Assignment 02 – Implementation Report

Author : Cristopher McIntyre Garcia
Email : cmcin019@uottawa.ca
S-N : 300025114

1. Explain your new search algorithm: Which algorithm you selected? Why? Explain the fitness function and the operators you have implemented for your algorithm.

The search algorithm chosen for the implementation component of this assignment is the hill climbing (HC) search algorithm. The reason being that this algorithm is simple yet effective at finding local optimums. Because we are generating input test cases, usually there are many local optimums that are viable options. And so our goal becomes to find a test case from the set of test cases that are optimally fit. Because of this, starting from a random vector and directly performing an exploitative search seems likely to find a solution faster than performing many explorative steps prior. Using randomness to continuously find new optimal solutions is only necessary when the solutions are very hard to find or if there aren't many of them.

We can add a termination constraint that does not allow the algorithm to end until it finds a solution that is optimal or if a certain number of random restarts have been conducted without finding a better solution. If none of these criteria are met, then the algorithm cannot terminate. It is important to have criteria that will allow the algorithm to terminate without the need to find an optimal solution in case one does not exist. Not finding an optimal solution means that the algorithm was unable to generate a test case that reached the branch we intended for it to reach. And so, this either means that the solution is hard to find or that this branch is impossible to reach. Either way, detecting this problem is important, and knowing which test case came closest is also important for further investigation.

The tweaking function used by the hill climbing algorithm can be implemented in a way that resembles the implementation of the exploratory method used by AVM. Firstly, starting at the beginning of a random vector, one will iterate through every element until a more optimal vector is found or all the elements have been checked. The way an element is checked is by adding or subtracting one from it and choosing the value that yields the best result if at least one of them is more optimal than the previous value. Once an element is modified, the local search returns to the first element and restarts the process of generating and comparing neighbours. This specific implementation makes our HC algorithm of the first ascent type.

The fitness of a test input is composed of its approach level to the desired branch and the branch distance. When comparing two vectors, their approach level will be compared first, and the vector with the lowest approach level will be considered more fit. If both vectors have the same approach level, in that case their branch distance will be compared. The vector with the lower branch difference will be more optimal. Note that only if the approach levels are the same will the branch distances be considered. This will allow the branch level to not overpower the approach level.

This is continued until a vector is found that has optimal fitness, that is, an approach level and a branch distance of zero. If this is not found and the algorithm reaches the end of the vector, the algorithm will perform a random restart. If the algorithm has performed a certain number of random restarts, then it will terminate and the most optimal vector will be returned.

2. Explain how you modified the existing implementation of avmf to implement your new algorithm. Describe which files and at what locations were modified and/or which new files you implemented?

Everything related to the fitness function stays the same, and the same *betterThan* function is used to compare different test inputs' fitness. This is because regardless of the search algorithm used, the goal is always the same: to generate test inputs that will reach a desired branch in an algorithm. Finding these test inputs becomes an optimization problem in which we try to minimize the approach level and branch distance as much as possible.

What changes is everything related to the search algorithm and the way they generate neighbours. A new class named *HillClimbingMethod* was created to house the search algorithm, similarly to how the *AlternatingVariableMethod* class (found in the same folder) houses the AVM algorithm. The algorithm is implemented in the *hillClimbingSearch* method, which takes in a single parameter. The parameter is the vector that we want to optimize. The fitness of the vector is calculated and compared with that of its neighbours until a better solution is found.

Finding new neighbours is done with the tweaking function that is implemented in the *performSearch* method from the *HillClimbingSearch* class. This is also a new class, and it is similar to the PatternSearch class (found in the same folder). The tweaking method modifies an element in the vector by adding or subtracting one from it. If the vector performs better when one is added to the element, the vector keeps the modification. If it does not perform better, then it checks if subtracting one from the element yields a better fitness. If that is the case, then the vector keeps the modification. If no improvements are made, the element in the vector stays unchanged.

Once the tweaking is done on an element, the fitness is calculated and compared to that of the original vector. If the new fitness is better than the original, then the new vector becomes the current vector and the algorithm restarts the search from the beginning. If no better fitness was found, the next element in the vector is tweaked, and so on.

3. Explain how we can run your new test generation algorithm on example java classes in the avmf directory.

Please run the following command from the *avmf-master* directory:

java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Triangle 1T

This command will use the implemented algorithm to generate a test case for the *classify* method in the *Triangle* class that passes by the 1T branch. The test object and target branch can also be modified:

java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Line 5F

This command will use the implemented algorithm to generate a test case for the *intersect* method in the *Line* class that passes by the 5F branch.