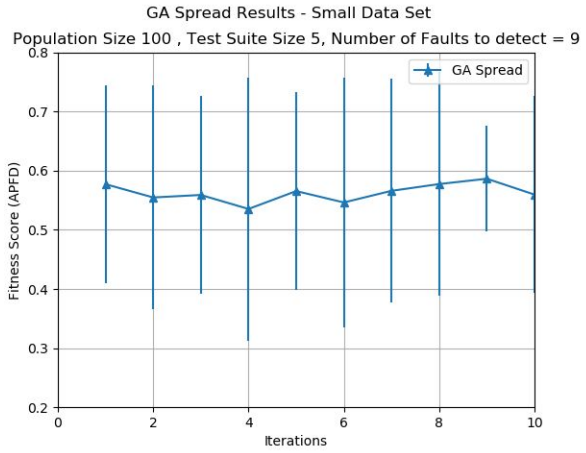

Figure 1: APFD spread for GA (small data set)

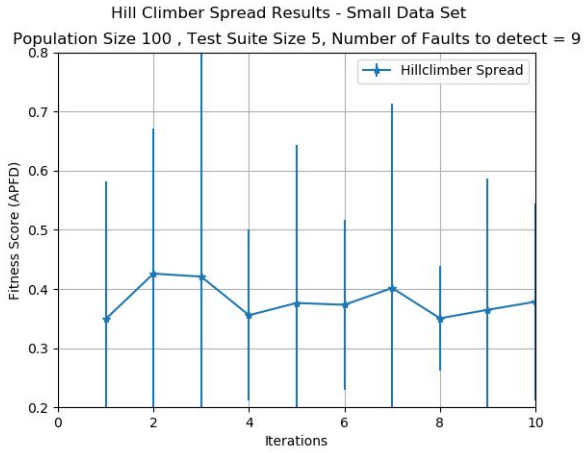

Figure 2: APFD spread for hill climber (small data set)

Figure 3: Mean variation of all methods (small data set)

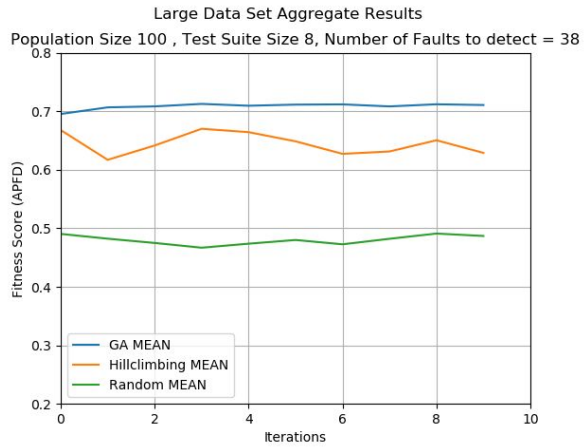

Figure 4: Mean variation for all methods (large data set)

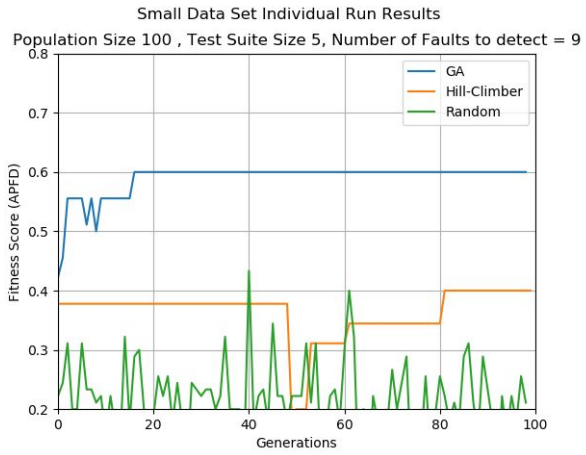

Figure 5: Individual runs of all methods (Small data set)

Figure 6: Individual runs of all methods (Large data set)

3.1 Genetic Algorithm

Figure 1 displays the spread of the GA and mainly shows the variation of the minimum and maximum APFD found in the GA solution. We can see the solution is nominal through the low range of each iteration, showing how the solution remained close to the average. The spread for the large data sets has been excluded but the trend was mirrored in these results.

Figures 3 and 4 displays the variation of the GA APFD for small and large data sets respectively. With the average consistently remaining between 0.54 and 0.6 for the small data set and plus 0.72 for the large, this is a nominal result.

3.2 Hill Climber

Figure 2 displays the spread of the hill climber method and demonstrates a fairly erratic variation of minima and maxima. Similar to the GA, the results were also mirrored in the large data set.

Figures 3 and 4 exhibit the variation of the hill climber APFD for small and large data sets respectively. We can see a greater variation in these results, with no real convergence occurring. This is possibly due to our approach in dealing with local maxima, which involves resetting the current solution after a time relative to the total number of generations has passed.

3.3 Evaluation of results

3.3.1 Genetic Algorithm

Figures 3 and 4 clearly demonstrate the GA's high APFD rating across both data sets compared to the hill climber and random searches. In Figures 5 and 6, we can see clearly the GA's swift approach to the maximum APFD. With a strong upward trend for the first 20 generations, it arrives at an APFD of approximately 0.6 which it hovers at for the rest of the run. For the large data set in Figure 6 we reach an APFD of approximately 0.72 which is consistent for the rest of the run. This rapid

convergence on the optimum solution could be connected to the strong bias in place during selection, towards the more desirable population members. It could also be attributed to the relatively low population count when compared to the amount of test cases.

Overall the GA performs optimally and outperforms both the hill climber and random methods. This was to be expected when using a more informed search method and demonstrates the suitability of the GA for regression testing.

3.3.2 Hill Climber

The hill climber performs fairly well with a general upwards trend for portions of the run, however the ability to reach desirable APFD levels is inhibited by the automatic resets. This is exacerbated by the relatively slow evolution of the solution, seen clearly in both small and large data-sets by extensive flat-lining, making it highly susceptible to the resets. The hill climber repeatedly lingers too long and therefore is reset by the failsafe. Therefore, whilst for appropriate applications the hill climber method could be a reliable method, it is an undesirable and fairly unpredictable approach compared to the GA for use in regression testing.

3.3.3 Random

The random solution performance matched expectations, with variations in peak and trough heights across the run. No real trend can be visualised, demonstrating its poor performance compared with the GA and hill climber methods. This clearly shows the benefits of more informed search algorithms.

Chapter 4: Contributions

Liam Devlin: 50.0%
David Smith: 50.0%