Coursework 2 – Report

# 1 Question and Example-based Essay

## 1.1 Random Initialisation

### (1)

Each solution is represented in the form of an array, where the size is the number of delivery locations and each entry consists of a delivery ID which corresponds to a specific delivery location, i.e. an instance of Location() which consists of an x and a y coordinate. All delivery locations are listed in order from line 8-13 in the form of their coordinates in the instance file for the square problem (see figure 1.1.1). Thereby, solution s1 which is represented as 0,1,2,3,4,5, corresponds to a route where all delivery locations are visited in the order of the square.pwp file.

A screenshot of a cell phone

Description automatically generated

Figure 1.1.1

### (2)

See figure 1.1.2 for a visualisation of the complete route of solution s1, including all locations, their types (delivery location, postal depot or home address) and their coordinates (x,y).

A close up of text on a white background

Description automatically generated

Figure 1.1.2

### (3)

To initialise a random solution s0 to a problem instance, initialiseSolution(int index) method is used to access the currently loaded instance read from a file (Square.pwp in this case) and to create a random solution for the said instance using the createSolution() method. In createSolution(), a PWPSolution() object is initialised from the PWPInstance() object using the mode parsed to the method from the enum InitialisationMode() (in this case, only the RANDOM mode was implemented). The method initialises an Integer List of the size of number of deliveries where entries ascend from 0 to (size-1) and the java.util.Collections method shuffle() is used to shuffle the entries in the list using the random seed parsed to the constructor of the PWPInstance. The list is then converted to an int array permutationArray[] which gives the random permutation of delivery locations to be used as the representation of the solution and to create its objective value. A new PWPSolution() object can now be created by parsing the representation and the objective value to it. To access the actual delivery location which each index of the representation correspond to, the getLocationForDelivery(int deliveryId) method can be used which returns a Location() object based on the parsed index.

### (4)

See figure 1.1.3 for a visualisation of the complete route of solution s0 (a randomly initialised solution created with the method explained above), including all locations, their types (delivery location, postal depot or home address) and their coordinates (x,y).

A close up of text on a white background

Description automatically generated

Figure 1.1.3

## Inversion Mutation

The basic structure of the apply() in inversionMutation() is the same as the apply() in the other heuristics. At the start, the termination criterium is identified as the int variable ‘limit’ which tells how many times to perform the heuristic. The int variable ‘noLocations’ is set to be the number of delivery locations and three doubles are initialised to be used for the objective value. Then comes a for loop which runs until the termination criteria is met and for each iteration it performs the following actions (see what happens to the example solution representation s = [0,4,1,3,5,2] in bold):

1. Calculate the objective value *objVal* of the solution.
2. Generate two random indices indexA and indexB to be used as the indices of the locations for the inversion, where the latter has to be larger than the former. **indexA = 1, indexB = 5.**
3. Subtract preCost and postCost from the objective value (cost for parts of the route which will be changed based on the indices indexA and indexB).
4. Perform inversion mutation on the current PWPSolutionInterface ‘solution’ by calling reverseSubarray() which takes the solution and returns the new solution where the subarray between indices indexA and indexB has been reversed according to the following steps:
   1. Create an Integer List of the subarray. **[4,1,3,5,2].**
   2. Reverse the list using the java.util.Collections method reverse(). **[2,5,3,1,4].**
   3. Rewrite the entries of the representation where indexA<=*index*<=indexB with values from the reversed list. **s = [0,2,5,3,1,4].**
   4. The method then updates the solution to contain the modified representation and then returns the new solution.
5. Calculate the new preCost and postCost and add them to the objective value.
6. Set the objective value of the new solution to be *objVal* which has now been modified through delta evaluation according to the inversion mutation performed on the representation.

After the for loop has finished, the solution will have the latest representation and objective value after multiple iterations of applying inversion mutation to it. The objective value is returned from the apply() method.

## 1.3 Delta Evaluation for Adjacent Swap

### 1.

si = [L1,L2,L3,L4,L5,L6]

Lx = L1

Ly = L2

si+1 = [L2,L1,L3,L4,L5,L6]

f(si+1) = f(si) – C(depot, L1) – C(L2,L3) + C(depot,L2) + C(L1,L3)

### 2.

si = [L1,L2,L3,L4,L5,L6]

Lx = L3

Ly = L4

si+1 = [L1,L2,L4,L3,L5,L6]

f(si+1) = f(si) – C(L2, L3) – C(L4,L5) + C(L2,L4) + C(L3,L5)

### 3.

si = [L1,L2,L3,L4,L5,L6]

Lx = L6

Ly = L1

si+1 = [L6,L2,L3,L4,L5,L1]

f(si+1) = f(si) – C(depot, L1) – C(L1,L2) - C(L5,L6) - C(L6,home) + C(depot, L6) + C(L6,L2) + C(L5,L1) + C(L1,home)

# 2 Tramstops-85

## (1)

To design my hyper-heuristic, I initially did Reinforcement Learning where I maintained a score for each low-level heuristic which was incremented if the heuristic gave an improving objective value and decremented if not. The heuristic selection was based on the one with the highest score. The algorithm got stuck early (at iteration 44) at heuristic 3 because its score had been incremented many times consecutively and therefore was always chosen for the next iteration.

For the final hyper-heuristic, I decided to go for the multi-point based Greedy search (GREEDY\_IE\_HH.java) where I created a for loop to iterate through all heuristics and store their returning objective value in an array. This method allowed to take advantage of all low-level heuristics. After the loop, I chose the best objective value and accepted the candidate solution that its heuristic gave.

## (2)

### a.

The greedy heuristic selection strategy was based on applying each low-level heuristic to the candidate solution and choosing the one that returned the lowest objective value. The candidate solution of that heuristic would then we copied to the current solution.

### b.

The move acceptance method was chosen as *improving or equal* to avoid getting stuck at local optima and to increase the number of solutions explored.

### c.

As the Greedy hyper-heuristic is multi-point based, it means it is compatible with all of the lower level heuristics. Therefore, they were all used in the implementation as using many different types of operators would increase the variety of the solutions.

## (3)

Run GREEDY\_IE\_VisualRunner.java.

At first, it initialises the memory size to 3 and initialises the first solution to the problem instance. Then it sets the parameters intensity of mutation and depth of search. It also initialises a counter for the number of iterations for comparison purposes. An array of doubles *aoObjVals* is then initialised which will be used to store the objective value of the candidate solution when each heuristic is applied.

In a while loop which runs until the time limit of a minute has been reached, the following actions are completed:

1. Iterate through all heuristics in a for loop, applying each to the candidate solution and storing the objective value of it in *aoObjVals*.
2. Get the index of the heuristic which gave the candidate solution of the lowest objective value.
3. Run that selected heuristic again so its candidate solution overwrites what candidate solution is in memory currently.
4. Accept the candidate solution if its objective value is improving or equal to the current value.
   1. Copy the candidate solution into the current solution.
5. Iteration is completed, increment iteration-counter.

When while loop is finished, the best solution value is printed, and the solution’s route is visualised.

# 3 Hyper-Heuristic Comparison

## 3.1 Ranking Comparison

See Spreadsheet Comparison.xlsx, sheet “Ranking Comparison”.

## 3.2 Statistical Comparison

See Spreadsheet Comparison.xlsx, sheet “Statistical Comparison”.