IMPLEMENTATION

DIVYA REDDY - 300290332
MAHEEDHAR VUNDELA - 300282606

To implement a new search algorithm into Alternate Variable Method Framework.

INTRODUCTION:

Alternating Variable Method (AVM)- The Alternating Variable Method (AVM) is a local search method that was originally used to generate numerical test data automatically. It is a quick and efficient local search technique. The method takes the solution as an input vector and attempts to optimise it through an individual search process.Iterated Pattern Search is the variable search process used in this approach (IPS). IPS uses two types of moves: exploratory moves and pattern moves. Exploratory moves help us determine the direction, while pattern moves help us get closer to the optimum value.
The optimal direction contributes to the pattern's growth until it no longer improves the objective value. In this case, AVM must test to see if the optimum value was missed. This technique varies depending on the type of search.

SEARCH ALGORITHM: HILL CLIMBING

Hill climbing is a heuristic search algorithm which continuously moves in an elevated direction to reach the peak of the hill (best solution). A node of hill climbing algorithm has two components which are state and value.

Algorithm

1. Examine the current state, Return success if it is a goal state

2. Continue the Loop until a new solution is found or no operators are left to apply

3. Apply the operator to the node in the current state

4. Check for the new state

    - If Current State = Goal State, Return success and exit
    - Else if New state is better than current state then Goto New state
    - Else return to step 2

5. Exit

Pseudo code

```
i=initial solution
While f(s) <= f(i) (s belongs to Neighbours(i)) do
Generates a Neighbours(i);
If fitness(s) > fitness(i) then
        Replaces s with i;
End if
```

Reason for choosing this algorithm:

We don't need to handle or maintain the search tree or graph because this method simply maintains a single current state. It can also be used to solve pure optimization issues, where the goal is to identify the best possible state given the objective function. It is also known as greedy local search because it only considers its good immediate neighbouring state and nothing else. We don't need to handle or maintain the search tree or graph because this method simply maintains a single current state.

Fitness Function:

The fitness function value is calculated using the approach-level and branch-distance metrics. These two numbers represent the number of branches that lead to the test target but were not executed by the test candidate, as well as the shortest distance required to satisfy the branch condition when the candidate's execution diverged from the target's path. The approach level improves the branch distance fitness, which helps connect each branch to a level. The number of branches that separate a branch from the test target helps determine its level.

IMPLEMENTATION:

HillClimbing.java in local search folder:
To initialize the vector using initialize() function and invoke the search “Hill Climbing”.

```
public static final int MAXIMA = 100;
private int n;
int final_vector;

public HillClimbing() {}

protected void performSearch() throws TerminationException {
    initialization();
    hillClimbing();
}
```

Examine the current state, Return success if it is a goal state

```
protected void initialization() throws TerminationException {
    System.out.println(vector);
    initial_value = objFun.evaluate(vector);
    final_vector = var.getValue();
}
```

Continue the Loop until a new solution is found or no operators are left to apply

Check for the new state

```
protected void hillClimbing() throws TerminationException {
    Boolean climb = true;
     //This loop iterates through the neighbour list and finds the optimal neighbour
    while (climb)
    {
       ArrayList<Integer> neighbours = getNeighbours(var.getValue());
       climb = false;

       for (Integer N : neighbours)
       {
          var.setValue(N);
          next_value = objFun.evaluate(vector);
          if (next_value.betterThan(initial_value))
          {
             initial_value = next_value;
             climb = true;
             final_vector = N;
          }
       }
       var.setValue(final_vector);
    }
}
```

The getNeighbours() method returns the closest neighbours of a given input. It iterates MAXIMA times, which is currently set to 100 but can be changed later to suit the test case. It returns all the neighbours of a given vector.

```
    public ArrayList<Integer> getNeighbours(int current)
    {
            ArrayList<Integer> neighbours = new ArrayList<>();
            for (int i = 0; i < MAXIMA ; i++)
              {
                  int randomNeighbour = (int) ((current / 2) + (current * Math.random()));
                  neighbours.add(randomNeighbour);
              }
            return neighbours;
    }

}
```

GenerateInputData.java

The user's input is used to determine the type of algorithm to use (which is already assigned to SEARCH NAME as "HillClimbing").

```
// CHANGE THE FOLLOWING CONSTANTS TO EXPLORE THEIR EFFECT ON THE SEARCH:
// - search constants
static final String SEARCH_NAME = "HillClimbing"; // can also be set at the command line
static final int MAX_EVALUATIONS = 100000;

public static void main(String[] args) {
  // Nested class to help parse command line arguments
  GenerateInputDataArgsParser argsParser = new GenerateInputDataArgsParser(args);
```

The AVM object receives the randomizer, initializer, vector, Objective function, and local search parameters.

```
    // set up the local search, which can be overridden at the command line
    LocalSearch localSearch = argsParser.parseSearchParam(SEARCH_NAME);

    // set up the objective function
    ObjectiveFunction objFun = testObject.getObjectiveFunction(target);

    // set up the vector
    Vector vector = testObject.getVector();

    // set up the termination policy
    TerminationPolicy terminationPolicy =
        TerminationPolicy.createMaxEvaluationsTerminationPolicy(MAX_EVALUATIONS);

    // set up random initialization of vectors
    RandomGenerator randomGenerator = new MersenneTwister();
    Initializer initializer = new RandomInitializer(randomGenerator);
```

AVM sets up all the input data and initialises the monitor. By establishing an async connection with AVM, the monitor acts as a continuous tracker of any action that occurs there.

AlternatingVariableMethod.java

This function creates an AVM instance.

localSearch A local search instance.
Tp - The termination policy to be used by the search.
initializer - The initializer to be used to initialize variables at the start of the search.

1. Performs an AVM search on any arbitrary vector, whether it is the complete vector that needs to be optimised or a variable that can function as a vector on its own. This vector needs to be improved. An exception will be raised if the search is stopped (as specified by the TerminationPolicy used to create this instance).
2. Invokes a search for a certain variable.

   (VariableSearch(var))

3. Once the goal function value does not improve, try a series of moves to gradually increase and reduce the size of a vector variable (current condition). Prior to making any moves, the goal value of the whole vector. increase Specifies whether the vector variable's size should be increased (true) or decreased (false) (value is false).
4. Every vector that is passed is then checked for improvements against previous inputs, and if any are found, a single variable from that vector is passed to variableSearch, which then sends it to atomic variable search, which sends the vector, along with the objective function, to the chosen local search algorithm, in our case hill climbing.

```
protected void alternatingVariableSearch(AbstractVector abstractVector)
    throws TerminationException {
  ObjectiveValue lastImprovement = objFun.evaluate(vector);
  int nonImprovement = 0;

  while (nonImprovement < abstractVector.size()) {

    // alternate through the variables
    int variableIndex = 0;
    while (variableIndex < abstractVector.size() && nonImprovement < abstractVector.size()) {

      // perform a local search on this variable
      Variable var = abstractVector.getVariable(variableIndex);
      variableSearch(var);

      // check if the current objective value has improved on the last
      ObjectiveValue current = objFun.evaluate(vector);
      if (current.betterThan(lastImprovement)) {
        lastImprovement = current;
        nonImprovement = 0;
      } else {
        nonImprovement++;
      }

      variableIndex++;
    }
  }
}
```

EVALUATION:

java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmframework.examples.GenerateInputData Triangle 6T

```
(base) divya@Divyas-MacBook-Pro avmf %  java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmfra
mework.examples.GenerateInputData Triangle 6T
8
[376, 406, 665]
[404, 406, 665]
[404, 406, 665]
[404, 406, 665]
[407, 406, 665]
[407, 407, 665]
[407, 407, 408]
[407, 407, 408]
[407, 407, 408]
Best solution: [407, 407, 407]
Best objective value: Approach Level=0, Branch Distance=0.0
Number of objective function evaluations: 1416 (unique: 1048)
Running time: 8ms
```

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

```
(base) divya@Divyas-MacBook-Pro avmf %  java -cp target/avmf-1.0-jar-with-dependencies.jar org.avmfra
mework.examples.GenerateInputData Triangle 1T
8
Best solution: [217, 152, 401]
Best objective value: Approach Level=0, Branch Distance=0.0
Number of objective function evaluations: 1 (unique: 1)
Running time: 1ms
```