Project. Two-day Travelling Problem

EE3980 Algorithms

104061212 馮立俞

June 15, 2018

Introduction

Given n cities and the distance between each pair of cities, a salesperson is planning a two-day trip to visit all cities once and return to the starting city. It is also determined to stay in a particular city at the end of the first day. In addition, in order to balance the travelling distance of the two days, it is desired to have travelling distance of both day as close to each other as possible. That is, given n cities, $c_1, c_2, ..., c_n$, and $A[1{:}n,\ 1{:}n]$ be the distance matrix, find two paths, $P_1 = <p_1^1, p_1^2, ..., p_1^k>$, and $P_2 = < p_2^1, p_2^2, ..., p_2^h>$ such that
1. $p_1^1 = p_2^h, p_1^k = p_2^1$ and $P_1 \cup P_2$ is a Hamiltonian cycle.
2.$| \sum_{i=1}^{k-1} A[p_1^i, p_1^{i+1}] - \sum_{j=1}^{h-1} A[p_2^j, p_1^{j+2}] |$ is minimum

However, efficiency is a major concern for this project, one should speed up the execution as long as possible.

Approach

It can be easily identified that the requirements are similar to that of Travelling Salesperson Problem(TSP), in which a Hamitonian cycle with least travelling cost is to be solved. Since TSP is already solved in hw11, the next thing we care is how to speed up the performance and split the travel into two days.

Recap: Solving TSP Using Branch and Bound

We can build a N-level tree to denote the TSP 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 Branch and Bound, 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 / each column by $c_{i,k}$ / $c_{k,j}$ and add $c_{i,k}$ / $c_{k,j}$ to current bound to estimate the new bound.

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

```
Algorithm DFS(V, level, matrix, bound){
  if(Having traveled 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)
        DFS(new city, level +1, matrix,bound);
      restore matrix //visit complete
    }
  }
}
```

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

Least Cost Method

In a branch and bound algorithm, how one traverse the nodes are crucial to the efficiency. It's known that both Depth First Search (DFS) and Breadth First Search (BFS) can guarantee to travel all reachable nodes in a graph. Therefore either we adopt DFS or BFS will not affect the correctness of answer.

In general, DFS outperform BFS since DFS reaches the leaf node directly and is capable of updating the current minimum earlier, which can then be used to compare with bounds, thus skips some nodes and speeds up the process. BFS, on the contrary, traverses all the nodes in n-1 levels until it reaches a leaf node and update the current minimum.

As a result, a naive BFS doesn't utilize the calculated bound much, so the efficiency is basically no better than brute-force approach.

In hw11, I adopted DFS to solve TSP; yet it took around 200 seconds of CPU time to solve the largest given task. To further speed up, I plan to utilize Least Cost method.

A Least Cost traversal strategy calculates the cost of traversal for all nodes in the next level. Then picks the node with least cost to traverse. In this case, least cost is equivalent to lowest lower bound. Since if we follow the nodes with low bounds, we are more likely to find a smaller new current minimum, and speed up the execution in this way. Least Cost method can be viewed as improved DFS, since it evaluate the cost of all children and prioritize them instead of blindly choose one and traverse.

The following is the pseudocode of LC solving TSP

```
Algorithm LC(V, level, matrix, bound){
  if(Having traveled all cities){
    if(cost < minCost)
      record new minCost and path
  }
  else{
    for ( all not-visited cities){
        compute bound
    }

    Sort(boundList); // Sort the bounds in non-decreasing order

    for ( all not-visited cities){
      //visit nodes according boundList
      if(bound <= minCost)
        LC(new city, level +1, matrix,bound);
    }
  }
}
```

Splitting tour into two parts

After solving TSP, we know the tour with least travel cost and the cost of such travel. We can split the tour into two parts with minimal difference in travel cost.

The method is quite simple, just find which split point is closer to half of the minimum cost. For instance, the following figure illustrates how to split a given tour. The rectangles represent a tour from one city to another, the length of rectangles are proportion to the cost of travel.

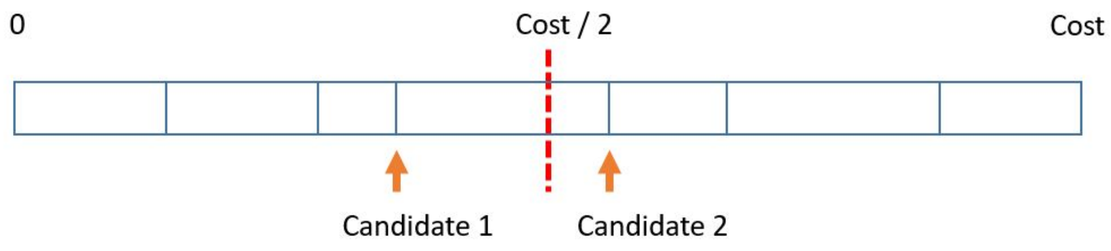

Figure 1: splitting a tour

To split the tour into two parts, there are two candidates for split point. It's obvious that candidate 2 is preferable since it's closer to Cost / 2.

The same method can be applied to splitting into more than two parts, just find the split point closest to k * Cost / (parts), k = 1,2,...(parts-1).

Results and Analysis

The execution result of *t1.dat* is as follows:

```
$time ./a.out < t1.dat
Two-day travelling plan:
 Day 1:
  Ann Arbor -> Iowa City
  Iowa City -> Manhattan
    Distance: 16
 Day 2:
  Manhattan -> Charlottesville
  Charlottesville -> Corvallis
  Corvallis -> Ann Arbor
    Distance: 12
 Total distance: 28
Day 1 and day 2 difference: 4
0.000u 0.003s 0:00.01 0.0%      0+0k 0+0io 0pf+0w
```

Then, we can compare with the result of hw11 to see how much we improved by adopting Least Cost over DFS. The below table shows the CPU time used by each algorithm on different task sizes.

Table 1: CPU time taken for two approaches. (in seconds)

N	DFS	LC
5	0	0
10	0.004	0
15	0.042	0.001
20	1.241	0.179
25	26.001	2.096
30	201.261	0.721

For $N = 30$, We can find an interesting fact that larger task size doesn't guarantee longer execution time for LC. This phenomenon could rise from our luckily finding a small current minimum early during execution, and skipped many branches thanks to it. This is more likely to happen for LC since LC skillfully picks the node with lowest bound.

Observations and Conclusion

1. Least Cost brings about noticeable speed up for branch and bound algorithms compared to DFS. Because if we want to skip more nodes, finding a low current minimum early is desired.

2. Larger task size doesn't guarantee longer execution time for LC.