CSI 5137 A: AI-enabled Software Verification and Testing
Assignment 2

Wei Li 0300113733 report date: Nov/2/2020

1 Homework Description

In this assignment we investigate and extend an implementation of a search-based test input generation approach, AVMFramework [1]. The questions of the assignment is answered in section 2, while a report describing the implementation is presented in section 3.

2 Answers

2.1 Question 1

How does the Alternating Variable Method (AVM) work? How is this algorithm different from the Hill Climbing or Simulated Annealing algorithms we discussed in the class? Please try to describe the main ideas and the working of AVM in less than one page (using a few paragraphs).

Answer: Like the paper [1], we limit the discussion of AVM to the search in vector space $\vec{x} \in \mathbb{Z}^n$. In other word, the algorithm searches the optimal $\vec{x}$ for some optimization problem where $\vec{x} = (x_1, \ldots, x_{len})$, $x_i \in \mathbb{Z}$.

Unlike other optimization methods where the whole vector is modified through applying **operations**, AVM modifies each of the variable x_i in the vector at a time. The algorithm search variable x_i in turn, and when all variables in the vector have been considered, the AVM go back to the first variable for another search until a termination condition is triggered. After AVM search, a local optimum can be found.

To search individual variable, the vanilla AVM use a algorithm called "Iterated Pattern Search" (IPS). Based on IPS, there are two variations "Geometric Search"(GS) and "Lattice Search"(LS). In IPS, there are two main processes: **exploratory moves** and **pattern moves**. In **exploratory moves**, a direction of positive(increasing the variable) or negative(decreasing the variable) that leads to an improvement to the objective function is found by making small change to the variable. After finding the correct direction, in **pattern moves**, the variable is increased or decreased step by step to the direction, and the step size should be increasing. The **pattern moves** stop when the step fails to improve the objective function, possibly overshoot the objective. Then IPS goes back to **exploratory moves** to establish a new direction and repeat the two processes. When there is no **exploratory move** that leads to an improvement, IPS terminates and the outer search loop proceeds to another variable.

GS and LS make a few change to the vanilla version. In GS, after overshooting the objective, GS use the new step and last step as upper bound and lower bound of a bracket of a binary search, in which the optimum is located. LS extends GS when unimodal assumption holds. Instead of using binary search in the bracket, it finds the optimum through adding Fibonacci numbers to x_i.

AVM is different from algorithms like Hill Climbing or Simulated Annealing in that there is no explicit definition of **operator**. Metaheuristics algorithms generally need to defined an operator to generate neighborhoods of an initial solution, and those neighborhoods are modified version of the full initial solution. In AVM, on the contrary, the optimization is done for each of the variables in a vector at a time. As a result, AVM may be more suitable for objective function like approach level+branch distance since there is usually only needs to modify one input variable at a time. Other than that, AVM is more similar to Hill Climbing than Simulated Annealing. For one thing, AVM is a local search algorithm and the algorithm is greedy. For another thing, the algorithm is deterministic, there is no stochastic element in the implementation.

2.2 Question 2

Show the control-flow graph for the classify method of the Triangle class and label each branch with an id such that your branch ids are consistent with those specified in **TriangleBranchTargetObjectiveFunction**. Note that each branch id should be a number followed by T (for the true branch) or F (for the false branch).

Answer:

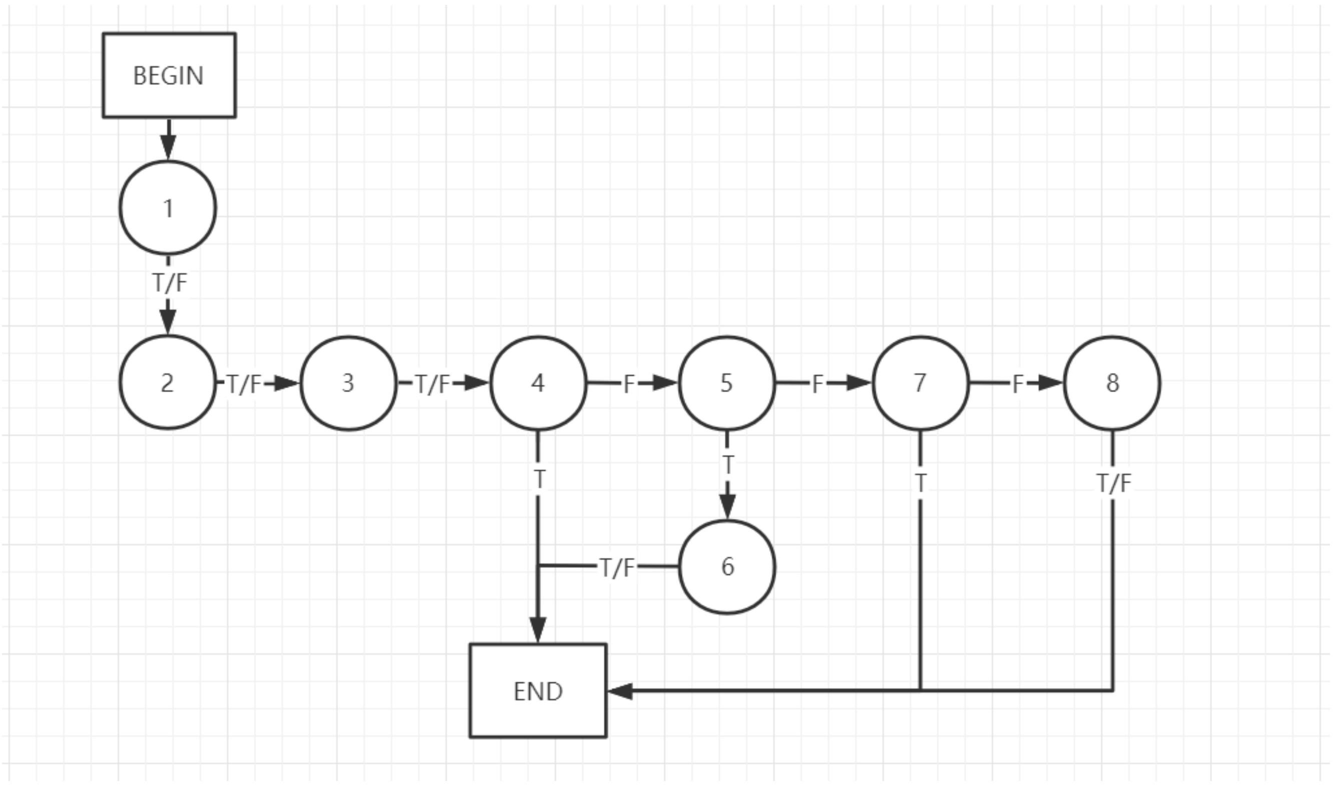

Figure 1: Control-flow graph of Triangle class

2.3 Question 3

Provide the smallest test suite that achieves branch coverage for the classify method of the Triangle class. For each test case in your test suite, specify the branches covered by the test case. Does this test suite achieve statement coverage as well?

Answer:
(94, 93, 86): covers {1T, 2T, 3T, 4F, 5F, 7F, 8F}
(251, 332, 964): covers { 1F, 2F, 3F, 4T}
(489, 489, 489): covers {1F, 2F, 3F, 4F, 5T, 6T}
(489, 489, 490): covers {1F, 2F, 3F, 4F, 5T, 6F}
(86, 94, 94): covers {1F, 2F, 3F, 4F, 5F, 7F, 8T}

The test suite achieve statement coverage, but not branch coverage. Since there is a bug in the **Triangle** class source code, the branch 7T is unreachable. Also all the AVMf optimization methods cannot optimize the objective function of **Triangle** 7T to 0. The branchs that are covered are {1T, 2T, 3T, 4T, 5T, 6T, 8T,

1F, 2F, 3F, 4F, 5F, 6F, 7F, 8F}
On the other hand, since all of the nodes are covered by the test suite, it achieves statement coverage.

2.4 Question 4

Provide a test suite for the intersect method of the Line class that achieves statement coverage, but not branch coverage. Explain which branches of the intersect method are not covered by your test suite.

Answer:
(10.9, 45.0, 23.7, 68.4, 31.1, 50.0, 5.9, 92.6): covers statements {1,2,3,4,5} and branches {1T, 2T, 3T, 4T, 5T}
(48.7, 39.2, 48.7, 39.2, 21.0, 39.5, 33.4, 5.8): covers statements {1,6,7} and branches {1F, 6F, 7T}

branches {6T, 7F, 2F, 3F, 4F, 5F} are not covered by the test suite.

2.5 Question 5

The fitness function to compare different candidate test inputs is defined based approach level and branch distance metrics. Explain how these two metrics are combined to compare different test input vector candidates and specify in which method of which Java class in the AVMf framework, this comparison is implemented.

Answer:
Given a test input x and a path generated by executing x, **approach level** is the minimum number of control nodes between the path and the coverage target t. However, since approach level is characteristic by a flat landscape and knowing the approach level does not help to modify the input data, an auxiliary metric is needed. **Branch distance** defines smooth numeric functions to approximate the target statement or branch in the program, and the function reach zero only when the target is covered. The two metrics are combined through adding approach level and normalized branch distance. Combining the two metrics as one fitness function gives a smoother landscape that allows better convergence and speed. In the search, minimizing the approach level gets the input close enough to the target, and minimizing the branch distance further modify the input so that it actually lands on the target.

In AVMf, given an input vector, the calculation of the two metrics is implemented in class **org.avmframework.examples.inputdatageneration.BranchTargetObjectiveFuntion**. The two metrics is used to instantiate objects of class **org.avmframework.examples.inputdatageneration.BranchTargetObjectiveValue**. The comparison is implemented in a method **compareTo** in the class.

2.6 Question 6

As discussed in the class, one way to combine approach level and branch distance metrics is to first nomralize branch distance and then add it to approach level. Why do we need to normalize the branch distance metric but not the approach level?

Answer:
Approach level is an integer greater or equal to 0. It is value reach 0 only when the test input is one step apart to the target or the test input has actually covered the target, otherwise, it is a value greater or equal to 1. By normalizing **branch distance** to a value smaller than 1, the search is encouraged to minimize the **approach level** small enough and then to minimize the **branch distance**. The reason why it is more appropriate is that there is no reason to modify the input to go through the condition of target statement/branch when the actual execution path is still far away from the target. The normalization reduces the search space and make the search faster.

3 Implementation Report

3.1 Implementation Task

Implement one of the local search algorithms we learned in the class (e.g., Hill Climbing and Simulated Anneal) in the AVMf framework and modify GenerateInputData.java to use your new local search algorithm to generate test input data.

3.2 About the Implementation

We implemented Hill Climbing as the new local search algorithm. The algorithm is implemented in the class **org.avmframework.localsearch.HillClimbingSearch**. The new **HillClimbingSearch** had been made the default search method in the test data generation task **org.avmframework.examples.GenerateInputData**. The Hill Climbing method can be integrated to the AVMf and be used to generate test data for **Line, Triangle** and **Calendar**.

The hill climbing can replace the original local searching include "IteratedPatternSearch", "GeometricSearch" and "LatticeSearch". It is still defined for single variable optimization in AVMf. The implementation idea is:

- Uniformly generate NEIGHBORHOOD_SIZE integers $M = \{m_1, m_2, \ldots\}$ within the NEIGHBORHOOD_WINDOW, which is [-NEIGHBORHOOD_WINDOW/2, NEIGHBORHOOD_WINDOW/2].
- Add each integer to the target variable x_i and evaluate the new solution. Identify the best objective value and its corresponding integer m_j, update the target variable of the initial solution as $x_i = x_i + m_j$
- Repeat the process until no integer in the batch improves the solution.

3.3 how to use

Since the HillClimbing has been made the default local search in **GenerateInputData**. You can run **org.avmframework.examples.GenerateInputData** directly with arguments of the problem(ie. **Triangle, Line, Calendar**) and the target (eg. **6T, 7F, ...**).

Alternatively, you can build the project as .jar and execute the program in shell eg:

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

It is observed that sometime the search fails to find the optimum within the default **MAX_EVALUATIONS**, which is 10^5, so we made the default **MAX_EVALUATIONS** 10^7 instead, which runs approximately 8s in our machine.

References

[1] P. McMinn and G. M. Kapfhammer, "Avmf: An open-source framework and implementation of the alternating variable method," in *SSBSE*, 2016.