Travelling Salesman Problem Solving

Student number: s2001696

March 25, 2020

1 2-Approximation Algorithm Declaration

Travelling salesman problem(TSP) is described as: given the graph G(V,E), where V are the vertexes, E are the edges and there exists (u,v)∈E, we should find an optimal Hamiltonian cycle , which satisfies the equation: $C(A) = min\sum_{(u,v)\in A} c(u, v)$

The approximation solutions are some of the best polynomial algorithms due to TSP problem, which of them is called 2-approximation algorithm, which is bounded to O($|E| * log|V|$)[1]. The prerequisite for the approximation algorithm is that the problem should satisfy the triangle inequality. Obviously, if we take the Euclidean distance between the cities as our metric(cost function), definitely it satisfies the following triangle inequality: $r(u, v) \leq r(u, w) + r(w, v)$, since the sum of the length of two sides of the triangle is always larger than the other side.

The 2-approximation algorithm is mainly based on Minimum Spanning Tree(MST) and the procedure of pre-order traversal. Given an undirected graph, we can choose the city that we start with. Suppose the starting point is the coordinate of the city in the first line of the .txt document. Suppose that U is the set that involves all the vertexes in the graph, V is the set that the cities have been visited once, while U-V is the set that the cities haven't been visited yet.

Beginning with taking the first city **a** from the set U, first we add it into V, and delete it from U-V. Finding the next vertex that is related to **a** and make the shortest path. Add **b** into V and delete it from U-V. So the next step is to find the third city **c**, which is related to either **a** or **b** and makes the total length the shortest, so it can be reached either from **a** or **b**. We can clearly see that this strategy is the same as Prim MST algorithm.

The next step is to traverse the whole tree pre-orderly. The reason why we do not try a post-order traversal or a in-order traversal is that 2-approximation algorithm is a greedy algorithm. And according to the property of a binary tree, the left leaf is always smaller than the right one if the node has a right leaf. But we are always trying to find the shortest path in the procedure, so we choose preorder traversal.

The final step is to calculate the distance of the returned queue of our algorithm.

Algorithm 1 Approx-TSP

```
1: procedure MyProcedure
2:     Select the first city as the root vertex
3:     loop:
4:     if visited_vertex ≠ ∅ then compute a minimum spanning tree from the root vertex
5:     end if
6:     all cities have been visited.
7:     Pre-order traversal
8: end procedure
```

2 Algorithm

The time complexity of the 2-approximation is about O($|E| * log|V|$), which is also equal to O($|V^2| * log|V|$). The time for traversing the whole vertex matrix spends about $|V^2|$, while traversing the tree can spend $log|V|$.

The higher bound for this algorithm is about O(V^3) while the lower bound is about O(V^2). Therefore, it's a polynomial time based approximation algorithm. And its solution is no more twice than the optimal value[1].

The function is being defined in the graph.py as **prim_MST_preorder(self)**. Just type **g.prim_MST_preorder()** and followed by **g.tourValue()**, we will get the solution. If we take cities50 as an example, it will shows 3477.305135479123

Algorithm 2 $Approx - TSP$

Require: $dists$, $number_of_cities$

1: **function** $prim_TSP_preorder(dists)$
2: $visited_id \leftarrow 0$
3: $T \leftarrow inf$
4: **while** $visited_numbers < number_of_cities$ **do**
5: $not_visited_id \leftarrow NIL$
6: $not_visited_id \leftarrow U - V$
7: $min_weight, min_from, min_to \leftarrow inf$
8: **for** $from_city \in visited_id$ **do**
9: **for** $to_city = 1 \rightarrow number_of_cities$ **do**
10: $weight = dists[from_city][to_city]$
11: **if** $from_city \neq to_city$ & $weight < min_weight$ & $to_city \in no_visited_id$ **then**
12: $min_to \leftarrow to_city$
13: $min_from \leftarrow from_city$
14: $min_weight \leftarrow dists[min_from][min_to]$
15: **end if**
16: **end for**
17: **end for**
18: $visited_id.append(min_to)$
19: $T[min_from][min_to] = dists[min_from][min_to]$
20: $T[min_to][min_from] = dists[min_to][min_from]$
21: **end while**
22: $is_visited \leftarrow False$
23: $stack \leftarrow 0$
24: $walk \leftarrow NIL$
25: **while** $stack.length > 0$ **do**
26: $node \leftarrow stack.pop()$
27: $walk.append(node)$
28: $is_visited.node \leftarrow True$
29: $nodes \leftarrow T.node! = inf$
30: **if** $nodes.length > 0$ **then**
31: **if** $is_visited.node == False$ **then**
32: $node_r \leftarrow reversed(nodes)$
33: $stack \leftarrow stack + node_r$
34: **end if**
35: **end if**
36: **end while**
37: **end function**

3 Experiments

3.1 Euclidean setting

According to the Euclidean setting, we promise that we have generated some of the graphs. The only thing is to make the coordinates smaller or larger. Here we take the "cities50.txt" as the experiment example. X and y coordinates are multiplied individually by the coefficient, which is tuned from 0.001 to 2. We admit a larger coefficient, but it will cost the calculation time. However, generally it will also show the same tendency as the coefficient goes to the infinity.

From the experiment, we can find that the two_opt heuristic is always the best algorithm, which is the closet to the optimal solution, followed by greedy algorithm and 2-approximation algorithm. However, swap heuristic performs bad, even the coefficient has been applied to. The reason is that swap algorithm would not find the global optimal route. Instead, after several iterations, it is mmuch harder to reach the local optimal route. It terminates at an early time.

We can also figure out that the effect of greedy algorithm is better than approximation algorithm but worse than two_opt. Since approximation algorithm is based on both greedy and minimum spanning tree. The MST might give another route that is different from what greedy results in. Totally, these three algorithms perform well no matter how we change the coefficient. The functions are **test_Euclidean_setting_x()** and **test_Euclidean_setting_y()** in tests.py.

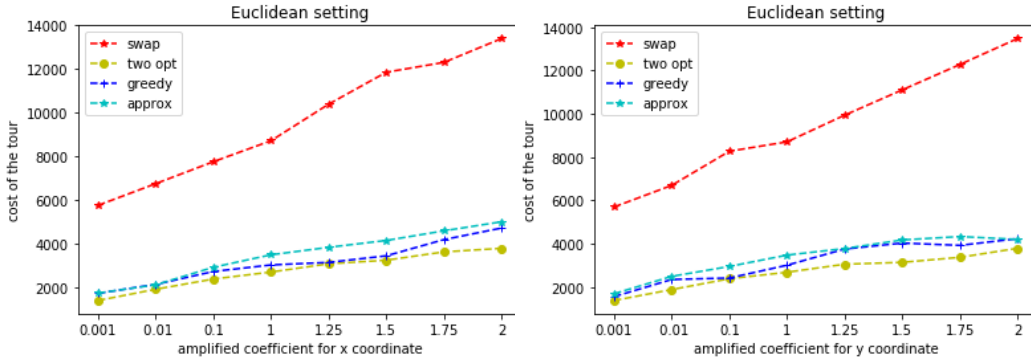

Figure 1: Euclidean setting for x coordinate

Figure 2: Euclidean setting for y coordinate

3.2 Metric setting

There are different metrics when the problem fits triangle inequality, such as Euclidean distance, Hamming distance and Chebyshev distance. We would like to find whether a different metric can bring something inspiring. Definitely, we find that the Chebyshev distance shows a lower cost of the city tour. Also, the two_opt, greedy and approximation algorithms perform well on each of three metrics. The function is **test_metric_setting()** in tests.py.

3.3 Non-metric setting

We know from the TSP problem that if the graph does not fit the triangle inequality, then it becomes a non-metric problem. In this case, we can simply multiply the distance by the weight(or probability, they are the same during the calculation). The result is shown in figure 4, where the greedy and two_opt algorithm keeps the best. The approximation performs badly because we have mentioned its prerequisite. It should fit the triangle inequality to ensure a less-than twice of the optimal value. Also the swap algorithm becomes better since the distance becomes more average that it's easier to reach the global optimal value. The function is **test_non_metric_setting()** in tests.py.

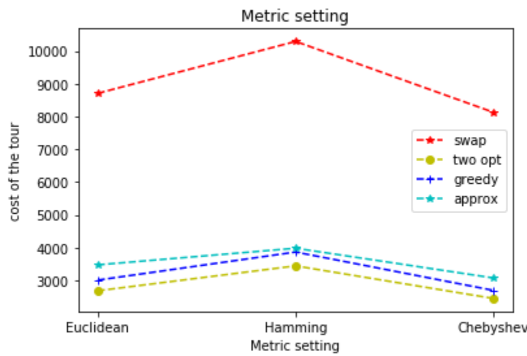

Figure 3: Different metrics

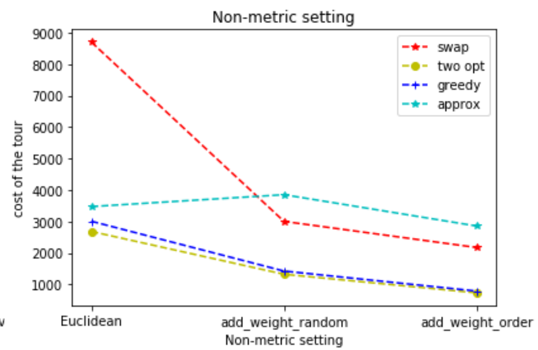

Figure 4: Non metric setting

3.4 dp algorithm

Dynamic programming is one of the exact solution for the TSP problem. However, the time complexity reaches $O(2^N * N^2)$.[1] So it's not the polynomial time solution. However, when the input size N is small enough, the task can still be completed in a very short time. And the comparison between heuristics and dp algorithm becomes legal.

In the experiment, the input size, which is the number of the cities, has been set in the range of 5 to 14. The result is shown in the Figure 5. We can find that the dynamic programming is always the optimal solution. When the input size is smaller than 7, the five algorithms show the same result. And when the number of the cities grows, we can find that two_opt heuristic curve is very close to that of dp one.

The consumption of the time is also compared. It's not surprising since we have mentioned its time complexity. It increases exponentially when the input size increases. The functions are **test_dp_tsp(g)** and **test_smaller_input_size()** in tests.py.

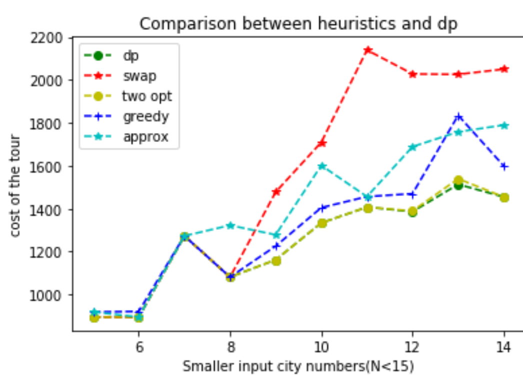

Figure 5: Comparison between heuristics and dp

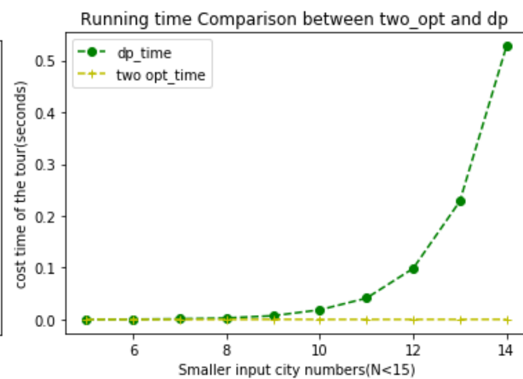

Figure 6: time comparison between two_opt and dp

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.