Final Exam – INF379 Metaheuristics

Solution for a Pickup and Delivery Problem with Time Windows (PDPTW) using Simulated Annealing

Table of Contents

Submission Details

My submission to the Exam consist of the following files:

Exam_pjo047.pdf - This document, containing a description of the submission, results as well as the reasoning behind the selected components and operators.

Exam_pjo047.m - The Matlab file containing the code with comments explaining the choices/algorithms chosen.

Results.xlsx - The result tables containing two sheets one with results and another with the best solutions from each run. A copy of the results are summarized also in this document.

Running the matlab code

The Matlab code Exam_pjo047.m can be run directly from the console using for example the following argument on my computer:

“ matlab -r 'seed=1;runs=1;iter=10000;run /home/preben/repo/inf379/FinalExam/Exam_pjo047' “

The *seed* is given to indicate the starting seed for the following runs. The *runs* indicate the amount of runs wanted for each instance. The *iter* indicates the amount of iterations wanted. More info about this further below. The path will have to be adapted to where the exam file is located. The instance txt files should be placed in the same folder with exactly the same names as the instances used in the Matlab file:

"Call_007_Vehicle_03.txt","Call_018_Vehicle_05.txt","Call_035_Vehicle_07.txt","Call_080_Vehicle_20.txt","Call_130_Vehicle_40.txt"

Main Components

The main Components of my chosen solution method are Simulated Annealing (SA) combined with a large neighborhood search operator. SA is a local search method that provide a good way to search neighborhoods and in combination with a large neighborhood reinsert operator this proves below to be quite an effective way of solving the given instances of PDPTW.

Simulated Annealing (SA)

I have chosen to use a Simulated annealing metaheuristics to solve this problem. The basic components of SA are having a cooling schedule that determine the probability of accepting a proposed solution s* over the current solution s. As s* always is being accepted if it is better than current s, the probability function is determining whether or not s* should be kept as the current solution if it brings the system to a worse state. A cooling schedule needs a starting temperature T0 and a finishing temperature Tf. It also needs a probability function which can be either linear, exponential or logarithmic (or other possibilities). The stopping criteria is based on how big the cooling schedule is (N) and the time spent in each temperature state *max_it* which is determined before starting up the algorithm. The complete algorithm can be summarized as follows (all pictures below: Hemmati 2018).

Initialization
    Parameters Setting (T_0 , T_f, Max_Ite, Cooling_Schedule)
    Generate an *Initial_solution* (randomly)
    Current_solution =Initial_solution; Best_solution =Initial_solution; T= T_0
While ($T > T_f$) do
    For *i* = 1 to *Max_Ite* do
        Generate *New_solution* = Neighbor(*Current_solution*)
        Evaluate the change in objective function level ΔE
        If $\Delta E < 0$ then
            Current_solution = New_solution
            Best_solution = New_solution
        Else
            Accept *Current_solution = New_solution* with probability $p = e^{\frac{-\Delta E}{T}}$
        End If
    End For
    Decrease parameter *T* according to *Cooling_Schedule*
End While

Cooling Schedule

I used an exponential cooling schedule in my solution. This makes the solution go from a like random walk in the beginning of the search and then moving over to a local search. This by far outdid a linear search and made the algorithm very robust. The T is determined with the following formula:

$$T_i = \frac{A}{i+1} + B$$

$$A = \frac{(T_0 - T_f)(N+1)}{N} ; B = T_0 - A$$

In my cooling schedule I have determined the *max_it* variable best when kept small. This might have something to do with my operators as they work best over a longer cooling schedule where I slowly lower the temperature and with it the threshold of accepting a solution. I therefore kept my max_it to 4 and the Cooling Schedule therefore has *iter*/4 different temperatures.

Determining T0 and Tf

To determine the T0 and Tf I used the following formula from the lectures:

$$T = \frac{-\Delta E}{\ln p} \qquad p_{max} = 0.99 \qquad p_{min} = 0.01$$

in $\Delta E \in [1, \max \Delta E]$

For T0 I used the maximum possible improvement which I determined using the following formula:

max **Δ**E = max(call_cost) – 2*mean(route_cost) - mean(unload_cost) – mean(load_cost)

where call_cost is the cost of not transporting a call, route_cost is the cost of getting to a certain node (to transport a call I need to go to two nodes), unload_cost is the cost of unloading a call and load_cost is the cost of loading a call. I figured the max improvement possible is very close to transporting the most expensive call minus the costs of doing the transport (ie. the average values). Since the cost of not transporting is significantly higher than the actual transportation cost this formula could also be simplified to the max(call_cost) however I choose to keep generic on the matter in case of a weird instance.

T0 is then simply *-max**Δ**E/ln(0.99)* which I calculate during each instance in my matlab code.

The Tf is set to the minimum improvement possible which in our case I have set to 1 as there definitely exist an improvement of 1 (see results of instance 1 which ends up in two different local optimum with 1 difference). The Tf is then set to *-1/ln(0.01)* in all cases.

Operators

The operators I have chosen in my method I divided up in two categories, feasible operators and non-feasible operators.

Feasible Operators

The feasible operators differ from the non feasible operators to the fact that they always return a feasible solution and therefore have no need of the feasibility check functions. I found the combination of fast operators such as the non feasible operators combined with the slower more reliable feasible operators to be a good and effective combination when solving a PDPTW problem.

Best-first-Insertion (35%) - Small

This operator always tries to insert a call from the unassigned calls, unless there are no unassigned calls, in which it removes a random call from the solution and reinserts it again in a random vehicle. To do this I have designed two *insertion functions*. The first insertion function tries to minimize the time to pickup the next call. Since time and costs are highly correlated this proves to be a very effective way of

inserting calls in a *best-fit* way. However it does not always succeed. Therefore I build another insertion function that uses a more *first fit(any-fit)* approach. This always tries to deliver the call that has the lower upper bound not caring if that call is far away or not to be sure the call gets delivered on time. This leads to more, but less effective, solutions. The best-first-insertion operator therefore tries to insert the selected call first after the best-fit principle, then if that doesn't work, in the first/any-fit way. The operator makes sure to test all vehicles and returns a null solution if the call could not be inserted. In which case the initial solution is returned to the program to move on to the next iteration.

Reinsertion (5%) - Large
This operator reinserts a certain amount of calls (dependent on the total amount of calls in the instance) to effectively move the current solution over a greater distance than the other operators (ie. a large neighborhood). This operator successfully moves a solution further away from a local optimum to greatly increase the chance of finding a better global optimum. The operator removes x amount of calls from a given solution and then reinserts them using the insertion functions mentioned above (tries best,then first fit). I found the amount of calls being moved best to be C/5 calls, where C is the total amount of call in the instance. If the call cannot be placed in any vehicle it is left in the dummy vehicle. This operator also always returns a feasible solution.

Non-Feasible Operators

The non feasible operators are much faster than the feasible ones but then needs to be checked for feasibility before being used. These operators makes sure I am not constraining myself too much to principle of the insertion functions, and therefore the feasibility constraints, and can find some deeper hidden solutions that the feasible operators cant find.

3-Exchange (30%) - Small
This operator is a swap type operator that swaps around within one vehicle (with at least two calls) the pickups and deliveries of 3 different calls. This operator has a small neighborhood is very fast but is not guaranteed to return a feasible solution.

Switcheroo (30%) - Small
This operator takes two random calls and switches both the pickup and delivery with each other. It is a very fast very effective operator however it does not always return a feasible solution. Like the Exchange operator it can find solutions that the insertion cannot and it has a small neighborhood.

Operator picker
I call on the operators using an operator picker function called operator. The operators are chosen with certain probabilities, the non feasible operators has a 30% chance of being picked each and the best-first insertion 35% chance, while the large reinsertion only has a 5% chance of being used. This is to avoid always changing around the large neighborhoods too often but also to avoid searching around a smaller neighborhood too long. Since the insertion operator is a bit better at finding feasible solutions I have chosen to run that a tad more often than the other two.

Results

I present the results from 10 runs of 20 000 iterations running on one core of an Intel i7 7500-U processor in less than 6 minutes. (Increasing the iterations gives marginally small improvements from 10 0000 to 20 000, therefore I decided to keep it at 20 000).

Table 1: Results from running 10 runs of each instance over 20 000 iterations

Inst	Run_1	Run_2	Run_3	Run_4	Run_5	Run_6	Run_7	Run_8	Run_9	Run_10
Call_007_Vehicle_	1256140	1256139	1256139	1256139	1256139	1256139	1256139	1256139	1256139	1256140
Call_018_Vehicle_	2799849	2495703	2400016	2400016	2631941	2495703	2654552	2637174	2641890	2495703
Call_035_Vehicle_	5194716	4954417	4875401	5196555	5158184	4852482	4888974	5208054	5285676	4798155
Call_080_Vehicle_	14220451	13711545	13497695	13849963	13719979	13551387	13653949	13284423	13321911	13518024
Call_130_Vehicle_	18728128	18361351	18656480	18719699	18647660	21476262	21522163	18294953	18964352	18416442

Table 2: Average and best improvements as well as initial objective and average time per run

Inst	Ini_obj	Avrg_Obj	Avrg_Impr	Best_Obj	Best_Impr	Avrg_time
Call_007_Vehicle_	3760286	1256139.2	66.5946	1256139	66.5946	0.78661
Call_018_Vehicle_	8761492	2565254.7	70.7213	2400016	72.6072	1.7745
Call_035_Vehicle_	18322178	5041261.4	72.4855	4798155	73.8123	3.2108
Call_080_Vehicle_	42211425	13632932.7	67.7032	13284423	68.5288	9.2668
Call_130_Vehicle_	75446687	19178749	74.5797	18294953	75.7512	16.0298

The best solution from each instance can be found below:

Call_07_Vehicle_03:
[4,4,6,1,6,1,0,5,5,2,2,0,3,7,7,3,0]

Call_18_Vehicle_05:
[4,14,7,14,4,3,7,3,0,15,15,6,17,17,6,0,11,16,16,11,10,9,10,9,0,12,12,1,8,1,8,2,2,0,18,5,5,18,0,13,13]

Call_35_Vehicle_07:
[18,24,18,24,0,5,13,5,13,30,30,0,23,23,3,3,11,11,20,1,20,26,1,26,0,25,25,16,16,10,14,10,14,15,15,4,4,0
,19,7,34,12,7,34,19,22,12,22,33,27,27,33,0,6,17,35,29,6,17,35,29,2,32,32,9,9,2,0,28,28,21,21,8,8,31,31
,0]

Call_80_Vehicle_20:
[42,40,40,42,0,59,72,59,72,51,51,65,18,65,18,0,20,20,68,68,28,28,13,35,35,13,0,67,53,67,53,17,17,43,
43,0,48,48,54,54,23,23,9,9,73,73,0,80,80,12,21,12,21,0,74,5,5,74,2,2,58,58,0,26,26,29,24,31,29,24,31,
75,75,0,57,55,55,36,36,57,1,1,0,47,47,39,39,64,64,0,52,52,30,30,0,50,50,61,7,61,7,78,78,0,46,46,56,56
,0,69,69,37,16,16,71,71,10,37,10,0,34,79,34,79,70,41,70,25,41,25,0,33,33,63,63,19,38,38,19,0,3,3,11,1
1,6,76,6,76,0,77,77,27,27,44,44,0,4,14,14,4,15,15,0,66,62,66,62,60,22,60,22,45,8,8,45,0,32,32,49,49]

Call_130_Vehicle_40:

[91,91,105,105,106,106,0,7,71,7,71,13,24,13,24,0,57,83,102,83,102,57,62,62,53,53,10,10,70,70,0,50,50,61,38,61,38,0,107,107,39,39,55,55,42,40,40,8,42,8,113,113,0,48,48,68,68,100,100,0,120,120,72,112,72,112,9,9,23,23,0,89,89,127,5,127,5,0,41,124,41,124,0,58,58,0,104,104,60,59,59,60,27,27,19,36,36,19,0,34,34,98,75,98,75,0,0,64,64,4,4,0,108,96,96,108,119,119,0,73,73,0,103,103,28,28,94,94,128,130,128,130,0,82,82,117,117,79,79,0,65,65,80,80,0,17,17,30,35,30,35,29,129,129,29,0,52,126,126,52,87,87,45,45,0,90,111,90,54,111,54,125,125,86,86,0,43,2,2,43,0,33,33,63,63,110,31,110,31,0,123,123,0,74,114,74,51,51,114,0,26,97,97,92,26,20,20,92,3,3,0,37,16,16,37,6,109,1,6,109,1,0,44,14,14,44,0,116,22,22,116,0,25,25,67,78,12,12,67,78,0,21,21,101,101,66,11,81,11,81,66,0,115,115,15,15,99,99,0,49,49,0,47,47,46,46,0,118,122,118,93,122,93,56,56,95,95,88,88,0,77,77,0,121,18,121,18,0,84,76,76,84,0,32,32,69,69,85,85,0]

Reference list

- A. Hemmati, Lecture notes from INF379 Metaheuristics, Spring 2018

- V. R. Bons, The pickup and delivery problem with time-dependent travel times, 2014

- M. I. Hosny, Investigating Heuristic and Meta-Heuristic Algorithms for Solving Pickup and Delivery Problems, 2010