Heuristics for the Travelling Salesman Problem

Pablo Miró Ruiz

April 5, 2020

1 Algorithm

The Travelling Salesman Problem (TSP) is NP-hard and one of the most well-known combinatorial optimization problems. The purpose of TSP is to find the shortest possible route which visits, from a collection of cities, each of them starting and finishing in the same one.

In order to implement an algorithms of my own which should be *polynomial-time* I inspired myself in section 35 *Approximation Algorithms* from CLRS 3rd edition, namely subsection 35.2, focused on the Travelling Salesman Problem.

The approach given in this book was to start from a given "root" vertex from the Graph, compute a minimum spanning tree and return the hamiltonian cycle H from the list of vertices given in the minimum spanning tree. I did some more research online and I found the *Double-Tree Algorithm* which was quite similar to the one described in the book and is proven to be a 2-approximation algorithm, ie. the cost of the tour can be, as a maximum, double the cost of the minimum spanning tree. The algorithm results as follows:

1. Find a Minimum Spanning Tree from the given adjacency matrix. Let T be the resulting MST.
2. Duplicate all edges (doubling). Find an Euler tour.
3. Shortcut the Eulerian tour.

Nevertheless, I wanted to implement it a bit differently so I changed steps 2 and 3 from the previous algorithm for the following ones, respectively:

- Build an n-ary tree from the edges from T.
- Run DFS to construct the permutation of the tour.

I did not know what a MST was and I found the idea very interesting because it contains all the vertices while minimizing the length of the edges and avoiding loops. In order to implement the MST I used Prim's algorithm which starts from a given root, calculates the distance from the visited nodes to the adjacents and selects the edge with smallest weight each time. Further, it only takes $\mathcal{O}(E \log V)$ time to compute it.

Once the MST is built I associate every node starting from the root to its children, considering these are the nodes to which it is connected to, with a default dictionary in python. With these mapping, I then build a Tree with the functions *build_Tree_recursion(self,dict,node)* and *build_Tree(self,parent,root)* so that I can build a traversal by exploring the tree from the root which I have previously selected (the root can be any node as Prim's algorithm can be started in any vertex).

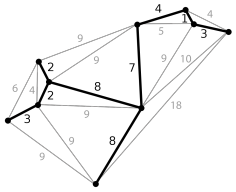

Figure 1: A MST

It is necessary to mention that in order to implement this, I used a class called *TreeNode* which has two attributes: value and children. This made my life simpler with respect to building the Tree.

With the tree built from the edges of the MST, I run Depth First Search algorithm to traverse the tree and keep a record of the nodes that have been visited so that we do not get any duplicates. If the node has not been visited it gets added to the permutation which, in the end, will be the one substituted in self.perm. The time complexity of DFS is $\mathcal{O}(E+V)$ as we studied in the course, so our algorithm will not be worse than *polynomial-time*.

The implementation of my own algorithm in Python can be found inside graph.py, namely in the function *myAlgorithm(self,root)* which implements the other subfunctions given the root from which we will create the MST. In the following section, we will consider *root = 0* for the experiments.

2 Experiments

2.1 Given data

To begin with the *Experiments* section, we will look how the algorithms perform with respect to the files provided by taking into account the cost of the whole path (first row) and the time it takes to run (second row). Firstly, we will start with the Euclidean graphs:

Euclidean	tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
cities25	6489.035 6.524μs	5027.406 99.77μs	2211.066 0.000952 s	2587.933 0.00011 s	2456.043 0.000216 s
cities50	11842.558 15.071μs	8707.056 0.000192 s	2781.273 0.0050195 s	3011.593 0.000398 s	3477.305 0.000638 s
cities75	18075.507 19.428μs	13126.072 0.000336 s	3354.912 0.0155799 s	3412.405 0.001254 s	4104.55 0.001402 s

Table 1: First results for the Euclidean case

Note: *tourValue()* uses the permutation $\{0, 1, 2, \ldots, N-1\}$ for its first run.

At first sight we can observe that the 2-Opt Heuristic gives the best results, even though it is the slowest of all. Furthermore, the algorithm implemented in Part C of the coursework gives results which are much better than *swapHeuristic()* while not taking too much time. It seems the Greedy approach is

the best one in terms of time/results and we may say it does make a little bit of sense because for maps of cities, going to the closest one each time is usually a good option. For the non-metric case we have the results in Table 2.

It is a bit difficult to compare every algorithm with only one test case because the only thing we can say is that Greedy gives us the best result, while myAlgorithm(0) the worst one. We will come to this case later in the report with our own test case implementation.

2.2 Own Testing

2.2.1 Euclidean Case

The test files given were OK, but four files is not enough. Hence, I created some new functions and used more graphs to compare the algorithms implemented in the coursework. I will proceed to describe the results and will mention tables which can be found later.

These tests and functions can be found in tests.py!

In order to check against more inputs, I defined *random_euclidean(n)* which uses the random library. Given an argument, an integer n, it makes a new file with n cities giving floating point x and y coordinates in the range $0.0 \rightarrow 2000.0$ using random.uniform(a,b).

Furthermore, I did some research and found some websites with different cities which could be used for trying the algorithms implemented. I parsed these files so that I could try the algorithms implemented and compare the results with the optimal tour given. In order to do so, I defined *parse_optimal_tour(filename)* and *parse_text_file(filename)*. It is to mention that we achieved really interesting results. I will reference enumerated tables which contain the costs and times for the different files used.

As a first comment, it is interesting to note that the algorithms keep the same evolution as before: 2-Opt gets the best cost, Greedy second, then myAlgorithm(0) and swapHeuristic() the last one, by a huge difference. It seems that swapping the values does improve the tour cost much, probably because for that to happen, adjacent values of the permutation should be closer to the city, and the bigger the input, the less likely for that to happen.

I found the closeness of Greedy() and myAlgorithm() very thought-provoking. After some thinking, I believe this happens on account of how the building of the MST is performed: It optimises every edge starting from a given root, so it is optimising the edges as it goes through the graph. Our Greedy() approach does the same, but directly considering the tour. Therefore, it makes sense indeed that the values of these two are quite related, even though myAlgorithm(0) takes a bit longer (probably because of the running of the DFS) and gets a less optimum tour.

One of the experiments I found most interesting was the one with 2392 cities. This is due to the fact that the optimal tour consists of the permutation $\{0, 1, 2, \ldots, N-1\}$, which is the one with which we initialise our graph! As a consequence, it seems really compelling to know whether an algorithm will get trapped and change the tour or not. As we can see in Table 3, the only algorithms which change the value of the tour are Greedy() and myAlgorithm(0). It seems that myAlgorithm(0) does so because it does not consider the given permutation, but starts a new one by building the MST from the start. Regarding Greedy(), it is possible that there is a better edge to choose, but which will end in a less optimal tour overall so it is not strange that this happened. On the other hand, swapHeuristic() and TwoOptHeuristic() do count on the given starting tour, and that is why they do not change it, since

there is not any improvement which can be made.

Speaking of time differences, I was taken aback by how much time it took 2-OptHeuristic to run when the input was greater than around 2500 cities (specifically with 10,000 cities in Table 5). Reviewing its code and the use given to tryReverse(), I believe this happens because when trying some crossings some new ones might be introduced, which will add more time and loops eventually.

On balance, the differences were most noticeable with big inputs. However, even with the given inputs and the 150 cities (Table 4) we can start to appreciate how each algorithm is going to behave.

2.2.2 Metric — Non-Metric Cases

Having analysed the Euclidean case, it is time to mention the other cases as well. In order to consider the triangle inequality I drew some graphs with a small amount of cities (4) and tried them on the algorithms. We can check in table 7 how they behave and we see that all but myAlgorithm(0) improve the tour cost.

On the other hand, I defined *non_metric(n)*, which creates a new file, *non_metric*, with n nodes and random distances between all of them using a Gaussian distribution function with the numpy library (numpy.random.normal(mean,std_deviation,size)).

Using a value of n = 2000 we can reaffirm the statements we used with the other cases: TwoOptHeuristic() takes a long time to run, Greedy() and myAlgorithm(0) are quite similar in running times, but Greedy() achieves a more optimum tour than myAlgorithm(0). On the other hand, swapHeuristic() stays way behind, even though it is the fastest to compute (since it only considers the adjacent nodes in the permutation).

We can still say that swapHeuristic() will only work properly when the optimum path is similar to the one given, but when the initial permutation is quite different the result will be really distinct to the optimum one. On the other hand, it is quite surprising to see that Greedy() outperforms 2-Opt in this case.

To sum up, having tried the algorithms with different inputs, we conclude that TwoOptHeuristic() gives the best results at the expense of time. Greedy() achieves pretty good answers in a decent amount of time, while myAlgorithm(0) always gets worse results relative to Greedy(). Even if the root is changed, the end result dos not vary that much. And last but not least, swapHeuristic() does not seem to work quite well for random graphs althought it does not take much time to run.

References

[1] CLRS: TSP, 1108 - 1117; Prim's Algorithm, 634 - 636

[2] TSPLIB: Symmetric traveling salesman problem (TSP)

[3] **Prim's Algorithm** → *Programiz* : `https://www.programiz.com/dsa/prim-algorithm`
→ *GeekForGeeks* : `https://www.youtube.com/watch?v=PzznKcMyu0Yfeature=emb_title`

tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
3.872 μs	9.404 μs	12.658 μs	13.128 μs	39.592 μs
9	9	9	8	10

Table 2: First results for the general case with Sixnodes

tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
378062.83	378062.83	378062.83	461207.49	528148.4
0.0010611 s	0.0027172 s	2.9629083 s	1.0830873 s	1.6479809 s

Table 3: (Euclidean) 2392 cities. Cost of optimal tour: 378062.83

tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
52812.15	40942.5	7060.85	8194.61	9202.46
52.171 μs	0.0010397 s	0.0766854 s	0.0039829 s	0.0055795 s

Table 4: (Euclidean) 150 cities. Cost of optimal tour: 6532.28

tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
10453981.78	8123149.28	164286.04	177813.2	202833.81
0.00479028 s	0.0972704 s	1004.4581719 s	34.0979325 s	45.0859355 s

Table 5: (Euclidean) Random 10,000 cities defined with *random_euclidean(10000)*

tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
3.872 μs	9.404 μs	12.658 μs	13.128 μs	39.592 μs
9	9	9	8	10

Table 6: (Euclidean) 2500 cities defined with *random_euclidean(2500)*

tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
11	10	10	10	11
3.704 μs	11.1 μs	11.7 μs	8.9 μs	44.6 μs

Table 7: 4 cities with triangle inequality

tourValue()	swapHeuristic()	TwoOptHeuristic()	Greedy()	myAlgorithm(0)
202788	151445	10925	6454	87764
0.001059 s	0.0162519 s	32.8997109 s	0.8193167 s	0.8968817 s

Table 8: (General non-metric) Random 2000 cities