AIM Project 2020 – Implementation of a HyFlex Compatible Postal Worker Problem (PWP) Domain

1 Introduction

The Postal Worker Problem (PWP) is a routing-type problem which states that given N number of postal deliveries, a post office depot, and the postal workers home address, the objective is to find a route which ensures that all postal deliveries are made exactly once such that the total tour length is minimised whereby the postal worker starts their shift at the start of the day at the post office depot, and finishes their day by returning to their home address once all deliveries have been made.

HyFlex is a software framework designed for the implementation and testing of iterative general-purpose search methods and will be/was covered in the "HyFlex Tutorial Lab" (13/03/2020).

2 Objectives

The aim of this project is to implement a HyFlex compatible PWP problem domain. A template code base is given to you to give you a starting point. You may wish to add more classes in your implementation; this is fine if each of the supplied classes and methods function as described in their requirements. There are several components which you must implement, some of which can be accomplished multiple ways with the hardest attracting more marks as detailed in each section. These will be highlighted in this document in **green** to highlight these areas. Below you will find a breakdown of each of these components along with a description of what you are expected to implement.

3 Deadline and Marks

The deadline for providing your implementations is 11/05/2020 – 15:00

You should submit a single zip folder containing all your source files, any libraries used, and the instance files – that is, so that we can run it without any compilation errors caused by missing files!

This project consists of an implementation, and a 2,000-word report with a maximum 4-page limit (using 11pt font!). The weightings for each are 80% implementation, and 20% report.

4 PWP Domain Components

4.1 Problem Representation

You are to use a permutation representation using an array of integers (`int[]`) for representing solutions to the PWP problem instances such that each integer corresponds to an individual postal delivery whose mapping is specified by the order each delivery address appears in the instance file(s).

4.2 INSTANCE READER

Problem instances are stored as *.pwp files and reside in the folder "`instances/pwp/*.pwp`". To be able to "load" an instance using HyFlex's `problem.loadInstance(int id)` method, you should fulfil two components:

1. `PWPInstanceReader` class.
2. `loadInstance(int id)` method.

4.2.1 PWPInstanceReader

You should implement the `readPWPInstance` method in the `PWPInstanceReader` class to read in an instance from a given file, returning a new `PWPInstance` Object. Each PWP instance file is a plain text written in accordance to the following structure where the structure is in bold, and data italic:

```
NAME : <name : string>
COMMENT : <comment : string>
POSTAL_OFFICE
<x coordinate : double> <y coordinate : double>
WORKER_ADDRESS
<x coordinate : double > <y coordinate : double >
POSTAL_ADDRESSES
<x coordinate : double > <y coordinate : double >
…
<x coordinate : double > <y coordinate : double >
EOF
```

The `readPWPInstance` method should return an instance of a `PWPInstance` Object which contains all the information relating to the problem instance. Moreover, it contains a factory method to create solutions to the current problem instance given an `InitialisationMode`. As discussed in Section 4.3, you only need to implement random initialisation.

To create a `PWPInstance`, five components are required: the total number of locations, an array of `Locations`, the `Location` of the post office depot, the `Location` of the worker's home address, and a seeded random number generator.

1. The number of locations can be calculated from the problem instance file.
2. Each `Location` object is essentially a wrapper for an x and y coordinate and should be populated depending on the information from the problem instance file specified by the Path supplied to the `readPWPInstance` method.
3. The random number generator is passed as an argument to the `readPWPInstance` method.

4.2.2 loadInstance(int id)

This method should map instance IDs as integers to each of the problem instances stored in the "`instances/pwp/*.pwp`" folder. Once the respective instance is located, it should use the `PWPInstanceReader` class to load the instance information as a `PWPInstance` Object.

At this point, it is a good idea to set the objective function to be used by each of the heuristics.

4.3 SOLUTION INITIALISATION

The order of postal delivery addresses should be chosen at random (dependent on a seeded random number generator).

4.4 Objective Function

To evaluate the cost of a solution, you should calculate the total distance of the postal workers journey starting at the postal depot and ending at their home. Distances between two locations should be calculated as their Euclidean distance. That is, in the below example, the total distance should be calculated using both sets of green and yellow routes.

Figure 1 - Sample solution illustrating the postal worker's delivery route and return journey.

Standard Evaluation

This is the easiest evaluator to implement. A standard evaluator takes the current solution and calculates the sum of distances between each location, setting the objective function value of the solution once the entire calculation is complete using the method below.

```
PWPSolution.setObjectiveFunctionValue(double objectiveFunctionValue);
```

4.4.1 Delta Evaluation

This is more complicated than the standard evaluation technique but is much faster and will attract more marks (if implemented correctly!). Delta evaluation exploits the fact that if a segment of a route is not changed, then neither will the distance of that route segment. Hence, only the path between locations that are removed and created need to be calculated. **Beware of the trips between the postal office and worker's home locations with the start and end points of the explicitly represented route!**

Each time a heuristic is applied to a solution, its objective value must be updated **during the application of the heuristic**. By subtracting the removed part(s) of the tour from the current objective function value, and adding back the newly created part(s) of the tour using the methods:

```
PWPSolution.getObjectiveFunctionValue();
```

```
PWPSolution.setObjectiveFunctionValue(double objectiveFunctionValue);
```

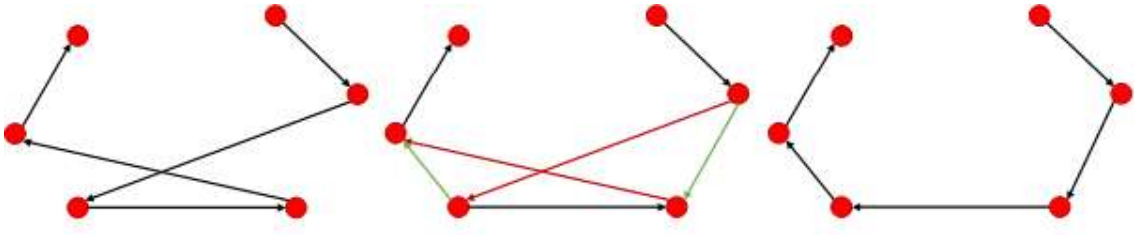

Figure 2 - Example of delta evaluation where the red paths are removed, and green paths are added.

> Note that when delta evaluation is used for crossover heuristics, the calculation time can be slower! We therefore only ask that delta evaluation is implemented for **mutation** and **local search** heuristics.

4.5 Solution Memory Management

In HyFlex, there is a set of solutions of size "`memorySize`" which is maintained by the search method. By default, this should be set to 2 however if `setMemorySize(int size)` is called, then the memory size should be changed accordingly. If a `memorySize` is specified less than or equal to 1, then the request to change the memory size should be ignored.

4.5.1 Tracking of Best Solution

A separate piece of memory should be used to store the best solution found during the search and should be updated after each time a heuristic is fully applied.

4.6 Heuristics

You are asked to implement several heuristics based on mutation, local search, and crossover. The templates for all heuristics can be found in the package "`com.aim.project.pwp.heuristics`". When implementing the AIM_PWP domain, we ask that each of the operators are mapped to heuristic indices as follows:

$$\begin{Bmatrix} 0 \rightarrow Inversion\ Mutation; 1 \rightarrow Adjacent\ Swap; 2 \rightarrow Random\ re\text{-}insertion; 3 \rightarrow Next\ Descent; \\ 4 \rightarrow Davis's\ Hill\ Climbing; 5 \rightarrow Ordered\ Crossover; 6 \rightarrow Cycle\ Crossover \end{Bmatrix}$$

For clarity, the mutation operators are highlighted **red**, local search **blue**, and crossover **green**.

Hint: it might be a good idea to implement such basic perturbations in a base class, one is already provided called `HeuristicOperators`, which your heuristic implementations can extend to inherit common functionality!

4.6.1 Adjacent Swap – AdjacentSwap.java

In adjacent swap, two locations which are adjacent to each other in the current route are switched as shown in Figure 3 by the blue locations. The first location should be selected randomly amongst all delivery locations and the second delivery location chosen as the location visited **after** the chosen location.

Exception if the last location is chosen, then it should be swapped with the first delivery location.

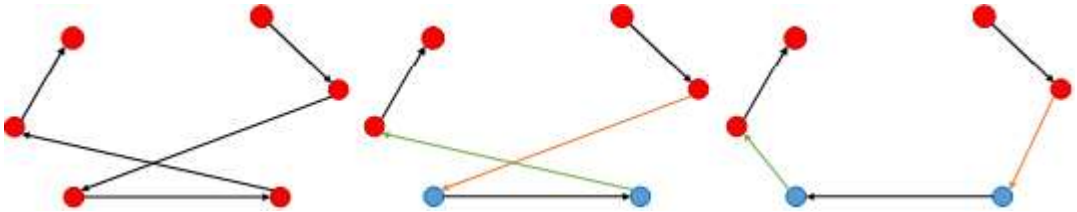

Figure 3 - Illustration of Adjacent Swap Operator

Intensity of Mutation

The intensity of mutation setting should be used to control the number of times that an adjacent swap is performed using the following mapping.

$$0.0 \leq IOM < 0.2 \Rightarrow times = 1$$

$$0.2 \leq IOM < 0.4 \Rightarrow times = 2$$

$$0.4 \leq IOM < 0.6 \Rightarrow times = 4$$

$$0.6 \leq IOM < 0.8 \Rightarrow times = 8$$

$$0.8 \leq IOM < 1.0 \Rightarrow times = 16$$

$$IOM = 1.0 \Rightarrow times = 32$$

4.6.2 Inversion Mutation – InversionMutation.java

In inversion mutation, two delivery locations are selected at random l_a, l_b as indicated by the blue dots and ordered based on their position in the current route. That is, the index of the first selected location $I[l_a]$ appears before $I[l_b]$. The route between these delivery locations, shown in green, is preserved but reversed as shown in Figure 4; the route either side but not adjacent to these delivery locations, shown in orange, is preserved; and the connections between $l_{a-1} \rightarrow l_a$ and $l_b \rightarrow l_{b+1}$ are swapped such that $l_{a-1} \rightarrow l_b$ and $l_a \rightarrow l_{b+1}$ as illustrated by the blue routes. Note that in this example, l_a was selected as the first delivery route, hence the route is updated such that $l_a \rightarrow l_{b+1}$, and l_b is now the first delivery location.

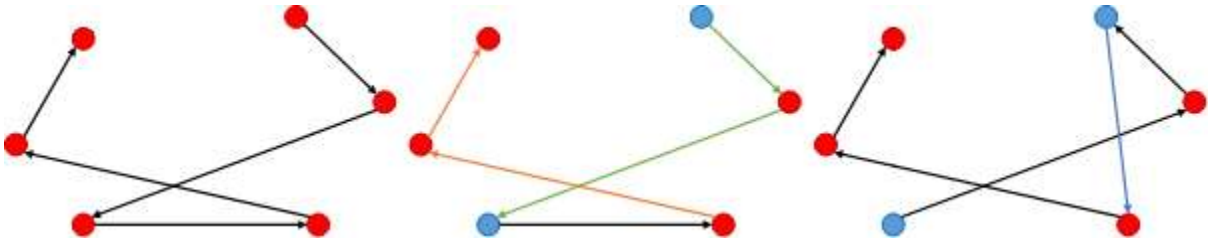

Figure 4 - Illustration of Inversion Mutation.

Intensity of Mutation

The intensity of mutation setting should be used to control the number of times that Inversion Mutation is performed using the following mapping.

$$0.0 \leq IOM < 0.2 \Rightarrow times = 1$$

$$0.2 \leq IOM < 0.4 \Rightarrow times = 2$$

$$0.4 \leq IOM < 0.6 \Rightarrow times = 3$$

$$0.6 \leq IOM < 0.8 \Rightarrow times = 4$$

$$0.8 \leq IOM < 1.0 \Rightarrow times = 5$$

$$IOM = 1.0 \Rightarrow times = 6$$

4.6.3 Random re-insertion – Reinsertion.java

A delivery location is selected at random, removed from the current route, and re-inserted at a different place. In Figure 5, the blue delivery location represents the chosen delivery location, and the yellow line indicates the two delivery locations in between which it has been re-inserted.

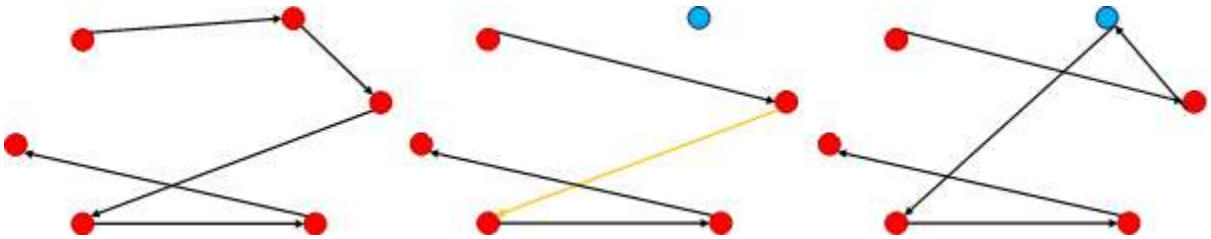

Figure 5 - Illustration of Random re-insertion.

It is up to you to design how the reinsertion will work, as long as:

1. A random delivery location is selected to be reinserted.
2. The selected delivery location can be reinserted into any position in the route.
3. The selected delivery location **is not** reinserted at the same position.

Intensity of Mutation

The intensity of mutation setting should be used to control the number of times that two-opt is performed using the following mapping.

1. $0.0 \leq IOM < 0.2 \Rightarrow times = 1$
2. $0.2 \leq IOM < 0.4 \Rightarrow times = 2$
3. $0.4 \leq IOM < 0.6 \Rightarrow times = 3$
4. $0.6 \leq IOM < 0.8 \Rightarrow times = 4$
5. $0.8 \leq IOM < 1.0 \Rightarrow times = 5$
6. $IOM = 1.0 \Rightarrow times = 6$

4.6.4 Next Descent – NextDescent.java

Next descent tries perturbations of the solution one-by-one and chooses the first of such which generates an improvement (strict improvement; OI) in the solution (if any). The perturbation operator we would like you to use is the Adjacent Swap as described in 4.6.1.

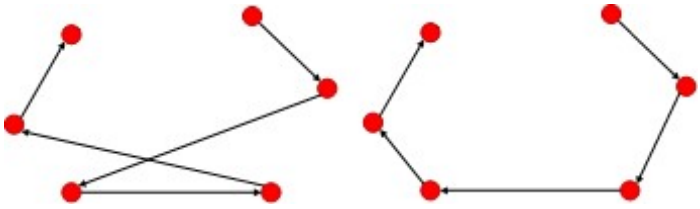

Figure 6 - Illustration of Steepest Descent.

Implementation Considerations

When Next Descent is applied, it is usually applied from the starting delivery location up and makes its way sequentially through the route until it makes its way back to the start. That is, the final delivery location is swapped with the first delivery location. This will favour swapping delivery locations earlier in the tour. We therefore would like you to implement Next Descent such that the starting point of the heuristic is selected randomly and performs a full pass of all delivery locations.

Depth of Search

The depth of search setting should be used to control how many times Next Descent should accept an improvement using the following mapping.

1. $0.0 \leq DOS < 0.2 \Rightarrow times = 1$
2. $0.2 \leq DOS < 0.4 \Rightarrow times = 2$
3. $0.4 \leq DOS < 0.6 \Rightarrow times = 3$
4. $0.6 \leq DOS < 0.8 \Rightarrow times = 4$
5. $0.8 \leq DOS < 1.0 \Rightarrow times = 5$
6. $DOS = 1.0 \Rightarrow times = 6$

Once next descent starts trying adjacent swaps, rather than terminating after the first improvement, each improvement should be persisted (accepted) and the algorithm continues until:

1. The number of accepted improvements exceeds $times$.
2. A complete loop of the solution has completed without finding any improvement.

4.6.5 Davis's Hill Climbing – DavissHillClimbing.java

Davis's Hill Climbing, like Davis's Bit Hill Climbing, performs a sequence of perturbations, persisting any which results in an improvement in the solution. For the perturbation operator, we would like you to use Adjacent Swap as described in 4.6.1. **Remember**, the ordering of the sequence should be randomised such that the order in which the delivery locations are tried is not the order of the current route. You should persist any swap which results in an improving or equal quality solution.

Depth of Search

The depth of search setting should be used to control the number of times that Davis's Hill Climbing is performed using the following mapping.

1. $0.0 \leq DOS < 0.2 \Rightarrow times = 1$
2. $0.2 \leq DOS < 0.4 \Rightarrow times = 2$
3. $0.4 \leq DOS < 0.6 \Rightarrow times = 3$
4. $0.6 \leq DOS < 0.8 \Rightarrow times = 4$
5. $0.8 \leq DOS < 1.0 \Rightarrow times = 5$
6. $DOS = 1.0 \Rightarrow times = 6$

4.6.6 Ordered Crossover – OX.java

See lecture slides "Evolutionary Algorithms II" using two cut points. Note that you should **not** be able to choose the cut points 0 and N simultaneously as this would result in no modification of the solution.

Intensity of Mutation

The intensity of mutation setting should be used to control the number of times that crossover is performed using the following mapping.

1. $0.0 \leq IOM < 0.2 \Rightarrow times = 1$
2. $0.2 \leq IOM < 0.4 \Rightarrow times = 2$
3. $0.4 \leq IOM < 0.6 \Rightarrow times = 3$
4. $0.6 \leq IOM < 0.8 \Rightarrow times = 4$
5. $0.8 \leq IOM < 1.0 \Rightarrow times = 5$
6. $IOM = 1.0 \Rightarrow times = 6$

4.6.7 Cycle Crossover – CX.java

See lecture slides "Evolutionary Algorithms II". The starting point for the cycle should be chosen randomly from parent 1, and you should perform only a single cycle. That is, if there are multiple cycles, only the first cycle is needed for this implementation.

Intensity of Mutation

The intensity of mutation setting should be used to control the number of times that crossover is performed using the following mapping.

1. $0.0 \leq IOM < 0.2 \Rightarrow times = 1$
2. $0.2 \leq IOM < 0.4 \Rightarrow times = 2$
3. $0.4 \leq IOM < 0.6 \Rightarrow times = 3$
4. $0.6 \leq IOM < 0.8 \Rightarrow times = 4$
5. $0.8 \leq IOM < 1.0 \Rightarrow times = 5$
6. $IOM = 1.0 \Rightarrow times = 6$

Operation of Crossover Operators

Ordinarily, a crossover operator will take two parent solutions are produce 2 offspring as shown in Figure 7. The HyFlex Framework however only ever produces one candidate solution. When applying a crossover heuristic, you should therefore take in the two parent solutions, perform the respective crossover operation, and return **one of the offspring chosen at random** by placing it into the child index specified in the call to the apply(…) method as shown in Figure 8.

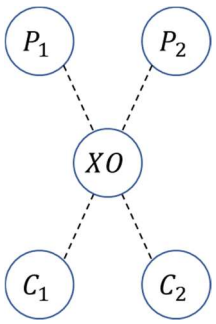

Figure 7 - Ordinary Operation of Crossover.

To apply multiple crossover operations when using non-default intensity of mutation settings, you should first get the solution representations of both P_1 and P_2. You should then apply crossover to produce **two offspring** and continue to apply crossover to each successive round of offspring until the correct number of successive crossover operations have been performed. At the end, you should choose a random one of these offspring to be the returned child C as shown in Figure 9.

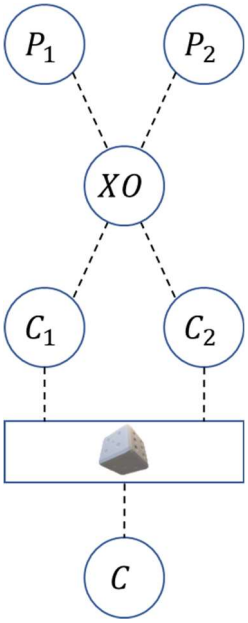

Figure 8 - Operation of Crossover using HyFlex.

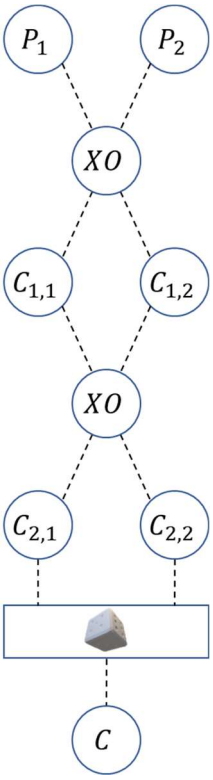

Figure 9 - Operation of Crossover with IOM > 0.2 under the HyFlex Framework.

5 PWP VISUALISER

A HyFlex compatible visualiser tool will be released soon which will enable you to visualise the solutions your algorithms have come up with. This tool is not important for your implementation and is independent from it, but it is nice to see how the solutions change over time and can be used for debugging purposes.

6 PROJECT SETUP

As with the labs, you will need to set up a new project (any name is fine), create the project hierarchy shown in Figure 10, and import the HyFlex Framework. You will also need to create an "instances" folder where the PWP instances are to be stored within the project directory.

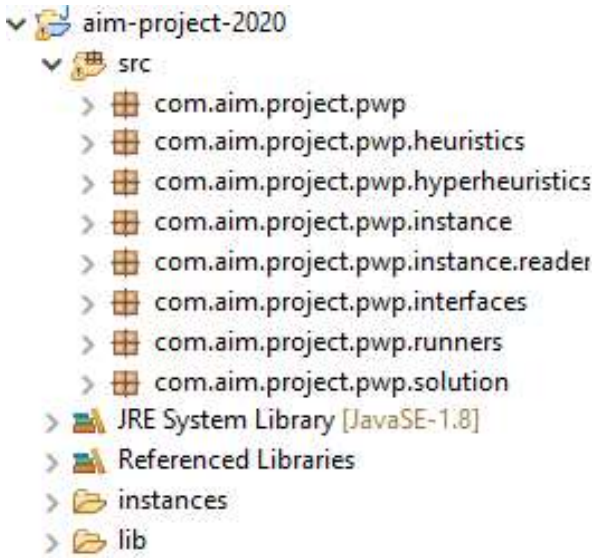

Figure 10 - Project structure.

You should not use any additionally libraries other than the standard java libraries and those provided on the Moodle page, and you must not rename any of the supplied packages.

7 HINTS

Please read these hints before emailing queries!

7.1 GETNUMBEROFHEURISTICS() QUIRK

Due to the way the HyFlex framework was designed, the `ProblemDomain` class calls the `getNumberOfHeuristics()` method on the implementing class during its initialisation **before** the implementing class has had chance to be constructed. Therefore, there is no choice but to hard code the value returned by this method.

8 SHAPES AND INSTANCES

8.1 SQAURE.PWP

This instance has the coordinates of a square using 8 points and is provided for testing purposes. That is, it should be trivial to find the optimal tour with objective value 35.0.

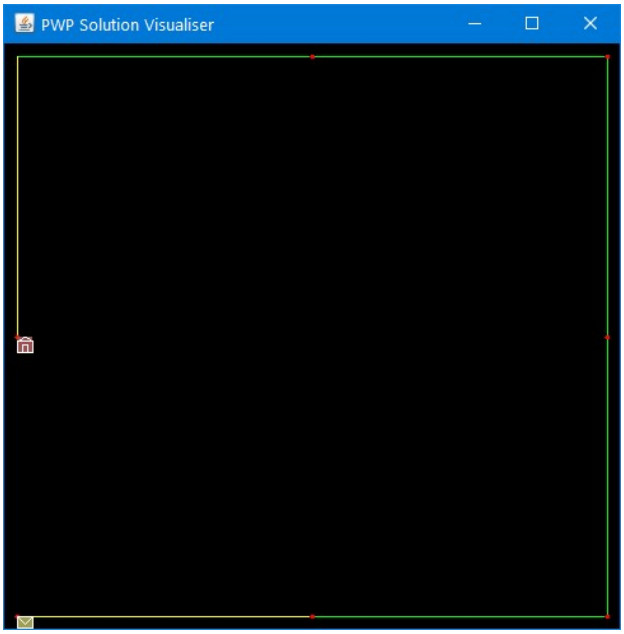

Figure 11 - Optimal tour for square.pwp.

More to be updated soon!