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

Discrete Structure for Computing(CO1007) - CC02

ASSIGNMENT REPORT

Traveling Saleman Problem

Instructor(s): Nguyen Van Minh Man, Mahidol University
Nguyen An Khuong, CSE-HCMUT
Tran Tuan Anh, CSE-HCMUT
Tran Hong Tai, CSE-HCMUT
Mai Xuan Toan, CSE-HCMUT
Student: Vo Hoang Phuc, 2252650

HO CHI MINH CITY, NOVEMBER 2024

Contents

1 Introduction

"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" - Wikipedia

In the next section, this essay presents in detail the usage of dynamic programming approach, specifically the Held-Karp algorithm, which can generate the best solution path for small to moderately sized TSP graphs.

2 Approach to TSP

2.1 Problem definition

Given:

- A graph $G = (V, E)$, where V is a set of n cities (vertices), and E is a set of edges representing distances between pairs of cities.
- A distance matrix graph$[i][j]$ that defines the cost of traveling from city i to city j. Missing edges (disconnections) are represented 0.

Objective:

Find a Hamiltonian cycle (a cycle visiting each vertex exactly once and returning to the starting vertex) with the minimum total cost.

2.2 Idea explanation

The intended approach uses Dynamic Programming method mixed with Bitmasking. Briefly speaking, Dynamic Programming or DP is an algorithmic technique used in computer science and mathematics to solve complex problems by breaking them down into smaller overlapping subproblems. Moreover, DP is a powerful technique for resolving optimization issues by segmenting them into more manageable, overlapping subproblems . DP solves each subproblem once and saves the solution for later use, usually in the form of a table or memoization, as opposed to solving each subproblem repeatedly. This method is more efficient than naive recursive or brute-force approaches because it avoids duplicate calculations.

Initially, we have:

- v is the current city.
- `mask` represents the subset of cities visited so far. Each subset of cities is represented by a binary number
  - If the i-th bit of mask is 1, the i-th vertex is included in the subset.
  - Example: For 4 cities, mask = 0110 represents a subset including cities 1 and 2 (0-indexed).
  - In initialization stage, mask = 1 » startVertex
- `f[v][mask]` stores the minimum cost to visit all cities in `mask`, starting at city v.
- parent[v][mask] stores the vertex that v visit.

Base case:

- When all cities have been visited (`mask == (1 « numVertices) - 1`), the cost to return to the starting city is the weight of the edge from the current city to the starting city (`graph[v][startVertex]`).
- If this edge does not exist (`graph[v][startVertex] == 0`), return a large value (`MAX`) to indicate infeasibility.

At recursive stage, for every unvisited city `u`, compute the cost to visit `u` from `v`, add it to the result of solving the subproblem for `u`, and update the DP table:

```
f[v][mask] = graph[v][u] + f[u][mask | (1 « u)]
```

- Here, `mask | (1 « u)` marks city `u` as visited.

- If this `newCost` is less than the current minimum `res`, update `res` and `parent[v][mask]` $=$ `u`.

After calculating minimun cost, we will use Backtracking to find out the solution path for TSP. Let the path be initialized as an empty string, i.e., `path` $=$ ””.

Let `current` be the index of the starting vertex, which is calculated as `current` $=$ `startVertex`.

Let `mask` be initialized with $1 \ll$ `current`,mean that startVertex is visited

While true:

- `path` $=$ `path` $+$ `current`..
- `next` $=$ `parent[current][mask]`.
- If `next` $= -1$, break the loop, as this indicates the end of the path.
- Otherwise, update the mask: `mask` $+= (1 \ll$ `next`$)$.
- Update `current` to the next vertex: `current` $=$ `next`.

Finally, add the starting vertex back to the path to complete the cycle: `path` $=$ `path` $+$ `startVertex`. Return the computed path.

2.3 Example

Let consider this example

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

Where:

- The rows and columns represent cities A, B, C, and D.

- The entry at position (i, j) represents the distance from city i to city j. Suppose A is startVertex, we need to find f[A][1] which represent the cost of the path start from A and visit all the unvisited vertex in mask $= 1 = 0001_2$.

- newCostA = graph[A][B] + f[B][0011_2]
- newCostA = graph[A][C] + f[C][0101_2]
- newCostA = graph[A][D] + f[D][1001_2]

For newCostA = graph[A][B] + f[B][0011_2], we have:

$$\begin{aligned} f[B][0011_2] &= \text{newCostAB} = \text{graph}[B][C] + f[C][0111_2] \\ &= \text{graph}[B][C] + \text{graph}[C][D] + f[D][1111_2] \\ &= \text{graph}[B][C] + \text{graph}[C][D] + \text{graph}[D][A] \quad = 4 + 2 + 5 = 11 \end{aligned}$$

or

$$\begin{aligned} f[B][0011_2] &= \text{newCostAB} = \text{graph}[B][D] + f[D][1011_2] \\ &= \text{graph}[B][D] + \text{graph}[D][C] + f[C][1111_2] \\ &= \text{graph}[B][D] + \text{graph}[D][C] + \text{graph}[C][A] \quad = 3 + 2 + 3 = 8 \end{aligned}$$

For newCostA = graph[A][C] + f[C][0101_2], we have:

$$\begin{aligned} f[C][0101_2] &= \text{newCostAC} = \text{graph}[C][B] + f[B][0111_2] \\ &= \text{graph}[C][B] + \text{graph}[B][D] + f[D][1111_2] \\ &= \text{graph}[C][B] + \text{graph}[B][D] + \text{graph}[D][A] \quad = 4 + 3 + 5 = 12 \end{aligned}$$

or

$$\begin{aligned} f[C][0101_2] &= \text{newCostAC} = \text{graph}[C][D] + f[D][1101_2] \\ &= \text{graph}[C][D] + \text{graph}[D][B] + f[B][1111_2] \\ &= \text{graph}[C][D] + \text{graph}[D][B] + \text{graph}[B][A] \quad = 2 + 3 + 2 = 7 \end{aligned}$$

For newCostA = graph[A][D] + f[D][1001_2], we have:

$$\begin{aligned} f[D][1001_2] &= \text{newCostAD} = \text{graph}[D][C] + f[C][1101_2] \\ &= \text{graph}[D][C] + \text{graph}[C][B] + f[B][1111_2] \\ &= \text{graph}[D][C] + \text{graph}[C][B] + \text{graph}[B][A] \quad = 2 + 4 + 2 = 8 \end{aligned}$$

or

$$\begin{aligned} f[D][1001_2] &= \text{newCostAD} = \text{graph}[D][B] + f[B][1011_2] \\ &= \text{graph}[D][B] + \text{graph}[B][C] + f[C][1111_2] \\ &= \text{graph}[D][B] + \text{graph}[B][C] + \text{graph}[C][A] \quad = 3 + 4 + 3 = 10 \end{aligned}$$

After get minimun cost of each formula, we have

- newCostA = graph[A][B] + 8
- newCostA = graph[A][C] + 7
- newCostA = graph[A][D] + 8

Then

- newCostA = 2 + 8 = 10
- newCostA = 3 + 7 = 10
- newCostA = 5 + 8 = 13

So the minimun cost will be 10 and we have 2 possible paths for this problem. Because our algorithm just update the minimum cost if newcost < minimun cost so we can just get the first solution as result but it does not matter that much since the 2 paths generate the same minimun cost

Backtracking Path:
f[A][0001_2] = graph[A][B] + f[B][0011_2]
Then parent[A][0001_2] = B
f[B][0011_2] = graph[B][D] + f[D][1011_2]
Then parent[B][0011_2] = D
f[D][1011_2] = graph[D][C] + f[C][1111_2]

Then parent[D][1011_2] = C and parent[C][1111_2] = A
Apply the algorithm described in last section, we will get A B D C A

2.4 Evaluation

Time complexity

Let consider function f[u][mask].

1. There are 2^n values for `mask`.
2. For each subset, there are n ending vertices.
3. For each mask, there are n ways to go from a vertex to that mask.

So, the time complexity is $O(n^2 \times 2^n)$.

Space complexity

There must be a 2-dimensional table to store value of f[u][mask] which u have n possible way to choose, and mask have 2^n values.

So, the Space complexity is $O(n \times 2^n)$.

Advantages

- This method guarantees the optimal solution for TSP.
- It gracefully handles disconnected graphs.
- It is computationally feasible for $n \leq 30$, making it suitable for many practical applications.

Disadvantages

- This algorithm is hard to implement as recursion is applied.
- Memory usage can be high for large graphs due to the exponential growth of the Dynamic Programming table

3 Conclusion

In this approach to solving the Traveling Salesman Problem (TSP), we utilized dynamic programming combined with bitmasking, which is a good method guaranteeing to generate a robust, exact solution for TSP of moderate size graph. Using dynamic programming, it significantly reduces computational overhead. The implementation is versatile and practical for real life applications. However, its scalability remains a challenge for larger graphs.

4 References

1. Rosen, Kenneth H. *Discrete Mathematics and Its Applications*, 8th edition. McGraw-Hill Education, 2023.