Discrete Mathematics

The Travelling salesman problem

Student: Cao Le Minh Khoa : 2352550
Ngày 9 tháng 6 năm 2024

Mục lục

1 Introduction

The Traveling Salesman Problem (TSP) entails discovering the most efficient route for a salesman to visit a specific set of cities only once and then return to the starting city. In a formal sense, the goal is to identify the optimal path that minimizes the overall distance traveled, considering a given list of cities and the distances between each pair. This problem is categorized as NP-hard, indicating that as the number of cities grows, the difficulty of finding an exact solution increases exponentially. Consequently, solving this problem for large datasets poses a significant challenge.

2 Problem definition

Given a list of n cities and a matrix g that show the distances between each two cities. For example, g[i][j] represents the distance between city i and city j, the goal is to find a tour that: starts and ends at a specific city, visits each city exactly once, minimizes the total travel distance.

3 Algorithm

This implementation employs the Held-Karp algorithm, which leverages dynamic programming with bitmask. This approach efficiently addresses the Traveling Salesman Problem (TSP) for a moderate number of cities by storing intermediate results, thereby preventing redundant calculations.

3.1 Recursive Function

The recursive function "tsp"computes the minimum tour cost using the following steps

1. If all cities have been visited, it returns the distance back to the starting city.
2. If a value has already been computed, it returns the cached result.
3. It iterates through all cities to identify the next unvisited city.
4. It updates the minimum cost and records the parent city for later path reconstruction.

3.2 Dynamic Programming with Bitmasking

The dynamic programming approach uses a 2D array dp, where dp[mask][i] denotes the minimum cost to visit the set of cities indicated by the bitmask mask, ending at city i. The bitmask mask is an integer in which each bit signifies whether a city has been visited (1) or not (0).

4 Code

The code is segmented into three components: "tsp,""Path_generator,"and "Traveling."

4.1 Function Definition

4.1.1 tsp

The "tsp"function calculates the minimum tour cost utilizing dynamic programming and bitmasking.

```cpp
int tsp(int mask, int pos, int start_index, int VISITED_ALL, int G[20][20],
 ↪  vector<vector<int>>& dp, vector<vector<int>>& parent) {
    if (mask == VISITED_ALL) {
        return G[pos][start_index] == 0 ? MAX : G[pos][start_index];
    }

    if (dp[pos][mask] != -1) {
        return dp[pos][mask];
    }

    int ans = MAX;
    for (int city = 0; city < dp.size(); city++) {
        if ((mask & (1 << city)) == 0 && G[pos][city] != 0) {
            int new_cost = G[pos][city] + tsp(mask | (1 << city), city,
             ↪  start_index, VISITED_ALL, G, dp, parent);
            if (new_cost < ans) {
                ans = new_cost;
                parent[pos][mask] = city;
            }
        }
    }
    return dp[pos][mask] = ans;
}
```

4.1.2 Traveling function

The "Traveling"function sets up the required data structures and invokes the "tsp"function to solve the Traveling Salesman Problem (TSP) starting from the provided city.

```cpp
string Traveling(int G[20][20], int n, char start) {
    int start_index = start - 'A';
    int VISITED_ALL = (1 << n) - 1;

    vector<vector<int>> dp(n, vector<int>(1 << n, -1));
    vector<vector<int>> parent(n, vector<int>(1 << n, -1));

    int min_cost = tsp(1 << start_index, start_index, start_index,
     ↪  VISITED_ALL, G, dp, parent);

    // Reconstruct the path
    int mask = 1 << start_index;
    int pos = start_index;
    vector<int> path;
    path.push_back(start_index);

    while (true) {
        pos = parent[pos][mask];
        if (pos == -1) break;
        path.push_back(pos);
        mask |= (1 << pos);
    }
```

```
    path.push_back(start_index);

    // Convert path to string
    stringstream ss;
    for (int i = 0; i < path.size(); ++i) {
        if (i != 0) ss << " ";
        ss << (char)(path[i] + 'A');
    }

    return ss.str();
}
```

5 Example

To utilize this code, you need an adjacency matrix representing the graph and specify the starting city. The following example illustrates how to invoke the "Traveling"function.

```
    #include "tsm.cpp"
#include "bellman.cpp"
#include <iostream>

using namespace std;

int main()
{
    int n = 4;
    int g[20][20] = {
        {0, 11, 14, 19},
        {11, 0, 34, 24},
        {14, 34, 0, 29},
        {19, 24, 29, 0}};

    char start = 'A';
    char goal = 'E';

    string tsp_result = Traveling(g, n, start);
    string BF_result = BF_Path(g, n, start, goal);

    cout << "Best Path: " << tsp_result << endl;
    cout << "Shortest Path using Bellman-Ford: " << BF_result << endl;

    return 0;
}
```

6 Detailed Calculation and Output

Given the adjacency matrix:

$$\begin{bmatrix} 0 & 11 & 14 & 19 \\ 11 & 0 & 34 & 24 \\ 14 & 34 & 0 & 29 \\ 19 & 24 & 29 & 0 \end{bmatrix}$$

6.1 Step by step

1. Start at city 'A' (index=0)
2. From 'A' (0):
   - 'B' (1): Distance 11, next state: 'A-B'
   - 'C' (2): Distance 14, next state 'A-C'
   - 'D' (3): Distance 19, next state 'A-D'
3. From 'A-B' (0-1):
   - From 'B' (1) to:
     - 'C' (2): Distance 34, next state: 'A-B-C'
     - 'D' (3): Distance 24, next state: 'A-B-D'
   - Adding distances, 'A-B-C' total: 11+34=45
   - Adding distances, 'A-B-D' total: 11+24=35
4. From 'A-B-C' (0-1-2):
   - From 'C' (2) to:
     - 'D' (3): Distance 29, next state 'A-B-C-D'
   - Adding distances, 'A-B-C-C' total: 45+29=74
5. From 'A-B-D' (0-1-3) to:
   - From 'D' () to:
     - 'C' (2): Distance 29, next state 'A-B-D-C'
   - Adding distances, 'A-B-D-C' total: 35+29=64
6. Continue similarly for paths starting from 'A-C' and 'A-D'

6.2 Reconstructed Path

The "Path_generator"function is utilized to reconstruct the path from the parent array "p".

6.3 Output

Given the provided adjacency matrix and starting city 'A', the anticipated output of the main function would be:

```
Best Path: A B D C A
```

Explaination

- The path starts from 'A', goes to 'B', then 'D', then 'C', and returns to 'A'.

- Tho total distance of this path is:
  - A to B: 11
  - B to D: 24
  - D to C: 29
  - C to A: 14
- Total distance: 11 + 24 + 29 + 14 = 78

Therefore, the best path which has the minimum toatal distance is A B D C A.