Advanced Topics in Software Engineering - Handin 2

Jack Neilson

October 14, 2018

1 Test Case Prioritisation

1.1 Introcudtion

Test case prioritisation is an interesting problem to be considered when deciding which tests should be run on a given program. At it's core, the problem is simply that running every test on a program is infeasible and as so the subset of tests that find the most faults should be run instead. This subset is not trivial to find however, as there is no way of knowing ahead of time exactly what faults a given program has - meaning the tests that would find these faults are also not know.

1.2 Selection

To solve this problem, a heuristic for how useful or otherwise a test will be must be used. Examples include coverage (i.e. the percentage of code that the test actually tests), and the average percentage of faults found (APFD), which may be found by running the test against programs with only known faults. These metrics may then be used to evaluate which tests are the most useful, and allows us to create suites of tests that give us the all of the most useful tests in a given timeframe or other such budget.

1.3 Ordering

The tests selected should be ordered such that the test most likely to find faults is ran first. This is another difficult task, as again the faults are not known beforehand. The ordering is important as some of the more important tests may render the rest of the set obsolete - for example, if the first test finds a buffer overflow in a cryptography module then presumably it would need to be fixed before the set of tests is ran again (which may then alter the optimal set of tests). This is where the APFD metric comes in handy, as it gives a higher score to the sets that order the more important tests first.

2 Implementation

2.1 Overview

To solve the problem outlined above, I implemented a genetic algorithm which randomly selects n tests from a set of tests which have been ran against a program with

9 known faults to give a single node, then repeated this process until I had the desired population size. This population was then evaluated, cross-bred and possibly mutated for a certain user-defined number of generations until the best node of the final generation is returned. Given a large enough sample size and number of generations, this node should contain the optimal (or at least, a close enough approximation) selection and order of tests from the given set.

2.2 Representation

Initially, each test is contained in a comma-seperated format. This is imported and transformed in to a set of nodes, which makes up the population. Each node of the population is a suite of tests, and is represented as a list containing a custom test object (definition below).

```
class Test:
    def __init__(self, name, data):
        self.name = name
        self.data = data
```

2.3 Functionality

2.3.1 Population Generation

As mentioned above the population is generated by importing the entire data set, then continually selecting tests to make up nodes of the population until the desired population size is reached. Care is taken not to allow duplicate tests in each test suite.

2.3.2 Evaluation

To evaluate a node in the population the APFD of its tests is calulated using the formula:

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

Where T is the position of the test in the suite, F_n is a fault, n is the number of tests in a suite, and m is the number of total possible faults a test can find. For faults that are not found in the test suite, T is equal to the number of tests in the suite plus one.

This formula gives a value between 0 and 1 to indicate how good a suite of tests is. It accounts for ordering, as tests which find more faults give a higher APFD if ordered first. It also accounts for tests finding duplicate faults, as only the first test to find a fault is included in $\sum TF$.

2.3.3 Crossbreeding

Care must be taken when breeding two nodes in a population to avoid duplicates, as running the same test twice would be pointess. To accomplish this, when breeding two nodes the first k elements of $node_1$ is taken, then $node_2$ is iterated over and elements which are not in the first k elements of $node_1$ are added to the new node, until the new node is the same size as both of its parents. In this implementation, k is equal to the floor of half of the lenght of the parent i.e. $k = floor(\frac{len(node)}{2})$

2.3.4 Mutation

Since the problem to be considered is an ordering problem rather than a selection problem, mutation simply swaps the ordering of two randomly selected elements in the list of tests belonging to a test suite. This is an important feature, as it allows the algorithm to be complete (given infinite time) and therefore also optimal. In real terms, they make the algorithm much more consistent in finding a decent solution as the population takes longer to converge (and also converges on a solution with a higher APFD on average).

2.4 Other Features

2.4.1 Termination

It is impossible to know the optimal set and order of solutions ahead of time, and generating and evaluating each possible permutation is obviously computationally infeasible. As such, three other termination conditions present themselves - population convergence, APFD threshold, and number of iterations. Testing for population convergence is computationally expensive, and is required every iteration. The solution is already very expensive to run, so adding more expensive code to terminate would not be a good idea. As for APFD thresholding, there is no guarantee that the algorithm would ever find a solution from the population above the threshold, allowing the program to run infinitely. Therefore, the decision was made to use number of iterations as the termination condition.

2.5 Ancestor Selection

The elitism strategy was used when selecting ancestors for breeding and possible mutation. To do this, the best n candidates from the population were taken and crossbred with each other. A helper function was developed to take the best n from a given population (shown below)

```
# Returns the best n candidates from a population (elitism strategy)
def get_best_n(population, number):
    return sorted(population, key=lambda x: evaluate(x),
        reverse=True)[:number]
```

3 Results

3.1 Genetic Algorithm

3.1.1 Small Data Set

Below is a table of the APFD of the optimal test suite and the selection of tests in that suite after 1000 iterations with a test suite size of 5, a mutation rate of 0.05, an initial population size of 50 and 10 ancestors taken per iteration. The population was selected from the small data set.

Run No.	APFD	Time Taken (s)
1	0.9198	0.638
2	0.8333	0.645
3	0.8333	0.629
4	0.8333	0.645
5	0.9198	0.634
6	0.9198	0.617
7	0.8333	0.647
8	0.8333	0.639
9	0.8333	0.640
10	0.8333	0.633

3.1.2 Large Data Set

Below is a table of 10 tests ran under the same conditions as the test above on the large data set, with the exception of the initial population being increased to 500.

Run No.	APFD	Time Taken (s)
1	0.5217	2.088
2	0.5509	1.746
3	0.5825	1.782
4	0.5970	2.256
5	0.6086	1.887
6	0.6068	2.004
7	0.5940	1.639
8	0.6163	2.293
9	0.6421	2.386
10	0.6579	2.142

3.2 Hill Climber

3.2.1 Small Data Set

Below is a table of 10 tests using the example hill climber algorith. The neighbour generation function was deliberately designed to return a list of few possible neighbours to cut down on computational cost. Again, the test suite size was 5.

Run No.	APFD	Time Taken (s)
1	0.3444	0.002
2	0.7000	0.001
3	0.3444	0.002
4	0.7889	0.002
5	0.7000	0.001
6	0.7000	0.002
7	0.7889	0.002
8	0.1444	0.001
9	0.2556	0.001
10	0.3444	0.001

3.2.2 Large Data Set

Again, this is a table of 10 tests using the hill climber against the large data set.

Run No.	APFD	Time Taken (s)
1	0.1052	0.015
2	0.0526	0.015
3	0.0368	0.015
4	0.2211	0.015
5	0.1158	0.015
6	0.2368	0.016
7	0.0316	0.015
8	0.0105	0.015
9	0.0105	0.015
10	0.0895	0.015

3.3 Random Search

In the interest of brevity, random search was only ran once with each data set for 1,000,000 iterations to reduce variance.

3.3.1 Small Data Set

Run No.	APFD	Time Taken (s)	Avg. APFD
1	0.8556	34.243	0.3967

3.3.2 Large Data Set

Run No.	APFD	Time Taken (s)	Avg. APFD
1	0.3895	85.8442	0.0711

4 Analysis

4.1 Genetic Algorithm

Overall, the genetic algorithm performed reasonably well. For the small data set, every run was performed in under a second, and the mean APFD was 0.85925. Every run also produced a decently viable test suite, with both the median and minimum APFD being 0.8333. I theorise this could be improved by adding some tests from the full data set that do not exist in the population during mutation, as this would allow the program to navigate the full search space given infinite iterations.

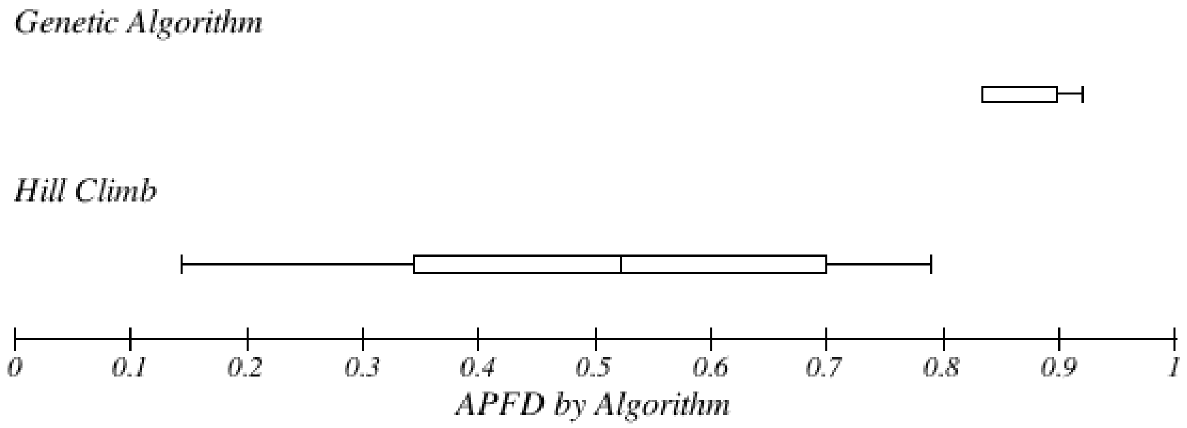
4.2 Hill Climber

The hill climbing algorithm performed much worse than the genetic algorithm. This is largely due to it searching much less of the search space, as well as the fact that it may get stuck in a local maxima. This is evidenced by the time it took for each run - the median time taken for each run over the small data set was 0.001, orders of magnitude less than the median time taken for the genetic algorithm (0.64525 seconds). The fact that much less space has been searched is reflected in the mean APFD value, which is only 0.5111. What makes this worse is the massive variance in the APFD of the test suites this algorithm produces, ranging from 0.1444 to 0.7889.

4.3 Random Search

As expected, the random search performed the worst out of the three algorithms. It managed to produce a test suite with an APFD of 0.8556 after 1,000,000 iterations, however this took 25 seconds to do and is only slightly higher than the minimum value of the genetic algorithm. What's more, the mean APFD of a test suite generated by random search was an abysmal 0.3967, showing that random search is clearly not a viable method of generating a test suite. This problem becomes much more apparent when using the large data suite, as it took over a minute to produce a suite with and APFD of only 0.3895.

4.4 Visualisation


Genetic Algorithm
Hill Climb
APFD by Algorithm