Report on AIM Project 2020
Implementation of a HyFlex Compatible Postal Worker Problem Domain

Question and Example Based Essay

Random Initialisation

The solutions to the PWP problem domain instances are represented by permutation representation. Here, I take "Square.pwp" instance file as an example to illustrate how the locations are mapped on to the solution. Reading from the file, all the postal delivery locations are stored in an array of 'Location[]' type with the same order. Postal office depot and home address, which are invariable, are stored in 'oPostalDepotLocation' and 'oHomeAddressLocation' respectively. This makes implementation of heuristics more convenient as we only need to focus on the change of paths between delivery locations.

In solution representation $s_1 = [0, 1, 2, 3, 4, 5]$, the elements match the locations stored in the array of locations. For instance, $s_1[0]$ represents the starting postal address (5,0) right after the postal office depot and $s_1[5]$ represents the last delivery location (0, 10) visited before home address. The route of s_1 is shown as *Figure 1*.

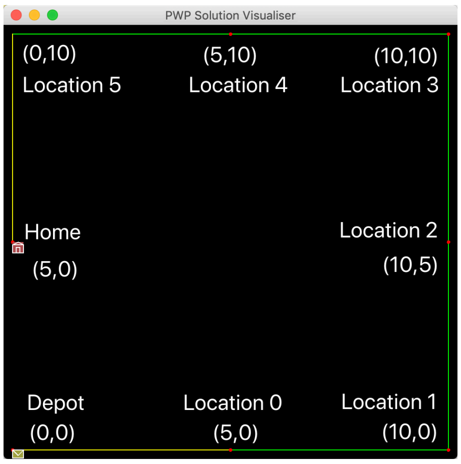

Figure 1

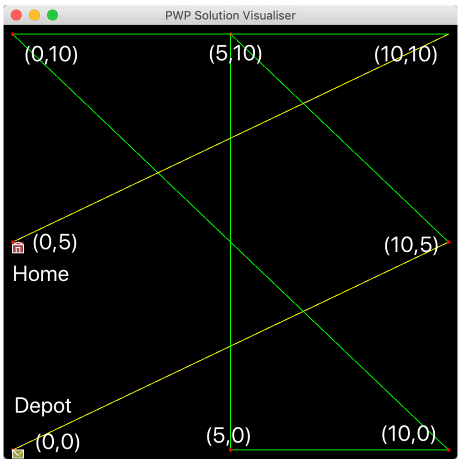

Figure 2

To generate the initial solution s_0, I create an ArrayList with elements from 0 to the number of postal addresses – 1, which is 5 in this case. And then I use 'Collections.*shuffle*' to shuffle the ArrayList with a given random seed and convert it into an array, which is the randomly initialised solution representation. *Figure 2* shows the solution representation $s_0 = [2, 4, 0, 1, 5, 3]$. As you can see, delivery location (10, 5) becomes the first police to be visited after leaving postal depot, while delivery location (10, 10) is changed to be the last.

Inversion Mutation

Inversion Mutation is applied to change the solution representation. Specifically, two locations l_a and l_b are selected randomly in the first place, where l_a appears before l_b based on the current route. Then, according to the selected locations, the route between them is preserved but reversed in an opposite direction and the rest routes remain the same. The whole procedure is repeated according to the intensity of mutation. My algorithm of this implementation is similar to quick sort, which is scanning through the sub-array from each side and swapping the locations each time until $index_{right} \leq index_{left}$. Taking $s = [0, 4, 1, 3, 5, 2]$ as an example, if the index of l_a is chosen to be 1 and the index of l_b is selected to be 5, the solution representation after inversion mutation will be $s' = [0, 2, 5, 3, 1, 4]$, where the sub-array $[4, 1, 3, 5, 2]$ is reversed. The steps are as follows:

a. The initial solution representation is

$$s = [0, 4, 1, 3, 5, 2]$$

b. Then I swap the elements of $index_{left} = 1$ and $index_{right} = 5$, and I obtain the solution

$$s = [0, 2, 1, 3, 5, 4]$$

c. Next, I swap the elements of $index_{left} = 2$ and $index_{right} = 4$. The solution representation is

$$s = [0, 2, 5, 3, 1, 4]$$

d. I find that $index_{left} == index_{right}$, so we stop here.

Updating the cost is also included in the method. For delta evaluation of Inversion Mutation, we only need to consider the edges $l_{a-1} \rightarrow l_a$ and $l_b \rightarrow l_{b+1}$, because the route between l_a and l_b is unchanged. The general expression to calculate the delta is

$$delta = cost(l_{a-1}, l_b) + cost(l_a, l_{b+1}) - cost(l_{a-1}, l_a) - cost(l_b, l_{b+1})$$

For special cases, if l_a is the starting delivery location or l_b is the final delivery location, we need to update the costs associated with home and postal office depot respectively.

Delta Evaluation for Adjacent Swap

When applying evaluation to obtain the cost, we need to consider the changed edges associated with the selected delivery locations. To update objective function value, we can simply add the delta to the original value, $f(s_{i+1}) = f(s_i) + delta$, which is much faster than traversing the whole array. Assuming a problem instance with 6 delivery locations $(L_1, L_2, L_3, L_4, L_5, L_6)$, we can think about the following cases:

a. Adjacent swap of locations L_1, L_2
Because l_1 is the starting delivery location in the original path, the edge between postal office depot and the first delivery location will be changed after adjacent swap. Also, edge $l_2 \rightarrow l_3$ will be replaced by $l_1 \rightarrow l_3$. Delta is calculated as:

$$delta = cost(Depot, l_2) + cost(l_1, l_3) - cost(Depot, l_1) - cost(l_2, l_3)$$

b. Adjacent swap of locations L_3, L_4
Since neither L_3 or L_4 are adjacent to home and postal office depot, we only consider the general way of delta evaluation in this case. If we swap those two locations, the edges associated with them will be altered. Edges $l_2 \rightarrow l_3$ and $l_4 \rightarrow l_5$ will be removed, while $l_2 \rightarrow l_4$ and $l_3 \rightarrow l_5$ will be added to the route. So, the delta in this case is expressed as:

$$delta = cost(l_2, l_4) + cost(l_3, l_5) - cost(l_2, l_3) - cost(l_4, l_5)$$

c. Adjacent swap of locations L_6, L_1
In this case, L_6 is connected to home and L_1 is linked to postal office depot in the original path. If they are swapped, edges $Depot \rightarrow l_1, l_6 \rightarrow Home, l_1 \rightarrow l_2, l_5 \rightarrow l_6$ will be changed to $Depot \rightarrow l_6$, $l_1 \rightarrow Home, l_6 \rightarrow l_2, l_5 \rightarrow l_1$.

$$\begin{gathered} delta = cost(Depot, l_6) + cost(l_1, Home) + cost(l_6, l_2) + cost(l_5, l_1) \\ -cost(Depot, l_1) - cost(l_6, Home) - cost(l_1, l_2) - cost(l_5, l_6) \end{gathered}$$

Tranmstops-85

My design of hyper-heuristic is a reinforcement learning based selection method with adaptive threshold move acceptance. Initially, I am inspired by the selection strategy and adaptive method in the Lab04 and Lab07 respectively. Since I would like to develop my own HH, I caught the core ideas from the implementation in the labs and designed it in a different fashion. To achieve higher performance, I have tested many parameters that need to manually set when designing. Finally, the overall performance suggests that my implementation does a great job in finding the shortest path. The result of my hyper-heuristic design is visualised in *Figure 3* and the pseudo code of my implementation is presented as well in *Figure 4.*

For heuristic selection strategy, I have applied reinforcement learning in my hyper-heuristic selection. Apart from that, I sperate the iteration process into two successive stages, where the selected mutation operator and local search operator are applied to the problem one after another. In my implementation, I create two arrays to maintain scores for mutation heuristics and local search heuristics respectively. After applying a heuristic with the highest score so far, the algorithm will check whether the result meets the acceptance criterion and then the corresponding score will be updated according to the performance of the selected heuristic. If there is an improvement, then increase the score, otherwise decrease the score of that heuristic (the higher the score is, more likely the heuristic will be selected).

The reason why I choose this strategy is that it can provide a rewarding mechanism to maximise the opportunity for agent to achieve the goal based on environments (Özcan et al., 2008). In other words, through predefined reward and penalty schemes for each lower heuristic, my program can learn to decide which heuristic to use to potentially find the shortest path. Another reason is that since mutation and local search may influence the quality of generated solution in different ways, applying them one after another can explicitly enforce diversification and intensification. If mutation and local search operators are evaluated in a same standard, which means only one heuristic is applied each time, it is likely that some operators may have less probability to be chosen. Therefore, two-stages processing approach can create a relatively fair selection environment for those operators.

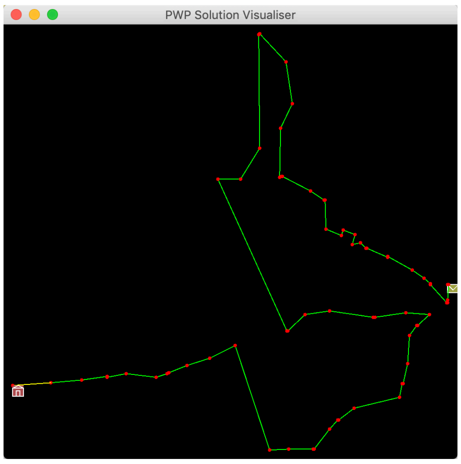

Figure 3 Best route forTramstops-85 found using my Hyper-Heuristic

For move acceptance, I have considered a stochastic adaptive threshold method, which is using Metropolis criterion with a cooling schedule. In detail, if solution is non-worsening or the random number is smaller than Boltzmann probability, then accept the solution and update the relevant information. After that, Lundy-Mees model as a cooling schedule with a suitable cooling factor 0.0001 is applied to update 'temperature' in each loop and prepare for the acceptance criterion for the next loop.

Inspired by simulated annealing algorithm, I have implemented it in such a way in that it can converges to a better optimum with relatively less computational time. The acceptance criterion can be adjusted according to running circumstance in each loop. With adaptive threshold, the program is able to captures the trade-off between the costs to accomplish an improving step and the degree of that improvement (Kheiri et al., 2016). Also, with the probability introduced, the solution is able to jump out of a local optimum by allowing worse solutions occasionally.

For choosing a set of low-level heuristics, I have considered to reuse the mutation and local search operators from SR_IE_HH except crossover operators. Since crossover will accept all the solution without guaranteeing improvement, which may influence the performance, I disregard them in my design. And another reason I give them up is that the process of crossover has a relatively high time complexity, since there is no delta evaluation involved to obtain the fitness and the program needs to take some time to traverse through two parent-solutions to produce a child-solution. For the rest of heuristic operators, they will be selected at each stage according to their historical performance.

In summary, my design of hyper-heuristic has a better performance for the instance TRAMSTOPS-85 compared to nearest-neighbour constructive heuristic. As a matter of result, given 1 minute of computational time, the mean average of the shortest route is $4.56339228430492 * 10^{-1}$ $(15\, s.f.)$, which is shorter than the shortest route can be found by the nearest-neighbour constructive heuristic method. The proposed use of heuristic selection strategy and move acceptance synergize well, producing a feasible and maintainable hyper heuristic for PWP domain search.

Pseudo Code 1 My Design of Hyper-Heuristic

Input: Problem domain

Output: Obejective Function Value

```
1:  function SOLVE(ProblemDomain problem)
2:
3:      problem initialisation;
4:      Create an array M_IDs to store indexes of Mutation Heuristic;
5:      Create an array LS_IDs to store indexes of LocalSearch Heuristics;
6:      Create an array M_Scores with length M_IDs.size;
7:      Create an array LS_Scores with length LS_IDs.size;
8:
9:      s0 = initial objective function value;
10:     s1 ← s0 ; s2 ← s0 ; temperature ← s0;
11:     repeat
12:         Get Mutation heuristic id with the highest score;
13:         s2 = apply( M_IDs[id]);
14:         if s2 < s0 then
15:             M_Scores[id] ++;
16:         else
17:             M_Scores[id] - -;
18:         end if
19:
20:         Get Local Search heuristic id with the highest score;
21:         s1 = apply( LS_IDs[id]);
22:         Create a random number r;
23:         if s1 <= s0 OR r < Boltzmann_probability(s0, s1, temperature) then
24:             if s1 < s0 then
25:                 LS_Scores[id] ++;
26:             else
27:                 LS_Scores[id] - -;
28:             end if
29:             s0 ← s1;
30:         end if
31:         Update temperature using Lundy and Mees model;
32:     until (time's up)
33:
34:     return s0;
35:
36: end function
```

Figure 4 Pseudo Code of My Hyper-Heuristic

Hyper-Heuristic Comparison

/* I wrote this part before the last Team's online discussion. You mentioned that this part wasn't needed in
*** the report. I believe the analysis of the HH is important, where I compare the results and investigate the**
*** test approach critically. I will be grateful if you can take some time to go through this part.**
***/**

Ranking Comparison

In this section, I am going to evaluate my own hyper-heuristic design by comparing it with the provided SR_IE_HH. To build a fair comparison environment, I have created a test framework to make the evaluation. Here are the relevant settings:

a. Testing instances
   - Carparks-40 (forming the "small" sized instance)
   - Tramstops-85 (forming the "medium" instance)
   - Trafficsignals-446 (forming the "large" sized instance)

b. Number of running trials per instance: 11

c. Time limit per trial: 1 minute

d. Testing method: pairwise ranking method

e. Running environment
   - Mac OS 10.14.6
   - Eclipse IDE with Java 10

Since my design of hyper-heuristic is single point based and does not apply crossover operators, so I have removed all the crossover operators in SR_IE_HH when testing. Here, a pairwise ranking method is introduced to rank each hyper-heuristic on a per trial basis. The initial solution is set to be the same for the same trial, but for the seeds in following trials are changed together based on a seeded pseudorandom number generator. In each trial, the best search method will obtain a rank of 0 and the worst will receive a rank of 1.

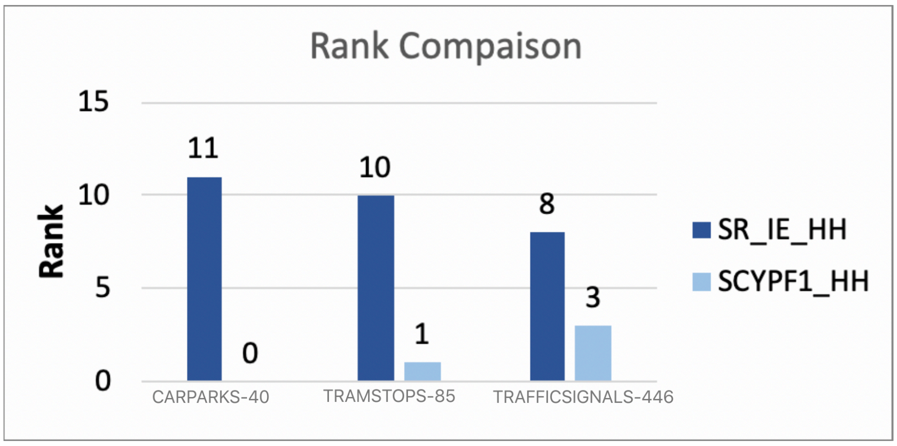

Figure 5 Rank Comparison for each instance

In *Figure 5*, the y-axis refers to sum of ranks and x-axis represents each instance. Lower numbers in the figure denote a higher placement in the ranking and indicate better performance. Generally, although there are some cases that SR_IE_HH performs better, my design has lower ranks in most trials according to experiments. Based on the mean score of each hyper-heuristic, we can see that my design of hyper-heuristic performs better than SR_IE_HH, whose average ranks are $1.21 * 10^{-1}$ and $8.79 * 10^{-1}$ $(3\ s.f.)$ respectively.

Statistical Comparison

Further analysis based on an appropriate statistical test have been carried out to evaluate the performances. Through some research, I have compared some statistical tests and found Wilcoxon signed-rank test is suitable for our experiment.

Since we try to compare hyper heuristic search methods by applying each to the same population (instance) with the same random seed for each trial, the experimental data is dependent and should be paired. So, the number of treatments for each test is two. Also, the distribution of data is skewed or unknown (not normally distributed) and our dataset size is small, so non-parametric test is more preferred (Yates et al., 2019). For all the reasons above, we can see Wilcoxon signed-rank test appears to be an optimal choice for comparing hyper-heuristics.

To obtain high-precision evaluation, all the experiments are conducted through MATLAB and are repeated for 11 running trials for each instance. There is a built-in function "$signrank()$" to help me do the statistical test.

$$[p, h] = signrank(x, y, 'tail', 'both');$$

Where

x, y are the arrays of results obtained from two hyper-heuristics;
'$tail'$ and $'both'$ means we set the test to be two-tailed;
p refers to p-value for the signed test;
h is a logical value referring to the test result.

"$signrank()$" can test the null hypothesis that data in the x and y comes from a distribution whose median is zero at the 5% significance level. And for the return value, h = 1 implies a rejection of the null hypothesis, and h = 0 indicates an acceptance of the null hypothesis at the 5% significance level.

The tests suggest that the comparison is statistically significant within a confidence interval 95% in small and medium instances. For the big size instance, the data is not significant different at $P < 0.05$. However, the testing configuration has a limitation that the small dataset (11 trials) is a little bit small and may not be able to reflect the whole. If I increase the number of trials. the testing result may be different. Also, based on the principle behind the testing approach, if a pair has the same score in both treatments, then the test will ignore that data and decreases the sample size. Therefore, testing on more data will make the statistical result reliable.

Overall, based on Wilcoxon signed-rank test, my design of hyper heuristic has a better performance on small and medium size instances and indicates a significant difference compared to SR_IE_HH. As for the large size instance, although my HH generally performs better than SR_IE_HH, the calculated P value shows that there is no significant difference at $p < 0.05$. However, the limitation of the testing configuration suggests that further investigation still needs to be done to obtain a solid evidence to show the statistical significance.

Reference

Kheiri A, Özcan E. An iterated multi-stage selection hyper-heuristic. European Journal of Operational Research. 2016 Apr 1;250(1):77-90. https://doi.org/10.1016/j.ejor.

Özcan, Ender, Mustafa Misir, Gabriela Ochoa and Edmund K. Burke. "A Reinforcement Learning - Great-Deluge Hyper-Heuristic for Examination Timetabling." *IJAMC* 1.1 (2010). https://doi:10.4018/jamc.2010102603

Yates, W.B., Keedwell, E.C. An analysis of heuristic subsequences for offline hyper-heuristic learning. *J Heuristics* 25, 399–430 (2019). https://doi.org/10.1007/s10732-018-09404-7