Title of Report

Table of Contents

[Random Initialisation 2](#_Toc36997187)

[Inversion Mutation 2](#_Toc36997188)

[Delta Evaluation for Adjacent Swap 3](#_Toc36997189)

[My Hyper-Heuristic 4](#_Toc36997190)

# Random Initialisation

Firstly, PWPInstanceReader creates a PWPInstance based off of the square.pwp file; storing relevant information such as an array of locations(aoLocations).  
Each PWPSolution contains an int array used to store the indexes of these locations in the order they are to be visited.

When a PWPSolution is initialised, it is set to a random permutation by first creating an array of indexes [0-PWPInstance.iNumberOfLocations] and then shuffling their order using Fisher–Yates shuffle as it is a simple and efficient way to produce an unbiased permutation.

When using seed ‘13032020l’ s0 is initialised to [4,5,3,1,2,0].

Visual Representation of both solutions is seen below

|  |  |
| --- | --- |
| S0:[4,5,3,1,2,0] | S1:[0,1,2,3,4,5] |
|  |  |

# Inversion Mutation

A do-while loop is used to ensure I have picked 2 different random points followed by using the ternary operator to assign the lowest point as the startIndex and the highest point as the endIndex.

I then create a new subarray of the elements from those 2 indexes within the solutionRepresentation.

Next that subarray is reversed utilising a for loop going through half of the subarrays size swapping the element at that offset from the start and that offset from the end.

Finally, the initial solution is updated by simply setting the values in the range between startIndex and endIndex to the newly reversed subarray.

Example:

* solution=[0,4,1,3,5,2]
* i1=4, i2=1
* startIndex=1, endIndex=4
* subArray=[4,1,3,5]
* subArray=[5,1,3,4]
* subArray=[5,3,1,4]
* solution=[0,5,3,1,4,2]

# Delta Evaluation for Adjacent Swap

To use delta evaluation for adjacent swap simply get the cost of the solution then subtract the cost of the edges to be changed (before they are changed) then add the cost of those newly changed edges; done using the index of the locations that have been swapped as the indexes (i1 and i2) don’t change, the locations inside them do

If statements should be used to check which edges are being changed:

* if i1 is the last element: C(i2,Depot)+C(i2,i2+1)+C(i1,Home)+C(i1-1,i1)
* else
  + if i1 is intermediate C(i1-1,i1)
  + else if i1 is first node C(i1,Depot)
  + if i2 is intermediate C(i2,i2+1)
  + else if i2 is last node C(i2,Home)
* Where i1 = random index and i2 = i1+1%size.

Example swaps with solution S=[L1,L2,L3,L4,L5,L6]

* Swap(L1,L2)
  + f(si+1)=f(si);
  + f(si+1)-=C(L1,Depot)+C(L2,L3);
  + f(si+1)+=C(L2,Depot)+C(L1,L3);
* Swap(L3,L4)
  + f(si+1)=f(si);
  + f(si+1)-=C(L2,L3)+C(L4,L5);
  + f(si+1)+=C(L2,L4)+C(L3,L5);
* Swap(L6,L1)
  + f(si+1)=f(si);
  + f(si+1)-=C(L1,Depot)+(L1,L2)+(L6,Home)+(L6,L5);
  + f(si+1)+=C(L6,Depot)+(L6,L2)+(L1,Home)+(L1,L5);

# My Hyper-Heuristic

My hyper-heuristic uses Roulette Wheel Selection, Simulated Annealing, considers all the provided low-level heuristics and has a memory size of 4 holding the best, current, candidate and parent solutions.

I chose to pair RWS with SA as they synergise well with one another.  
At the start all heuristics will have an equal chance to be selected by RWS and SA will accept more non-improving moves, encouraging exploration.  
Whilst as time goes on RWS will have adjusted such that the heuristics which are more likely to result in an improvement are more likely to be picked and SA will accept more improving moves, encouraging exploitation.  
I also opted to consider all low-level heuristics for the same reason; exploring all heuristics at the start then automatically adjusting to be more likely to pick the heuristics that are resulting in improvements being made.

In the main loop, first Roulette Wheel Selection is used to determine which heuristic it is going to apply for this iteration; that heuristic is then applied to the current solution and stored in the candidate solution.

If the candidate solution is a strict improvement on the current solution, then it increments the roulette wheels score for that heuristic, so it is more likely to be picked again and updates the current solution. Then if the candidate is also an improvement on the best solution that is also updated.

Else we check if the candidate has a longer path than the current, if it does then the roulette wheels score for this heuristic is decremented; done so that a heuristic isn’t punished for resulting in an equal cost solution. Next the Boltzmann probability is applied to determine whether to update the current solution to the worse candidate solution; this encourages more exploration of the search domain helping to prevent getting stuck in a local optima early on (whilst the temperature is still ‘hot’). Finally the temperature is then advanced, setting it to the ratio of elapsedTime to timeLimit; meaning the longer the less time left the ‘cooler’ the temperature and thus the less likely the algorithm is to accept a non-improving solution; useful as the less time there is left the more exploitation we want to do as we want to find the best solution we can within the time limit.