EE3980 Algorithms

Homework 12. Travelling Salesperson Problem

By 105061212 王家駿

1. Introduction

In this homework, we solve for the travelling salesperson problem by using the branch and bound algorithm. The problem is to start from the first city, visit every city exactly once, and finally go back to the first city. Then we accumulate the distance of between the two cities along the path. Our goal is to find a path that minimize the total distance. By the branch and bound algorithm with least-cost-first search, we could solve the problem more efficiently by eliminating the possible path to some extent.

2. Implementation

2.1. Data Structure and method of algorithm

One way to solve the problem is to find all the possibility of the path. For the input size of N cities, this method would lead to $(\mathrm{N}-1)*(\mathrm{N}-2)*\cdots*2*1=(\mathrm{N}-1)!$ different paths, which is large and not necessary.

To improve the efficiency, we use the branch and bound method to reduce the possibility we have to try. At every step we try, we calculate the least bound of the distance (cost), and choose the step of minimum least bound for the next try. Since at each step, we always take the path with least cost, we could get the minimum distance when all city was visited, because the rest paths take the longer distance than the selected one. By this method, we could eliminate the possibilities of other paths which takes the larger lower bound earlier. The detail would be described in the TSP function.

To store the information, we use a structure *step* to represent each step of traversal. The structure contains a two-dimensional array *arr* to store the distances between every two cities after cost reduction, a one-dimensional array *path_arr* to store the order of cities which have been visited, an integer *path_size* to store the number of visited cities, and an integer *cost* for the lower bound so far.

Every time when we traverse for a new possible step of path, we would use a new element of step structure, which inherits the value of the previous step, and revise it with the constraints of the problem.

2.2. readInput

In the readInput function, we read the input data and store them in suitable data structure.

At first, we have to read the names of N cities, so we label the cities from 1 to N, and store the names as strings in an array. However, since some names contain spaces, if we read them by strings, we could only get the characters before the first space. Thus, we read the names by characters. Only until we read the newline character do we finish getting the name of a city.

Then, we read the distance data to a two-dimensional array. The array value A[i][j] means the distance from the ith city to the jth city. If the value is zero, there's no path between the cities, and we set the value to the maximum value.

2.3. copyCostArray

There are some functions to implement on revising the data in *step* structure. One is the copyCostArray function. In this function, we copy the *arr* array of the input structure to the output structure.

2.4. copyPathArray

This function copies the *path_arr* array of the input structure to the output structure, and also inherits the value of *path_size*, the size of this array.

2.5. initNewStep

This function initializes a new *step* structure when we are traversing a new step. We allocate dynamic memories for the *arr* array and *path_arr* array, and also set the *cost* value to zero.

2.6. costReduce

```
Algorithm costReduce(A, cost)           // cost reduction on distance array
{
    // reduce by row
    for i := 1 to N do {
        mini := min value from A[i][1] to A[i][N]
        for j := 1 to N do {
            A[i][j] := A[i][j] - mini;
        }
        cost := cost + mini;            // add to cost
    }

    // reduce by column
    for i := 1 to N do {
        mini := min value from A[1][i] to A[N][i]
        for j := 1 to N do {
            A[j][i] := A[j][i] - mini;
        }
        cost := cost + mini;            // add to cost
    }
}
```

In the *arr* array in the structure, it contains the distance data between every two cities. For the ith row of the array, the value A[i][j] stands for the distance from i to j. Now we set minimum value in the ith row as *mini*, when we try to leave the ith city, the distance of path we take must be greater than *mini*, so the lower bound of leaving the ith city is *mini*. Then we record this value and reduce the above values by *mini*. After we applied this method on every row, we could get the lower bound of leaving every city.

Similarly, we could do the same method on columns, and we would get the lower bound of entering every city. After doing the reductions, we sum up all the subtraction, the value stands for the minimum value of entering and leaving every city, which is the lower bound of total distance.

2.7. adjustArray

After selecting a path from the ith city to the jth city, we could revise the *arr* array with some constraints by this function.

(1) Since we had left the ith city, we couldn’t leave the city again, so we could set the distance of leaving the ith city to maximum value. That is, A[i][k] = MAX for k from 1 to N.

(2) Since we had entered the jth city, we couldn’t enter the city again, so we could set the distance of entering the jth city to maximum value. That is, A[k][j] = MAX for k from 1 to N.

(3) Since we couldn’t go back to the first city unless all the city had been visited, we could set the distance from the jth city to the first city to maximum value. That is, A[j][1] = MAX if there is city not been visited yet.

By applying these constraints, we could simplify the *arr* array to some extent, which could reduce the branch of traversal. In addition, we also implement costReduce function after the array have been adjusted to update the lower bound.

2.8. TSP (travelling salesperson problem) algorithm

```
Algorithm TSP()
{
    // initialize the first element
    s := initNewStep();
    copyCostArray(data, s);                 // from data to s
    costReduce();
    push s into the list

    while true do {
        t := min element in the list

        // check if it is end
        if all the cities had been visited then {
            print out result and return
        }

        // for all child, find out cost
        for i := 1 to N do {                // for all child
            // feasible path
            if there is a path from end point of t to i then {
                s := initNewStep();         // initialize new step
                copyCostArray(t, s);        // from t to s
                copyPathArray(t, s);        // from t to s
                put i at the end of path array
                add the edge distance to cost
                adjustArray();              // adjust array by constraints
                push s into the list
            }
        }

        remove t from the list
    }
}
```

In this function, we implement the branch and bound method with least-cost-first search to solve the travelling salesperson problem.

We use a list to store the undone steps. Every time when we go for the next step, we search for the element t in the list with minimum lower bound (least-cost-first search). If the step t had visited all the cities, the algorithm ends with finding out the optimal solution.

Otherwise, we would try the possible path based on t. We create new steps from the end point of t, and calculate the lower bound of the new steps by adjustArray function. Then we put these steps into the list for the next iteration, and also remove t from the list since it had done and replaced by the new steps.

To estimate the time complexity, in the worst case, we would need to traverse through all possibility, which is (N - 1)! times as mentioned above. However, the actual number of times of traversal depends on the arrangement and value of input distance array. We could only ensure that it takes no more steps than (usually far less than) the brute-force approach.

3. Executing results

We run the testing data from t1.dat to t6.dat with different input data size, and we also record the average CPU time used additionally.

Number of cities (*N*)	**Minimum distance**	**Time used**
5	28	0.123 ms
10	84	2.305 ms
15	105	8.661 ms
20	132	474.2 ms
25	153	35.13 s
30	166	4.914 s

4. Result analysis and conclusion

From the executing time, we could observe that the CPU time used increases when the input data size grows up in most cases. However, the fifth testing data spends much time than others. It matches out estimation that the actual time used is affected by the arrangement of input data greatly. Moreover, the traversing method would also influence the actual executing time. There might be other ways faster than the least-cost-first method for this testing data.

In our implementation, we use an array to represent the list in the TSP function. Every time we take the minimum element, we need to iterate through the array, which contributes the time complexity of O(L), where L is the size of the list. Yet, we could use other type of data structure. If we use the minheap to represent the list, we could get the minimum element directly by just fetching the first element. However, when we add a new element to the list, we have to rearrange the heap to maintain the minheap, and the step has the time complexity of O(L*log(L)). Thus, the other type of data structure might or might not the previous one, depending on the arrangement of input data.

At the worst case, there would be (N-1)! steps for traversal. Thus, since we record the value of every step, we have to allocate at most (N-1)! spaces for the *step* structure, which is a large number despite the fact that it doesn’t need such more spaces at most cases. The drawback could make it hard to use with large input data size.