10802EE 398000 Algorithms

Homework 12. Travelling Salesperson Problem

● Introduction

In this homework, I will write a C program to solve the Travelling Salesperson Problem (TSP). The Travelling Salesperson Problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is **the shortest possible route that visits each city and returns to the origin city**?" This problem can be stated as below:

Def: Let G = (V, E) be a directed graph, with |V| = n and c_{ij} be the cost of edge $\langle i, j\rangle \in E$, $c_{ij} = \infty$ if $\langle i, j\rangle \notin E$. Without loss of generality, we can assume every tour start from vertex 1. So, the solution space is S = $\{(1, \pi, 1) \mid \pi$ is a permutation of $(2, 3, \cdots, n)\}$. For any solutions, $(1, i_1, i_2, \cdots, i_{n-1}, 1) \in S$, $\langle i_j, i_{j+1}\rangle \in E$, $0 \le j \le n-1$ and $i_0 = i_n = 1$. The objective is to **find a path with the minimum cost**.
Definition 1 the definition of Traveling Salesperson Problem

This problem can be solved by using **Backtracking**. However, it is quite time-consuming (O(n!)). Thus, I will use **Branch and Bound** in this homework. **Branch and Bound** method is applicable to all state space search methods. **Bounding functions** are used to help reducing the number of subtrees to be explored. Thus, it can reduce the CPU time.

How to find the bound in TSP problem? **First, we have to get the reduced cost matrix from the input cost matrix.** There are three steps in **the reduced matrix technique**: 1. Find the minimum costs of each row and reduce the cost of each row by them simultaneously. 2. Find the minimum

costs of each column and reduce the cost of each column by them simultaneously. 3. Sum them up. The result is the cost of the root (starting node). More precisely, it's the lower bound of the root. The reduced cost matrix will then become the cost matrix of the root. After doing this, **if we pick an edge ⟨i, j⟩**, we can use the following steps to get the cost matrix and the corresponding cost:

Step: Suppose an edge ⟨i, j⟩ is selected, the cost of the path is increased by $c_{i,j}$.

- All other edges ⟨i, k⟩, k != j cannot be selected. Thus, set $c_{i,k} = \infty$, $1 \le k \le n$. (Row i).
- All edges ⟨k, j⟩, k != i, cannot be selected. Thus, set $c_{k,j} = \infty$, $1 \le k \le n$. (Column j).
- The edge ⟨j, 1⟩ cannot be selected (unless j is the only vertex not selected). Thus, set $c_{j,1} = \infty$.
- Perform **reduced matrix technique** to the resulting matrix to get the lower bound, r.
- Then the lower bound of path cost of selecting edge ⟨i,j⟩ is $R + c_{i,j} + r$, where R is the lower bound before selecting edge ⟨i, j⟩.

Definition 2 the steps to get the cost matrix and the corresponding cost.

We can use these steps iteratively to get the lower bound of all nodes. After traveling to the leaf node (that is, all vertices have already been travelled), we can update the **upper bound** if the cost of this node is less than the current upper bound. With the upper bound, we can avoid travelling some unnecessary nodes (nodes with the cost greater than the upper bound).

There are three ways to choose the next node to visit: **Last-In-First-Out (DFS), First-In-First-Out (BFS), and Least-Cost (LC)**. First-In-First-Out is not in my consideration because we can update the upper bound only after we go to the leaf. The upper bound should be updated as soon as

possible (It is infinite at the beginning). The sooner we get the upper bound, the more unnecessary nodes we can avoid travelling. Thus, I will only compare the CPU time between DFS and LC strategies and show results in **Results** part.

There are six sets of data used to test my program: t1.dat, t2.dat, t3.dat, t4.dat, t5.dat, and t6.dat. The first line of each file is the number of cities the salesperson needs to travel, followed by the names of the cities. After that, a 2-dimensional matrix is given that depicts the distance between different cities. The starting and ending city of the travelling plan is the first city on the list.

I will describe my algorithm, analyze the complexities of my program in **Approach** part. Besides, I will show results of all input files and state my conclusions in **Results** part.

● Approach

I use the Least-Cost strategy in the code I submit. That is, I pick the node with the least cost of all live nodes. The term “live node” indicates that it hasn’t been visited but are adjacent to the node that has already been visited. The structure of the node is shown in the next page.

The index is the vertex number stored in this node. The visit[V] is an array to check whether vertex is visited or not. The p[V] will store the path from the root to this node. The count will store how many vertices have already been visited. If count is equal to the number of vertex, we have found the leaf and are able to update the upper bound. The table[V][V] will store the cost matrix of this node.

```
typedef struct _Node {   // the structure for LC-search
    int index;           // the vertex stored in this node
    int* visit;          // visit[i] indicates whether i-th node is visited
    int* p;              // the current path

    int cost;            // the cost of this node
    int count;           // the number of vertices which have been visited
    int** table;         // the corresponding cost matrix
} Node;
```

Algorithm 1 structure of Node.

The most important function in the code is LC_search. Before calling LC_search, we have to call LC_call. The pseudocode of LC_call is shown below:

```
// get the minimum distance and its corresponding route
// Input: the input graph G (the input cost table)
// Output: ansPath, upperBound (the cost of ansPath) , final (the last vertex of the path).
1 Algorithm LC_call(G, ansPath, upperBound)
2{
3       initialize visit array, ansPath array, and initialPath (defaults = -1);
4       copy the input graph G to tempTable[V][V];
5       set the upper bound to infinity;
6       cost = get_cost(tempTable);    // get the reduced cost table and the cost
7       make the root node;
8       LC_search(G, root, initialPath);    // start to search
9       free the memory of visit and initialPath;
10 }
```

Algorithm 2 LC_call.

The main purpose of LC_call is to make the root node. First, I copy the graph to tempTable because the graph cannot be modified. Second, I get the cost using get_cost which is shown in the next page. Third, I use the initialized visit array, initialPath array, cost, and tempTable to generate the root node. After making the root node, we can start to do the LC_search.

```
// get the cost and update the matrix
// Input: the cost table c
// Output: the reduced cost table and the cost using the reduced matrix technique
1 Algorithm get_cost(c)
2{
3       cost := 0;
4       // get the cost of rows and the row-reduced matrix;
5       for i := 1 to V (the number of total vertices) do {
6               find the minimum of the i-th row;
7               reduce the i-th row with minimum;
8               cost := cost + minimum;
9       }
10      // get the cost of columns and the row-/column- reduced matrix
11      for i := 1 to V (the number of total vertices) do {
12              find the minimum of the i-th column;
13              reduce the i-th column with minimum;
14              cost := cost + minimum;
15      }
16      return cost;
17 }
```

Algorithm 3 get_cost.

LC_search is shown in the next page. It is the implementation of **Definition 2**. First, I set the starting point (the root). Then, the program will enter the while loop until the heap is empty. In the loop, I find all adjacent vertices of the current visiting node (v), get the corresponding cost. If the cost is less than upperBound, then I make the node and put it into the min heap.

If we have already travelled to the leaf node, I check whether the upperBound needs to be updated or not. If it needs to be updated, then I also update the ansPath.

I pick which node to visit in Line 29-33. First, remove the node with the minimum cost from the heap. Then, check whether the cost is less than upperBound. If so, I will visit that node.

```
// do the Least-Cost Search
// Input: the input graph G, vertex (it's actually the root node), p (the initial path)
// Output: ansPath, upperBound (the cost of ansPath), final (the last vertex of the path).
1 Algorithm LC_search (G, vertex, p)
2 {
3       u = vertex;       // set the starting point
4       v = u->vertex;
5       set (u->visit)[v] to true;
6       p[v] = 0;
7       repeat {
8             if (we have already travelled to the leaf and upperBound > the cost of the leaf) then {
9                   upperBound := the cost of the leaf;
10                  copy the path of the leaf node to ansPath;
11                  set final to the index of the leaf;
12            }
13            for w := 1 to V (the number of total vertices) do {
14                  if (G[v][w] != infinity and (u->visit)[w] == 0) then {
15                        copy the graph from (u->table) to tempTable[V][V];
16                        c_cost := tempTable[v][w];
17                        set tempTable[v][k] and tempTable[k][w] to infinity;
19                        set the edge from the root to u to infinity;
20                        cost = c_cost + u->cost + get_cost(tempTable);
21                        if (cost < upperBound) then {
22                              make the newNode;
23                              (newNode->visit)[newNode->index] := 1;
24                              (newNode->p)[w] = u->index;
25                              insert the newNode to the min heap;
26                        }
27            }
28            judge := 1;
29            while (the heap is not empty and judge == 1) {
30                  u = remove the node in the min heap;
31                  if (upperBound >= u->cost) then
32                        v := u->index; judge := 0;
33            }
34      } until (the heap is empty);
35 }
```

Algorithm 4 LC_search.

The time complexity of LC_search is O(V!) (V is the number of vertices), which is still same as using the backtracking. The space complexity is $O(V^2 * V!)$, where V^2 is the space complexity of the table, and V! is the maximum possible number of nodes. However, the actual CPU time and the actual memory used will not be that much. The reason is that using **Branch and Bound** can avoid visiting some unnecessary nodes.

- **Results**

	t1.txt	t2.txt	t3.txt	t4.txt	t5.txt	t6.txt
Total travelling distance	28	84	105	132	153	166
CPU Time Using LC	3.40939e-05 seconds	2.14505e-03 seconds	6.59204e-03 seconds	3.52329e-01 seconds	8.35542e+00 seconds	2.54702e+00 seconds
Nodes be visited	4	44	51	1730	20805	3208
CPU Time Using DFS	3.00407e-05 seconds	9.00388e-03 seconds	8.73799e-02 seconds	2.08725e+00 seconds	4.60937e+01 seconds	3.60728e+02 seconds
Nodes be visited	11	450	1549	18008	177435	797885

Table 1 results table.

Obviously, using LC is faster than DFS almost in every test case. The only exception is in t1.txt. Although the LC has the less nodes be visited, the CPU time is larger. The reason is that selecting the next live node is more consuming in LC approach. I use the min heap in the program which is O(lg n). There must be some faster ways to choose the next node to visit. Also, there must be some methods to make less copy in the program. Copying the table is quite time-consuming. These are some ways I can make this program better in the future.