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

As detailed above, my hyper-heuristic uses a memory based selection method which is an adaption of roulette wheel selection, instead of incrementing/decrementing the heuristics score if it improved/worsened; it uses the average percentage change caused by the hyper-heuristic over n moves , allowing it to make more accurate and informed choices based on the heuristics performance as well as not permanently punishing a heuristic if it performed poorly in the past.

My selection method works by maintaining a buffer of size n, storing the last n heuristic runs and their resulting percentage change. When selecting a heuristic, it uses the average score of each heuristic as the probability of picking that heuristic, and if a heuristic isn’t a buffer it is given the default probability of 1. The buffer is an ArrayList being used as a queue- when a score has been updated it is added to the ArrayList and if it’s size is now greater than n, it pops the element from the back (first in first out).  
This method allows my hyper-heuristic to adapt to the different search spaces it enters in the domain, being more likely to perform the heuristic that’s performing the best in the given search space but also giving the heuristics that haven’t performed within n moves (presumably due to them having a low score whilst they were on the buffer due to them being used in a search space that they weren’t well suited for) another chance to run as the problem may now have entered a better suited search space for that heuristic.

After experimentation with a few different acceptance methods I ended up deciding to use the Simple Acceptance of improving only as it resulted in a much better best solution at the end of each run on the larger instances.  
I believe this is because the time limit for the hyper-heuristic to run in was only 1 minute and thus the simple acceptance firstly ran quicker and so it allowed for more iterations of the hyper-heuristic, but secondly in that time it was unlikely to reach a local optima so exploitation was generally more useful than the exploration encouraged by approaches such as simulated annealing. If there was a larger problem with a longer time to run a more complex acceptance strategy may have performed better.

In the main loop, first my memory-based selection is run to pick the heuristic to run. That heuristic is then applied to the current solution and stored in the candidate solution next my memory-based selection is updated with the heuristics ID and percentage change from the currents cost to the candidates cost. After which I simply check if the candidate is a strict improvement on the current, and if it is, I accept it by copying the candidate to the current.