Assignment 2

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

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).

ANS: The Alternating Variable Method (AVM) serves as a **local search technique** initially designed for automated numerical test data generation. Recognized for its speed and efficiency, AVM employs an iterative pattern search that involves variable search patterns. The objective is to approach the optimal value by executing exploratory moves on individual variables within the vector, incrementally refining the objective value.

A critical aspect of AVM lies in the directional selection, achieved by executing pattern moves based on exploratory moves. This process enhances objective values, determining whether to proceed in a positive or negative direction. Directional changes occur as the objective value of the pattern moves upward. The iteration persists until there is no improvement in the vector space, indicating the end of the search. To illustrate, consider the Traveling Salesman Problem (TSP): exploratory moves are performed on cities, considering pattern moves that decrease the distance between cities (objective value).

The procedural steps of AVM involve:

1. **Exploratory Moves**: Increment or decrement variables in the vector by a value of 1.
2. **Pattern Moves**: If exploratory moves improve objective values (positive or negative), a direction is generated.
3. **Continuation of Pattern Moves**: Increase the size of pattern moves with an increase in objective value. If the pattern move size doesn't increase, it suggests overshooting the optimal value.
4. **Loop Break**: If the pattern move size surpasses the difference between the current value of xi and the optimal value, the loop breaks, and exploratory moves start anew in a different direction. If exploratory moves do not improve, the search begins with a new variable in the vector.

AVM incorporates three distinct searches:
- **Iterative Search**: When overshooting occurs, move in the opposite direction to discover the optimal solution, continuing until all resources are exhausted or the desired result is achieved.
- **Geometric Search**: After overshooting, establish an upper and lower limit bracket, then apply a linear search to locate the vector providing the best value.
- **Lattice Search (LS)**: Similar to geometric search, LS converges on the best solution by incorporating Fibonacci numbers to steps that raise the value from the lower side of the bracket.

In contrast to AVM, Hill Climbing increments variables randomly, verifying objective values until reaching a point where any increment no longer improves the objective value, considered a local optimum. AVM's distinctiveness lies in its systematic exploration and refinement of vector variables, offering an iterative, pattern-based approach to optimization, differing from the more random exploration of Hill Climbing or the probabilistic approach of Simulated Annealing.

2.2 Question 2.

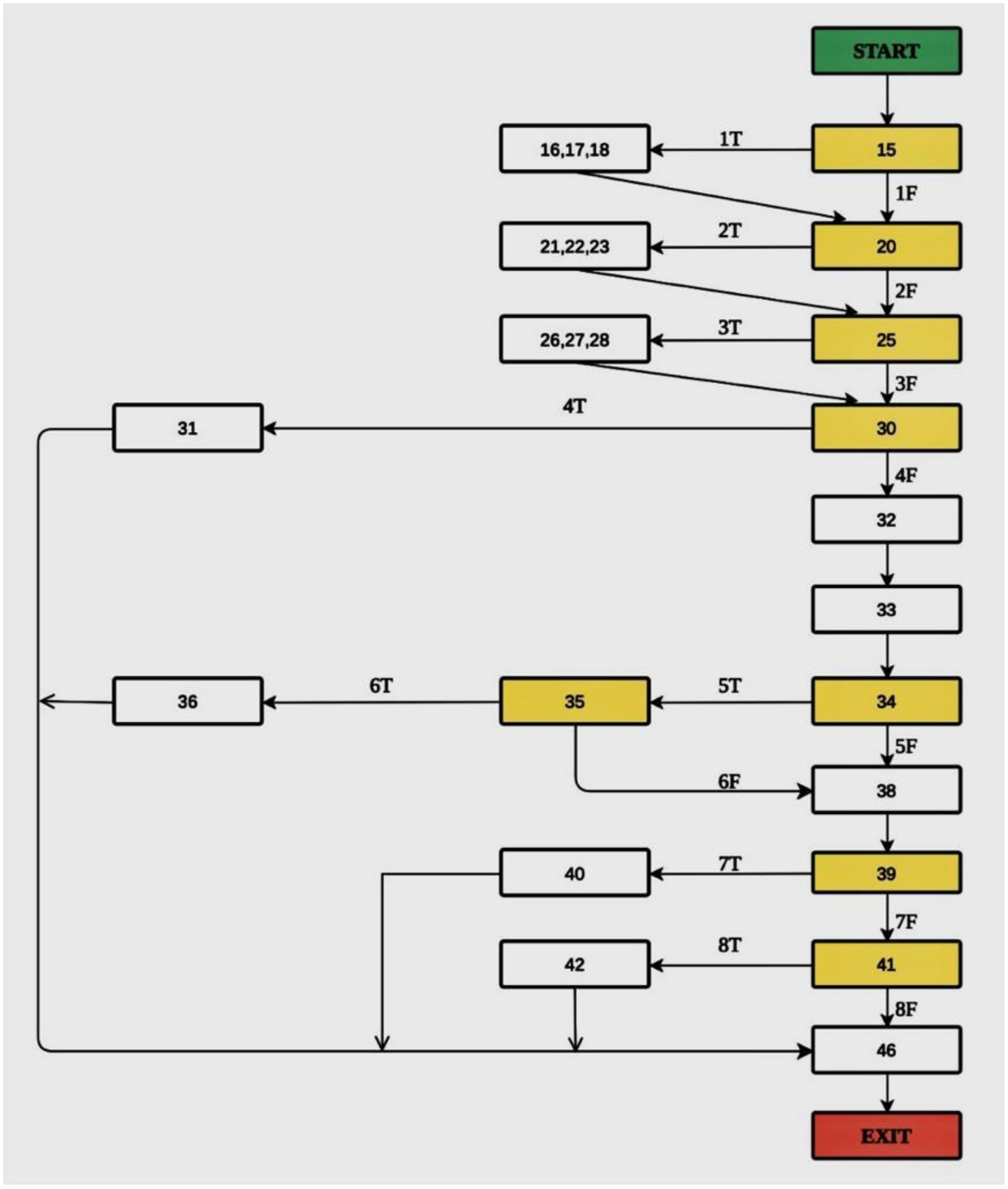
Show the control-flow graph for the classify method of the Triangle class and label each branch with an id such that your branch ids are consistent with those specified in TriangleBranchTargetObjectiveFunction. Note that each branch id should be a number followed by T (for the true branch) or F (for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

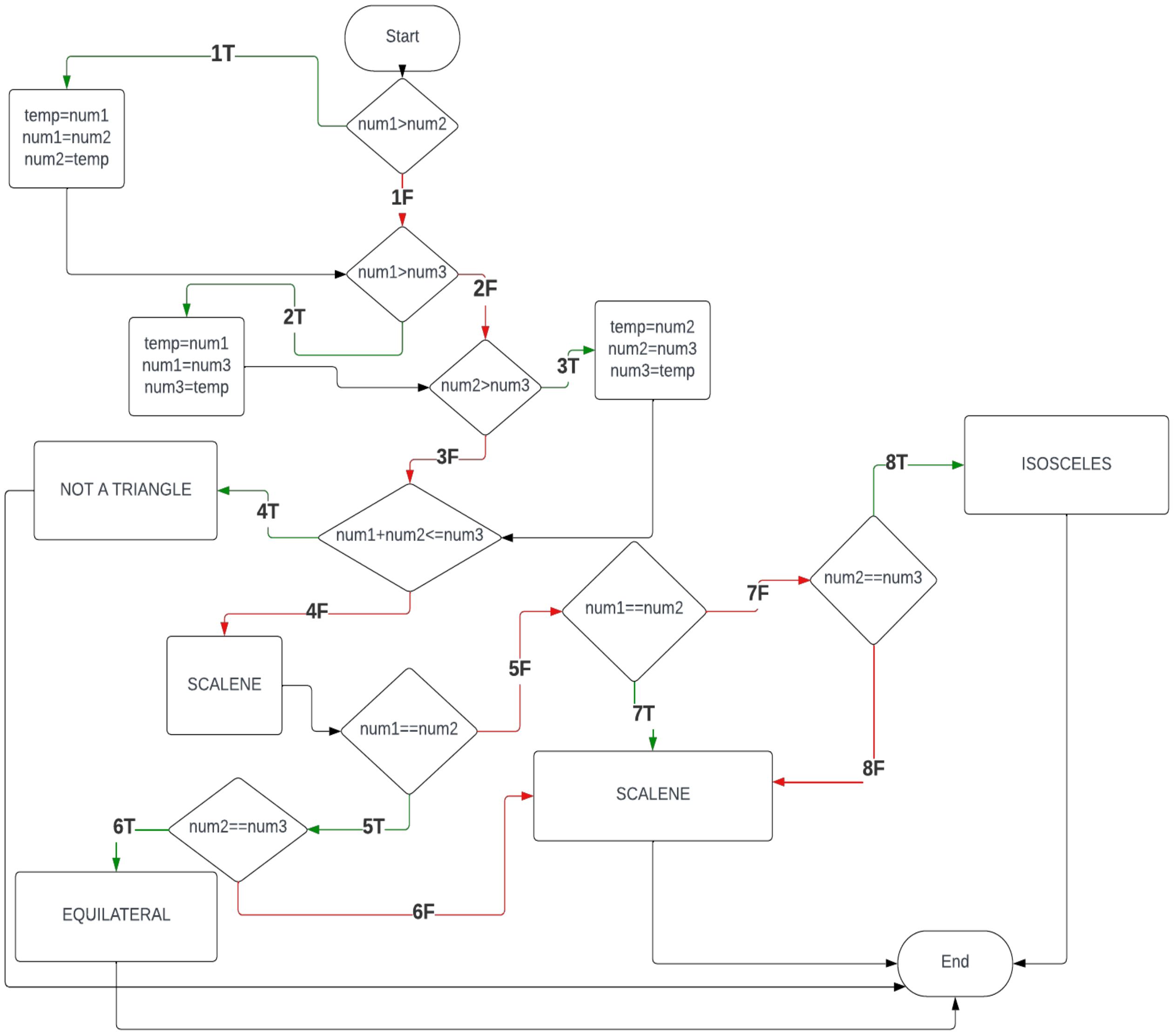
ANS: Triangle has branch IDs ranging from 1T/F to 8T/F. Branches are highlighted green in below code.

Triangle.java

```java
package org.avmframework.examples.inputdatageneration.triangle;

public class Triangle {

  public enum TriangleType {
    NOT_A_TRIANGLE,
    SCALENE,
    EQUILATERAL,
    ISOSCELES;
  }

  public static TriangleType classify(int num1, int num2, int num3) {
    TriangleType type;
```

```
                                                                  (14)
    if (num1 > num2) {                                            (15)
      int temp = num1;                                            (16)
      num1 = num2;                                                (17)
      num2 = temp;                                                (18)
    }                                                             (19)
    if (num1 > num3) {                                            (20)
      int temp = num1;                                            (21)
      num1 = num3;                                                (22)
      num3 = temp;                                                (24)
    }                                                             (25)
    if (num2 > num3) {                                            (26)
      int temp = num2;                                            (28)
      num2 = num3;                                                (29)
      num3 = temp;                                                (30)
    }                                                             (31)
    if (num1 + num2 <= num3) {                                    (32)
      type = TriangleType.NOT_A_TRIANGLE;                         (33)
    } else {                                                      (34)
      type = TriangleType.SCALENE;                                (35)
      if (num1 == num2) {                                         (36)
        if (num2 == num3) {                                       (37)
          type = TriangleType.EQUILATERAL;                        (38)
        }                                                         (39)
      } else {                                                    (40)
        if (num1 == num2) {                                       (41)
          type = TriangleType.ISOSCELES;                          (42)
        } else if (num2 == num3) {                                (43)
          type = TriangleType.ISOSCELES;                          (44)
        }                                                         (45)
      }                                                           (46)
    }                                                             (47)
    return type;                                                  (48)
  }                                                               (49)
}                                                                 (50)
```


START
15
1T
16,17,18
1F
20
2T
21,22,23
2F
25
3T
26,27,28
3F
30
4T
31
4F
32
33
34
5T
35
6T
36
5F
6F
38
39
7T
40
7F
41
8T
42
8F
46
EXIT


Start
num1>num2
1T
temp=num1
num1=num2
num2=temp
1F
num1>num3
2T
2F
temp=num1
num1=num3
num3=temp
num2>num3
3T
temp=num2
num2=num3
num3=temp
3F
NOT A TRIANGLE
4T
num1+num2<=num3
4F
SCALENE
num1==num2
5F
5T
num1==num2
7F
7T
num2==num3
8T
ISOSCELES
8F
SCALENE
num2==num3
6T
6F
EQUILATERAL
End

2.3 Question 3.

Provide a test suite that achieves branch coverage for the classify method of the Triangle class.

A test suite is a collection or set of test cases that are designed to assess the functionality. From the code, we know that:

1. If NUM1 =NUM2==NUM3 → EQUILATERAL.
2. If any 2 numbers in NUM1, NUM2, NUM3 are equal → ISOCELES.
3. If NUM1!=NUM2!=NUM3 → SCALENE.
4. If the sum of any two is less than the third in NUM1, NUM2, NUM3 → NOT A TRIANGLE.

Test Case	Num 1	Num 2	Num 3	Type	Branches covered
T1	230	230	230	EQUILATERAL	1F,2F,3F,4F,5T,6T
T2	80	120	150	SCALENE	1F, 2F, 3F, 4F, 5F, 7F, 8F
T3	80	100	150	SCALENE	1F, 2F, 3F, 4F
T4	87	87	123	ISOCELES	1F, 2F, 3F, 4F, 5T, 6F,
T5	890	295	501	NOT A TRIANGLE	1T,2T,3F,4T
T6	123	67	123	ISOCELES	1F, 2F, 3F, 4F, 5F, 7F, 8T
T7	152	131	147	SCALENE	1T, 2F, 3T, 4F, 5F, 7F, 8F
T8	500	200	100	NOT A TRIANGLE	1T,2F,3T,4T
T9	347	353	137	SCALENE	1F, 2T, 3T, 4T

2.4 Question 4.

Provide a smallest test suite that can achieve statement coverage for the code below. Does this test suite achieve branch coverage as well? If yes, for each test case in your test suite, specify the branches covered by the test case. Otherwise, provide a smallest test suite that can achieve branch coverage for this code.

```
Add (int a, int b) {
if (b > a) {          → 1
b = b - a
Print b
}
if (a > b) {          → 2
b = a - b
Print b
}
if (a = b) {          → 3
if (a = 0) {          → 4
Print ’0’
}
}
}
```

There are 4 conditions: (a<b), (a>b), (a==b) and (a==0). For branch coverage, we need test cases that exercise both the true and false branches of each decision point. **Yes**, this test suite achieves branch coverage.

The smallest test suite that can achieve statement coverage for the above code is:
Test Case 1: Add(5, 10);
Test Case 2: Add(10, 5);
Test Case 3: Add(0, 0);

Test Case	A	B	Branch_Covered	Statements covered

T1	5	10	**1T**,2F,3F
			Explanation:
			True branch of the first if (b > a)
			False branch of the second if (a > b)
			False branch of the third if (a == b)
				if(b>a),
				b = b - a,
				Print b

T2	10	5	1F,**2T**,3F
			Explanation:
			False branch of the first if (b > a)
			True branch of the second if (a > b)
			False branch of the third if (a == b)
				if (a > b) {
				b = a - b
				Print b

T3	0	0	1F,2F,**3T,4T**
			Explanation:
			False branch of the first if (b > a)
			False branch of the second if (a > b)
			True branch of the third if (a == b)
			True branch of the nested if (a == 0)
				if (a = b) {
				if (a = 0) {
				Print '0'

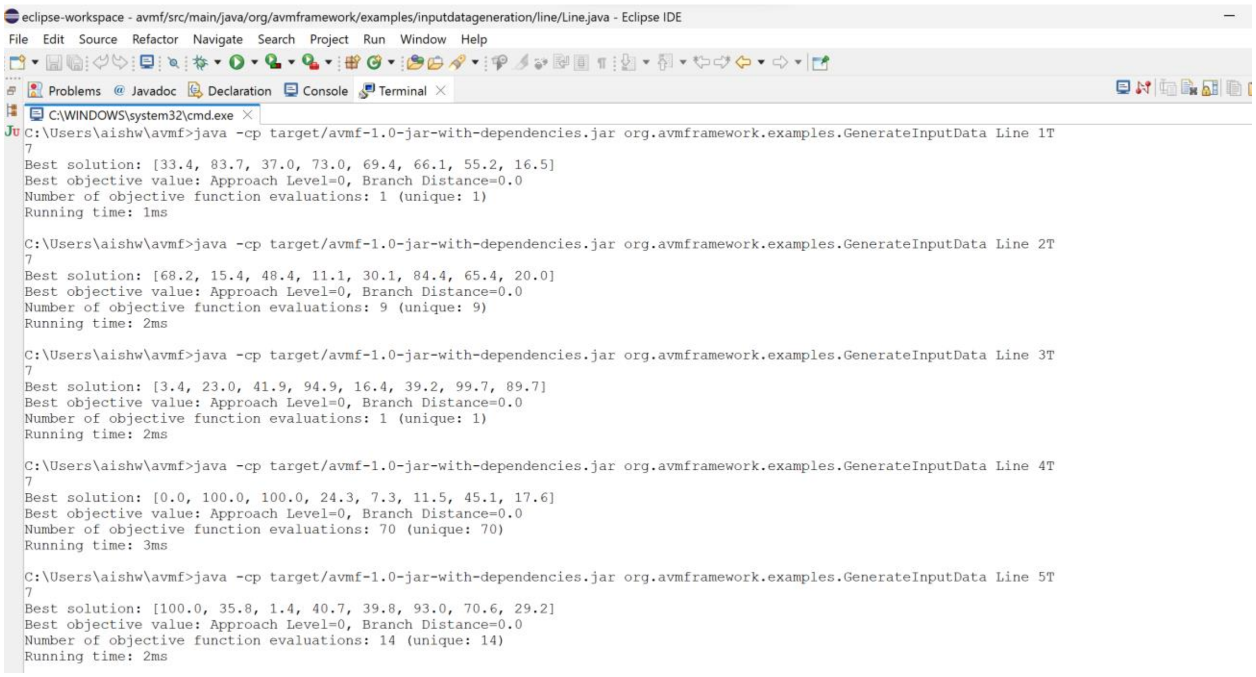
2.5 Question 5.

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.

For these branches, the statement is covered by the following:

Statement Coverage	Branches
26-31	1T,2T,3T,4F,5F
26-39	1T,2T,3T,4T,5T
43	6T

47	1F, 6F, 7T
50	1F, 6F, 7F


eclipse-workspace - avmf/src/main/java/org/avmframework/examples/inputdatageneration/line/Line.java - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
Problems
Javadoc
Declaration
Console
Terminal
C:\WINDOWS\system32\cmd.exe

```
C:\Users\aishw\avmf>java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Line 1T
7
Best solution: [33.4, 83.7, 37.0, 73.0, 69.4, 66.1, 55.2, 16.5]
Best objective value: Approach Level=0, Branch Distance=0.0
Number of objective function evaluations: 1 (unique: 1)
Running time: 1ms

C:\Users\aishw\avmf>java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Line 2T
7
Best solution: [68.2, 15.4, 48.4, 11.1, 30.1, 84.4, 65.4, 20.0]
Best objective value: Approach Level=0, Branch Distance=0.0
Number of objective function evaluations: 9 (unique: 9)
Running time: 2ms

C:\Users\aishw\avmf>java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Line 3T
7
Best solution: [3.4, 23.0, 41.9, 94.9, 16.4, 39.2, 99.7, 89.7]
Best objective value: Approach Level=0, Branch Distance=0.0
Number of objective function evaluations: 1 (unique: 1)
Running time: 2ms

C:\Users\aishw\avmf>java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Line 4T
7
Best solution: [0.0, 100.0, 100.0, 24.3, 7.3, 11.5, 45.1, 17.6]
Best objective value: Approach Level=0, Branch Distance=0.0
Number of objective function evaluations: 70 (unique: 70)
Running time: 3ms

C:\Users\aishw\avmf>java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Line 5T
7
Best solution: [100.0, 35.8, 1.4, 40.7, 39.8, 93.0, 70.6, 29.2]
Best objective value: Approach Level=0, Branch Distance=0.0
Number of objective function evaluations: 14 (unique: 14)
Running time: 2ms
```

2.6 Question 6.

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.

- The class "AlternatingVariableMethod.java" includes a method named "alternatingVariableSearch(). This compares the vector variables to optimize the vector.
- The metrics computation is handled in the "objective" folder located at "avmframework/avmf/src/main/java/org/avmframework/objective."
- The DistanceFunction.java calculates the distance between two numbers or nodes.
- NormalizeFunction.java employs alpha and beta to normalize branch distance.
- NumericObjectiveValue.java assesses optimal and current values using the "higherIsBetter" variable.
- Both NumericObjectiveValue.java and ObjectiveFunction.java in the AVM framework monitor and evaluate the vector, facilitating objective value comparison.
- The AVM object, utilizing a Monitor instance, keeps track of the candidate solution with the best objective value, the number of completed objective function evaluations, and other relevant data.
- The implementation for generating input data is utilized in GenerateInputData.java.

2.7 Question 7.

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

To prevent the approach level from being overshadowed, the branch distance is normalized and scaled to fall within the [0,1] range. In contrast, the approach level is represented as an integer. Without normalization of the branch distance, it has the potential to become disproportionately large, thus overshadowing variations in the approach level. The normalization of branch distance ensures that minimizing the approach level has a more significant influence on the fitness value, automatically giving it higher priority.

The normalization process facilitates a fair combination and weighting of these metrics in the overall evaluation of a candidate solution's fitness.

2.8 Implementation Task.

Separate files are attached for this task. It included a report and implementation files.