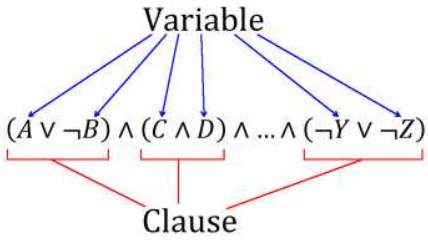
G52AIM Framework API

1 INTRODUCTION

This framework is a Java framework designed for the teaching of heuristic decision support/AI methods. It includes the MAX-SAT problem domain as an example of an NP-hard optimisation problem. A MAX-SAT problem is defined as a Boolean formula in **c**onjunctive **n**ormal **f**orm (CNF) which is a conjunction of disjunctive clauses. Each clause itself contains one or more **variables**.



A solution is represented by a binary string (sequence of bits) where each bit represents a truth value of each variable. In the above example, we have the variables $\{A, B, C, D, Y, Z\}$. A solution to this could be represented as 011011, meaning that B, C, Y, and Z are true, and A and D are false.

The objective for MAX-SAT is to maximise the number of **satisfied** clauses. This framework aims to **minimise** the objective value; hence a modified objective function is used which is to minimise the number of unsatisfied clauses.

2 API SPECIFICATION

Below are all the Classes and methods etc. that you will need to use in the following lab sessions. Classes/methods/etc. which are used by each `LabRunner`, and hence not required to complete the courseworks will not be included in this specification but can be found in the Javadoc comments. This document will be your reference to the framework's API. Any additional information will be supplied in each lab's respective exercise sheets.

2.1 SAT

`Creates a new instance of the MAX-SAT problem and initialises solutions in each memory index.`

```
// Returns true if the termination criterion has lapsed, else returns false.
public boolean hasTimeExpired();

// Copies the solution from the originIndex to the destinationIndex.
public void copySolution(int originIndex, int destinationIndex);

// Evaluates and returns the objective value of the solution in memory index index.
public double getObjectiveFunctionValue(int index);
```

```
// Flips the truth value of the variable variable in the current solution index (0).
public void bitFlip(int variable);

// Flips the truth value of the variable variable in memory index memoryIndex.
public void bitflip(int variable, int memoryIndex);

// Exchanges the bits in index variableIndex between the solutions in indicies
solutionIndexA and solutionIndexB.
public void exchangeBits(int solutionIndexA, int solutionIndexB, int variableIndex);

// Gets the objective value of the best solution found.
public double getBestSolutionValue();

// Creates a new solution in memory index index initialised as a random binary
string.
public void createRandomSolution(int index);

// Gets the number of variables in the current problem instance.
public int getNumberOfVariables();

// Gets the number of memes associated with the solutions.
public int getNumberOfMemes();

// Gets the Meme at index memeIndex for the solution in memory index solutionIndex.
public Meme getMeme(int solutionIndex, int memeIndex);
```

2.2 MEME

```
// Returns the current option of the Meme.
public int getMemeOption();

// Sets the option of the Meme returning false if the supplied option exceeds the
total number of possible options of this Meme.
public boolean setMemeOption(int option);
```

2.3 ARRAYMETHODS

```
// Returns a shuffled copy of the input arraylist, array, using the supplied random
number generator, random.
public static ArrayList<Integer> shuffle(ArrayList<Integer> array, Random random);

// Returns a shuffled copy of the input array, array, using the supplied random
number generator, random.
public static int[] shuffle(int[] array, Random random);
```

2.4 SATHeuristic

2.4.1 Global Variables

```
// The random number generator.
final Random random;

// The memory index where the solution currently being operated on is.
final int CURRENT_SOLUTION_INDEX;

// The memory index where a backup of the current solution is stored.
final int BACKUP_SOLUTION_INDEX;
```

2.4.2 Methods

```
// Applies this heuristic to the problem at memory index CURRENT_SOLUTION_INDEX.
public abstract void applyHeuristic(SAT problem);
```

2.5 PopulationHeuristic

2.5.1 Global Variables

```
// Reference to the problem being solved.
final SAT problem;

// The random number generator.
final Random random;
```

2.5.2 Methods

```
// Applies this heuristic to the problem at memory index solutionIndex.
Public abstract void applyHeuristic(int solutionIndex);
```

2.6 SinglePointSearchMethod

Creates a search method with a population size of 2; one for the current solution, and one for the backup solution. *Note that when an instance of a SinglePointSearchMethod is created, the solution in the current solution index is copied to the backup solution index.*

2.6.1 Global Variables

```
// Reference to the problem being solved.
final SAT problem;

// The random number generator.
final Random random;

// The memory index where the solution currently being operated on is.
final int CURRENT_SOLUTION_INDEX;

// The memory index where a backup of the current solution is stored.
final int BACKUP_SOLUTION_INDEX;
```

2.6.2 Methods

// Inner loop of the search method. **Note that the while loop of each search method has been lifted to the LabRunner Classes to allow for data collection.** The responsibility of this loop is therefore to apply the logic for each iteration of the search method and to ensure that the accepted solution is stored in the **CURRENT_SOLUTION_INDEX**.

```
protected abstract void runMainLoop();
```

2.7 PopulationBasedSearchMethod

Creates a search method with a specified population size. Internally, there are $2 \times POPULATION\_SIZE + 1$ memory indices for storing solutions as follows:

- $0 \rightarrow POPULATION\_SIZE - 1$ are used for storing the **current/accepted** population.
- $POP\_SIZE \rightarrow 2 \times POP\_SIZE - 1$ are used for storing intermediate solutions/**offspring**.
- $2 \times POP\_SIZE$ is used for storing a backup solution if required.

2.7.1 Global Variables

```
// Reference to the problem being solved.
final SAT problem;

// The random number generator.
final Random random;

// The population size. Note the memory scheme detailed above!
final int POPULATION_SIZE;

// The memory index where a backup solution is stored.
final int BACKUP_SOLUTION_INDEX;
```

2.7.2 Methods

// Inner loop of the search method. **Note that the while loop of each search method has been lifted to the LabRunner Classes to allow for data collection.** The responsibility of this loop is therefore to apply the logic for each iteration of the search method and to ensure that the accepted population is stored in the indices $0 \rightarrow POPULATION\_SIZE - 1$.

```
protected abstract void runMainLoop();
```

3 EXAMPLES

3.1 CREATING A RANDOM BIT HILL CLIMBING HEURISTIC

Pseudo-code for Random Bit HC.

```
1. | bit <- random ∈ [0, N-1];
2. | s' <- bitflip(s, bit);
3. | if(f(s') ≤ f(s)) {
4. |   s <- s';
5. | }
```

Solution

```
7  public class RandomMutationHillClimbing extends SATHeuristic {
8
9      public RandomMutationHillClimbing(Random random) {
10
11         super(random);
12     }
13
14     public void applyHeuristic(SAT problem) {
15
16         //record current solution fitness
17         double currentFitness = problem.getObjectiveFunctionValue(CURRENT_SOLUTION_INDEX);
18
19         //apply random mutation
20         int variable = random.nextInt(problem.getNumberOfVariables());
21         problem.bitFlip(variable, CURRENT_SOLUTION_INDEX);
22
23         //record new solution fitness
24         double newFitness = problem.getObjectiveFunctionValue(CURRENT_SOLUTION_INDEX);
25
26         //accept if new not worse than current else revert to original
27         if(newFitness > currentFitness) {
28
29             problem.bitFlip(variable, CURRENT_SOLUTION_INDEX);
30         }
31     }
32
33     public String getHeuristicName() {
34
35         return "Random Mutation Hill Climbing";
36     }
37
38 }
39
```

Figure 1 - Code for Random Bit Hill Climbing Example.

Description

Since we are creating a new heuristic, we must extend the `SATHeuristic` class. Within the `applyHeuristic` method, we start by evaluating the objective value of the current solution, $f(s)$, storing this as the `currentFitness`. The next task is to flip a random bit. This is achieved by generating a random number between 0 and the total number of variables in the problem. Therefore, we must use the `nextInt(int upperBound)` method of the supplied Random number generator `random` and pass it the number of variables. Next, we re-evaluate the solution in the current solution index to get the objective value of the candidate solution, $f(s')$. We then want to accept this candidate solution if and only if $f(s') \leq f(s)$, hence we first check for the opposite. This is because the current solution index already contains s'. Thus if $f(s') > f(s)$, we must revert the bit flip operation as in Line 29. The accepted solution now resides in the current solution index and we are finished.

3.2 Creating a Naïve Acceptance Metaheuristic

Pseudo-code for Naïve Acceptance

```
1. | s <- initialiseSolution();
2. | while( !hasTimeExpired() ) {
3. |   s' <- h(s);
4. |   P <- random ∈ [0,1];
5. |   if(f(s') < f(s) || P < 0.5) {
6. |     s <- s';
7. |   }
8. | }
```

Solution

```
11 public class NaiveAcceptance extends SinglePointSearchMethod {
12
13     // define the move operator(s)
14     private final SATHeuristic h;
15
16     public NaiveAcceptance(SAT problem, Random random) {
17
18         super(problem, random);
19         this.h = new RandomBitFlipHeuristic(random);
20     }
21
22     @Override
23     protected void runMainLoop() {
24
25         // generate the candidate solution
26         h.applyHeuristic(problem);
27
28         // generate a random number in [0,1]
29         double P = random.nextDouble();
30
31         // current solution (s) backed up in the BACKUP_SOLUTION_INDEX
32         double current = problem.getObjectiveFunctionValue(BACKUP_SOLUTION_INDEX);
33
34         // candidate solution (s') in the CURRENT_SOLUTION_INDEX
35         double candidate = problem.getObjectiveFunctionValue(CURRENT_SOLUTION_INDEX);
36
37         // Naive Acceptance
38         if(candidate < current || P < 0.5) {
39
40             // accept solution - retain the solution in CURRENT and create a backup
41             problem.copySolution(CURRENT_SOLUTION_INDEX, BACKUP_SOLUTION_INDEX);
42         } else {
43
44             // reject solution - write over the solution in CURRENT with the backup
45             problem.copySolution(BACKUP_SOLUTION_INDEX, CURRENT_SOLUTION_INDEX);
46         }
47     }
48
49     public String toString() {
50         return "Naive Acceptance";
51     }
52 }
53
```

Figure 2 - Code for Naive Acceptance Example.

Description

Since we are creating a new (single point based) metaheuristic, we must extend the `SinglePointSearchMethod` Class. Remember that solution initialisation and the while loop are lifted into the test frame, hence we only need to implement the inner loop (Lines 3-7) 'ensuring that the accepted solution is stored in the CURRENT_SOLUTION_INDEX', as stated in Section 2.6.2.

When a `SinglePointSearchMethod` is created, the solution in memory index `CURRENT_SOLUTION_INDEX` is copied to memory index `BACKUP_SOLUTION_INDEX`.

At each iteration, **you must ensure this assertion by copying solutions between the two memory indices in the accept/reject branches**; it is then safe to assume that the current solution (s) is backed up in the memory index BACKUP_SOLUTION_INDEX and that this is the same solution as that in memory index CURRENT_SOLUTION_INDEX.

To begin (code Line 26), we apply the heuristic to the current problem as in Pseudocode Line 3. Next we have to generate a random number $P \in [0,1]$. This is achieved by using the `nextDouble()` function of the random number generator. Then we evaluate the objective values of the current and candidate solutions, remembering that the current solution (s) is backed up in memory index `BACKUP_SOLUTION_INDEX` and the candidate solution (s') is the solution we are operating on; hence it is in memory index `CURRENT_SOLUTION_INDEX`. We then apply the Naïve Acceptance move acceptance method which accepts a candidate move if $f(s') < f(s)$ or with a 50% probability if it is not. We then accept the move $(s \leftarrow s')$ by creating a backup of the solution in the current solution index, or reject the move $(s \leftarrow s)$ by overwriting the solution in the current solution index with the backup we made earlier in the backup solution index.