INF273 – Final Submission

Lukas Schramm 4th May 2022

1 Functions

Currently, I am maintaining the following helper functions in my Utils file.[1]

- load_problem: Given a path to a file, it reads the content of the file into a dictionary of information.
- feasibility_check: It takes a solution (list) and a problem dictionary and checks if the solution is feasible. If it is not feasible, it outputs the reason why. It does not check validity. I hope it works correctly
- cost_function: It takes a solution (list) and a problem dictionary and calculates the cost of the function. As feasibility_check it does not check if the original solution was valid.
- splitting_a_list_at_zeros: Helper function which splits a solution into vehicles and if needed a dummy vehicle.
- initial_solution: Generates an initial default solution to start with. This is always a solution where the dummy vehicle handles all calls.
- random_solution: Generates a random solution. The generator itself is quite bad in my view because I overtuned it a bit. It automatically gives one vehicle exactly one call and the rest goes to the dummy vehicle. That way I got solutions for file 3 and 4 but the solutions for all files are quite bad.[2]
- blind_random_search: Takes a problem and a number of iterations to find the best out of n random feasible solutions if any is found.
- blind_search_latex_generator: This function runs the blind_random_search and writes the data into LaTeXtables since I am obviously too lazy to do it myself.
- latex_add_line: Adds a new result line into an results table of this file.
- latex_replace_line: Change the optimal solution and its seed in that file.
- feasibility_helper: A function which works similar to the feasibility helper but calculates the feasibility only for one vehicle.
- cost_helper: A function which calculates the cost only for one vehicle.

[1] The green ones are changes or additions from the last assignment

[2] But since we do not need that random solution generator any longer I keep it like that.

- cost_helper_transport_only: A function which calculates the cost only for one vehicle, but also only the transporting cost. This is perfect for adding a call at different positions and calculating the cost.
- greedy_insert_one_call_one_vehicle: A helper function which greedily finds the best insertion position for a call for one vehicle.
- helper_regret-k_insert_one_call_vehicle: A helper function which calculates the regret-k values for all valid insertion positions of a call into a vehicle.

  The following six functions three removal and insertion heuristics which then are combined to my full neighbouring functions.

- remove_random_call: Removes n random calls from it's current position.
- remove_highest_cost_call: Removes the n calls with the highest cost from its current position. This function has a random choice function based on the exponential probability distribution which chooses calls randomly, but weighted by how costly they are.
- remove_dummy_call: Removes n calls from the dummy vehicle.
- insert_regret-k: This function performs regret-k and inserts n calls into it. This function does not yet have a random functionality, which probability will be added.
- insert_greedy: Performs greedy insert for n calls. This function does not yet have a random functionality, which probability will be added.
- insert_back_to_dummy: It takes n calls and puts them back into the dummy vehicle.
- solution_to_hashable_tuple_2d and solution_to_hashable_tuple_1d: Since lists (and 2D lists) are not hashable, there now is a function converting 2D lists into 2D tuples such that they can be hashed and used in sets and dictionaries. This makes it possible to look them up fastly, like if solution have been seen before or what their previously calculated cost is.

Neighbour functions: Since I implemented three removal and two insertion algorithms, I have the possibility to combine them to six different neighbours. In the previous assignment, I choose three of them and named them Steven, Jackie and Sebastian. For assignment 5, I have implemented the other three, being named Steinar, Stian and Karina.

My neighbouring functions are the following six, combining pairs of the above removal and insertion functions.

- Steven: Removing n random calls and inserting those greedily.
- Jackie: Removing n highest cost calls and inserting them with regret-k.
- Sebastian: Removing n dummy calls and inserting them greedily.
- Steinar: Removing n random calls and inserting them with regret-k.

- Stian: Removing n highest cost calls and inserting those greedily.
- Karina: Removing n dummy calls and inserting them with regret-k.

If there are any questions or nice recommendations to get a better structure, just send me a message.

Remark on the running time: I have tried to get down the running time of my algorithm for a while, but struggle a bit with it. However, I have recently implemented a functionality which remembers information about solutions already seen. Both feasibility and cost of these solutions are now stored in a global dictionary, such that nothing gets calculated twice. This way, I managed to get the running time by 50-70% for most runs. This does not mean that the algorithm now became fast, but now I am happy with it. I will however try to improve it further. If the currently reported time is too big for the final exam, please inform me in time about it.

Moreover, there is a file for Heuristics where I collect all of the important algorithms and their helper functions.

- alter_solution_steven: Neighbour function, see above.
- alter_solution_jackie: Neighbour function, see above.
- alter_solution_sebastian: Neighbour function, see above.
- alter_solution_steinar: Neighbour function, see above.
- alter_solution_stian: Neighbour function, see above.
- alter_solution_karina: Neighbour function, see above.
- local_search: This function takes a problem, an initial solution, a number of iterations (10.000) and the allowed neighbouring function and performs a local search
- simulated_annealing: This function takes a problem, an initial solution, a number of iterations (10.000) and the allowed neighbouring function and performs a simulated annealing
- local_search_sim_annealing_latex: This function takes as input the allowed neighbouring function(s), the heuristics method, the problem and the number of iterations and performs the heuristics on randomly chosen seeds. It then calculates the average time and objective and runs the LaTeXfunctions to change the tables of this PDF
- adaptive_algorithm: This function is the adaptive algorithm being shown in the lecture slides 12-20. It performs 10.000 iterations and includes an escape algorithm which gets executed if the algorithm does not find a better solution for more than 100 iterations. Every 200 iterations, the current weights of all neighbours get updated. This is based on the formula shown on the slides. The scoring function is based on three values. A neighbour function scores 4 points if it finds a new global best, 2 points if it finds a solution better than the current one and 1 point if it finds a new solution never seen before.
- escape_algorithm: The escape algorithm takes some handchosen of the six neighbour functions and runs on the current solution of the adaptive algorithm. It runs up to 20 times or until it finds a new global best and accepts every feasible solution in those iterations.

2 Result tables

Table 1: Call_7_Vehicle_3

Method	Average objective	Best objective	Improvement (%)	Running time
Random search	2289893.35	2120884	34.59%	0.62s
Local Search-1-insert	1416012.10	1134176	65.02%	0.96s
Local Search-2-exchange	1243141.90	1134176	65.02%	0.99s
Local Search-3-exchange	1238358.70	1134176	65.02%	0.88s
Simulated Annealing-1-insert	1301446.80	1134176	65.02%	1.16s
Simulated Annealing-2-exchange	1314847.60	1134176	65.02%	0.79s
Simulated Annealing-3-exchange	1240012.20	1134176	65.02%	0.74s
SA-new operators (equal weights)	1134176.00	1134176	65.02%	15.54s
SA-new operators (tuned weights)	1134176.00	1134176	65.02%	11.03s
Adaptive Algorithm	1134176.00	1134176	65.02%	9.02s
Final Report	1134176.00	1134176	65.02%	7.71s

Listing 1: Optimal solution call_7_vehicle_3

```
1 sol = [4, 4, 7, 7, 0, 2, 2, 0, 1, 5, 5, 3, 3, 1, 0, 6, 6]
2 seeds = [316961720, 318580301, 764304586, 133611434, 873596552,
    603170979, 411840014, 559930731, 242886443, 347244978]
```

Table 2: Call_18_Vehicle_5

Method	Average objective	Best objective	Improvement (%)	Running time
Random search	7195792.08	6215552	29.80%	0.80s
Local Search-1-insert	2985468.70	2535568	71.70%	1.33s
Local Search-2-exchange	3114722.90	2512517	71.96%	1.17s
Local Search-3-exchange	3380136.20	2878735	67.87%	1.11s
Simulated Annealing-1-insert	3064756.80	2668336	70.22%	1.63s
Simulated Annealing-2-exchange	2911592.60	2525599	71.81%	1.34s
Simulated Annealing-3-exchange	3124333.50	2835545	68.35%	1.08s
SA-new operators (equal weights)	2613101.40	2389436	71.78%	38.40s
SA-new operators (tuned weights)	2388456.00	2374420	73.50%	38.89s
Adaptive Algorithm	2374420.00	2374420	73.50%	24.38s
Final Report	2374420.00	2374420	73.50%	30.81s

Listing 2: Optimal solution call_18_vehicle_5

```
1 sol = [4, 4, 15, 15, 11, 11, 16, 16, 0, 6, 6, 5, 18, 5, 14, 17, 17, 14,
    18, 0, 9, 8, 8, 9, 13, 13, 0, 7, 7, 3, 3, 10, 1, 10, 1, 0, 12, 12, 0,
     2, 2]
2 seeds = [492867642, 844650198, 729451312, 877452767, 754712203,
    965542029, 838887199, 906552834, 278741590, 388604696]
```

Table 3: Call_35_Vehicle_7

Method	Average objective	Best objective	Improvement (%)	Running time
Random search	15924073.22	14436028	20.19%	1.09s
Local Search-1-insert	7495649.90	7005670	61.90%	1.18s
Local Search-2-exchange	8173130.50	6859563	62.70%	1.18s
Local Search-3-exchange	7846618.10	6819851	62.91%	1.23s
Simulated Annealing-1-insert	7649784.40	7089562	61.44%	1.37s
Simulated Annealing-2-exchange	7609380.90	6652294	63.82%	1.23s
Simulated Annealing-3-exchange	7956510.00	6870483	62.64%	1.37s
SA-new operators (equal weights)	5603329.30	5239643	70.28%	127.86s
SA-new operators (tuned weights)	5684032.33	5592499	68.68%	116.81s
Adaptive Algorithm	5398705.70	5040942	72.59%	94.40s
Final Report	5118523.10	4980722	72.91%	94.02s

Listing 3: Optimal solution call_35_vehicle_7

```
1 sol = [4, 4, 16, 16, 3, 3, 21, 21, 0, 12, 7, 7, 10, 10, 12, 28, 28, 29,
    29, 0, 19, 19, 24, 24, 33, 33, 5, 5, 2, 2, 20, 20, 0, 15, 34, 15, 30,
     34, 30, 27, 27, 22, 22, 25, 25, 31, 31, 0, 9, 14, 17, 35, 17, 9, 14,
     13, 35, 13, 26, 32, 32, 26, 0, 23, 23, 8, 11, 11, 18, 8, 18, 0, 1,
    1, 0, 6, 6]
2 seeds = [220444141, 288716607, 377897665, 393365757, 444432674,
    454079388, 259110931, 558930740, 859492581, 554724584]
```

Table 4: Call_80_Vehicle_20

Method	Average objective	Best objective	Improvement (%)	Running time
Random search	39584864.24	37697832	18.28%	2.44s
Local Search-1-insert	17631174.10	16254766	65.25%	3.57s
Local Search-2-exchange	18107201.60	17381583	62.84%	3.00s
Local Search-3-exchange	18725020.20	16763949	64.16%	3.01s
Simulated Annealing-1-insert	17288988.70	15640209	66.56%	3.69s
Simulated Annealing-2-exchange	17611125.00	16291560	65.17%	2.82s
Simulated Annealing-3-exchange	18606671.00	16496296	64.73%	2.76s
SA-new operators (equal weights)	12188485.00	11207964	75.13%	221.34s
SA-new operators (tuned weights)	12509274.67	12249170	73.14%	216.27s
Adaptive Algorithm	11504103.60	11074639	76.32%	198.98s
Final Report	11367761.10	10844117	76.81%	254.77s

Listing 4: Optimal solution call_80_vehicle_20

```
1 sol = [41, 70, 41, 51, 70, 45, 51, 45, 65, 12, 65, 12, 36, 50, 14, 36,
    14, 50, 0, 54, 63, 54, 63, 49, 49, 72, 73, 73, 10, 72, 10, 0, 18, 18,
     35, 46, 35, 46, 16, 16, 0, 11, 11, 61, 42, 61, 3, 3, 42, 0, 62, 62,
    74, 67, 74, 67, 28, 28, 33, 33, 0, 22, 68, 22, 34, 34, 68, 20, 20, 0,
     15, 15, 64, 64, 5, 17, 5, 58, 17, 58, 0, 39, 39, 55, 55, 80, 80, 7,
    7, 0, 8, 8, 69, 48, 48, 69, 0, 25, 25, 4, 4, 26, 37, 37, 26, 0, 29,
    29, 77, 77, 0, 23, 59, 23, 79, 59, 27, 27, 79, 0, 38, 38, 78, 78, 2,
    2, 24, 24, 0, 71, 9, 71, 9, 19, 19, 52,
2 56, 56, 52, 0, 66, 1, 66, 1, 0, 32, 21, 32, 21, 47, 47, 6, 6, 0, 53, 30,
     53, 30, 76, 44, 44, 76, 0, 57, 57, 75, 75, 0, 43, 43, 0, 60, 40, 60,
     40, 13, 13, 31, 31, 0]
3 seeds = [785650946, 980380899, 540682256, 307432919, 589791489,
    285675507, 688529980, 971235212, 451996879, 252521591]
```

Table 5: Call_130_Vehicle_40

Method	Average objective	Best objective	Improvement (%)	Running time
Random search	76627567.00	76627567	0.00%	4.52s
Local Search-1-insert	28309714.70	25950511	66.13%	6.63s
Local Search-2-exchange	28157453.50	26420742	65.52%	4.89s
Local Search-3-exchange	29951029.40	28441536	62.88%	5.52s
Simulated Annealing-1-insert	27700390.50	26539400	65.37%	5.41s
Simulated Annealing-2-exchange	28875623.30	27391443	64.25%	4.97s
Simulated Annealing-3-exchange	30539879.10	28741918	62.49%	4.87s
SA-new operators (equal weights)	19505440.40	18877168	75.13%	351.41s
SA-new operators (tuned weights)	19244346.33	18455413	75.63%	371.13s
Adaptive Algorithm	18173223.10	17985318	76.53%	259.27s
Final Report	17719119.10	17370291	77.33%	398.54s

Listing 5: Optimal solution call_130_vehicle_40

```
1 sol = [123, 98, 98, 96, 96, 123, 25, 25, 9, 9, 56, 56, 64, 64, 0, 42,
    114, 42, 50, 1, 114, 50, 1, 0, 105, 105, 62, 62, 112, 37, 112, 46,
    36, 46, 77, 36, 37, 77, 0, 88, 88, 90, 82, 125, 90, 82, 65, 125, 127,
     65, 127, 0, 120, 61, 120, 61, 59, 59, 35, 12, 35, 12, 0, 74, 74,
    118, 118, 63, 83, 83, 63, 0, 67, 99, 99, 67, 110, 110, 48, 48, 0,
    129, 129, 45, 40, 45, 40, 101, 7, 7, 101, 10, 53, 30, 10, 30, 53, 0,
    84, 84, 117, 117, 81, 94, 94, 81, 0, 34, 68, 70, 34, 70, 68, 20, 20,
    0, 15, 15, 16, 16, 41, 41, 23, 23, 107, 107, 26, 26, 39, 39, 0, 3, 3,
     121, 121, 29, 17, 17, 29, 0, 103, 103, 55, 109, 109, 55, 0, 115, 69,
     69, 115, 0, 11, 11, 104, 79, 79, 104, 0, 66, 66, 0, 126, 126, 130,
    130, 28, 28, 0, 4, 4, 0, 5, 5, 22, 22, 0, 97, 97, 0, 49, 27, 49, 27,
    108, 108, 128,
2  128, 0, 87, 87, 47, 47, 76, 76, 0, 2, 73, 2, 13, 13, 73, 0, 106, 113,
     95, 95, 113, 106, 0, 86, 86, 100, 100, 89, 43, 43, 89, 0, 21, 32,
     21, 32, 78, 78, 0, 72, 60, 60, 124, 124, 92, 92, 72, 0, 6, 44, 6,
     44, 0, 85, 85, 0, 0, 116, 116, 57, 57, 0, 54, 111, 111, 54, 0, 8, 8,
      0, 38, 38, 31, 51, 51, 31, 0, 33, 102, 102, 33, 0, 58, 18, 58, 18,
     0, 80, 80, 119, 119, 0, 19, 19, 0, 93, 93, 24, 24, 0, 75, 75, 52,
     122, 91, 122, 91, 52, 71, 14, 71, 14, 0]
3 seeds = [398521026, 14243230, 298362426, 324295529, 498970133,
    709260715, 106389256, 781740669, 264304938, 939948533]
```