HW 11. Traveling Salesperson Problem

EE3980 Algorithms

104061212 馮立俞

May 27, 2018

Introduction

Solve the Traveling Salesperson Problem. As efficiently as possible.

The Traveling Salesperson Problem(TSP) consists of a list of cities and the cost to travel from one city to another. The objective of TSP is to find the least cost path that travel each city exactly once except for the starting city, where his travel starts and ends. The problem can be solved using brute force approach. Yet its $O(n!)$ complexity is somewhat frightening.

Approach

Backtracking

We can build a N-level tree to denote the traveling process. In such tree, the tree edge from level i to level $i+1$ means the salesperson's i^{th} movement. To reach all leaf nodes, we can utilize DFS or BFS, yet $O(N!)$ operations are required if we perform it naively. However, using Backtracking, we can save some effort by calculating the lower bound of a path. If the bound is higher than current shortest path, we'll simply skip trying the path.

Calculating Lower Bound

Given a cost matrix, it can reduced as following to calculate the lower bound.

Note that $c_{i,j}$ is the cost from vertex i to vertex j Thus, if $c_{i,k} = \min_{j=1}^{n} c_{i,j}$, then $c_{i,k}$ is the minimum cost leaving vertex i. And, if $c_{k,j} = \min_{i=1}^{n} c_{i,j}$, then $c_{k,j}$ is the minimum cost entering vertex j.

As we reach a not-visited node. We first add the previous edge $c_{i,j}$ to our current bound. Then, mark row i and column j and $c_{j,i}$ as infinity as they're no longer needed for travel. Finally, reduce each row or each column by $c_{i,k}$ or $c_{k,j}$ and add them to current bound to estimate the new bound.

As we include calculating bound into naive DFS, the improved DFS has following structure.

```
Algorithm TSP(V, level, matrix, bound){
  if(Travel all cities){
    if(cost < minCost)
      record new minCost and path
  }
  else{
    Visit(V);
    for ( all not visited cities){
      compute bound( matrix, bound)
      if(bound <= minCost)
        TSP(new city, level +1, matrix,bound);
      restore matrix //visit complete
    }
  }
}
```

It's a recursion function with Maximum depth N, so the space complexity is $O(N)$.

Validate the correctness

We can use brute force approach to verify that the solution found from backtracking is correct. To generate all permutation of N elements, Algorithm from Narayana Pandita can be utilized. The procedures are:

---

1. Find the largest index j such that $A[j] < A[j+1]$. If no such index exists, the permutation is the last permutation.

2. Find the largest index k such that $A[j] < A[k]$.

3. Swap $A[j]$ with $A[k]$.

4. Reverse the sequence from $A[j+1]$ up to and including the last element $A[N-1]$.

After a permutation is formed. We can calculate the cost according to the path.

Results and Analysis

The execution result is as follows:

```
t1(5)
Total Cost = 28
0.000u 0.001s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
t2(10)
Total Cost = 84
0.004u 0.000s 0:00.00 0.0%      0+0k 0+0io 0pf+0w
t3(15)
Total Cost = 105
0.042u 0.000s 0:00.04 100.0%    0+0k 8+0io 0pf+0w
t4(20)
Total Cost = 132
1.241u 0.017s 0:01.26 99.2%     0+0k 0+0io 0pf+0w
t5(25)
Total Cost = 153
26.001u 0.241s 0:26.25 99.9%    0+0k 0+0io 0pf+0w
t6(30)
Total Cost = 166
201.261u 1.443s 3:22.73 99.9%   0+0k 8+0io 0pf+0w
```

Then, we can plot the above result into a chart.

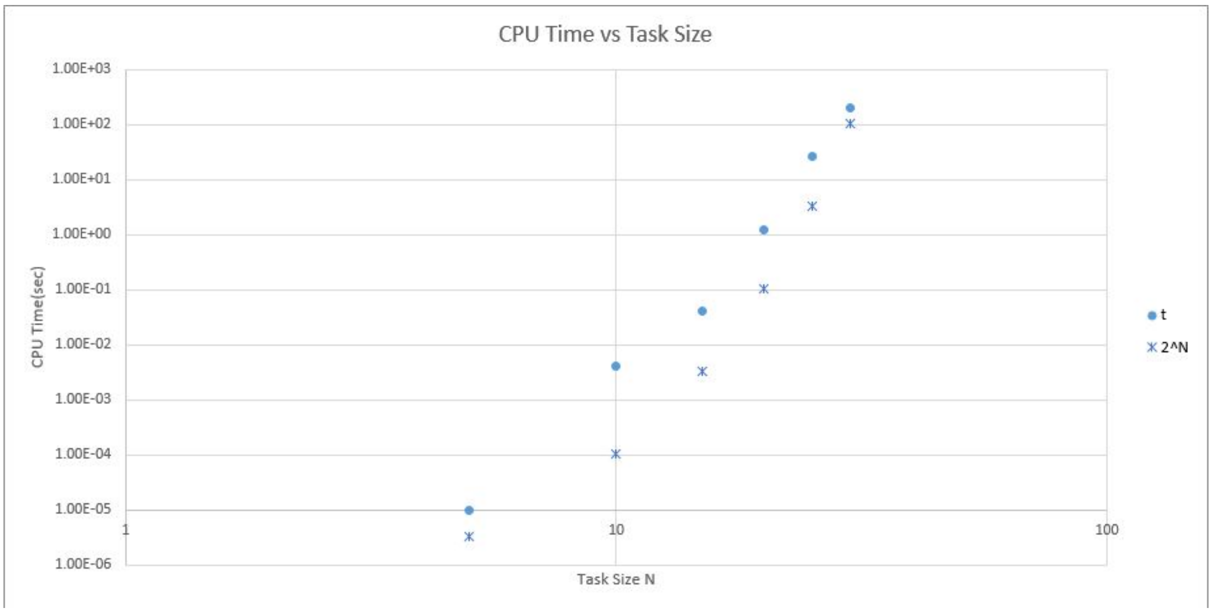

Figure 1: CPU Time needed to find optimal solution v.s task size, $lg - lg$ scale

We can see from the figure that the complexity is comparable with exponential. Yet actually our backtracking effort has reduce the coefficient quite a lot. As comparison, a $N = 15$ task can have Pandita's algorithm work for days. So our validation stops at file t2 ($N = 10$).

Observations and Conclusion

1 Backtracking can seemingly be incorporated into Pandita's algorithm. This has a great benefit – the recursion can be replaced by iteration. I didn't manage to work it out before deadline though.

2 Coefficient in time complexity matters. Reducing coefficient is where backtracking can be used.