University of Padova

Lab exercise report

Methods and Models for Combinatorial Optimization

Samuele Rizzato

2024-2025

1 Problem of the exercise

A company produces boards to make electric panels and for each board a drill moves over it stops at the desired positions and makes the holes. Given the hole positions a company asks to find the hole sequence that minimizes the total drilling time, taking into account that the time needed for making an hole is the same and constant for all the holes. This problem can be represented on a weighted complete graph in which the nodes correspond to the positions of the holes and the edges represent the trajectory of the drill as it moves from one hole to another, with each edge assigned a weight corresponding to the time taken by the drill to move between the holes. This problem can be seen, using this graph, as a Traveling Salesman Problem.

2 Datasets

The datasets used for parts I and II can be found inside the *data* folder, and they store the weight matrices of the graphs. The method `createWeights`, used to create the datasets, utilizes a 50x50 matrix to represent valid positions, with values ranging from 0 to 200. Since the problem is related to electric boards, hole positions were generated in rectangular or linear shapes for each instance to make them more realistic and simulate how they would appear on a real board. The method `createRectangles` stops adding rectangles to an instance once the specified number of nodes is reached or surpassed. In the case of extra points, those are removed. After adding all the necessary nodes, the weight matrix is computed based on the Euclidean distance between each pair of nodes (`computeDistances`) and is saved to a file named *tsp(nodes).(instance).dat* (`writeWeightsToFile`). The code that creates weight matrices can be found inside the `WeightsFileManager` class and here are some examples:

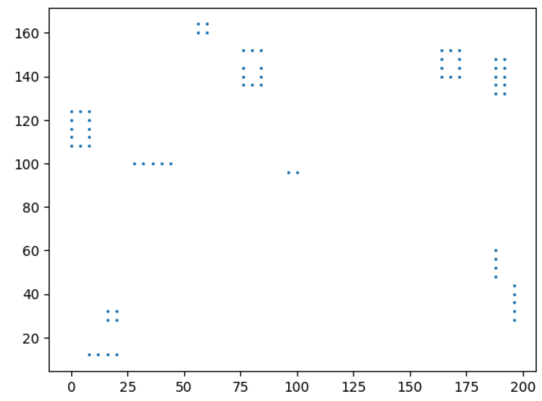
Figure 1: tsp70.3.dat nodes position

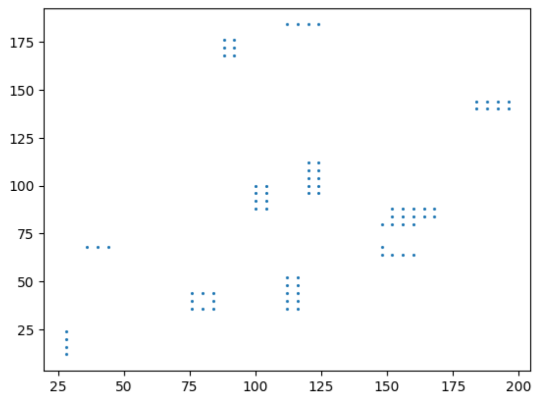
Figure 2: tsp80.2.dat nodes position

For each number of nodes were generated 5 instances.

3 Part I

3.1 Introduction

In the first part of the exercise a network flow model on the graph is given to solve the TSP and it is asked to implement it using cplex.

3.2 Cplex solver

The cplex solver class uses two matrices *map_x* and *map_y* to store the position of the variables in the cplex model

```
std::vector<std::vector<int>> map_x;
std::vector<std::vector<int>> map_y;
```

and it also implements the following methods:

- `setupLP`: adds every variable and every constraint one at a time using respectively `CPXnewcols` and `CPXaddrows` inside for loops;
- `solve`: solves the model created with setupLP. This method is just a wrapper around the function `CPXmipopt`;
- `getMapX`: returns the positions of x variables in the cplex model;
- `getMapY`: returns the positions of y variables in the cplex model;

A consideration during the implementation was to avoid adding to the model variables for edges of the form (i, j) *where* $i = j$ since these are not considered in a TSP problem they can be excluded to save memory space.
Also there is a function named `printVariables` to print the non-zero decision variables, i.e., the variables that are used in the solution. It uses `CPXgetx` in combination with `CPXgetnumcols` to retrieve all the decision variables and then uses *map_x* and *map_y* to print them in order.

`getMapX` and `getMapY` are just used to print the variables of the solved model.

3.3 Results

The computer on which the cplex solver was run had this configuration:

- Processor: Intel i5 7400 3 GHz
- RAM: 8 GB
- OS: Windows 10

Tests were conducted using the datasets that can be found inside the folder *data* and the following table shows the results:

nodes	obj. value	time (seconds)
10	260.072	0.287
	386.911	0.066
	424.029	0.069
	107.895	0.112
	493.713	0.065
40	610.798	3.639
	748.170	2.941
	547.100	4.827
	333.851	1.201
	581.563	4.191
70	706.746	63.477
	826.065	55.940
	835.257	335.287
	862.998	72.306
	725.344	275.810
80	804.685	574.386
	745.759	49.342
	818.519	304.217
	774.866	633.261
	760.650	64.811

Table 1: Cplex solver's results

Although the instances for each number of nodes are limited to 5, we can observe that, on average:

- 10 nodes require 0.119s;
- 40 nodes require 3.359s;
- 70 nodes require 160.564s;
- 80 nodes require 325.203s.

We can see that the bigger the instance the more the time is required to solve the model. Another thing to notice is that tests with bigger instances gave very different results in terms of time (e.g. the second test with 70 nodes required 55.94 seconds meanwhile the last test on the same number of nodes required 4 minutes and 35 seconds). This model, even though it gives the exact solution, is not very fast and it is not suitable for cases in which we have little time to provide a solution or there are very big instances.

4 Part II

4.1 Introduction

In the second part of the exercise it was asked to implement an algorithm as an alternative to the mathematical programming model and to test the performance of the implemented method.

4.2 Algorithm

The implemented method is a local search algorithm and it was designed as follows:

- Initial solution is given by a random combination of the sequence or the best insertion algorithm;
- Solution is represented by a sequence of nodes (path representation);
- Moves applied to get neighbourhoods are 2-opt;
- Evaluation function of a 2-opt move is computed as $var = c_{hj} + c_{il} - c_{hi} - c_{jl}$, it computes the variation cost adding edges $(h,j)(i,l)$ and removing $(h,i)(j,l)$;
- Exploration strategy chooses the first improving neighbour;
- The method stops when there are no more improving neighbours.

There is also a variant of this local search that uses random initial solutions + multistart to escape local optimum.

4.3 Implementation details

Best insertion uses the nearest nodes as the initial pair and at each iteration selects among the nodes to be visited the one minimizing the insertion cost among all possible positions in the cycle.

$$r = argmin_{i \in V \setminus C}\{c_{ir} + c_{rj} - c_{ij} :\ i, j \text{ } nodes\ consecutive\ in\ C\}$$

Best insertion was implemented in a less efficient way with a complexity of $O(n^3)$ instead of $O(n^2)$ for simplicity.

`LSSolver` class implements the algorithm with the listed methods:

- `initRnd`: initializes a solution with a random combination of the passed sequence;
- `initBestInsertion`: initializes a solution with the best insertion algorithm;
- `findNearestNodesIndices`: finds the nearest nodes indeces in the sequence, it is used by `initBestInsertion` to get the initial pair;
- `solve`: executes local search starting from the passed initial solution and stops when there are no more improving neighbours;
- `rndMultistartSolve`: executes a random initial solution local search multiple times, with the number of starts specified as a parameter;
- `apply2optMove`: applies a 2-opt move to a subsequence of the passed solution;
- `findFirstImprovementNeighbour`: incrementally evaluates neighbours and stops when it finds an improving move or there aren't improving moves;
- `evaluate`: evaluates a solution (it is used when printing a solution value).

4.4 Results

The computer configuration used for the tests is as described in section 3.3.

4.4.1 Solutions comparison

Random initial solution version of local search was run 10 times on each dataset and here are the results:

nodes	opt. solution	average value	standard deviation	max	min
10	260.072	260.072	0.000	260.072	260.072
	386.911	388.359	1.450	390.357	386.911
	424.029	424.029	0.000	424.029	424.029
	107.895	108.826	1.499	111.000	107.895
	493.713	493.713	0.000	493.713	493.713
40	610.798	620.030	5.778	634.481	614.819
	748.170	752.843	2.791	757.251	748.630
	547.100	558.553	9.298	574.517	548.053
	333.851	337.331	2.203	342.663	335.375
	581.563	595.568	14.652	624.694	583.144
70	706.746	723.184	4.645	732.432	716.261
	826.065	839.122	9.326	851.644	830.341
	835.257	862.429	8.525	881.218	852.971
	862.998	914.662	38.826	953.046	863.300
	725.344	755.099	32.236	802.815	731.538
80	804.685	827.599	27.818	903.967	807.946
	745.759	782.455	25.736	834.620	758.280
	818.519	841.131	27.783	919.235	826.304
	774.866	809.379	27.336	854.507	783.843
	760.650	784.851	11.296	797.522	761.781

Table 2: Results of random initial solution local search.

Average error

- 10: 0.247%
- 40: 1.536%
- 70: 3.449%
- 80: 3.633%

We can see that the average solution is optimal or not so far from the optimal when we have a small number of nodes; however, as nodes increase, the average error also starts to increase. The results show that solutions, on some instances, may be farther away from the mean or spread out making algorithm's behavior difficult to predict since the local search accepts the first improving neighbour we also depend more on the initial solution which is drawn at random.

The following table shows results of local search using best insertion initial solution:

nodes	opt. solution	bi ls solution
10	260.072	260.072
	386.911	386.911
	424.029	424.029
	107.895	107.895
	493.713	493.713
40	610.798	625.272
	748.170	753.570
	547.100	558.627
	333.851	363.648
	581.563	611.756
70	706.746	716.172
	826.065	946.491
	835.257	880.150
	862.998	952.477
	725.344	738.855
80	804.685	830.511
	745.759	771.613
	818.519	916.127
	774.866	821.541
	760.650	778.802

Table 3: Results of best insertion initial solution local search.

Average error

- 10: 0%
- 40: 3.863%
- 70: 6.703%
- 80: 5.402%

As for random initial solutions the average error gets bigger as the number of nodes increases but, if we compare average errors, results are worse in every case except for 10 nodes, so from this results error seems to grow faster.

4.4.2 Time comparison

The following table shows the average time (measured in seconds) of local search on the datasets and compares it with the linear programming solver:

nodes	cplex solver	bi ls	ri ls
10	0.119	0.0000094	0.00003
40	3.359	0.00020	0.0014
70	160.564	0.00132	0.0107
80	325.203	0.00186	0.0151

Table 4: Temporal results of the methods and the model in seconds.

As we can see local search was clearly faster in both versions than the cplex solver. Best insertion was the fastest and this could be because local search had to do less iterations since the best insertion algorithm, usually, gave better solutions than random ones the method was able to reach quickly a local optimum.

Both local searches as we saw are faster compared to the exact method, however the random initial solution local search provided on average better results than the best insertion one but is uncertain since different runs could give different results e.g. on *tsp70.4* the worst error was 10.43% and the best one was 0.03%.

4.5 Diversification with random multistart

A multistart approach was used to improve the results and the stability of the random init. local search, in this way since we start from different points we can explore more regions of the solution space and take the best local optimum found.

Random multistart was run 10 times on each instance and it used one parameter which was the number of starts in a single run. For parameter calibration were used the instances that can be found inside the folder *data/training* as the training set (there are three instances for each number of nodes) and the rest of the datasets that can be found inside the folder *data* as the test set.

4.5.1 Parameter calibration

After trying with 5, 10, 15, 20, 25, 30 starts on the training set, 5 starts already seemed enough to yield better results than the previous method and with 40, 70 and 80 nodes the solutions did not improve significantly after a certain threshold, for example with 40 nodes after 15 starts there were better solutions (figures 3 and 4) but the method required more time and the improvement was little. The parameters for the various problem sizes are provided in file *parameters.dat*.

While doing parameter calibration, i noticed an issue that may arise when applying this method on large instances. Specifically, the running time of a single local search tends to increase, while the algorithm's stability decreases (maybe because the search space contains more or harder to escape local optima). As a result, more starts would be required, which would lead to an overall increase in the running time.

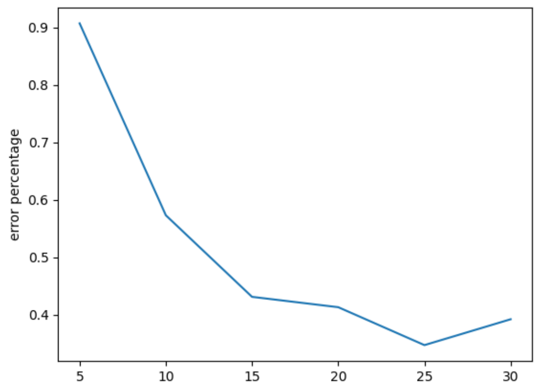

Figure 3: Multistart error with different number of starts on tsp40.6

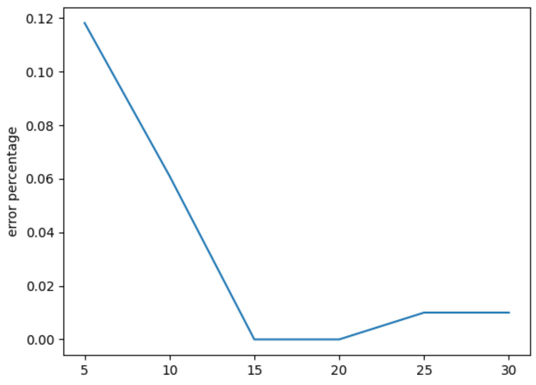

Figure 4: Multistart error with different number of starts on tsp40.7

4.5.2 Solutions comparison

These are the training results after parameter calibration:

nodes	opt. solution	average value	standard deviation	max	min
10	311.524	311.524	0.000	311.524	311.524
	152.171	152.171	0.000	152.171	152.171
	56.000	56.000	0.000	56.000	56.000
40	512.048	514.255	1.229	515.993	513.090
	538.666	538.666	0.000	538.666	538.666
	537.043	537.458	0.306	537.988	537.043
70	824.134	826.062	0.719	827.897	824.855
	742.054	745.543	2.314	750.364	742.054
	878.138	884.422	4.002	889.428	879.341
80	863.153	866.733	2.478	871.457	864.349
	896.608	902.034	3.330	907.592	896.608
	1010.93	1015.74	3.526	1019.92	1011.94

Table 5: Multistart results on the training set.

Average error

- 10: 0%
- 40: 0.169%
- 70: 0.485%
- 80: 0.499%

The results on the training set provided better average errors and more stability than just one random initial solution local search as expected.

Test set results:

nodes	opt. solution	average value	standard deviation	max	min
10	260.072	260.072	0.000	260.072	260.072
	386.911	386.911	0.000	386.911	386.911
	424.029	424.029	0.000	424.029	424.029
	107.895	107.895	0.000	107.895	107.895
	493.713	493.713	0.000	493.713	493.713
40	610.798	613.520	1.588	614.887	610.798
	748.170	748.847	0.597	749.818	748.170
	547.100	547.195	0.301	548.053	547.100
	333.851	335.185	0.901	336.146	333.851
	581.563	583.001	1.254	585.648	582.116
70	706.746	716.359	1.264	718.102	713.923
	826.065	827.589	1.329	829.516	826.273
	835.257	848.096	4.016	853.435	840.175
	862.998	863.997	0.788	865.643	863.300
	725.344	728.743	1.060	730.327	727.083
80	804.685	809.740	1.443	811.991	807.946
	745.759	754.956	2.838	760.517	750.464
	818.519	824.667	2.228	827.691	820.049
	774.866	780.084	1.673	782.715	777.017
	760.650	766.928	2.666	770.493	763.026

Table 6: Multistart results on the test set.

Average error

- 10: 0%
- 40: 0.24%
- 70: 0.733%
- 80: 0.822%

Test set results were worse than training possibly hinting to use more instances to calibrate parameters. There was also less stability on bigger instances like in the training phase. Although the results were not as good as those from training, the random multistart approach provided better solutions compared to both the random and best insertion methods.

4.5.3 Time comparison

In table 7, multistart is compared with previous methods and the cplex solver average running times (in seconds). Random multistart is still very fast compared to cplex solver but slower then the other methods since the time required is $ri\ ls\ time * number\ of\ starts$.

nodes	cplex solver	bi ls	ri ls	random + multistart
10	0.119	0.0000094	0.00003	0.000146
40	3.359	0.00020	0.0014	0.0242
70	160.564	0.00132	0.0107	0.1928
80	325.203	0.00186	0.0151	0.2688

Table 7: Temporal results of multistart, the other methods and the model in seconds.

5 Conclusion

The exact method gives the optimal solution but is slower compared to the local search approach, which one is better depends on the size of the problem and the company requirements e.g. if the company needs a fast response and doesn't need the optimal solution an exact method approach would be slower compared to local search. The best approach for the heuristic method was random multistart since it was a good tradeoff between solution quality, running time and development time (programming + parameter calibration) and the worst was local search + best insertion because, even though it provided good time performance, programming the best insertion algorithm was not straightforward and the solution quality on average was worse than the random approach.

Finally if the provided instance is not big and we don't need to provide a fast response the exact method would be better otherwise random multistart could be used to deliver good results in a limited amount of time.

6 Instructions

To try the solvers enter the `/solvers` folder and run:

```
make
./main
```

Once you run you should specify the number of nodes, the version of the file you want to test (datasets are already inside the `/data` folder) and the method, you should see something like this:

```
Insert nodes:
> 70
Insert version:
> 1
* Weights matrix initialized
1) Cplex
2) Random init. local search
3) Best insertion init. local search
```

```
4) Random init. + multistart local search
Choose method [1/2/3/4]: 3
```

In the example the selected file has 70 nodes, it is the version 1 and the method is best insertion init. local search. If the chosen method is random multistart you should also specify the number of starts provided in the `parameters.dat` file.

To try datasets creation in the `/exercise` folder run:

```
make
./main
```

Important: If you specify a dataset that already exists (i.e., any file in the `/data` folder), it will be overwritten.