GA – Regression Testing Report – Garry James McBride

Introduction/Overview

Regression testing is the process of testing changes to computer programs to make sure that the older programming still works with the new changes. Supplied with test results from multiple in two matrix files (215 tests in the small matrix file & 10897 tests in the larger matrix file) the objective is to work out a way of calculating which test sequence will produce the best results regarding fitness of the faults and how quickly they are found. With regression testing comes a high cost, but that was not considered with this assignment, an opportunity to find the highest fitness possible has presented itself.

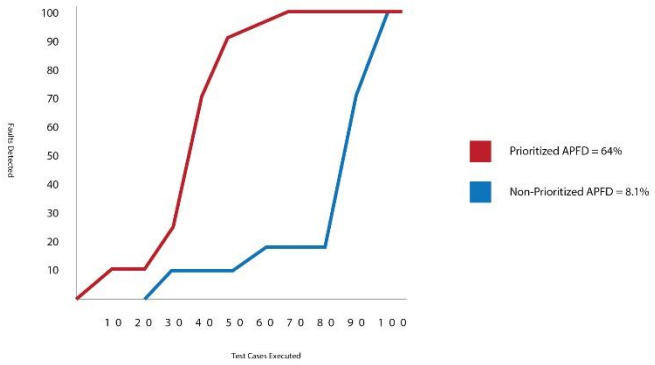
A Genetic Algorithm was the implementation chosen to find the results. Genetic Algorithm (GA) is defined as a search technique used in computing to find exact approximate solutions to optimization and search problems. Used in finance, GA's are a reliable tool for finding results for the issue at hand, precisely we define a fitness function, then we try iteration after iteration to generate good (that maximise or minimise the fitness function) solutions(genes). The GA was written in Python 3.

Fitness

Using Python to split the data into elements (keys: test number, Values: test results) and combined into a random choice of 5 tests together, a population is created so many test results are together and also mixed with others, to help find the best possible results from not only all possible results from a certain range of 5 suites, but also results from those suites mixed with others, and this is only to start with.

Solutions have to measure in terms of a rating, when a target has been used, the fitness value will be compared or matched regarding its similarity to said target, but when there is no target, and data to take into consideration, the fitness function must be calculated with a more specific value, a percentage takes into account possible difference among sizes of data and their fitness. Being more accurate than counts, percentages in fact allow us to compare groups that not similar and are more statistically significant, and superior.

Average Percentage of Faults Detected (APFD) assumes we have we have a variety of test suites, with faults they are going to uncover. APFD metric denotes the weighted average of the faults detected. The metric of APDF is widely adopted in evaluating test case prioritization techniques. APFD is a comparison drawn between prioritized and non-prioritized cases, they show the value collected or obtained is more for prioritized than non-prioritized. Prioritized is more effective for finding the best possible fitness value, shown below is an example of both cases:



The diagram below is an example of the APFD formula with a combination of 5 tests with 9 values:

T1	0	0	0	(1)	0	0	0	0	(1)
T2	(1)	(1)	(1)	0	0	(1)	(1)	(1)	0
T3	0	0	1	0	(1)	0	0	1	0
T4	0	0	0	1	1	0	0	0	1
T5	0	0	1	0	1	0	0	1	0

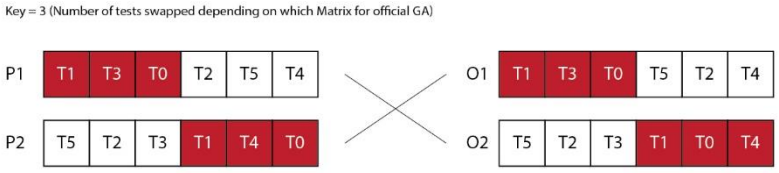
$$\textbf{APFD Formula} = 1 - \frac{2+2+2+1+3+2+2+2+1}{5*9} \qquad 1 + \frac{1}{2*5}$$

Selection

The selection process chose a random solution of 5 tests combined, based on the highest and then does the same, selecting another randomly selected test suite of solutions, creating two parents that can then be crossed over and lead to mutation.

Crossover

Crossing the solutions, highly dependent on depensation, the solutions must ensure duplicates aren't made when swapping. The main function of crossover is to combine the genetic information of two parents to generate new offspring, as seen below:



Mutation

The mutation (like crossover) is also highly dependent on representation. Small changes to the solution are made, increasing fitness value in smaller places since the value is based on the solution (as a whole). Finding aspects of the solution that can be improved can be done with crossover, but when its smaller, and more precise, the implementation of mutation had to be introduced to the Genetic Algorithm.

Overview Summery

The algorithm consists of functions, that are called at the end in a function named "algorithm". Functions that do not appear in the algorithm function, are called inside other functions that are called inside the Algorithm. As an example (this appears in both Regression and Hill Climber) create_population is called inside the Algorithm but individual_combine is called inside create_population, thus functions call upon other functions so that in the algorithm, they are present working separately or in the case above, together.

Implementation

Using Python, the Regression Testing Algorithm was designed in JetBrains PyCharm where the code highlights and informs the user, so they can understand what they have wrote and where it is being used. Variables can be highlighted, then seen throughout the file, this helps the user find where certain functions are linked or implemented in whether it be another function. The Software is colour coded, structuring the file for the user to understand what certain aspects are, if they are comments or code that is doing a certain task. PyCharm also help making the code look presentable with telling the user to use double spaces, or warnings of potential syntax errors before running the code and finding an error, and then must go back and fix said error. Although it does give detail on which line, or function is not recognized or there is an issue with it.

Regression Test

The GA for Regression testing starts by reading one of the matrix files, splitting the data into lines and then separating the elements with commas. A dictionary was then created to separate the test numbers from the numbers if a fault was detected or not. The dictionary then recognizes the test numbers as Keys, and then test results as values.

A function was then created to combine 5 randomly selected tests together, which before this a population was created (depending on which matrix) in a range of number of populations of 5 randomly selected tests together. The fitness function (based on the APFD Formula) calculates the fitness of the randomly selected combined 5 tests, and the population is sorted to show a growth in fitness value starting from the weakest to the strongest.

To increase fitness in certain parts of the tests, cross over and mutation were created, separating the tests into 2 parents, and then into children so they can then be mutated further to allow the fitness value to increase in the smallest areas of the test suites. They are then added to a New generation of crossed over test suites which are then mutated before being called upon in the Algorithm function.

The Algorithm function prints generations starting from the weakest found test suite after being mutated, and then proceeds to find the strongest. After 100 generation of being the same fitness function, the Algorithm stops, and the strongest test suite is presented beside its fitness value.

Hill Climber

The GA for Hill Climber (same as the regression test) starts by reading one of the matrix files, splitting the data into lines and then separating the elements with commas. A dictionary was then created to separate the test numbers from the numbers if a fault was detected or not. The dictionary then recognizes the test numbers as Keys, and then test results as values.

A function was then created to combine 5 randomly selected tests together, which before this a population was created (depending on which matrix) in a range of number of populations of 5 randomly selected tests together. A function is then created to swap around the suite, so faults are detected quicker, this is like crossover but not as effective, as it is only the tests in one suite being swapped around, unlike cross over where other suites are mixed from two parents into their children (new suite).

A function is then created to select another randomly selected 5 tests, combined in a solution, so the original selected solution can be compared with the new one from this function, if the fitness is higher, the newly selected neighbour will replace the first selected suite, and if the fitness is less than the first, the climb will finish with the first selected combination, which is unlikely unless the

first suite has a high fitness to start with, making the odds of finding another with a higher fitness very unlikely.

The fitness function (based on the APFD Formula) calculates the fitness of the randomly selected combined 5 tests, and the population is sorted to show a growth in fitness value starting from the weakest to the strongest.

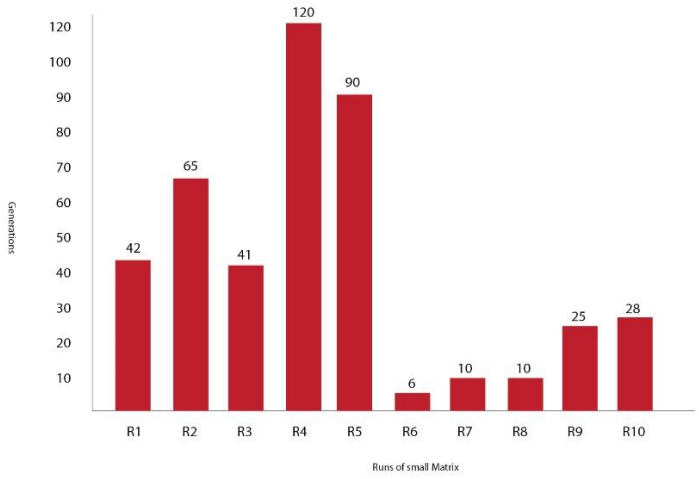
An algorithm is now implemented so that the highest fitness test suite is selected and can go no further, to go further run the code again and it could find a higher fitness test suite leading to another.

Results

After around 40-50 generations the Algorithm found the best possible fitness value in the small matrix, as example is shown below:

generations 39 best individual is ['t125', 't147', 't211', 't150', 't156'] with fitness 0.7666666666666667

The Algorithm is designs to end after 100 generations of the same fitness value. The Algorithm was executed 10 times to ensure the results weren't solely down to chance. After 10 runs, the average fitness value was calculated after 43.8 Generations. The results vary per run as seen below for the small matrix shown in red:



From 1-10 the runs seem to start off with a low count of generations, and then seem to jump high, to only come back down even lower than the initial run, only to them start to climb again. The results are strange as its almost like they are taking longer but drastically increase in detecting the highest fitness value from around 100 generations to 6. The generations then start to climb higher but as much slower pace than the first rise between runs 1-5. The data below shows the generations before 100 was deducted because of the function in the algorithm where after 100 generations of the same fitness value it stops.

R1 generations 42 best individual is ['t125', 't147', 't211', 't150', 't156'] with fitness 0.7666666666666667

R2 generations 165 best individual is ['t105', 't147', 't167', 't134', 't150'] with fitness 0.7444444444444444

R3 generations 141 best individual is ['t83', 't167', 't150', 't147', 't82'] with fitness 0.7666666666666667

R4 generations 220 best individual is ['t215', 't147', 't150', 't178', 't201'] with fitness 0.7666666666666667

R5 generations 190 best individual is ['t81', 't150', 't179', 't147', 't51'] with fitness 0.7666666666666667

R6 generations 106 best individual is ['t105', 't150', 't166', 't147', 't175'] with fitness 0.7666666666666667

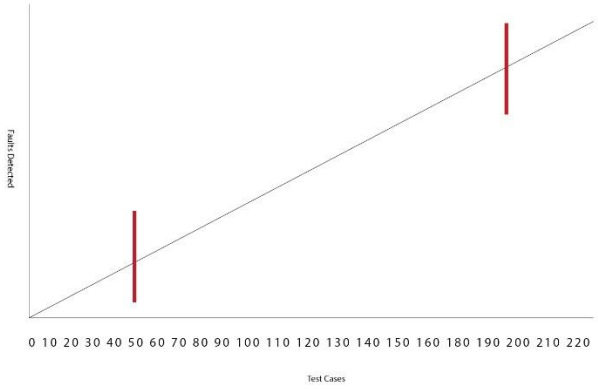
R7 generations 110 best individual is ['t105', 't150', 't147', 't146', 't167'] with fitness 0.7666666666666667

R8 generations 110 best individual is ['t134', 't150', 't147', 't202', 't151'] with fitness 0.7666666666666667

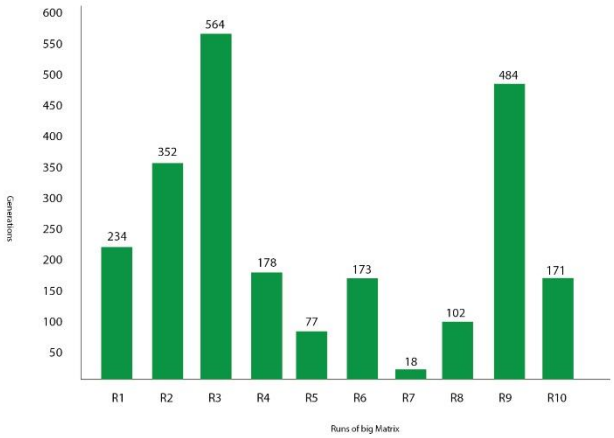
R9 generations 125 best individual is ['t79', 't171', 't150', 't147', 't49'] with fitness 0.7666666666666667

R10 generations 128 best individual is ['t111', 't147', 't203', 't150', 't49'] with fitness 0.7666666666666667

From the results, the average fitness value is 0.76, and the most common test is T150, it appears in every suite, along with T147. These seem to be regulars within the suites. Both tests seem to be near each other as well in the suite, also T150 is in the second element from Run5 – Run8, and this is where shown in the graph the generations start to improve. The Suites also seem to be around half way through the data set (shown below), since ranging from the first Test (T1) to the last Test (T215) faults seem to be have been detected more the higher the test number. The test results in the middle of the data set look to be more inclined to accept mutation and replace weaker parts of the suites better than more faults detected with the higher tests.



Shown below is the runs recorded from the bigger matrix. Like the smaller matrix (shown in green), the after a few runs the bigger matrix improves drastically, and has an average of 242 generations. Once again, the results vary from runs, it is better to do as many runs as possible for an average of generations as the results seem to vary but they do not seem to jump to crazy results past a certain generation. *(See next page for graph)*



R1 generations 334 best individual is ['t1042', 't10363', 't1014', 't10043', 't10145', 't1018', 't10487', 't10552', 't10383', 't10369', 't10035', 't10554', 't10204', 't10438', 't10226', 't10185', 't10200', 't10598', 't1078', 't10751', 't1008', 't10125', 't10658', 't1044', 't10799', 't10236', 't10711', 't10819', 't10532', 't10695', 't1029', 't1003', 't10537', 't10645', 't10830', 't10505', 't1011'] with fitness 0.7076813655761024

R2 generations 452 best individual is ['t1083', 't10057', 't10122', 't10483', 't10371', 't1014', 't1080', 't10359', 't1042', 't10087', 't10577', 't10036', 't1044', 't10383', 't10625', 't10878', 't10136', 't10558', 't10402', 't10332', 't10163', 't10811', 't10497', 't10276', 't10030', 't10405', 't10151', 't10643', 't10455', 't10819', 't10172', 't10890', 't10823', 't10555', 't10889', 't10630', 't10719'] with fitness 0.7155049786628734

R3 generations 664 best individual is ['t10057', 't10122', 't10614', 't1083', 't1014', 't1080', 't10613', 't10618', 't10749', 't10143', 't10411', 't1018', 't10889', 't10551', 't10278', 't10363', 't10791', 't10272', 't10306', 't10508', 't10383', 't10421', 't1085', 't1027', 't10163', 't10789', 't10663', 't10483', 't1023', 't10445', 't10428', 't10590', 't1086', 't10773', 't10680', 't10451', 't10404'] with fitness 0.7155049786628734

R4 generations 278 best individual is ['t10371', 't10129', 't1080', 't10123', 't10041', 't10421', 't1083', 't1033', 't10047', 't10035', 't10369', 't10313', 't10852', 't10463', 't10048', 't10746', 't10512', 't10887', 't10057', 't1015', 't10269', 't10875', 't10382', 't10287', 't10351', 't0', 't10394', 't10894', 't10675', 't10517', 't1', 't10023', 't10704', 't10181', 't10372', 't10541', 't10586'] with fitness 0.7055476529160739

R5 generations 177 best individual is ['t10369', 't1075', 't10383', 't10124', 't1014', 't10057', 't10490', 't10136', 't1045', 't1013', 't10133', 't10261', 't1042', 't10470', 't10506', 't10572', 't10756', 't10817', 't10423', 't10142', 't10051', 't10301', 't10271', 't10039', 't10244', 't10674', 't10277', 't10543', 't10725', 't10802', 't10733', 't10588', 't10560', 't10603', 't10591', 't10145', 't10694'] with fitness 0.7076813655761024

R6 generations 273 best individual is ['t10369', 't1083', 't10540', 't10057', 't10123', 't1080', 't1024', 't10113', 't10371', 't10363', 't1058', 't10356', 't10134', 't10538', 't10338', 't10795', 't10275', 't10566', 't10047', 't10778', 't10355', 't10528', 't10790', 't1042', 't1050', 't10885', 't10163', 't10244', 't10771', 't10459', 't10091', 't10079', 't10690', 't10000', 't10464', 't1031', 't10286'] with fitness 0.7083926031294452

R7 generations 118 best individual is ['t10371', 't1083', 't10057', 't10128', 't10565', 't10133', 't1009', 't10363', 't10328', 't10521', 't10139', 't10031', 't1040', 't10751', 't1010', 't10677', 't10881', 't10375', 't10292', 't10767', 't10749', 't10404', 't10858', 't10746', 't1015', 't1000', 't10500', 't106', 't10572', 't10', 't10662', 't10081', 't10063', 't10338', 't10234', 't1075', 't10305'] with fitness 0.716927453769559

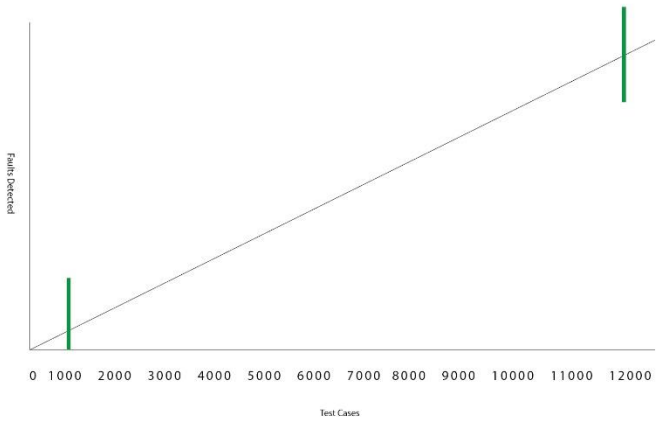
R8 generations 202 best individual is ['t1042', 't1014', 't10139', 't10363', 't1075', 't10369', 't10443', 't10145', 't10041', 't10147', 't1000', 't10480', 't10316', 't10536', 't10825', 't10464', 't10561', 't1078', 't10057', 't10587', 't10788', 't10646', 't10256', 't10547', 't10481', 't10097', 't10770', 't10027', 't10763', 't10543', 't1001', 't10516', 't10651', 't10845', 't10010', 't10270', 't10275'] with fitness 0.7147937411095305

R9 generations 584 best individual is ['t1083', 't10057', 't1014', 't1080', 't10122', 't10560', 't10033', 't10048', 't10397', 't10072', 't10416', 't10605', 't10147', 't10369', 't10817', 't10173', 't10435', 't10548', 't10180', 't10588', 't10864', 't10566', 't10102', 't10140', 't1018', 't10023', 't10432', 't10847', 't10160', 't10776', 't10642', 't10641', 't10152', 't10658', 't10440', 't1084', 't10130'] with fitness 0.7211948790896159

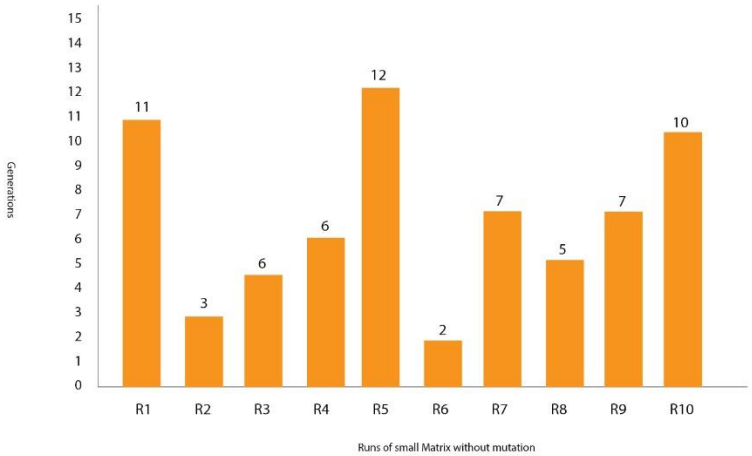
R10 generations 271 best individual is ['t1008', 't1083', 't10051', 't10363', 't10123', 't10424', 't1042', 't10551', 't10728', 't1018', 't10057', 't1087', 't10138', 't1033', 't10109', 't10304', 't10184', 't10642', 't10181', 't10248', 't10714', 't10416', 't10744', 't10446', 't10608', 't10506', 't10539', 't105', 't10390', 't10017', 't10786', 't10219', 't10567', 't10305', 't1016', 't10252', 't10820'] with fitness 0.702702702702702

From the results the best fitness value calculated was around 0.70, for a larger matrix, you'd think the results would detect more faults and the fitness strength would improve because of more tests

in one suite, 32 more tests to be precise compared to the smaller matrix. But it looks like this is not the case, its actually better to use less tests in one suite to calculate fitness, of the results, the reason for this is because mutation and cross over allow improvement in detecting faults, no matter how many tests are in one suite, and mutation would most likely have not covered the majority of the suite because of the scale of it.



Again, from the diagram shown above the tests that occur most are located not at the start or the end of the algorithm. I noticed that without mutation, the fitness value was calculated between 0.60 – 0.76, which I found strange as mutation would surely improve the fitness value by a large amount, but cross over has done a good job of detecting faults without mutation. This could potentially offer a good solution at a lower cost. Shown below is a graph for the small matrix without mutation (shown below).



Random searched was performed with both matrix's and had some interesting results. The fitness value with random search was not far off the Algorithm. The search was run 10 times on each matrix with the fitness value for small matrix around 0.65% and the bigger matrix received a fitness value of 0.57%. It seems the cost of running a full genetic Algorithm with functions like mutation or cross over, will find the best possible solution, but compared to the algorithm without these functions,

and also the implementation of a random search, unless the budget for regression testing is high, a good solution can be found without the added cost of a larger algorithm with functions to only increase the solution by less than a quarter.