NAME	STUDENT NUMBER	EMAIL ID
Aishwarya Manapuram	300322316	amana022@uottawa.ca
Ahmed Farooqui	300334347	afaro053@uottawa.ca

2.8 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. Do not modify the input interface of GenerateInputData.java though. That is, your modified version of GenerateInputData.java should still generate test inputs for branch ”1T” of Triangle by passing 1T Triangle to it as input. Note that you do not need to use the optional search as input to determine the type of search algorithm since GenerateInputData.java should call the new search algorithm that you have implemented by default.

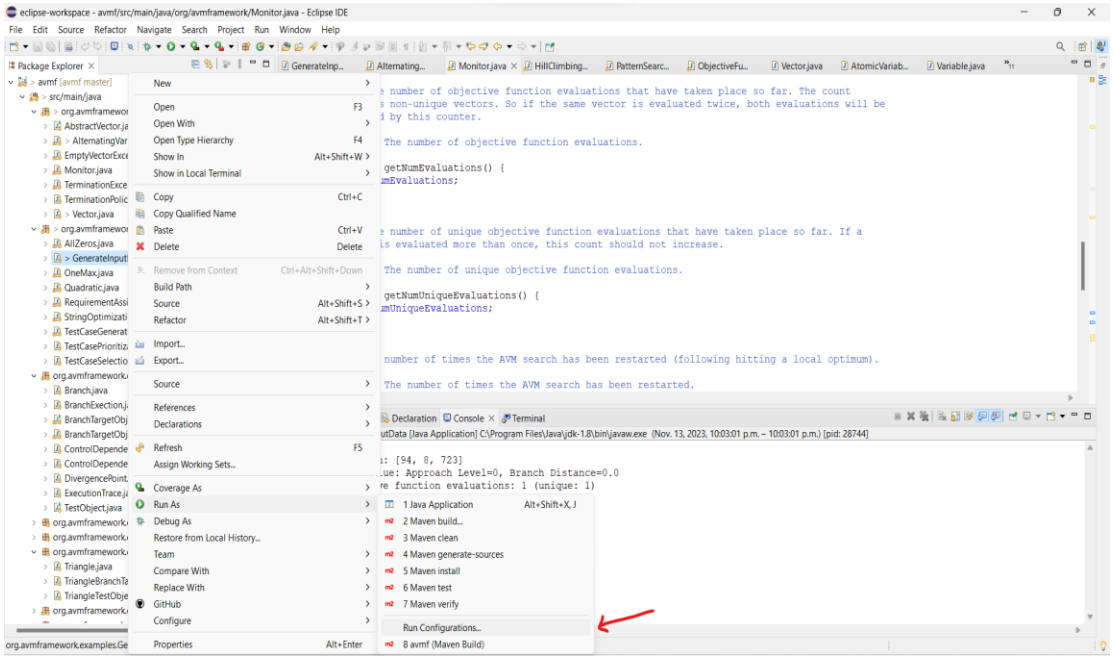
For the implementation task, you should submit the following deliverables by the submission deadline:

• **Implementation:** The modified AVMf framework that generates test inputs using a new local search algorithm that you have implemented. Make sure that your submission is self-contained and compiles and executes without any problem.

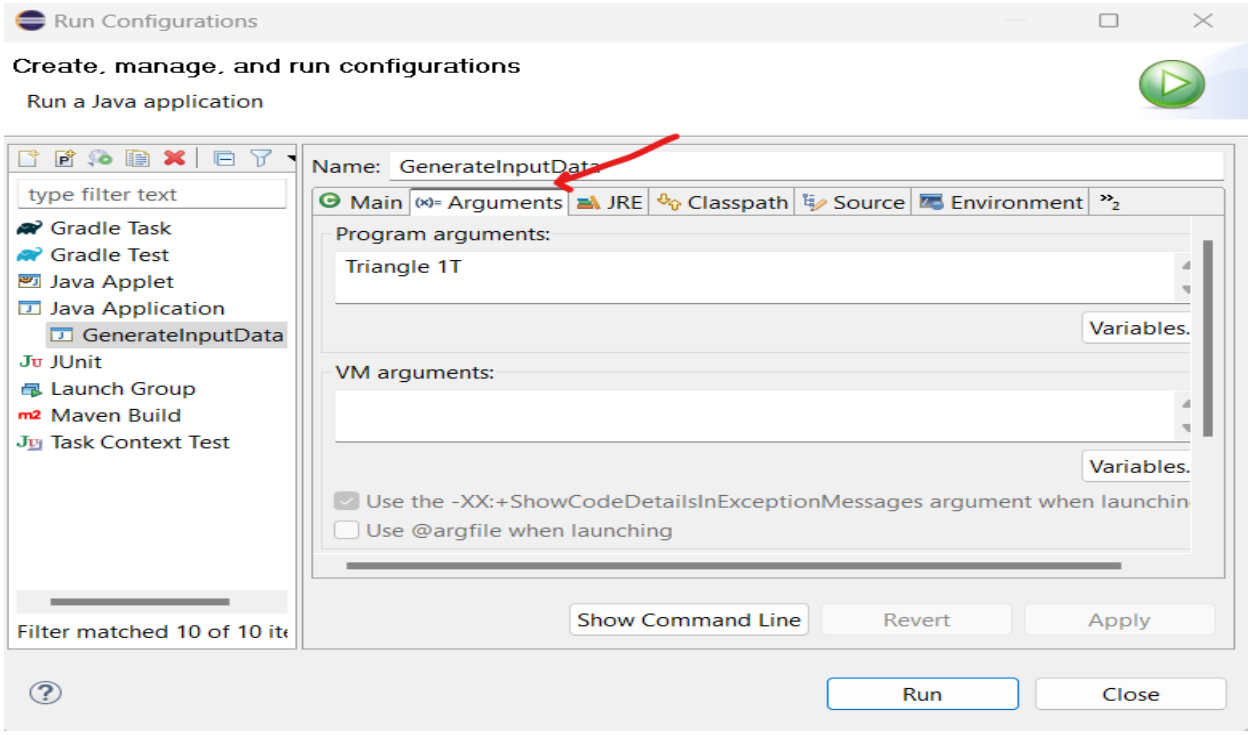
• **Report:** A written report that contains a detailed description of your new search algorithm and how it is used for test generation.

Instructions to Run

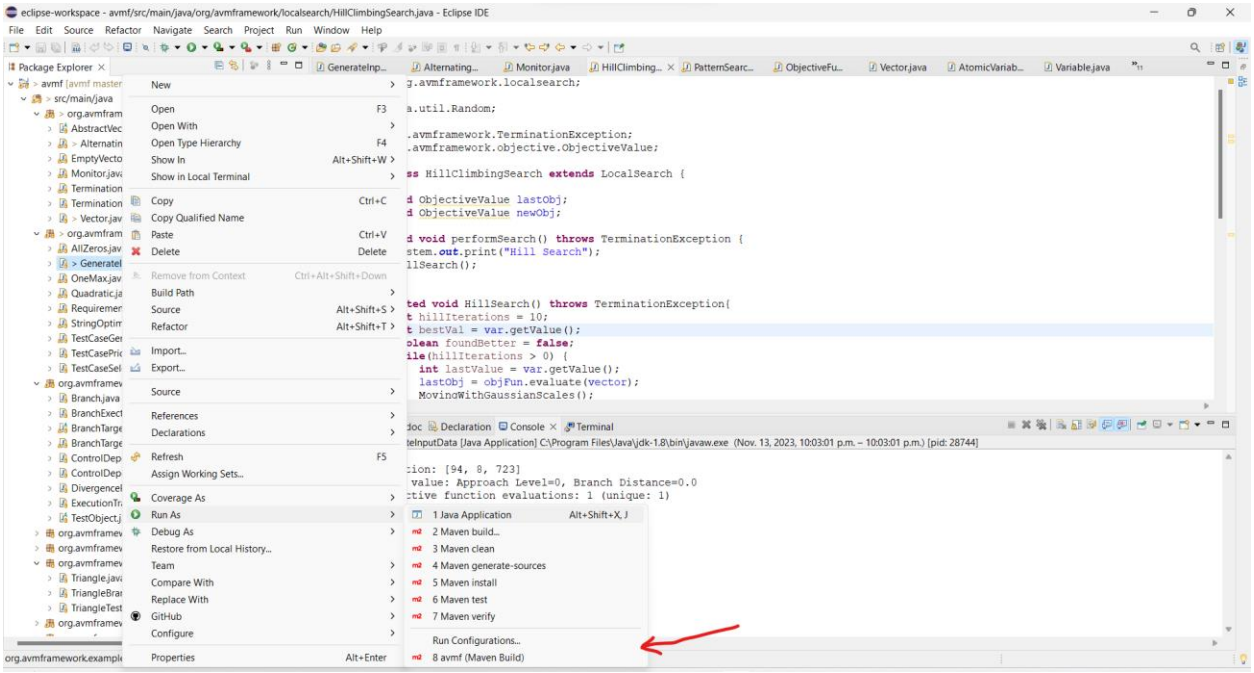
1. Set the run configurations in the GenerateInputData.java



2. Set the command line arguments that you want to pass.



3. Run as Java Application



Approach

Algorithm: **Hill Climbing**

In our HillClimbingSearch.java, we noticed that each integer from the vector is passed once at a time. We try to find the neighbors of this integer and see if it produces the right result.

The way we do it is by finding 10 neighbors of the first integer and replacing it with the neighbor and seeing if the objective function value obtained is better or not? If it is, then we consider that iteration to be successful and we continue searching. If not, then we can go ahead and apply the hill climbing on the second integer which would be passed.

The algorithm we used is Perturbation with scaling where we generate 3 random scales and multiply it with the integer and then normalize it with Gaussian value.

Reason for multiplying by 40:
The program multiplies the neighbor value obtained by 40. This is to ensure that the property of the triangle is still maintained in the solution i.e. a + b > c. While this scaling using 40 is not a foolproof solution, it does reasonably well. However, setting up a check for all solutions obtained would be better going forward.

In general, for a sample size of just 3 integers, the neighbor generation algorithm is not so significant, however, for a problem like this, swapping can obtain a solution without changing any of the values but that is not necessarily the aim.

Code

A new file named HillClimbingSearch.java was created in the org.avmframework.examples package. The default search method was changed to HillClimbingSearch in the GenerateInputData.java file.

org.avmframework.examples.HillClimbingSearch.java

```
package org.avmframework.localsearch;

import java.util.Random;

import org.avmframework.TerminationException;
import org.avmframework.objective.ObjectiveValue;

public class HillClimbingSearch extends LocalSearch {

  protected ObjectiveValue lastObj;
  protected ObjectiveValue newObj;

  protected void performSearch() throws TerminationException {
```

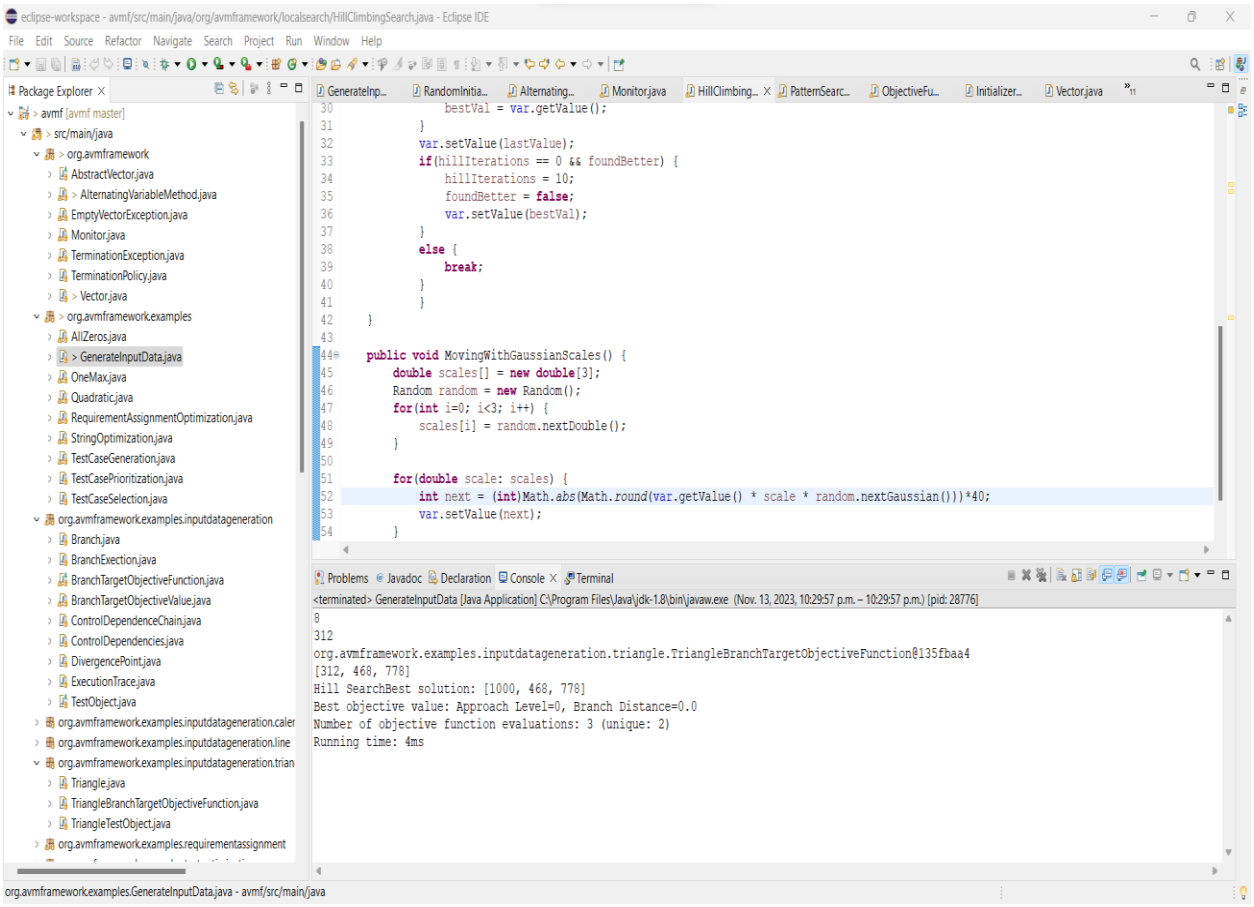
```
                System.out.print("Hill Search");
            HillSearch();
}

        protected void HillSearch() throws TerminationException{
                int hillIterations = 10;
                int bestVal = var.getValue();
                boolean foundBetter = false;
                while(hillIterations > 0) {
                        int lastValue = var.getValue();
                        lastObj = objFun.evaluate(vector);
                        MovingWithGaussianScales();
                        newObj = objFun.evaluate(vector);
                        hillIterations--;
                        if(newObj.betterThan(lastObj)) {
                                foundBetter = true;
                                bestVal = var.getValue();
                        }
                        var.setValue(lastValue);
                        if(hillIterations == 0 && foundBetter) {
                                hillIterations = 10;
                                foundBetter = false;
                                var.setValue(bestVal);
                        }
                        else {
                                break;
                        }
                        }
        }

        public void MovingWithGaussianScales() {
                double scales[] = new double[3];
                Random random = new Random();
                for(int i=0; i<3; i++) {
                        scales[i] = random.nextDouble();
                }

                for(double scale: scales) {
                        int next = (int)Math.abs(Math.round(var.getValue() * scale *
random.nextGaussian()))*40;
                        var.setValue(next);
                }       }}
```

Sample Output:



Note: Implementation files are attached separately.