EE3980 Algorithms

Hw12 Traveling Salesperson Problem

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing, implementing, and observing 1 algorithm. The goal of the algorithm is to calculate the minimum distance needed for the salesperson to travel though all the cities.

During the analysis process, I will first introduce why and how branch-and-bound method can be applied to this task. Then, I will be using counting method to calculate the time complexities of the algorithm. Finally, I will calculate their space complexity for the total spaces used by the algorithm.

The implementation of the algorithm on C code will find the minimum distances needed and each of their paths for 6 files. Furthermore, I will evaluate their CPU times.

Analysis:

1. Direct thoughts: brute-force and dynamic programming

First, it is clearly that generate whole traveling possibilities should solve the problem. However, since there is (n - 1)! combinations to travel through n cities, this approach costs too much time.

Therefore, we can try dynamic programming instead. After traveling to a city, this city is removed from the list of cities needed to be travel, and it becomes a subproblem and can be applied to dynamic programming. So how is dynamic programing not suitable to solve this problem? The reason is that each choice of selecting the next destination generates multiple choices additionally, and each of them should be compared. Dynamic programming generates an answer after all, however it costs O(2^N x N) on time complexity.

2. Variables set:

N	Number of cities needed to be traveled to.
route[N][N]	Distance matrix of each city.
Path[N]	Path of traveling.
Node {city, cost, *next}	Linked list to store each level's cost.

3. Branch-and-bound method:

Branch-and-bound method is applicable to all state space search methods when the problem meets following two criteria.

i. All children of a search node are generated before any other live node is explored.

ii. Bounding functions are used to help reducing the number of subtrees to be explored.

Traveling salesperson problem meets (i.) since the solution generated by traveling to city A first is explored before any other solutions that traveling to other cities else first. It meets (ii.) because we want to minimize the distance needed for traveling, therefore we can develop a bounding function.

a. Route reduction:

The bounding function, i.e. the cost of each individual choice, is the summary of **distance reduced from current route[N][N] matrix, the distances of this specific path and the previous costs**. Take the matrix below for an example:

	A	B	C	D	E
A	∞	20	30	10	11
B	15	∞	16	4	2
C	3	5	∞	2	4
D	19	16	18	∞	3
E	16	4	7	16	∞

It represents the distances between each city. The first step is to reduce the matrix and get the initial costs. This process is to calculate the minimum costs to travel through all the cities for comparisons when finding the lower bound later.

∞	10	17	0	1	(-10)
12	∞	11	2	0	(-2)
0	3	∞	0	2	(-2)
15	3	12	∞	0	(-3)
11	0	0	12	∞	(-4)
(-1)		(-3)			Total: (-25)

We travel from city A. After this step, we can see that there are 4 cities to travel and we must choose which of them is the next one to travel. Therefore, we must calculate their costs.

For example, if we choose city B, there is no other options for traveling back to city A or city B, therefore all distance on row[A], col[B] and route[B][A] will be ∞.

	A	B	C	D	E
A	∞	∞	∞	∞	∞
B	∞	∞	11	2	0
C	0	∞	∞	0	2
D	15	∞	12	∞	0
E	11	∞	0	12	∞

After that, we calculate the cost using the previous method.

	A	B	C	D	E	
A	∞	∞	∞	∞	∞	
B	∞	∞	11	2	0	
C	0	∞	∞	0	2	
D	15	∞	12	∞	0	
E	11	∞	0	12	∞	
						Total: (-0)

The cost for choosing city B is (0) + route[A][B] (10) + original cost (25), which is 35.

b. Derive a tree and perform Least Cost Search

After calculating all the choices of traveling, we can derive a cost tree as so:

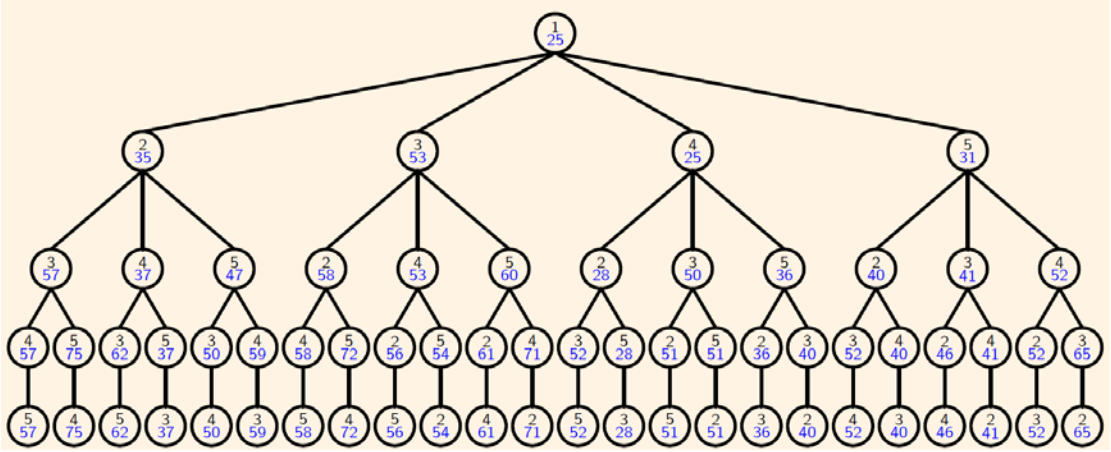


fig from lec72.pdf

By choosing the minimum cost through Least Cost Search, we can find that by traveling through A->D->B->E->C, there is a minimum distance, which is 28 in this case.

4. Cost(route[N][N]):

It is the algorithm to calculate the cost for route[N][N].

```
// Calculate costs for the input route
// Input: int route[N][N]
// Output: int cost
Algorithm Cost(r)
{
    for i := 1 to n do {
        find min in route[i][];
        cost += min;
    }
    reset min;
    for i := 1 to n do {
        find min in route[][i];
        cost += min;
    }
    return cost;
}
```

The total **time complexity is O(N)**. **The space complexity is O(N * N)** for route[N][N] array.

5. **FindAllCost(route[N][N], src, last_cost):**

This algorithm calculates all possible costs after choosing to travel to a city.

```
1.  // Calculate all costs when at city src
2.  // Input: int route[N][N], int src, int last_cost
3.  // Output: Node *costs
4.  Algorithm FindAllCost(r, src, last_cost)
5.  {
6.      costs = NULL;
7.      for i := 1 to n do {
8.          if route[src][i] != INF do {
9.              MASK route[src][], route[][i] and route[i][src];
10.             cost := Cost(route) + route[i][src] + last_cost;
11.             cost->next = costs;
12.             costs = cost;
13.             UNMASK route[src][], route[][i] and route[i][src];
14.         }
15.     }
16.     return costs;
17. }
```

The total **time complexity is O(N * N)** since there's a Cost() in n iterations. The space complexity is O(N * N).

6. **LCSearch(route[N][N], level, src, curr_cost):**

The following algorithm is the main function to solve traveling salesperson function:

```
1.  // Recursive function to solve traveling salesman problem
```

```
2.  // Input: route[N][N], int level, int src, int curr_cost
3.  // Output: distance, path[N]
4.  Algorithm LCSearch(route, level, src, curr_cost)
5.  {
6.      if level == N - 1 do { // finish traveling
7.          path[level] := 0;
8.          if distance > (curr_cost + route[src][0]) do {
9.              distance := curr_cost + route[src][0];
10.         }
11.         return;
12.     }
13.
14.     Child := Child ADD(FindAllCosts(cost, src, curr_cost));
15.
16.     new_route := route;
17.
18.     while child != NULL do {
19.
20.         find min_node in child
21.         Delete min_node in child
22.
23.         if min_node->cost > distance do {
24.             return; // cost exceeded lower bound
25.         }
26.
27.         MASK new_route[src][], new_route[][i] and new_route[i][src];
28.         Cost(route);
29.         path[level] := min_node->city;
30.         LCSearch(new_route, level + 1, min_node->city, min_node->cost);
31.     }
32. }
```

First, we must find all the possible nodes when we are traveling from src to anywhere on line 14. Then, we find the option with the least distance since we want to minimize the traveling distance, on line 18 and 19. If the costs on

current level are all larger than the costs on the previous level, it goes back to the upper level. Finally, when in the last level, it calculates the final distance and record the paths.

The time complexity for this has a great range. If the data is hard to calculate, it needs (N - 1)! calculations, which is very long. However, in regular data, we can easily eliminate lots of solutions by checking the lower bounds and calculate those solutions with lesser distances only.

Therefore, the worst time complexity for this algorithm is O(N! * N^2) for total computation time and FindAllCosts() and the worst space complexity for this is O(N! * N^2). Same as brute-force. The best case happens when there is only 1 traversing path. The time complexity should be O(N^4) for (1 + 2 + … + N – 1) times O(N^2) for FindAllCosts(). The space complexity should also be O(N^4).

7. Time & Space:

	LCSearch *()*	
	Best	**Worst**
Time complexity	**O(N^4)**	**O(N! * N^2)**
Space complexity	**O(N^4)**	**O(N! * N^2)**

Implementation:

1. Workflow:

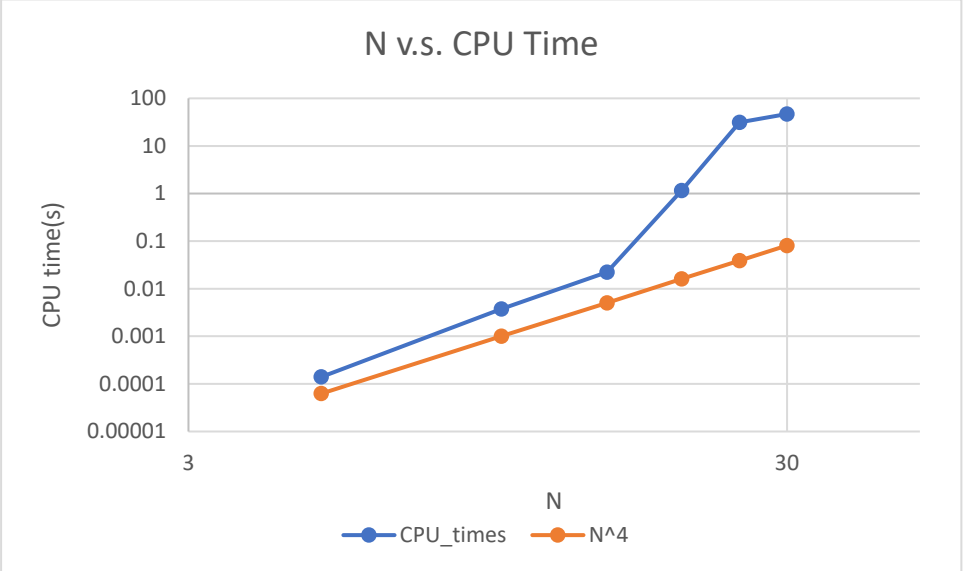
To get measure the CPU time more accurately, the workflow is as followed:

```
1. Algorithm workflow(X, Y)
2. {
3.     T = GetTime();
4.     TravelingSalesPerson();
5.     T = (GetTime() – T);
6. }
```

2. Result:

	N	distance	CPU time(s)
File 1	5	28	0.000139952
File 2	10	84	0.00374889
File 3	15	105	0.0221438
File 4	20	132	1.1564
File 5	25	153	31.2267
File 6	30	166	46.9566

Observation:



We can see that there is no correlation between N and CPU time, other than the fact that when N is larger, CPU time is larger. From the result I get, we can conclude that the graph's complexity got harder from file 4.

The reason for this is because, this algorithm relies on the complexity of the

distance map. If there is a path to travel very clearly, i.e., LCSearch does not need to travel back to find other optimal solutions. Therefore, I conclude that the average time complexity should be O(N^4) for easy cases.

Conclusions:

1. Traveling Salesperson problem can be implemented by branch and bounds.

2. Time and space complexities of *LCSearch()*:

	LCSearch *()*	
	Best	**Worst**
Time complexity	**O(N^4)**	**O(N! * N^2)**
Space complexity	**O(N^4)**	**O(N! * N^2)**

3. The implemented time complexity cannot be told.