Title of Report

Table of Contents

[Random Initialisation 2](#_Toc37167482)

[Inversion Mutation 2](#_Toc37167483)

[Delta Evaluation for Adjacent Swap 3](#_Toc37167484)

[My Hyper-Heuristic 4](#_Toc37167485)

[Development Process 4](#_Toc37167486)

[Final Version 4](#_Toc37167487)

# Random Initialisation

Firstly, PWPInstanceReader creates a PWPInstance based off 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 int array of length matching number of locations in the instance, representing the indexes of the different locations, 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].

The shortest route, s1 is known to be [0,1,2,3,4,5]

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 set two counters representing the indexes of the locations to swap, i initialised to the startIndex and j initialised to the endIndex.  
I then have a while loop repreat whilst i<j, swapping the locations at i and j then incrementing i and decrementing j.

Example:

* solution=[0,4,1,3,5,2]
* i1=4, i2=1
* startIndex=1, endIndex=4
* i=1, j=4 [ENTER LOOP]
* solution=[0,5,1,3,4,2]
* i=2, j=3 [CONTINUE LOOP]
* **solution=[0,5,3,1,4,2]**
* i=3, j=2 [BREAK LOOP]

# Delta Evaluation for Adjacent Swap

To use delta evaluation simply get the known cost of the solution before the application of the heuristic, then subtract the cost of the edges to be changed (before they are changed) then add the cost of those newly changed edges.

|  |  |
| --- | --- |
|  | Pictured to the left:  If you are swapping the first and the last element then you must first remove the cost of i1 to the element before it and i1 to the element after it (home), and the cost of i2 to the element after it and i2 to the element before it (depot). You then swap ‘i1’ and ‘i2’ into the same calculations and add those new edges back on.  If you are swapping any other 2 elements simply remove the cost of i1 and the element before it (checking if it’s the depot) and the cost of i2 and the element after it (checking if it’s the home). Then same as before you swap ‘i1’ and ‘i2’ you then swap ‘i1’ and ‘i2’ into the same calculations and add those new edges back on. |

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

## Development Process

I initially chose to pair Roulette Wheel Selection with Simulated Annealing believing the 2 would synergise well with one another due to them both encouraging exploration at the start and exploitation at the end; RWS adjusting to give better performing heuristics a greater chance at running and SA ‘cooling’ down over time, accepting less worsening moves.

I then realised that as, over a short period of time, RWS was unlikely to pick a worse solution it was actually working against SA so I switched it to a simpler, accept strictly improving moves only approach and saw the performance improve.

Due to the limitations of RWS I opted to create my own memory-based approach.  
The initial version would create a separate memory buffer for each heuristic, recording the scores from the last n times that heuristic was run then using those cumulative scores as the probability for picking each heuristic.   
This performed better than RWS but I realised it could be further improved as a heuristic could theoretically be used in an area of the search domain where it isn’t well suited and be given poor scores so it is unlikely to be picked again even if the hyper-heuristic has now moved to an area of the search domain well suited for that heuristic.  
In order to combat that I made a new version which has a single memory buffer storing pairs for the last n heuristics applied and their resulting percentage change. The probability of picking a heuristic is equal to the average percentage change it caused over the last n runs stored; and crucially if a heuristic hasn’t run in the last n runs its probability is set to 1 (meaning it is less likely to be picked than one which is known to cause an improvement but is more likely to be picked than one which is known not to). This means that all heuristics will be given a second chance if they resulted in a bad score.

## Final Version

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 be applied 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 less run 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.