BỘ GIÁO DỤC VÀ ĐẠO TẠO

TRƯỜNG ĐẠI HỌC BÁCH KHOA TP. HỒ CHÍ MINH

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

KHOA KHOA HỌC VÀ ỨNG DỤNG

REPORT

Discrete Math

<u>TOPIC:</u>

Travelling Sale Man Path Calculation

PROFESSOR: Trần Tuấn Anh

Student Name: Hồ Quốc Huy

ID: 2352379

CLASS: CC06

SUBMISSION DATE: 09/06/2024

I. Problem:

- The travelling salesman problem, also known as the travelling salesperson problem (TSP), 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 exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.

- The travelling purchaser problem and the vehicle routing problem are both generalizations of TSP.

- In the theory of computational complexity, the decision version of the TSP (where given a length L, the task is to decide whether the graph has a tour whose length is at most L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (but no more than exponentially) with the number of cities.

- The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, many heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely, and even problems with millions of cities can be approximated within a small fraction of 1%.[1]

- The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources want to minimize the time spent moving the telescope between the sources; in such problems, the TSP can be embedded inside an optimal control problem. In many applications, additional constraints such as limited resources or time windows may be imposed.

(https://en.wikipedia.org/wiki/Travelling_salesman_problem)

II. THE BRANCH AND BOUND METHOD

- The branch and bound method provides a correct solution for the given problem. It is suitable for use when the number of vertices of graphs does not exceed 60, which is sufficient when it comes to problems such as package delivery.

- This method is based on the idea that the sets are divided into two disjoint subsets at every step of the process - branching. Furthermore, one subset contains the path between the two selected towns and the other subset does not. For each of these subsets the lower restriction (bound) is calculated for the duration or for travel expenses. Finally, the subset which exceeds the estimated lower bound is eliminated.

- The procedure of branching is represented by a tree where the top is marked by the branching points of the set of solutions, and the edges mark the path between the two contiguous vertices in the graph which are used to model the problem and inside which the shortest Hamilton cycle is pursued.

- The knots in the branching tree are labelled with the “etiquettes” which represent the lowest binding point for the objective function.

- The steps of the algorithm for the Travelling Salesman Problem using the bound and branch method are as follows:

  Step 1: Drawing a table
  A table of distance between the given vertices is drawn. The distance between the vertices i and j are marked with $d_{ij}d_{ij}$. If the two vertices are not contiguous they are marked with $d_{ij} = \infty$. The $d_{ii} = \infty$ mark to signify and prevent the algorithm to choose the path $i \to i$ is also appointed.

  Step 2: The reduction of the table
  In every row of the table, the smallest element is located and marked with $r_i r_i$:

  $$r_i = min_j d_{ij}\,, j = 1, \dots, n$$

  The smallest elements in columns are calculated using the following formula:

  $$c_j = min_i\left(d_{ij} - r_i\right).$$

  The reduction of the table is calculated with the formula:

  $$d'_{ij} = d_{ij} - r_i - c_j$$

  Step 3: The calculation of the lower boundary
  The lower boundary of the duration of the travel is calculated

$$b = \sum_{i=1}^{n} r_i + \sum_{j=1}^{n} c_j$$

and assigned, which bounds to the knots of the branching tree as etiquette.

Step 3': The calculation of the lower boundary

b is equalized with the etiquette of the knot in which the branching is performed.

Step 4: Branching
The top from which the branching will start is chosen. All the addresses in the reduced table are found, the (i,j) paths, for which we can state that $d'_{ij} = 0$. For each of these, the "penalty" for not utilizing the trajectory (i,j) is calculated. The penalty is calculated with the formula:

$$\pi_{ij} = min_j d'_{ij} + min_i d'_{ij}$$

The branching is initiated with the trajectory which is assigned the maximum "penalty".

$$\pi = max_{ij} \pi_{ij}$$

(p,q) are used to mark the route with the maximum penalty.

Step 5: Etiquette calculating
The etiquette of the knot contiguous to the edge which has a "weighting" (p,q) is calculated. In the reduction table $d_{qp} = \infty$ is assigned to signify the restriction towards the traveller, preventing him from returning from town q to town p. Furthermore, all the possibilities of closing the cycle prior to passing all the vertices of the graph need to be blocked. The next step is to remove the p line and q column from that table and repeat steps 2 and 3. The sum of the reduced elements is marked with σ.

Step 6: Drawing the branching tree
In the branching tree, we assign the b etiquette to the knot from which the branching started. The edges that exit this knot are assigned with "weightings" (p,q) and non(p,q). The contiguous knot to the non(p,q) edge is assigned with the $b + \pi b + \pi$ etiquette. The contiguous knot to the (p,q) edge is on the other hand assigned with the $b + \sigma b + \sigma$ etiquette.

Step 7: Locating the knot with the smallest etiquette
One should try to pinpoint the knot with the smallest etiquette in the branching tree and repeat the procedure – steps 1-6 are the same with the exception of step 3 which is replaced with step 3'.

The algorithm is completed when the table contains only routes which, if not used, lead to the solution that is the trajectory of infinite duration.

III. CODE

```
int res_path[50];
bool passed[20];
int final_des;

void copyToFinal(int current_path[], int n) {
    for (int i = 0; i < n; i++){
        res_path[i] = current_path[i];
        res_path[n] = current_path[0];
    }
}

int fst_Min(int G[20][20], int i, int n) {
    int min = 999999;
    //int min=INT_MAX;
    int a=0;
    for (int k = 0; k < n; k++){
        if (G[i][k] < min && i != k)
            min = G[i][k];
    }
            return min;
}

int sec_Min(int G[20][20], int i, int n) {
    int first = 99999, second = 99999;
    int a=0;
    bool temp = true;
    int b=2;
    for (int j = 0; j < n; j++) {
        if (i == j)
            continue;
        switch(G[i][j] <= first) {
            case true:
                second = first;
                first = G[i][j];
                break;
            case false:
                if (G[i][j] <= second && G[i][j] != first)
                    second = G[i][j];
                break;
        }
    }
    return second;
}
```

```
void TSPRec(int G[20][20], int current_bound, int current_weighted, int frequent,
int current_path[], int n) {
    if (frequent == n) {
        if (G[current_path[frequent - 1]][current_path[0]] != 0) {
            int curr_res = current_weighted + G[current_path[frequent -
1]][current_path[0]];
            if (curr_res < final_des) {
                copyToFinal(current_path, n);
                final_des = curr_res;
            }
        }
        return;
    }
    int temp=0;
    char temp1;
    //for(int i=0;i<n;i++)
    //cout<<current_path[i]<<" ";
    //cout<<endl;
    for (int i = 0; i < n; i++) {
        if (G[current_path[frequent - 1]][i] != 0 && !passed[i]) {
            int temp = current_bound;
            current_weighted += G[current_path[frequent - 1]][i];

            switch(frequent == 1) {
                case true:
                    current_bound -= (fst_Min(G, current_path[frequent - 1], n) +
fst_Min(G, i, n)) / 2;
                    break;
                case false:
                    current_bound -= (sec_Min(G, current_path[frequent - 1], n) +
fst_Min(G, i, n)) / 2;
                    break;
            }

            if (current_bound + current_weighted < final_des) {
                current_path[frequent] = i;
                passed[i] = true;
                TSPRec(G, current_bound, current_weighted, frequent + 1,
current_path, n);
            }

            current_weighted -= G[current_path[frequent - 1]][i];
            current_bound = temp;
            for (int j = 0; j < n; j++)
                passed[j] = false;
            for (int j = 0; j <= frequent - 1; j++)
```

```
                passed[current_path[j]] = true;
            }
        }
    bool yes = true;
}

string Traveling(int G[20][20], int n, char start) {
    final_des = 999999;
    // int G[20][20]
    // int n
    int temp=0;
    char temp1;


    int current_path[20 + 1];
    for (int i = 0; i < 20 + 1; i++) {
        current_path[i] = -1;
    }
    //*************** */
    for (int i = 0; i < 20; i++) {
        passed[i] = false;
    }
    //*************** */
    int current_bound = 0;
    for (int i = 0; i < n; i++)
        current_bound += (fst_Min(G, i, n) + sec_Min(G, i, n));
    current_bound = (current_bound & 1) ? current_bound / 2 + 1 : current_bound /
2;

    int start_index = start - 'A';
    int tempro = 0;
    passed[start_index] = true;
    current_path[0] = start_index;

    TSPRec(G, current_bound, 0, 1, current_path, n);
    temp+=0;
    string result = "";
    for (int i = 0; i <= n; i++) {
        if (i > 0) result += " ";
        result += (char)('A' + res_path[i]);
    }
    return result;
}
```

1. **“copyToFinal” Function:**
    - This function copies the current path to the final path array.
    - It takes two parameters: curr_path[], the current path array, and n, the size of the path.
    - Inside the function, it iterates through each element of curr_path[] and copies it to final_path[].
    - Finally, it ensures that the last element of final_path[] is set to the first element of curr_path[].

2. **“firstMin” Function**:
    - This function calculates the minimum value in a row of the adjacency matrix G.
    - It takes three parameters: G, the adjacency matrix, i, the current row, and n, the size of the matrix.
    - Inside the function, it iterates through each element of the i-th row of G and finds the minimum value.
    - It skips the diagonal elements (where i == k).
    - The minimum value found is returned.

3. **“secondMin” Function:**
    - This function calculates the second minimum value in a row of the adjacency matrix G.
    - It takes three parameters: G, the adjacency matrix, i, the current row, and n, the size of the matrix.
    - Inside the function, it iterates through each element of the i-th row of G and finds the second minimum value.
    - Similar to firstMin, it skips the diagonal elements and updates first and second accordingly.
    - The second minimum value found is returned.

4. **“TSPRec” Function:**
    - This is the main recursive function to solve the Traveling Salesman Problem (TSP).
    - It takes several parameters: G, the adjacency matrix, curr_bound, the current lower bound, curr_weight, the current weight of the path, level, the current level in the search tree, curr_path[], the current path, and n, the size of the matrix.
    - It recursively explores all possible paths using backtracking.
    - At each level, it checks if the current path forms a complete tour. If yes, it updates the final result and final path if the cost is lower than the current best result.
    - It prunes unnecessary branches of the search tree based on the lower bound.
    - It updates the current weight and bound based on the chosen edge.
    - Finally, it backtracks by undoing the changes made during exploration.

5. **“Traveling” Function:**
    - This is the entry point of the TSP solver.

- It takes three parameters: G, the adjacency matrix, n, the size of the matrix, and start, the starting node.
- It initializes global variables, constructs the initial path, and calculates the initial lower bound.
- It calls the TSPRec function to find the optimal path.
- Finally, it constructs the string representation of the optimal path and returns it.