# Assignment 02 – Short Questions

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

**Question 1**

The Alternating Variable Method (AVM) is an approach that uses local search to generate test inputs. More specifically, the AVM searches for an input by optimizing a vector to maximize or minimize a given objective function. Each element of the vector is iterated through and subjected to an individual search process consisting of exploratory and pattern moves.

The exploratory move determines the direction (+ or -) in which to modify an element that will result in a fitter solution. It does so by increasing and decreasing the current element by one, and whichever move leads to an improvement in the objective function will dictate the direction. If no move leads to an improvement in the objective function, this means an optimum has been reached and the element should not change. The pattern move is an increase in size of the element in the direction chosen earlier. The increments also increase by a factor of two. The pattern search continues while the objective function improves. If a pattern move does not improve the objective function, this means the search overshot, and so the optimum is between the last element and the current.

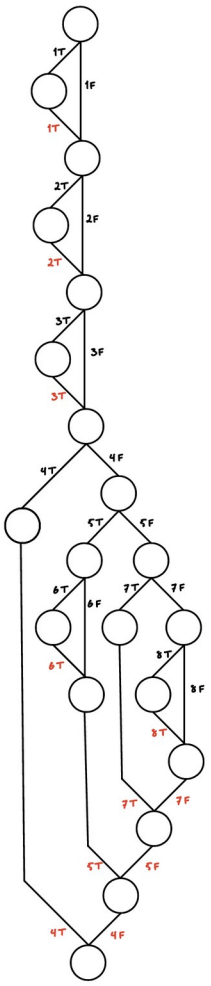
There are several variations of the AVM approach that handle the next steps differently. Once the search has overshot, the “Iterated Pattern Search” variation will revert to the exploratory move and repeat the process until an optimum is found. The “Geometric Search” will bracket the upper and lower limits of the variable and perform a binary search to find the element value that yields the largest improvement in the objective function. The “Lattice Search” will converge towards an optimum through moves that increase the element from the lower value of the bracket through the addition of Fibonacci numbers to the element. When no more moves are able to improve the objective function, the next element in the vector is considered. Once every element has been considered, the search loops back to the beginning and starts again. Once an entire cycle with no changes is complete, an optimum is found. The search can then be restarted from another random vector. This process continues until the termination criteria are met.

The AVM approach is similar to the Hill Climbing (HC) algorithm in the sense that neighbouring solutions are considered. These algorithms differ in the way they choose to generate their neighbours. In Steepest Ascent HC and Random Ascent HC, all possible neighbours are generated, and one is later chosen depending on the algorithm criteria. This is different from AVM which only explores one neighbour at a time, which is more similar to the First Ascent HC algorithm. In the First Ascent HC, the algorithm generates neighbours until it finds one that improves the objective function. It generates its neighbours by using a tweak function to modify the current solution. The tweak function can be implemented in the same way as the exploratory move used in AVM. This would entail the algorithm traversing the vector, modifying each element one at a time by adding or subtracting one from them, and resetting the element if a better solution is not found. Once a better solution is found, the search restarts from the beginning of the vector. AVM differs from this by using pattern moves to optimize an element before moving to another.

The difference between the AVM approach and the Simulated Annealing (SA) algorithm is the temperature component. SA will initially visit many less optimal neighbours randomly in the hopes of finding different local optimums. As time goes on, the explorative search cools down and a more exploitative method is utilized. This method helps the search algorithm find multiple local optimums, one of which could potentially be the global optimum. AVM does not consider worse-performing neighbours, and only after it has found a local optimum will it do a random restart. However, due to the large steps taken during the pattern moves, it is able to explore large amounts of the search space if starting far away from an optimum.

**Question 2**

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

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

  if (num1 > num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
  }
  if (num1 > num3) {
    int temp = num1;
    num1 = num3;
    num3 = temp;
  }
  if (num2 > num3) {
    int temp = num2;
    num2 = num3;
    num3 = temp;
  }
  if (num1 + num2 <= num3) {
    type = TriangleType.NOT_A_TRIANGLE;
  } else {
    type = TriangleType.SCALENE;
    if (num1 == num2) {
      if (num2 == num3) {
        type = TriangleType.EQUILATERAL;
      }
    } else {
      if (num1 == num2) {
        type = TriangleType.ISOSCELES;
      } else if (num2 == num3) {
        type = TriangleType.ISOSCELES;
      }
    }
  }
  return type;
}
```




The classify function takes three integers as input parameters. Firstly, the function compares the integers via three if statements, creating the first three “true” and first three “false” branches. After this, an if-else statement is used to compare the inputs, creating the fourth “true” and fourth “false” branches.

There are no additional if statements after taking the fourth “true” branch, and so it leads to termination. After taking the fourth “false” branch, there is a second if-else statement, which creates the fifth “true” and fifth “false” branches. There is an additional if statement after taking the fifth “true” branch, which creates the sixth “true” and sixth “false” branches, which lead to termination.

After taking the fifth “false” branch, an if-else-if statement is called. This statement can be treated as an if-else statement, with the else statement containing an additional if statement. The if-else statement creates the seventh “true” and seventh “false” branches, and the if statement in the else statement creates the eighth “true” and eighth “false” branches. These statements then lead to termination.

**Question 3**

Full branch coverage of the classify method is unachievable. The seventh "true" branch can only be reached if the fifth branch is false. However, if the fifth branch is "false," so will the seventh branch because they are making the same comparison. Thusly, branch coverage for the classify method will not include the seventh "true" branch.

** Note that in the framework, I believe that the authors made a mistake on line 65 of the TriangleBranchTargetObjectiveFunction.java file. The line should read:*

*if (trace.greaterThan(2, num1, num3)){*

*Without this modification, the branch distance of branch 2T never reaches 0.*

In the following test suite, every test case is used to cover one specific branch.

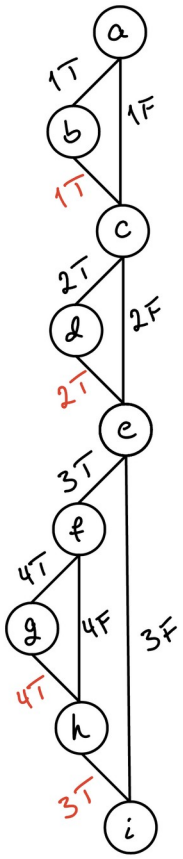
| Branches | | Test suite |
|---|---|---|
| 1st | True | (471, 399, 598) |
| | False | (472, 696, 693) |
| 2nd | True | (893, 893, 360) |
| | False | (49, 339, 621) |
| 3rd | True | (408, 947, 659) |
| | False | (44, 30, 169) |
| 4th | True | (1000, 603, 378) |
| | False | (776, 808, 580) |
| 5th | True | (736, 736, 755) |
| | False | (889, 740, 356) |
| 6th | True | (793, 793, 793) |
| | False | (618, 618, 755) |
| 7th | True | Unachievable |
| | False | (536, 901, 534) |
| 8th | True | (549, 448, 549) |
| | False | (662, 508, 538) |

This is redundant because certain test cases will likely cover a subset of the branches covered by other test cases. A simplified version of the test suit above is shown below.

| Branches | Test suite |
|---|---|
| 1 T – 2 T – 3 T – 4 T: | (1000, 603, 378) |
| 1 T – 2 F – 3 F – 4 F – 5 F – 7 F – 8 F: | (471, 399, 598) |
| 1 F – 2 F – 3 F – 4 F – 5 T – 6 T: | (793, 793, 793) |
| 1 F – 2 F – 3 F – 4 F – 5 T – 6 F: | (736, 736, 755) |
| 1 F – 2 F – 3 F – 4 F – 5 F – 8 T: | (785, 585, 584) |

**Question 4**

```
Add (int a, int b) {
        if (b > a) {
           b = b - a
           Print b
        }
        if (a > b) {
           b = a - b
           Print b
        }
        if (a = b) {
           if (a = 0) {
              Print '0'
           }
        }
}
```




Statement coverage test suite :

| Nodes | Test suite |
|---|---|
| Nodes a-c-e-f-g-h-i: | (0, 0) |
| Nodes a-b-c-d-e-i: | (2, 3) |

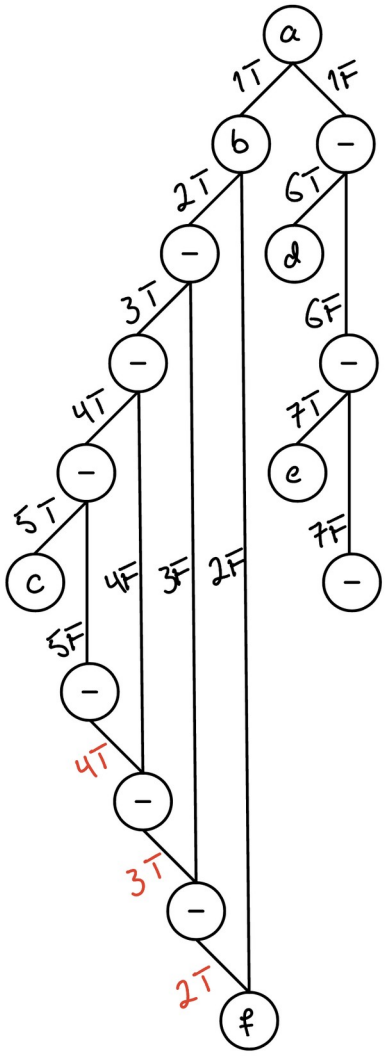
Every node (statement) is visited at least once. However, branch 4F is never visited. This is because all the nodes that would be visited using branch 4F are also visited when using branch 4T. We are therefore able to achieve statement coverage without needing to traverse branch 4F.

Branch coverage test suite :

| Branches | Test suite |
|---|---|
| Branches 1F-2F-3T-4T: (0, 0) | |
| Branches 1T-2T-3F: | (2, 3) |
| Branches 1F-2F-3T-4F: (1, 1) | |

**Question 5**

```java
public Line(double x1, double y1, double x2, double y2) {
  this.x1 = x1;
  this.y1 = y1;
  this.x2 = x2;
  this.y2 = y2;
}

public boolean intersect(Line line1, Line line2) {

  double u1t = (line2.x2 - line2.x1) * (line1.y1 - line2.y1)
      - (line2.y2 - line2.y1) * (line1.x1 - line2.x1);
  double u2t = (line1.x2 - line1.x1) * (line1.y1 - line2.y1)
      - (line1.y2 - line1.y1) * (line1.x1 - line2.x1);
  double u2 = (line2.y2 - line2.y1) * (line1.x2 - line1.x1)
      - (line2.x2 - line2.x1) * (line1.y2 - line1.y1);

  if (u2 != 0) {
    double u1 = u1t / u2;
    double u02 = u2t / u2;

    if (0 <= u1) {
      if (u1 <= 1) {
        if (0 <= u02) {
          if (u02 <= 1) {
            return true;
          }
        }
      }
    }
    return false;

  } else {
    if (u1t == 0) {
      return true;
    }

    if (u2t == 0) {
      return true;
    }

    return false;
  }
}
```




A test case here consists of 2 Lines, which both consist of four doubles.

Nodes a-b-c:
Branches 1T – 2T – 3T – 4T – 5T:
Line1 [26.2, 23.5, 61.0, 61.7]
Line2 [39.3, 85.2, 74.2, 40.8]

Nodes a-b-f:
Branches 1T – 2F:
Line1 [0.0, 55.9, 29.1, 18.0]
Line2 [43.0, 54.8, 16.4, 63.1]

Nodes a-d:
Branches 1F – 6T:
Line1 [100.0, 62.7, 100.0, 62.7]
Line2 [100.0, 62.7, 70.6, 25.5]

Nodes a-e:
Branches 1F – 6F – 7T:
    Line1 [0.0, 73.2, 0.0, 73.2]
    Line2 [10.3, 63.8, 85.6, 67.8]

Nodes a-g:
Branches 1F – 6F – 7F:
    Line1 [70.5, 89.9, 70.8, 69.9]
    Line2 [76.9, 78.9, 77.5, 38.9]

Branches not covered:
5F, 4F and 3F

These branches are not visited because they cover the same statements that were covered in the second test case, which never reaches these branches. More specifically, these branches would cover nodes a, b and f in the control-flow graph. These nodes are also covered in the second test case, in which branch 2F is utilized. It is therefore not necessary to cover the extra branches if our goal is solely to achieve full statement coverage.

**Question 6**

(From the ObjectiveValue class)

```
public abstract class ObjectiveValue<T extends ObjectiveValue> implements Comparable<T> {

  public abstract boolean isOptimal();

  public boolean betterThan(T other) {
    return compareTo(other) > 0;
  }

  public boolean sameAs(T other) {
    return compareTo(other) == 0;
  }

  public boolean worseThan(T other) {
    return compareTo(other) < 0;
  }
}
```

(From the BranchTargetObjectiveValue class which extends the ObjectiveValue class)

```
@Override
public boolean isOptimal() {
  return approachLevel == 0 && branchDistance == 0;
}

@Override
public int compareTo(Object obj) {
  BranchTargetObjectiveValue other = (BranchTargetObjectiveValue) obj;
  if (approachLevel < other.approachLevel) {
    return 1;
  } else if (approachLevel > other.approachLevel) {
    return -1;
  } else {
    if (branchDistance < other.branchDistance) {
      return 1;
    } else if (branchDistance > other.branchDistance) {
      return -1;
    }
  }
  return 0;
}
```

(From the BranchTargetObjectiveFunction class)

```
public abstract class BranchTargetObjectiveFunction extends ObjectiveFunction {

  protected Branch target;
  protected ControlDependenceChain controlDependenceChain;
  protected ExecutionTrace trace;

  public BranchTargetObjectiveFunction(Branch target) {
    this.target = target;
    this.controlDependenceChain = getControlDependenceChainForTarget();
  }

  protected abstract ControlDependenceChain getControlDependenceChainForTarget();

  protected ObjectiveValue computeObjectiveValue(Vector vector) {
    trace = new ExecutionTrace();
    executeTestObject(vector);
    DivergencePoint divergencePoint = controlDependenceChain.getDivergencePoint(trace);

    int approachLevel = 0;
    double distance = 0;
    if (divergencePoint != null) {
      approachLevel = divergencePoint.chainIndex;
      distance = trace.getBranchExecution(divergencePoint.traceIndex).getDistanceToAlternative();
    }
    return new BranchTargetObjectiveValue(approachLevel, distance);
  }

  protected abstract void executeTestObject(Vector vector);
}
```

The approach level and branch distance objective values are computed in the *computeObjectiveValue* method of the *BranchTargetObjectiveFunction* class. They are computed by tracing the execution path of a test input. Given the path traced by the test input and the path of the target branch, we can find at which node the paths diverge. This is done using the *getDivergencePoint* method from the *ControlDependenceChain* class. The index of the node where the divergence happened becomes the approach level value.

The branch distance is then computed by retrieving the predicate of the test input at the divergence node and calling the *getDistanceToAlternative* method on it. What this does is compute the difference between the test predicate and the predicate of the target at the divergence node. It does so by following heuristics to convert the difference in predicates into a distance. These heuristics can be found in the *DistanceFunction* class. The metrics are not summed at this stage, nor is the branch distance normalized.

Comparing the approach level and branch distance happens in the *compareTo* method in the *BranchTargetObjectiveValue* class. This class extends the *ObjectiveValue* class, which uses the *compareTo* method when determining which of two vectors has a better fitness value. In the *comparTo* method, first the approach levels are compared. If the test input has a smaller approach level than the other, the method returns 1. If the test input has a larger approach level than the other, the method returns -1. If the approach levels are the same, the method then compares the branch distances.

The reason why the branch distances are not normalized is because they are not directly added to the approach levels. Instead, they are compared separately, and only when the approach levels are the same. This in turn makes the approach levels more important, and does not allow the branch distances to overtake them. If the distances are the same, the method returns 0, meaning that they share the same fitness. If the test input has a lower branch distance than the other, it returns 1. If the test input has a higher branch distance than the other, it returns -1.

**Question 7**

Normalizing the branch distance will naturally bring the value to a number between 0 and 1. One way of normalizing the branch distance is by using the following equation:

$$x^{new} = x \,/\, (x + \quad ).$$

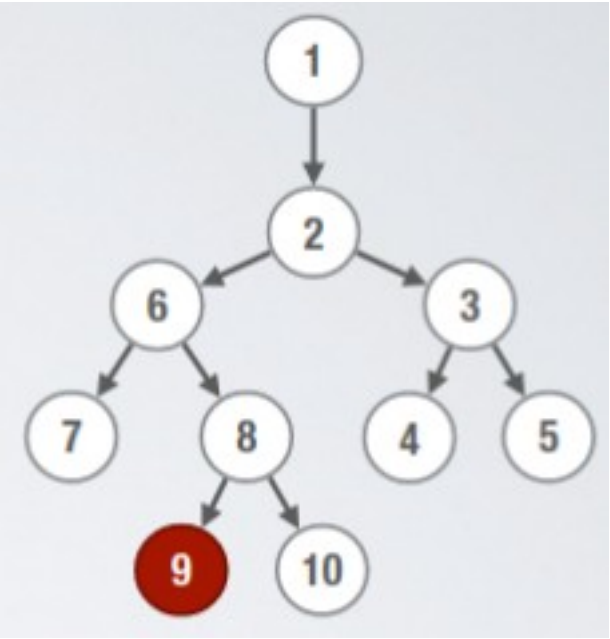
By using this normalization method, we rest assured that using a larger value of $x$ will yield a larger value of $x^{new}$,

$$x \,/\, (x + \quad ) < (x + a) \,/\, (x + a + \quad ), \textit{ for any given } a > \quad .$$

This is a good way of making very large numbers smaller so that they do not overpower the rest of our calculations. By normalizing the branch distance, which is capable of being a very large number, the approach level metric will not be overtaken. In other words, even if solution $a$ has a smaller branch distance than solution $b$, if solution $b$ has a smaller approach level, then it will be favoured over solution $a$ regardless of their branch distances.

In the case of AVMf, because they compare the branch distance after the approach level and don't combine the two through addition, it is not necessary to normalize. If a test input has a better approach level than another, the branch distances are not considered. They are only considered when the approach levels of two test inputs are exactly the same. In that case, it does not matter if the branch distances are a lot larger than the approach levels because they alone will dictate the difference in fitness.

```
class Triangle {

 void computeTriangleType() {
1.if (isTriangle()){
2.  if (side1 == side2) {
3.       if (side2 == side3)
4.           type = "EQUILATERAL";
         else
5.           type = "ISOSCELES";
     } else {
6.       if (side1 == side3) {
7.           type = "ISOSCELES";
       } else {
8.          if (side2 == side3)
9.            type = "ISOSCELES";
            else
10.            checkRightAngle();
          }
        }
    }// if isTriangle()
 }
```



Below, we compare the fitness of three different test cases for the algorithm above.

| Not Normalizing BD | Normalizing BD |
|---|---|
| x = (2, 2, 3)<br>Path(x) = <1,2,3,5><br>AL = 2<br>BD = 1<br>Fitness = AL + BD = 2 + 1 = 3<br><br>y = (2, 3, 5)<br>Path(y) = <1,2,3,5><br>AL = 0<br>BD = 2<br>Fitness = AL + BD = 0 + 2 = 2<br><br>z = (2, 3, 10)<br>Path(z) = <1,2,3,5><br>AL = 0<br>BD = 7<br>Fitness = AL + BD = 0 + 7 = 7 | x = (2, 2, 3)<br>Path(x) = <1,2,3,5><br>AL = 2<br>BD = 1 / 2 = .5<br>Fitness = AL + BD = 2 + .5 = 2.5<br><br>y = (2, 3, 5)<br>Path(y) = <1,2,3,5><br>AL = 0<br>BD = 2 / 3 = .67<br>Fitness = AL + BD = 0 + .67 = .67<br><br>z = (2, 3, 10)<br>Path(z) = <1,2,3,5><br>AL = 0<br>BD = 7 / 8 = .875<br>Fitness = AL + BD = 0 + .875 = .875 |
| Results by fitness<br>y – x – z | Results by fitness<br>y – z – x |

Clearly, when the branch distances are not normalized, they can sometimes overpower the approach levels. This is not desirable because the approach level is a metric that reveals how close to the target a test case has gotten. The approach level should therefore be considered over the branch distance, which does not always happen when the branch distance is not normalized. That said, it is only necessary to normalize when adding both metrics together. If they are compared separately, then the branch distance will not overpower the approach level. This is shown below.

| Comparing AL |
| --- |
| x = (2, 2, 3)<br>Path(x) = <1,2,3,5><br>AL = 2<br><br>y = (2, 3, 5)<br>Path(y) = <1,2,3,5><br>AL = 0<br><br>z = (2, 3, 10)<br>Path(z) = <1,2,3,5><br>AL = 0 |
| Results by fitness<br>y and z – x |
| Comparing BD |
| y = (2, 3, 5)<br>Path(y) = <1,2,3,5><br>BD = 2<br><br>z = (2, 3, 10)<br>Path(z) = <1,2,3,5><br>BD = 7 |
| Results by fitness<br>y – z – x |

The reason why the approach level does not need to be normalized is because it is not affected by anything other than the diverging path length. Unlike the branch distance, the value of the approach level cannot exceed that of the depth of the control-flow graph. And so, regardless of the magnitude of the test inputs, the value of the approach level will never be too large. That being said, the approach level is already being considered over the branch distance when the latter is normalized, and so having large results would not hinder the fitness. Also, if we were to normalize the approach level, the results might be worse-performing due to the branch distance having a more similar (or even higher) level of impact on the fitness.

End.