1. Implementation

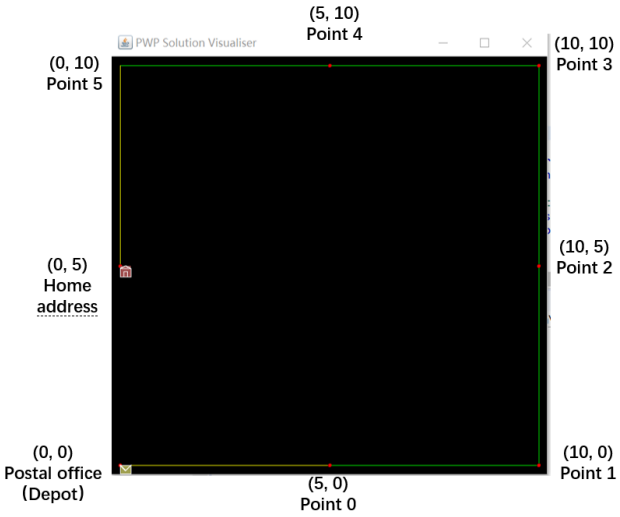
a) Random Initialization

i. Mapping

1. The *PWPInstanceReader class* will read the .pwp file one line after another. The **order of the coordinate** in the .pwp file represents the **order of the location in the instance**, which is the **elements** in the solution representation. For example, in the **s1=[0,1,2,3,4,5],** 0 represents point 0 in the file and 3 represents point 3, as showed in the graph.
2. The solution contains **no duplicate elements** since each address only need to be visited once.
3. The elements of solution represent the index of corresponding points in the instance. The instance class store the location of all the points.

ii. Draws

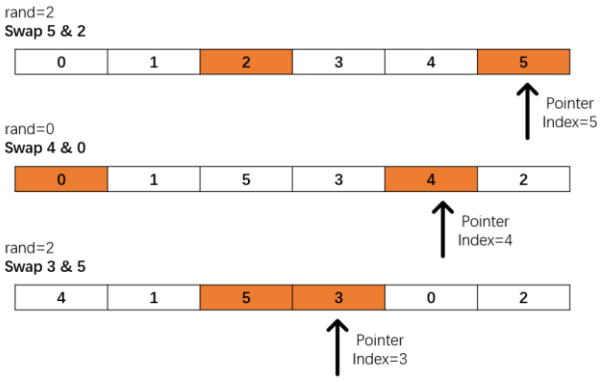
The solution **s1=[0,1,2,3,4,5]** is as followed



iii. The perturbation representation of PWP solution is generated with the following algorithm (implemented in **PWPInstance** class). Below is the general process.

The **initializeSolution (int index)** will first call out the **createSolution (InitializationType mode)** method with **InitializationType.Random** as parameter. The **createSolution (InitializationType mode)** will then call the following function which generates a perturbation array. (Uses **s0=[3,4,1,5,0,2]** as an example)

```
Public int[] PWPperturbationInitializer(int size)
```

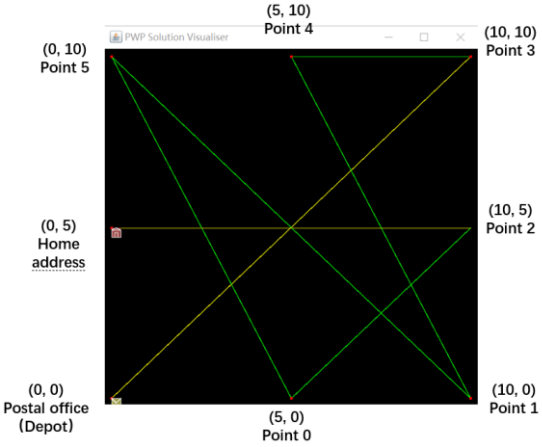


In the PWPpertubationInitializer (int size):

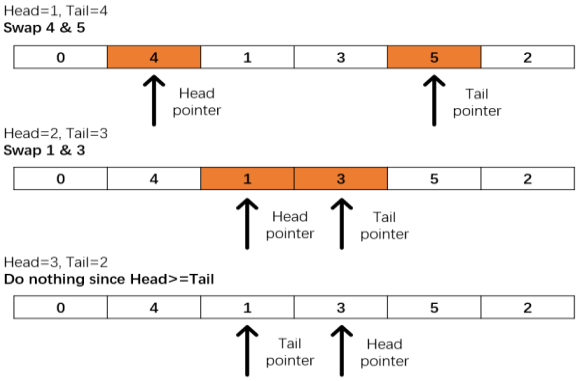
1. First generate an ascending ordered array which is the same as **s1=[0,1,2,3,4,5]**.
2. Initialize a pointer pointing to the last element of the array, and **generate a random number within the pointer index**.
3. Swap the element of the working pointer pointing to with the element in **random number index** (orange elements in the picture).
4. Go back to step ii until the pointer reaches the head of the array.
5. Finally, the array **s0 become [3,4,1,5,0,2]** which is shuffled

iv. Initialized solution

The visualized solution of s0 is as follow.



b) Inversion Mutation



Steps:

i. Randomly generate 2 number within size, pointer **head** to represent the smaller one and **tail** to represent bigger one.

ii. While **head < tail**, swap the element that head & tail pointing to.

iii. **head++** and **tail--.**

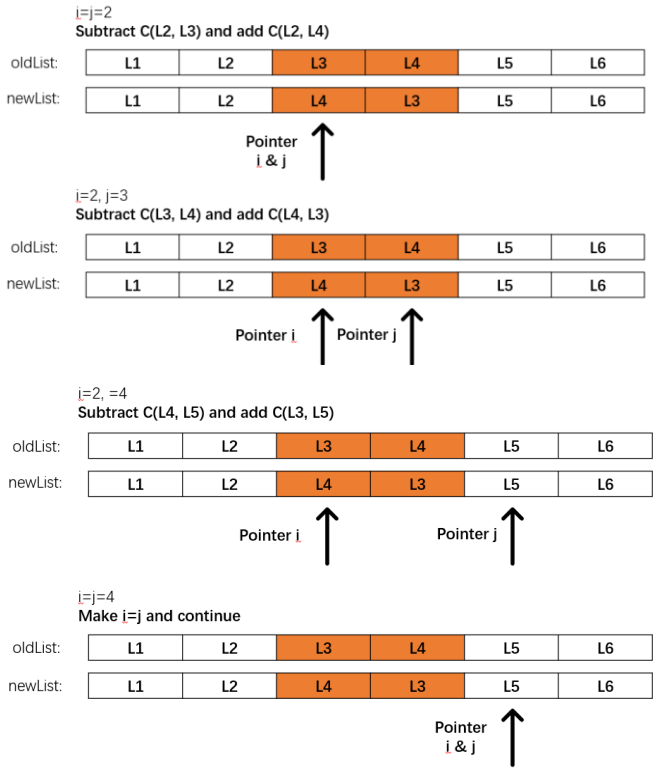
iv. If the intensity of mutation is not reached, go back to **step i.**

c) Delta Evaluation

The implementation of delta evaluation involves double pointer and three kinds of situations, which are showed below. The **traversal pointer i** goes through the whole array and finds out any unmatched elements in both array while the **working pointer j** which is responsible for getting the calculation value.

The delta Evaluation is implemented in ***HeuristicOperators class*** which can be called by all its sub-class (all mutation and local search algorithm)

i. Adjacent swap of locations L3, L4

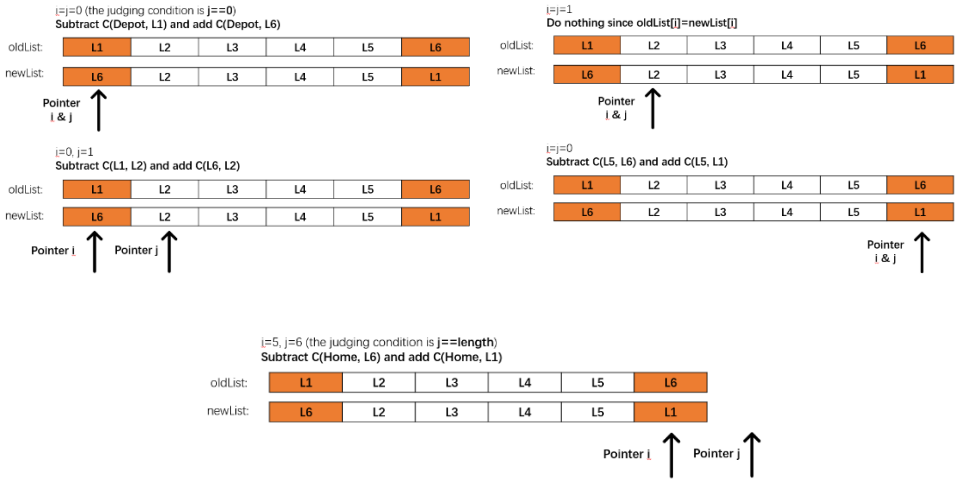


1. In normal scenario, when **i** finds a pair of different value, **j** will be initialized to the **same value as i** and updates the cost between where it points to and the former one by subtracting form old list and add the one in new list. When
2. When i doesn't find a difference in both list, **j will not be initialized, therefore will not update the value.**

ii. Adjacent swap of locations L1, L2

1. When the traversal pointer i find a difference in the first element, pointer j will be initialized to be **0 (j=i)**. The program will then detect j==0 and update the value with jth element and "postal office", which is the "last element" where j pointing to.
2. The remaining traversal is the same as scenario i.

iii. Adjacent swap of locations L6, L1



1. When i is at the front of the list, it remains the same as scenario ii.
2. When i is at the end of the list, j will be initialize to be 6. It automatically updates the value with former position and j++
3. When j==size in the list, which actually is the position of "home". The program will detect j==size and update the value with "home location".

d) Others

i. The *testScript* package

The *testScript* package includes tests for heuristic algorithms (6 low-level heuristic) and the problem domain, which aims to provide debug tools for these algorithms and doing some basis functional tests.

2. SR_ME_HH hyper heuristic

a) Introduction

The hyper heuristic model that I implement is the **memetic-based simple random heuristic.** The algorithm bases primarily on the memetic algorithm which applies **crossover, mutation and local search** to a problem domain orderly. The decision of specific heuristic use in each process is determined by random number.

The main algorithm is implemented in **SR_ME_HH** class in the ***hyperHeuristics***

package. **SR_ME_runner** class implemented in *runner* package which can provide single trail test for **SR_ME_HH** heuristic. **HH_compare** can run the algorithm in multiple trails and calculate the average cost of the found best solutions.

b) Implementation

i. Heuristic selection strategy

In each stage (crossover, mutation and local search), randomly choose one of the corresponding type of solutions available in the problem domain.

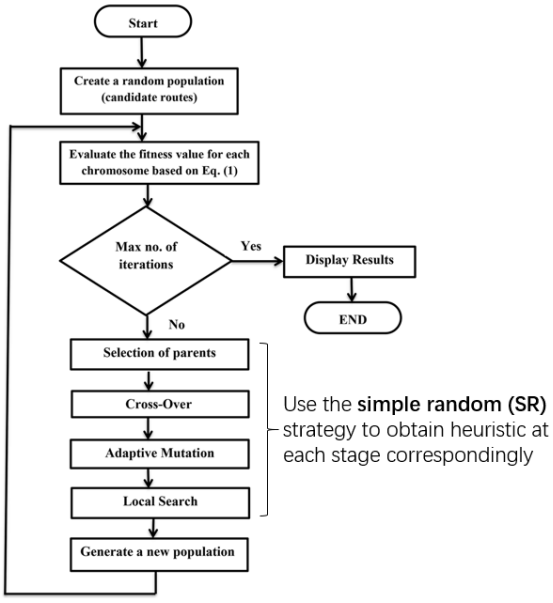
ii. Move acceptance

Naïve move acceptance strategy is applied. Accept only equal or progressive move at each round

iii. Low-level heuristics considered

Consider all low-level heuristic available in the problem domain. These algorithms shared equal opportunities to be chosen at each stage. For example, **OX** and **CX** are used for crossover. The SE_ME_HH will generate random number within algorithm size. In this case CX and OX both have 50% possibilities to be chosen. At the next stage, **re-insertion mutation**, **inversion mutation** and **adjacent swap** have equal possibilities to be chosen. It remains the same for **david's hill climbing** and **next decent** method.

iv. The rough structure is below:



c) Explanation & Reason

i. The hyper-heuristic algorithm makes use of the advantage of meta-heuristic which can "jump out"The mutation ensures that the heuristic will not be trapped in local optimal.

ii. The parents of each round are chosen randomly and the population size is set to be 5 which provides greater freedom for "allele" to flow in the gene pool and creates more possibilities of combination.

iii. The naïve acceptance strategy is applied since mutation can ensure the escape

of local optimal.

iv. I have tried to use the **Reinforcement Learning** method to train the model. However, the result is not satisfied enough since for crossover operation, **CX** and **OX** always result in penalization and local searches always produce rewards. So, the score does not significantly influence the result. This means at each process, every method shares equal opportunities to be chosen. However, the process of scoring and Roulette Wheel selecting methods will greatly occupy the time for progressing. Within a time limitation (1 min in this scenario), for small and middle size of instance (Carparks-40 and Tramstops-85), the use of simple random can greatly shortens the time for one round and performs more trails. This makes the result closer the optimal.

d) Settings & result

i. Average length:

1. Carparks-40=`0.3154983554666612`
2. Tramstops-85=`0.4782213600381234`
3. Trafficsignals-446=`1.9060134894453484`

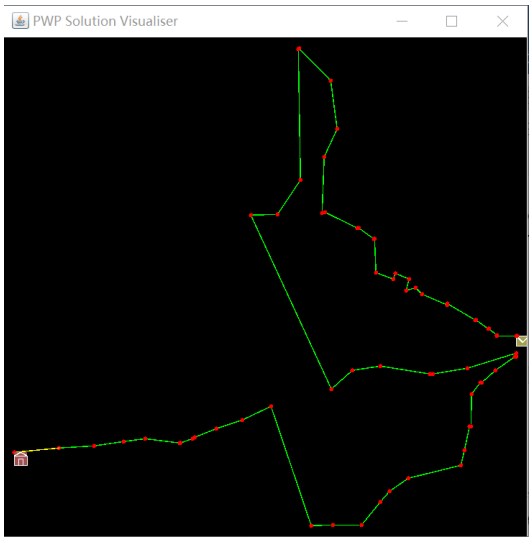
ii. Best solution length:

1. Carparks-40=031010422
2. Tramstops-85=0.459415374
3. Trafficsignals-446=1.906013489

iii. Time setting: 1s for 1 trail

iv. Trails: 11

v. The found best solution's graph is as follow (Tramstops-85):



3. Hyper-heuristic Comparison

a) Ranking comparison

The test framework is implemented in ***HH_compare*** class in the *runners* package.