VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DISCRETE STRUCTURE

Assignment 1

Using Dynamic Programming To Solve The Traveling Saleman Problem

Instructors: Tran Tuan Anh, *CSE-HCMUT*

Author: Nguyen Huu Phuc

ID: 2352938

Email: phuc.nguyen0310@hcmut.edu.vn

Ho Chi Minh City, May 2024

Contents

1 Introduction

The Traveling Salesman Problem (TSP) is a classic optimization problem in which a salesman seeks the shortest possible route that visits a set of cities exactly once and returns to the starting city. This problem is known to be NP-hard, meaning there is no known polynomial-time solution.

2 Problem Description

Given a graph represented by an adjacency matrix G with n nodes (cities), the objective is to determine the shortest possible route that visits each city exactly once and returns to the starting city. The adjacency matrix $G[i][j]$ holds the cost (distance) of traveling from city i to city j. The solution needs to:

1. Compute the minimum travel cost.
2. Determine the optimal path (route) that incurs this minimum cost.

Various approaches have been developed to tackle TSP, including dynamic programming, ant colony optimization, branch & bound methods, etc. This report outlines a dynamic programming approach to solving TSP.

3 Dynamic Programming Approach

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the problem can be divided into overlapping subproblems that can be solved independently. DP solves each subproblem only once and stores the results for future use, avoiding the need to recompute them.

3.1 Example

Consider 4 cities A, B, C, and D with the start city being A. There are six possible routes: ABCDA, ABDCA, ACBDA, ACDBA, ADBCA, and ADCBA. To find the optimal route, we calculate the distance starting from A plus the optimal route of the subset $\{B, C, D\}$ and then return to A. We need to find the shortest path within the subset $\{B, C, D\}$ and add the distance back to A. This involves determining the optimal route for B, C, and D as follows:

- Calculate the distance from B plus the optimal route for the subset $\{C, D\}$ (CD or DC) and then back to A.
- Calculate the distance from C plus the optimal route for the subset $\{B, D\}$ (BD or DB) and then back to A.
- Calculate the distance from D plus the optimal route for the subset $\{B, C\}$ (BC or CB) and then back to A.

By comparing these options, we can determine the most efficient route within the subset and ultimately find the shortest overall path.

3.2 Detailed Explanation

Given the distance matrix on the right side ($n = 4$, start city = A):

	A	B	C	D
A	0	1	1	1
B	2	0	4	2
C	2	3	0	3
D	5	2	4	0

Initially, create a matrix *value* of size $2^n \times n$ where n is the number of cities. 2^n is the total number of subsets, and n is used to store the optimal distances of the subsets based on the starting points. Along with this, create a matrix *previ* of size $2^n \times n$ to store the sequence of subsets.

	value				previ			
	D	C	B	A	D	C	B	A
	3	2	1	0	3	2	1	0
0010	INF	INF	2	INF	INF	INF	INF	INF
0100	INF	2	INF	INF	INF	INF	INF	INF
1000	5	INF	INF	INF	INF	INF	INF	INF
0110	INF	5	6	INF	INF	1	2	INF
1010	4	INF	7	INF	1	INF	3	INF
1100	6	8	INF	INF	2	3	INF	INF
1110	8	7	8	INF	1	3	3	INF
1111	INF	INF	INF	8	INF	INF	INF	2

The *value* and *previ* matrices are initialized as 999999, which is INF in the table. A, B, C, D (0, 1, 2, 3) can be represented in binary as 1000, 0100, 0010, 0001 (which is reversed in the table so the order is increasing). However, as we have 2^4 rows, we only use half of them. This is because there are 7 subsets including the start city A and 0000. (But at first, we need to initialize *value*[0001][0] to calculate the case subset of 1 city.)

3.2.1 Case 1: For subsets with 1 city

Fill in the distance from that city to the start city in the corresponding columns. In this table, the distance from B to A is 2 (noted as AB), from C to A is 2 (noted as AC), and from D to A is 5 (noted as AD).

3.2.2 Case 2: For subsets with 2 cities

Fill in the distance from one city (x) to the other city (y) and then back to the start city in the corresponding columns (meaning the last city x). In 0110 (CB), we consider two routes: ABC (x=C) and ACB (x=B). We can calculate these distances, which are CB + BA = 5 (ABC) and BC + CA = 6 (ACB). Then we fill in the *previ* table with y=B(1) for ABC and y=C(2) for ACB in the corresponding columns.

3.2.3 Case 3: For subsets with 3 cities

Fill in the distance from one city (x) to the subset of 2 cities (y) and then back to the start city in the corresponding columns (meaning the last city x). In 1110 (DCB), we consider three routes with a subset within: A(BC)D (x=D), A(BD)C (x=C), and A(CD)B (x=B). Consider A(BC)D: we need to choose between ABCD or ACBD. From the results in case 2, we compare DC + CBA = 4 + 5 = 9 and DB + BCA = 2 + 6 = 8, so we choose 8 to fill in *value*[1110][3] and fill in *previ*[1110][3] = 1, which represents ACBD.

3.2.4 Case 4: Calculate the final result

From case 3, we achieve three routes: ACBD, ABDC, ACDB. Now we calculate the total of a cycle: ACBDA = AD + DBCA = 1 + 8 = 9, ABDCA = AC + CDBA = 1 + 7 = 8, ACDBA = AB + BDCA = 1 + 8 = 9. Finally, we choose ABDCA as the shortest cycle with a cost of 8.

4 Code Implementation

Functions and Their Roles

DPtable

Purpose: Fills the DP table (`value`) with the minimum cost of traveling to each subset of cities ending at a specific city.

Parameters:

- `G[20][20]`: The adjacency matrix representing distances between cities.
- `n`: The number of cities.
- `value`: The DP table storing minimum travel costs.
- `previ`: The table storing the previous city in the optimal path for reconstructing the route.

Logic:

- Iterates over all subsets of cities.
- For each subset, iterates over all possible end cities.
- Calculates and updates the minimum travel cost to reach the end city from any other city in the subset.

findMinCost

Purpose: Determines the minimum cost of completing the tour and identifies the last city before returning to the start.

Parameters:

- `G[20][20]`, `n`, `value`, `startIdx`: Same as above.
- `minCost`: Output parameter for the minimum cost.
- `end`: Output parameter for the last city before returning to the start.

Logic:

- Iterates over all cities to find the minimum cost of returning to the start city from any city in the full subset of cities.

reconstructPath

Purpose: Reconstructs the optimal path from the `previ` table.

Parameters:

- `n`, `previ`, `endCity`, `startIdx`: Number of cities, previous city table, last city in the path, and starting city index.

Logic:

- Traces back from the end city using the `previ` table to reconstruct the full path.
- Adds the start city at the end to complete the tour.

Traveling

Purpose: Main function to solve the TSP and return the optimal route as a string.

Parameters:

- `G[20][20]`, `n`, `start`: Distance matrix, number of cities, and starting city.

Logic:

- Initializes the DP and `previ` tables.
- Calls `DPtable` to fill the DP table.
- Calls `findMinCost` to get the minimum cost and the end city.
- Calls `reconstructPath` to get the optimal path.
- Returns the path as a string.

```
1  void DPtable(int G[20][20], int n, vector<vector<int>>& value, vector<vector<int>>& previ)
2  {
3      // Iterate over all subsets of nodes (1<<n also means 2^n)
4      for (int sub = 1; sub < (1 << n); sub++)
5      {
6          // Iterate over all nodes to find the end node of the current subset
7          for (int i = 0; i < n; i++)
8          {
9              // operator & check if there is (1<<i) in sub
10             if ((sub & (1 << i)))
11             {
12                 // Iterate over all nodes to find the previous node in the subset
13                 for (int j = 0; j < n; j++)
14                 {
15                     if (i != j && (sub & (1 << j)) && G[j][i] != 0)
16                     {
17                         // Previous subset excluding the current node (operator ^ means xor)
18                         int prev_sub = sub ^ (1 << i);
19                         if (value[prev_sub][j] != 999999 && value[prev_sub][j] + G[j][i] < value[sub][i])
20                         {
21                             // Update the DP table with the minimum cost & prev table
22                             value[sub][i] = value[prev_sub][j] + G[j][i];
23                             previ[sub][i] = j;
24                         }
25                     }
26                 }
27             }
28         }
29     }
30 }
31
32 void findMinCost(int G[20][20], int n, vector<vector<int>> value, int startIdx, int &minCost, int &end)
33 {
34     minCost = 999999;
35     end = 999999;
36     for (int i = 0; i < n; i++)
37     {
38         if (i != startIdx) // Skip the start node
39         {
40             // Check if there is a route that visits all cities and ends at city i,
41             // the distance from the last city of the route to the start city is valid,
42             // total distance of route and from the last city of the route to the start city < minCost
```

```
            if (value[(1 << n) - 1][i] != 999999 && G[i][startIdx] != 0
            && value[(1 << n) - 1][i] + G[i][startIdx] < minCost)
            {
                // Update the minimum cost & end node
                minCost = value[(1 << n) - 1][i] + G[i][startIdx];
                end = i;
            }
        }
    }
}

vector<int> reconstructPath(int n, const vector<vector<int>>& previ, int endCity, int startIdx)
{
    vector<int> path;
    int current = endCity;

    // Subset includes all nodes
    int sub = (1 << n) - 1;

    // Reconstruct the path until the start node is reached
    while (current != 999999)
    {
        // Add the current node to the path
        path.push_back(current);

        // Get the prev node
        int temp = previ[sub][current];

        // Update the subset to exclude the current node
        sub = sub ^ (1 << current);

        // Move to the prev node
        current = temp;
    }

    // Reverse the path to get the correct order
    reverse(path.begin(), path.end());

    // Add the start node at the end to complete the tour
    path.push_back(startIdx);

    return path;
}

string Traveling(int G[20][20], int n, char start)
{
    int startIdx = start - 'A';

    // DP table stores min cost (initialize as 999999)
    vector<vector<int>> value(1 << n, vector<int>(n, 999999));

    // Table to store the prev node for path reconstruction
    vector<vector<int>> previ(1 << n, vector<int>(n, 999999));

    // Initialize the start position
    value[1 << startIdx][startIdx] = 0;

    // Fill the DP table
    DPtable(G, n, value, previ);

```

```
    // Find the min cost to return to the start node from any node
    // minCost is the minimum distance to the start city and end is the last city before returning to the start city
    int minCost, end;
    findMinCost(G,n,value,startIdx,minCost,end);

    // Reconstruct the path
    vector<int> path = reconstructPath(n, previ, end, startIdx);

    // Print the minimum cost and path
    string result;
    for (int i = 0; i < path.size(); i++)
    {
        // Print the path excluding the first start node (it is the same as the last node)
        char city = 'A' + path[i];
        result = result + city + " ";
    }

    // cout<<minCost<<'\n';
    return result;
}
```

5 Pros and Cons

5.1 Pros

- **Optimal Substructure**: Dynamic programming efficiently exploits the TSP's optimal substructure property by solving subproblems and storing results to avoid redundant calculations.
- **Exact Solutions**: Unlike heuristic methods, dynamic programming guarantees an exact solution, which is critical in scenarios where precision is paramount.
- **Reduces Redundancy**: By storing solutions to subproblems, dynamic programming minimizes the redundancy of recalculating the same values, significantly reducing computational effort compared to brute force methods.

5.2 Cons

- **High Computational Complexity**: The time complexity of dynamic programming for TSP is $O(n^2 \cdot 2^n)$, where n is the number of cities. This exponential complexity makes it impractical for large datasets.
- **Space Complexity**: The space requirement is also high, with $O(n \cdot 2^n)$ memory usage to store the solutions of subproblems, which can be prohibitive for a large number of cities.
- **Scalability Issues**: Due to its computational and space complexity, dynamic programming is not suitable for TSP instances with a large number of cities, limiting its practical applications.

6 Summary

Dynamic programming provides an exact solution to the TSP by leveraging the problem's optimal substructure and reducing redundancy through memoization. However, its high computational and space complexity restrict its use to smaller instances of the problem, making it less practical for larger datasets compared to heuristic or approximation methods.

7 Conclusion

The provided code implements a dynamic programming approach to solve the TSP efficiently for small to moderate-sized graphs. This method guarantees finding the optimal solution by exploring all possible

subsets and paths systematically. However, the time and space complexity are exponential, making it impractical for very large graphs. Further optimizations or heuristic methods may be considered for larger instances of TSP.

References

[1] GeeksforGeeks, *Travelling Salesman Problem using Dynamic Programming*, [Online]. Available: https://www.geeksforgeeks.org/travelling-salesman-problem-using-dynamic-programming/. [Accessed: 09-Jun-2024].

[2] Wikipedia, *Held–Karp algorithm*, [Online]. Available: https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm. [Accessed: 09-Jun-2024].

[3] R. H. T. W. Nguyen, *Dynamic Programming Algorithms for the TSP*, Nagoya University, [Online]. Available: https://www.math.nagoya-u.ac.jp/~richard/teaching/s2020/Quang1.pdf. [Accessed: 09-Jun-2024].