Algorithms to Solve Real Travel Planning Problems

Zhou Liu Jifan Xie Jing Ming

August 18, 2022

1 Introduction

Planning a trip is a practical problem in our daily life. People have the requirements that they visit as more spots as they can with the minimum cost on transportation, fuel and price, or minimum distance. The graph theory provides us a powerful tool to represent and solve this problem. We model all the spots as vertices and the distance/transportation/cost between spots as edges in a graph. Our goal is to find the shortest-path between the given source-destination pair or all pairs in the graph. We plan to solve the problem from easy to hard by setting different situations. At last, we are going to find a shortest possible route that visits each spot exactly once and goes back to the starting spot. This is modeled as the NP-hard travelling salesman problem. We will try to solve this problem using Dynamic Programming on small problem size and using Greedy Algorithm to reach approximate optimal solutions.

1.1 Motivation

Why our teammates find this topic interesting:

Zhou Liu: Every time I go out, I will firstly check Google map to find the shortest path to save time and fuel. Sometimes there is only one place I need to go, and I can just choose the shortest one among all possible paths. Sometimes I have several places to go and I want to plan in advance to find an optimal plan. With our study in CS5800, I realized such daily situations can be abstracted as finding optimized path(s) in a graph, and there are many algorithms can be applied depending on different situations, constraints and aims. Thus, in this project, I will clarify the different conditions and explore which and how an algorithm can be employed to solve such a problem.

Jifan Xie: My friend will visit me for a week this summer. I plan to show her around Boston with a list of spots. Planning the journey is very tiring, since I haven't lived in Boston for long, and I don't know the city very well. I needed to search how to get to each spot from home, and how to get to a spot from another spot. In the meantime, we found more good places to go in the planning process, so that I had to search the way to the new spots and modify the plan again and again. I hope there is a magic algorithm that could generate an optimal travel plan from a list of spots, freeing me from this annoying work and saving my time. If this problem can be solved in this project, it will play a big role in this journey.

Jing Ming: Early this year, I met the challenge of solving a shortest-path problem in the code competition Code Jam to I/O for Women 2022. The problem gives a path travelling through every node in a graph. The goal is to find the cheapest way to see all sights at least once. At that time, I found it difficult because I knew little about Dynamic Programming and Greedy Algorithms. Now I am learning these two algorithms in the CS5800 course. It is the best time to overcome the obstacle I met in the competition. By the way, the shortest-path problems consist of many variants. We can have the chance to create self-related questions by setting specific constraints, like taking visit time into consideration when making the best travel plan using shortest-distance algorithms. In this project, I expect to analyze different shortest-path algorithms under various application scenarios and create the best algorithm to solve the self-defined question.

2 Problem Description

Generate the best plan for travelling a list of n spots (the positions of spots are known), which:

1. Decide whether the spots can be reached by existing methods, such as walking or public transportation. – Decide whether there exists a path between two vertices.
2. If a spot is accessible, generate the best path to get to the spot from home. – Generate the shortest path from one vertex to another vertex.
3. Generate the best plan for all spots. – Generate the shortest path of accessing each vertex once and come back to the starting vertex in the end.

2.1 Problem Scope

Our project will focus on the travelling scenes, and discuss several different conditions and aims, under which people will make optimal decisions following different algorithms. In our project, we put forward 4 situations with clear conditions and analyze different methods under such situations. As there may be more complicated conditions in the real life, here we just care about the main conditions for modelling and analysis.

To clarify, we model the question with the basic graph structure, G = {V, E}. V represents the spots we want to visit, and E can represent direct distance, costs, and public transportation under different conditions.

1. The first question is to check the existence of a path. Given the start spot and the end spot, check whether there is a path from the start to the end. We always meet with the such situation during travelling. After visiting a spot, we need to first check whether we can go to the other spot we want. Here, each edge represents there is a path between 2 vertices.
2. The second question is to calculate the shortest path between 2 spots. When travelling from just one spot to the other, we want to find the shortest path from the start to the end to save time and fuel. Here, the edge represents costs from one vertex to the other with weights.
3. The third question is to plan ahead rather than find the next spot during travelling. Given all the spots we want to visit, we calculate the minimum distance between each pair of spots. After we visit spot A, we can choose to visit the place that can be arrived from A within the minimum distance. Here, the edge represents the direct distance between 2 vertices.
4. The last question is the travelling salesman problem. For some aggressive travellers, given all the spots they want to visit, they want to make an optimal plan in advance, with which they can visit all the spots within the minimum distance in total. Here, the edge represents the direct distance between 2 vertices.

3 Path Existence Problem

When we travel, we may encounter a situation where we can't get to our destination directly. Especially for people who use public transport, it is very common to change between different lines. In this case, we need to know whether the destination can be reached from the current location.

To solve this problem, we need a graph $G = (V, E)$ first. Its vertices are the spots (e.g. bus/subway stations). If we can access another vertex from a vertex, there exists an edge between them. The edges are undirected, and we don't care about their weights here. Then, this problem can be solved by BFS (Breadth-First Search, see [CLRS22] chapter 22.2) or DFS (Depth-First Search, see [CLRS22] chapter 22.3). The time complexity of both algorithms is $\mathcal{O}(V + E)$.

In this specific problem, BFS is better than DFS in most case, since BFS gives a path with the minimum number of steps, that is, it can provide the path with the least number of transfers. However, BFS requires more space complexity, $\mathcal{O}(V)$, while DFS requires only $\mathcal{O}(\log V)$ space complexity. DFS is better when space usage is more demanding.

4 Single Source Shortest Path Problem

Consider this situation, you move to a new city and plan to travel around using vacation days. You make a list of interested spots and stuck at planning the trip. People easily get tired if they spend so

many time on the transportation. So you may want to find a way to group these spots and visit them one group a day. Your desire is to visit all spots with the least travelling time.

This problem is a typical single source shortest path problem. First, we create a graph $G = (V, E)$. The source vertex is your home and all the interested spots forms the other vertices in a graph. If your are able to travel from one spot to another spot, connect these two spots to generate an edge in the graph. Due to the presented situation, the weight of edge is the travelling time and the edge must be directed since the travelling time differs in forward and backward direction. Second, your goal can be modeled as finding the path from your home to every other spots with the shortest cost. Then, every day you can choose one path from home to a destination and visit all the spots on it at once. It is guaranteed that the travelling time from your home to every spots is minimized.

To solve this single source shortest path problem, the algorithm selection depends on the edge representations. The edge is directed or undirected, positive weighted or has negative weights. As discussed above, the edge is directed when denotes the travelling time. When it denotes the distance between two spots, the edge becomes undirected. For other aspects like money and gasoline, the edge is always positive. However, as electric car becomes more attractive today, people may concern electricity usage during their travel if they are driving a electric car. One fact about the electric car is that it consumes electricity driving on flat land or up hills while it recovers energy going down hills. This leads to a negative weighted edge in the graph if one spot is at the top of a mountain and another is at the foot.

For directed or undirected positive weighted graph, we can use Dijkstra's algorithm to find the shortest path from single source to other vertices as described in [CLRS22] chapter 24. This algorithm is based on the greedy approach that it repeatedly select the vertex u with the minimum distance among the unvisited vertices set $V - S$, mark it as visited $u \in S$ and relax all edges leaving u. In the problem set 6 question 1.3, we use loop invariant to prove its correctness. Its time complexity depends on the min-priority queue implementation because it has to pick the vertex with minimum distance in each iteration and the distance is dynamically updated during the process. The time complexity is $\mathcal{O}(V^2)$ for array implementation, $\mathcal{O}(E \log V + V \log V)$ for binary heap implementation and $\mathcal{O}(E + V \log V)$ for Fibonacci heap implementation.

For directed negative weighted graph, we can use Bellman-Ford algorithm to find the shortest path if there is no negative-weight cycles. This algorithm is based on the idea of Dynamic Programming. It progressively decreases vertices' estimated distance by the weight of founded shortest path from the source to other vertices until reach the shortest path of whole graph. The Bellman equation for this problem is $OPT_v = \min\{OPT_v, OPT_u + Edge_{(u,v)}\}$. Problem set 9 question 1.2 proves its correctness. Its time complexity is $\mathcal{O}(V \times E)$.

To conclude, Dijkstra's algorithm suits for directed or undirected positive weighted graph with a considerably low time complexity while Bellman-Ford algorithm finds the solution for directed negative weighted graph with a slower running time.

5 All Pairs Shortest Path Problem

Before travelling, we want a map recording the shortest path between each pairs of spots. With such a map, after I visit spot x, and I plan to visit spot y, I can easily get the shortest path from x to y without re-calculation. Or after I visit spot x, and I just want to find another spot that is nearest to x, I can use this map and choose such spot whose shortest path to x is the shortest among all other spots.

Given such situations and problem, we referred to the Floyd Marshall Algorithm to pre-calculate such a map.

The basic idea of Floyd Marshall Algorithm is dynamic programming. It noticed that for two places x, y, its distance can be "relaxed" if there exists another place k, distance[x][k] + distance[k][y] ¡ distance[x][y], then the distance can be updated.[CLRS22]

Thus, start from the initial distance map, dp, we loop intermediate point from 0 to n - 1, and relax dp[i][j] for all i, j from 0 to n - 1 by the recurrence relationship that dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]). Finally, the dp is the map we want recording all shortest path between each pairs.

An implementation is uploaded to GitHub: Floyd Marshall Algorithm Implementation

6 Travelling Salesman Problem

6.1 Dynamic Programming

Though Travelling Salesman Problem is a NP-Complete problem, we can still employ non-polynomial algorithms to obtain a precise answer for small data inputs, especially suitable for the travelling situation with limited spots. It is easy to come up with the brute force method, listing all the possible paths, calculate each length and choose the minimum one. For N spots, this brute force method will take O(N!) time because there are N! possibilities of paths.

With the study of dynamic programming in shortest paths problem, for example, Floyd Marshall problem, we were inspired to solve TSP by smaller sub problem. Suppose we already have the shortest path from spot 0 to spot 1, spot 2, ... spot n and back to spot 0, then, the path from spot 1, spot 2 to spot n and back to spot 0 must also be a shortest path, which is easy to be proved by contradiction. If there is another shorter path within spot 1, ... spot n, we can find a shorter path within spot 0, spot 1, ... spot n, which contradicts the assumption that the given path is the shortest path. So we tried to solve the TSP problem by a dynamic programming method, whose time complexity is smaller than the brute force method.[Bel62]

6.1.1 Model

We model the given situation to a graph. Each vertex representing a spot, and each edge records the distance between vertex i and vertex j. If there is no direct path from vertex i to j, we record it as INF.

In our algorithm, we use an adjacent matrix to record the graph because we assume that the path between each pair of vertices is bidirectional.

6.1.2 Algorithm

Algorithm 1 Dynamic Programming Algorithm for TSP

```
procedure TSP-Dynamic Programming(G)
    Create a 2d array dp to record the procedure, the row represents a different set of cities, and the column represents all cities.
    Initialize the base case dp[i][0]
    Loop each j in row, loop each i in column, loop each city, calculate dp[i][j] by the state transition equation
    return dp matrix
end procedure
```

The implementation has been uploaded on GitHub TSP by DP method
Here I will specify 3 key points for the implementation.

1. Definition of dp matrix

   The first step is to define the dp matrix and its meaning. We define dp[i][V]:

   i represents a vertex in the graph

   V represents a set of vertices

   dp[i][V] represents the shortest path value from vertex i to start s, visiting all vertices in V for one and only one time

2. DP state transition equation

   Let C_{ij} represents the distance from vertex i to vertex j.

   If V = $\emptyset$ and i $\neq$ s, dp[i][V] = C_{is}

   If V $\neq \emptyset$ and k $\in$ V, dp[i][V] = min(C_{ik} + dp[k][V - {k}])

3. Bitmask and state compression in DP

   We use a 2d matrix to record dp result, but V is a set, which will take more space to store. Thus, to reduce space complexity, we use a state compression skill by bitmask operation.

   Let the number of all vertices is N. For set V, we use a N-length binary number to represent it. If v_i in V, the bit i is 1, else 0. For storage, we convert the binary number to the corresponding decimal number. For example, if N = 4 and V = $\{1, 2, 3, 4\}$, V can be represented by 1111 and stored as 15.

6.1.3 Proof of Correctness

1. Sub problem:

   Let dis(i, V) represents the shortest path from vertex v_i to start v_0, visiting all vertices in V for one and only one time.

   To calculate dis(i, V), we notice that we can find any vertex in V, let it v_k, the dis(i, V) can be reduced to dis(0, V - $\{v_k\}$) + C_{ik} if dis(i, V) < dis(0, V - $\{v_k\}$) + C_{ik}, where C_{ik} represents the distance between vertex v_i to v_k without visiting other vertices. Thus, we find the sub problem working on smaller set V - $\{v_k\}$.

   Let S = set of all vertices in the graph. Thus, the answer we what is dis(0, S), which can be solved by working on sub problems on S - $\{v_k\}$ recursively.

2. Base case:

   For base case, let's consider dis(i, $\emptyset$), it means the shortest path from i to start v_0 visiting no other vertices, thus it equals to the distance directly from v_i to v_0.

3. Recurrence relationship:

   If V is not empty, we will calculate dis(i, V) by solving on sub problem, i.e. dis(i, V - v_k). As dis(i, V) represents the shortest path, we will loop all v_k in V and find the minimum answer of dis(i, V - v_k) + C_{ik}. Thus, the recurrence relationship is

   dis(i, V) = min(dis(i, V - v_k) + C_{ik}) for all v_k in V.

4. How to solve:

   With the analysis above, we can start from the base case, getting all dis(i, $\emptyset$) by the given graph.

   Then we follow the recurrence relationship to add vertex to V and finally get the dis(0, S) by a bottom up dynamic programming method.

   Here is an example: To calculate dis(0, $\{1, 2, 3, 4\}$), we will calculate dis(1, $\{2, 3, 4\}$) + C_{01}, dis(2, $\{1, 3, 4\}$ + c_{02}, dis(3, $\{1, 2, 4\}$) + c_{03}, dis(4, $\{1, 2, 3\}$) + c_{04}, the minimum of which is dis(0, $\{1, 2, 3, 4\}$).

   To calculate dis(1, $\{2, 3, 4\}$), we need to calculate dis(2, $\{3, 4\}$) + c_{12}, dis(3, $\{2, 4\}$) + c_{13}, dis(4, $\{2, 3\}$) + c_{14} and find the minimum one. Similar procedure for dis(2, $\{1, 3, 4\}$), dis(3, $\{1, 2, 4\}$) and dis(4, $\{1, 2, 3\}$).

   To calculate dis(2, $\{3, 4\}$), we need to calculate dis(3, $\{4\}$), then to dis(4, $\emptyset$), which is the base case. Thus, by a bottom up dynamic programming method, we will start from the base case, increase the V by recurrence relationship and finally get the dis(0, S) we want.

6.1.4 Time and Space Complexity

1. Time complexity

   Let the number of vertices or spots is N. For the first level loop, we loop each spot/vertex, which takes O(N) time.

   For second level loop, we loop all situations of V, which takes O(2^N) time as there are 2^N possibilities for subsets of set of N elements.

   For third level loop of intermediate vertex to relax edges, it takes O(N) time.

   Thus, in total, this algorithm takes O(2^N * N^2) time.

2. Space complexity

   Considering the dp matrix, it takes O(2^N * N) space.

   Considering the graph's adjacent matrix, it takes O(N^2) space.

   Thus, in total, it takes O(2^N * N) space.

6.2 Greedy Algorithm

Although dynamic programming can obtain the optimal solution, its time complexity is relatively large. However, in daily life, rather than pursuing the optimal solution, we may need to obtain an approximate solution in a relatively short time, especially when the amount of data is large. Greedy algorithms we studied in Module 6-7 can do that. The algorithm is called Nearest Neighbor Heuristic, which is very straightforward: visit the nearest unvisited spot each time, until all the spots have been visited. Although the optimal solution may not be obtained, it can provide a combination of local optimal solutions in a small time complexity.

6.2.1 Model

The instance of this problem can be modeled into a complete graph $G = (V, E)$. Each vertex representing a spot, and the weight of each edge is the distance of the two spots linked by this edge. The graph is represented by an adjacent matrix A, in which the weight of edge between vertex i and vertex j is stored in A_{ij} and A_{ji}. A_{ii} is set to be 0.

6.2.2 Algorithm

The implementation of the greedy algorithm along with the visualization is uploaded on GitHub TSP by Greedy Algorithm

The Greedy algorithm goes as below, the input is a graph $G = (V, E)$ and the starting point *source*. The graph G is represented by a matrix, in which $G[i][j]$ is the distance between vertex i and vertex j:

Algorithm 2 Greedy Algorithm

```
1: function TSP-GREEDY(G, source)
2:     Initialization: path = [source], cost = 0, visited = [0] * V
3:     Mark the source visited: visited[source] = 1
4:     curr = source
5:     Loop for V-1 times to find the next V-1 spots:
6:     for k in range(V-1) do
7:         for i in range(V) do
8:             Find the smallest G[curr][i] in G[curr] with visited[i] = 0:
9:             next-dis = G[curr][i]
10:            next = i
11:        end for
12:        path.append(next)
13:        visited[next] = 1
14:        cost += next-dis
15:        curr = next
16:    end for
17:    path.append(source), return to the source
18:    return path, cost
19: end function
```

The returned path is the TSP tour given by greedy algorithm, and cost is the total weight of the tour.

6.2.3 Approximation Ratio

To find out how close the approximate solution of the greedy algorithm to the optimal solution is complicated. In fact, the Nearest Neighbor Heuristic does not have a constant approximate ratio for

the Traveling Salesman Problem. Judith Brecklinghaus and Stefan Hougardy have proved in [BH15] that the approximation ratio of the greedy algorithm for the Traveling Salesman Problem is $\Theta(\log n)$. Since the approximation ratio is a logarithm function to n, as the number of vertices grows, the difference between the approximate solution and the optimal solution grows. Thus, when the size of input is large, we may end up with a less satisfactory result.

6.2.4 Time Complexity

1. Time Complexity

   The first level loop traverse V-1 times to find the next V-1 spots after the source, which takes $\mathcal{O}(V)$ time. The second level loop traverse all spots to find the nearest unvisited one and decide which spot to go next, which takes $\mathcal{O}(V)$ time. Thus, in total, this algorithm takes $\mathcal{O}(V^2)$ time.

2. Space Complexity

   The space is mainly used in building the complete graph in this algorithm. The adjacent matrix of the graph takes $\mathcal{O}(V^2)$ space.

6.3 Approximation Algorithm

One way to solve NP-complete problems is to find its near-optimal solution. There are many approximation algorithms designed to solve Travelling Salesman NP-complete problem. We have learned 2-approximation solution in the week12 lecture and proved its guaranteed approximation ratio in the problem set 12 question 5. Here, we will introduce an improved approximation algorithm, Christofides – Serdyukov algorithm [Chr76]. This is a heuristic approach, guaranteed to give the solution which is at most 1.5 times the optimal to the metric TSP problem. Though this algorithm is independently discovered by Nicos Christofides and Anatoliy I. Serdyukov in 1976, it is still known as the best approximation algorithm for metric TSP problem.

Christofides algorithm is inspired by 2-approximation algorithm. It reserves the first part of finding minimum spanning tree(MST) in the graph. However, it additionally finds the minimum weight perfect matching(MWPM) in the induced subgraph generated by the odd vertices in the MST. Combine MST and MWPM results to form a multi graph. At last, find the tour on this multi graph. It shares the same idea as the 2-approximation algorithm, but with the finding of MWPM it can reduce the approximation ratio to 1.5.

6.3.1 Model

Since our goal is to find the shortest tour which visits each spot exactly once except the start vertex and the shortest refers to the shortest distance, we can model the problem as a undirected weighted graph $G = (V, E)$. The vertices set contains all interested spots. The edge weight represents the distance between two spots and it is non-negative. Because the distance exists between every two spots pair, this graph is a complete graph that every vertex is connected with each other. Also, the distance is the Euclidean Distance between two spots, it satisfies the Triangle Inequality. This forms a special case of TSP problem called metric TSP and Christofides algorithm requires the problem to be metric TSP.

To conclude, we build an undirected positive weighted complete graph for this metric TSP problem. The instance of this problem is the complete graph $G = (V, E)$. Let $d(u, v)$ denotes the non-negative distance between vertex u and vertex v. $d(u, v)$ satisfies:

- $d(u, v) \geq 0$ and $d(u, v) = 0 \iff u = v$
- $d(u, v) = d(v, u)$
- $d(u, v) \leq d(u, w) + d(w, v)$ (Triangle Inequality)

Let $d(A)$ denote the total distance of the edges in the subset $A \in E$:

$$d(A) = \sum_{(u,v)\in A} d(u, v)$$

Our goal is to find Hamiltonian Circuit with the minimum total distance $d(A)$.

6.3.2 Algorithm

The implementation has been uploaded on GitHub TSP Christofides Algorithm Implementation

The Christofides algorithm goes as below, the input is a graph $G = (V, E)$:

Algorithm 3 Christofides Algorithm

1: **procedure** TSP-CHRISTOFIDES(G)
2: Create a Minimum Spanning Tree T of G
▷ Using Prim's Algorithm in the implementation
3: Find the set of vertices O with odd degree in T
4: Create a Minimum-Weight Perfect Matching P in induced subgraph given by vertices from O
5: Combine the edges in T and P to form a connected multigraph M
6: Generate the Eulerian Circuit E from M
7: Generate the Hamiltonian Circuit H from E by skipping repeated vertices except start vertex
8: **return** Hamiltonian Circuit H
9: **end procedure**

The returned Hamiltonian Circuit H is the founded approximation TSP tour to this metric TSP problem.

6.3.3 Proof of Correctness

We will prove that Christofides algorithm is a polynomial-time 1.5-approximation algorithm for the metric TSP problem.

Proof: Let H^* denote an optimal TSP tour for the given set of vertices. Since deleting one edge of the tour H^* we can receive a spanning tree and $d(u, v) \geq 0$, the MST T we create in line 2 of procedure TSP-CHRISTOFIDES must provides a lower bound on the total distance of the optimal tour H^*:

$$d(T) \leq d(H^*)$$

Line 3 in procedure TSP-CHRISTOFIDES finds the vertices with odd degree in T. So the found vertices set O contains leaves and maybe some inner vertices of T. By handshaking lemma that sum of the degree of all nodes must be even, the cardinality $\mid O \mid$ of this odd-degree vertices set O is even. Thus, a perfect matching always exists in the induced subgraph given by the vertices set O. Since the input graph is complete, the subgraph is complete either and there is $\frac{(2n)!}{2^n \cdot n!}$ number of perfect matching for complete graph K_{2n}. Therefore, we can find a minimum weight perfect matching P in polynomial-time even if we search with the brute force approach.

Let N^* represent an optimal TSP tour on the odd vertices set O. Let N_1 and N_2 be two alternatively selected perfect matching in the induced subgraph of O which $d(N_1)+d(N_2) = d(N^*)$. Since the perfect matching P we invented in line 4 has minimum weight, $d(P) \leq \min(d(N_1), d(N_2))$. Combine the former two inequality, we get

$$d(P) \leq \frac{1}{2} d(N^*)$$

Because we can create optimal TSP tour N^* on subgraph from the optimal tour H^* by skipping the even degree vertices and the graph follows triangle inequality, it is guaranteed that $d(N^*) \leq d(H^*)$. Therefore, the minimum weight perfect matching P satisfies:

$$d(P) \leq \frac{1}{2} d(H^*)$$

By combining the edges from T and P in line 5, we now create a multigraph M whose edge weights follows:

$$d(M) = d(T) + d(P) \leq d(H^*) + \frac{1}{2} d(H^*) = \frac{3}{2} d(H^*)$$

Here we can see the fact that each edge in the multigraph M has even degree. Because the perfect matching P consists of one edge for each vertex in the odd degree vertices set O and O is a vertices

subset in T. By combining T and P, one edge is added to every odd degree vertex in T which increments the degree by 1.

Since all vertices has even degree in M, the Eulerian circuit E exists in the connected multigraph M. Because E visits every edge exactly once, the total distance of E does not exceed M:

$$d(E) \leq d(M) \leq \frac{3}{2} d(H^*)$$

At last, we generate the Hamiltonian circuit H from the Eulerian circuit E in line 7 by skipping repeatedly visited vertices. For example, let d be one repeatedly visited vertex in E and E is like a, b, c, b, d, a. To convert E to H, we have to delete edge (c, b) and edge (b, d) while adding the edge (c, d). Since the graph G satisfies triangle inequality, $d(c, d) \leq d(c, b) + d(b, d)$. Therefore, the circuit H follows:

$$d(H) \leq d(E) \leq \frac{3}{2} d(H^*)$$

Hence, the result H of procedure TSP-CHRISTOFIDES is no bigger than 1.5 times the optimal TSP tour H^*. ■

6.3.4 Time Complexity

For line 2 in the algorithm 6.3.2, we use Prim's algorithm to find the minimum spanning tree T. We have learned this algorithm in [CLRS22] chapter 23.2. We choose this algorithm because in problem set 7 question 2 it recommends to use Prim's algorithm when the graph is dense. Here, the graph in the model is a complete graph, it is a dense graph with $n * (n-1)/2$ edges. Followed by this, we first implement the Prim's algorithm with Fibonacci heap (see TSP-Christofides Fibonacci Heap Implementation on GitHub), this takes $\mathcal{O}(E + V \log V)$ time. Because $\mathsf{E} = \mathsf{V} * (\mathsf{V} - 1)/2$ in the complete graph, the Prim's algorithm with Fibonacci heap takes $\mathcal{O}(V^2)$ time. Then we implement the Prim's algorithm with binary heap (see TSP Christofides Algorithm Implementation on GitHub), this takes $\mathcal{O}(V \log V + E \log V)$ time. Taking edge set cardinality into consideration, the Prim's algorithm with binary heap takes $\mathcal{O}(V^2 \log V)$ time in total. The Fibonacci heap is faster than binary heap theoretically. But in practice when $\mid V \mid$ is not very big, the binary heap implementation runs faster than the Fibonacci heap. That said, both implementations are no larger than $\mathcal{O}(V^3)$.

For line 3 find odd vertices set O, we traverse all edges in T to calculate the degree for each vertex and then traverse all vertices to find the odd degree vertices in the implementation. Both iterations take $\mathcal{O}(V)$ time because MST T is a tree which contains $\mid V \mid$ vertices and $\mid V \mid -1$ edges.

For line 4 create minimum weight perfect matching, we generate an induced subgraph using vertices in O and then find M on this subgraph. The subgraph generation takes $\mathcal{O}(V^2)$ because we have to traverse every edge which connects every vertices pair in O. Then we use MAX_WEIGHT_MATCHING method in Python NetworkX library (see Reference) to find the perfect matching. This function use Edmonds Blossom algorithm to get solution and takes $\mathcal{O}(V^3)$ time. So in total this line takes $\mathcal{O}(V^3)$ time.

Noticed that the vertices size of T and M is $\mid V \mid$ and the vertices are both ordered in a line. So the edges set size of T and M is also $\mid V \mid$ that an edge exists between each two adjacent vertices. Combine T and M in line 5 by appending one array to the other array takes $\mathcal{O}(V)$ time. The generation of the Eulerian circuit in line 6 costs $\mathcal{O}(V)$ time if we use Hierholzer's algorithm. And the last step in line 7 takes linear time either that we simply traverse through E and delete repeated vertices except the start vertex.

To combine all steps, only the line 4 takes $\mathcal{O}(V^3)$ time which is the upper bound for this algorithm. Therefore, the overall growth of the Christofides algorithm is $\mathcal{O}(V^3)$. This is faster than the previous TSP-Dynamic Programming algorithm.

6.3.5 Space Complexity

The major space consumption in this algorithm is to build the undirected complete graph which takes $\mathcal{O}(V^2)$ space.

7 Conclusion

Under a general travelling scene, there are various conditions and no unified algorithm can solve the shortest path problem under all conditions. Thus, we enumerated multiple situations and proposed suitable algorithms for different situations.

To decide whether a spot is accessible, you can choose BFS or DFS to get an available path. BFS will provide a path with the minimum number of steps, while DFS will use less space.

To plan a trip starting from your home with the shortest cost, you can choose Dijkstra's algorithm if the cost is positive from one spot to another spot or you can choose Bellman-Ford algorithm if some of the costs are negative.

To calculate each pairs' shortest path within all cities for trip convenience, the Floyd Marshall algorithm based on dynamic programming can be applied if all distances are positive.

To plan a tour that visits all spots exactly once, we analyzed three different algorithms to solve this TSP problem precisely and approximately. When applied to practical situation, we can choose different algorithms depending on the data size, i.e. how many cities we want to visit.

For small data input, you can choose a dynamic programming method to gain the precise answer. Though the time and space complexity is not polynomial, it works better than a direct brute force method. However, this method cannot be applied to a bigger data input. In our test, the small data input has only 9 cities, and we can get the answer in 0.009 seconds. But for the larger data input with 34 cities, we can not even get the answer because of the high time complexity.

For large data input, you can choose a greedy algorithm, nearest neighbor heuristic, to get an approximate solution. Its time complexity is only $\mathcal{O}(N^2)$, which is the smallest among the three methods, so that it can calculate very fast even with a large size of input. However, this algorithm is $\Theta(\log N)$-approximation. Its solution may not be close enough to the optimal solution, especially with a large input.

For Christofides approximation algorithm, you can choose Prim's algorithm with binary heap implementation to create the minimum spanning tree if the data size is not big enough. When the size becomes extremely big, please choose Prim's algorithm with Fibonacci heap implementation to lower the time complexity.

7.1 Weakness and Limitations

Our analysis has some limitations:

- For the implementation of TSP problem, we can include more datasets for testing, and make more comparisons among different solutions given different datasets.
- Due to time constraints and the lack of large datasets with known optimal solutions, we do not explore much about the performance of greedy algorithms and analyze its approximation to the optimal solution.
- The lower bound for Christofides algorithm's approximation ratio is 3/2. So it is hard to improve this algorithm based on its structure.

7.2 Suggestion for Future

- Since the approximation ratio of the greedy algorithm is not ideal, we can try to optimize this algorithm to get closer to the optimal solution.
- For the TSP approximation algorithm, maybe we can try to find another kinds of tree instead of the minimum spanning tree to create an algorithm with a better approximation ratio than Christofides algorithm.
- We found our study very useful for travel planning, and we can try to build web or mobile applications based on our algorithm implementation in the future.

7.3 What we've learned in this project

Zhou Liu: With such a systematic study in the optimized algorithms under the travelling planning scene, I gained a deep understanding of solving problems in graph by searching, greedy and dynamic programming algorithms. Moreover, I gained valuable intuition and skills to consider how to solve an optimized problem in a graph, which inspired me to solve the TSP problem by dynamic programming method, and will continue inspiring me we I meet with similar problems in my future study and work.

Jifan Xie: This project allows me to apply various algorithms related to graphs in specific life scenarios, as well as applying greedy algorithms, dynamic programming, approximate algorithms to graph related problems. In the Traveling Salesman Problem, I also learned that the approximation ratio of the greedy algorithm is $\Theta(\log N)$ instead of constant. I spent a lot of time trying to understand its proof. In addition, the multiple solutions to the Traveling Salesman Problem show that we can use different algorithms to meet different priorities, such as an optimal solution, shorter running time, less space used, or find a balance among them. All these have inspired me for my future study and work.

Jing Ming: The best thing I learned from this project is to understand and implement the Christofides algorithm with all I learned in the lecture. It is a great chance to review the lecture including Heap, Graph theory, Greedy Algorithm, Shortest Path Algorithm, Minimum Spanning Tree Algorithm and NP-completeness. I believe this project will benefit me in my future study and work where I need to propose an algorithm with faster runtime and lower space.

References

[Bel62] Richard Bellman, *Dynamic programming treatment of the travelling salesman problem*, Journal of the ACM (JACM) **9** (1962), no. 1, 61–63.

[BH15] Judith Brecklinghaus and Stefan Hougardy, *The approximation ratio of the greedy algorithm for the metric traveling salesman problem*, Operations Research Letters **43** (2015), no. 3, 259–261.

[Chr76] Nicos Christofides, *Worst-case analysis of a new heuristic for the travelling salesman problem*, Tech. report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to algorithms*, MIT press, 2022.