Automated Design of Optimisation Algorithms
User Manual

Student ID: 20215171

April 2023

Contents

1 Introduction

In this document, we provide an overview of the software implemented and provide detailed instructions on how to customize and extend it. The program is designed to solve the Maximum Weighted Clique Problem (MWCP) using the Conditional Markov Chain Search (CMCS) framework and Simulated Annealing (SA) algorithm for optimizing CMCS configurations.

By following this manual, you will be able to extend and modify this software to implement different heuristic components or change existing functionality. For a detailed explanation of the methodologies and theoretical background of this project please refer to the main dissertation.

The benchmark instances are placed inside the folder:/COMP3003/DIMACS

The CMCS configurations are placed inside the folder:/COMP3003/Configurations

Java docs are inside the folder:/COMP3003/docs

2 System Overview

The software is divided into main components responsible for implementing the CMCS framework, SA algorithm, and objective functions, as well as sub-components for loading instances, transition matrices, and handling the various heuristic components.

The main components of the Software are:

- "Main", the Main class acts as the entry point for the program, controlling its functionality through user input and initializes and manages the components.
- "CMCS", a java class implementing the CMCS framework which takes an input of list of heuristic components H, M^{succ} transition matrix, M^{fail} transition matrix, termination time and adjacency matrix representing the problem instance.
- "SimulatedAnnealing", a java class implementing the SA algorithm for optimizing CMCS configurations. The input is a set of MWCP instances and list of heuristic components H.
- "ObjectiveFunctionSA", a java class implementing the objective function for SA which is crucial for configuration the optimization process, the objective function is implementation is done according to methods outlined in the dissertation.
- "ObjectiveFunctionMWCP", this class calculates the objective value of a

solution for a specific MWCP instance. It takes input of a solution and an adjacency matrix representing the problem instance and returns the objective value of the solution.

- "EvaluateCMCS", this component evaluates a configuration on a set of MWCP instances for a specified number of runs. This class is only used for evaluation of configurations and producing results and is not used in the SA algorithm.

The sub components of the program are:

- "AdjacencyMatrixLoader", this class loads a single MWCP instance, the instance must be in the DIMACS (ASCII, undirected) format. This component takes input the filename of the instance and returns an adjacency matrix for this instance.

- "InstancesLoader", this java class loads a set of instances. It takes as an input a file path which is searched for DIMACS MWCP instances and each instance found is loaded using the "AdjacencyMatrixLoader" class, along with the adjacency matrix associated with the instance also the instance name is stored and the number of total instances is stored. All the instances along with their name and index are stored in an array of adjacency matrices which is the output.

- "TransitionMatrixLoader", this class loads a CMCS configuration from a .txt file. Both transition matrices, M^{succ} and M^{fail}, are stored in the same file. The class loads the matrices and returns an array containing both matrices which can then be used in the "CMCS" and "SimulatedAnnealing" components.

- "CliqueValidation", this class was used for the initial stages of development for testing the validity of MWCP solutions. It takes a solution and an adjacency matrix representing a MWCP instance. It checks if the solution is valid, i.e, all the vertices included in the solution form a valid clique in the graph represented by the adjacency matrix.

- "Heuristic", an interface which serves as a standard blueprint for the different heuristic components, which allows the CMCS framework to work with the heuristic components in a uniform way. This makes it easy to switch between heuristics during CMCS search process.

  - "HillClimber1", a class implementing the "Heuristic" interface, this component implements the "HillClimber1". This component improves the solution stochastically, terminating only when a local maximum is reached.

  - "HillClimber2", a class implementing the "Heuristic" interface, this component implements the "HillClimber2" component. This component improves the solution using a greedy best improvement approach, terminating only when a local maximum is reached.

  - "Mutator1", a class implementing the "Heuristic" interface, this component implements the "Mutation1" component. This component flips 1/5 of the vertices included in the solution clique.
  - "Mutator2", a class implementing the "Heuristic" interface, this component implements the "Mutation2" component. This component flips 1/4 of the vertices included in the solution clique.
  - "Mutator3", a class implementing the "Heuristic" interface, this component implements the "Mutation3" component . This component flips 1/2 of the vertices included in the solution clique.
  - "Mutator4", a class implementing the "Heuristic" interface, this component implements the "Mutation4" component. This component flips all the vertices to 0 and initializes a new solution by choosing a random vertex and flipping it to 1.

3 Customizing and Extending the Program

This section provides an overview of possible extension to the software and how to implement additional heuristic components.

Adding New Heuristic Components to the program is relatively easy and requires only a few steps. This is because the CMCS framework treats the heuristic components as black box algorithms. The only condition is that the new component solve for the max weight clique problem.

Before adding the component it must take the following parameters:

- A binary encoded array representing a solution.
- An adjacency matrix representing the problem instance.
- A 'ObjectiveFunctionMWCP' object which is the objective function.

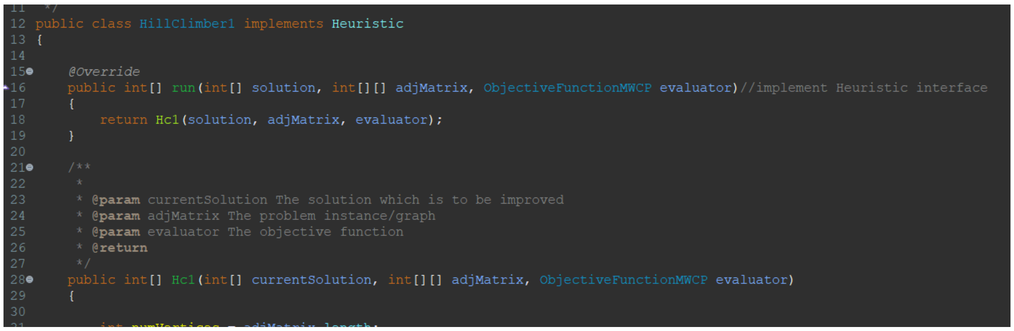
```
(int[] currentSolution, int[][] adjMatrix, ObjectiveFunctionMWCP evaluator)
```

Figure 1: Required parameters

The steps to adding a new heuristic component:

- Add a new class implementing the new heuristic component. If the class is in a different package it must be imported in 'Main' class.

- The new class must be implementing the Heuristic interface.



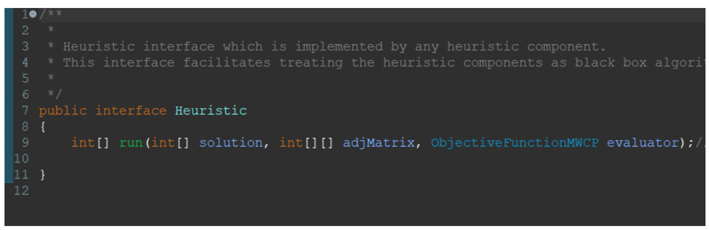
```
public class HillClimber1 implements Heuristic
{

    @Override
    public int[] run(int[] solution, int[][] adjMatrix, ObjectiveFunctionMWCP evaluator)//implement Heuristic interface
    {
        return Hc1(solution, adjMatrix, evaluator);
    }

    /**
     *
     * @param currentSolution The solution which is to be improved
     * @param adjMatrix The problem instance/graph
     * @param evaluator The objective function
     * @return
     */
    public int[] Hc1(int[] currentSolution, int[][] adjMatrix, ObjectiveFunctionMWCP evaluator)
    {

```

Figure 2: Example of the required implementation of the ’Heuristic’ interface



```
/**
 *
 * Heuristic interface which is implemented by any heuristic component.
 * This interface facilitates treating the heuristic components as black box algori
 *
 */
public interface Heuristic
{
    int[] run(int[] solution, int[][] adjMatrix, ObjectiveFunctionMWCP evaluator);/

}

```

Figure 3: The ’Heuristic’ Interface

- Finally, a new object of the new heuristic class must be instantiated in the ’Main’ class. The instance of the new class must then be added to the ’heuristics’ list as shown:

```java
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException
    {
        Scanner scanner = new Scanner(System.in);//Scanner object instantiation, is usec

        /* Instantiating heuristic components
         * Any heuristic component must be instantiated here
         */
        HillClimber1 hillClimber1 = new HillClimber1();
        HillClimber2 hillClimber2 = new HillClimber2();
        Mutator1 mutator1 = new Mutator1();
        Mutator2 mutator2 = new Mutator2();
        Mutator3 mutator3 = new Mutator3();
        Mutator4 mutator4 = new Mutator4();

        /* Adding the heuristic components to a list of type "Heuristic"
         * Any heuristic component must be added to this list to be usable in CMCS
         * The order of this list is significant as it translate to the index of each cc
         */
        List<Heuristic> heuristics = new ArrayList<>();
        heuristics.add(hillClimber1);
        heuristics.add(hillClimber2);
        heuristics.add(mutator1);
        heuristics.add(mutator2);
        heuristics.add(mutator3);
        heuristics.add(mutator4);

```

Figure 4: Adding heuristic component to 'heuristics' list