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

DISCRETE STRUCTURE (CO1007)

Assignment

Exploring the Travelling Salesman Problem: A Comprehensive Approach

---

Advisor(s):	Nguyễn Văn Minh Mẫn, Mahidol University	
	Nguyễn An Khương, CSE-HCMUT	
	Trần Tuấn Anh,CSE-HCMUT	
	Nguyễn Tiến Thịnh, CSE-HCMUT	
	Trần Hồng Tài, CSE-HCMUT	
	Mai Xuân Toàn, CSE-HCMUT	
Student(s):	Nguyễn Hồng Phúc	ID 2252639

HO CHI MINH CITY, JUNE 2024

Contents

1 Introduction:

1.1 Graph

These discrete structures consist of vertices (nodes) and edges that connect these vertices. Graphs come in various forms, depending on specific characteristics:

- **Directed vs. Undirected Graphs:**In directed graphs, edges have a direction(one-way), while undirected graphs have bidirectional edges.
- **Multigraphs:**These allow multiple edges to connect the same pair of vertices.
- **Loops:** Some graphs permit loops (edges connecting a vertex to itself).

1.2 Applications

Graphs serve as powerful models across diverse disciplines. Here are some applications:

- **Street Navigation:**Using graph models, we can determine if it's feasible to walk all streets in a city without retracing our steps.
- **Map Coloring:** Graphs help find the minimum number of colors needed to color regions on a map without adjacent regions having the same color.
- **Circuit Design:** We assess whether a circuit can be implemented on a planar circuit board using graph theory.
- **Chemical Structures:** Graphs distinguish between chemical compounds with identical molecular formulas but different structures.
- **Network Connectivity:** Graph models verify whether two computers are connected via a communication link.
- **Weighted Graphs:** Assigning weights to edges allows us to solve problems like finding the shortest path between cities in a transportation network.
- **Scheduling and Channel Assignment:** Graphs aid in scheduling exams and allocating channels to television stations.

Graph theory's versatility makes it an indispensable tool for problem-solving and analysis.

1.3 Path

Definition 1. *Let n be a non-negative integer and G an undirected graph. A path of length n from u to v in G is a sequence of n edges e1, ..., en of G for which there exists a sequence $x0 = u$, x1, ..., xn1, $xn = v$ of vertices such that ei has, for $i = 1$, ..., n, the endpoints xi1 and xi. When the graph is simple, we denote this path by its vertex sequence x0, x1, ..., xn (because listing these vertices uniquely determines the path). The path is a circuit if it begins and ends at the same vertex, that is, if $u = v$, and has a length greater than zero. The path or circuit is said to pass through the vertices x1, x2, ..., xn1 or traverse the edges e1, e2, ..., en. A path or circuit is simple if it does not contain the same edge more than once.*

Determining a path of least length between two vertices in a network is one such problem. To be more specific, let the length of a path in a weighted graph be the sum of the weights of the edges of this path. The question is: What is a shortest path, that is, a path of least length, between two given vertices?

2 The Travelling salesman problem

2.1 Introduction

The Travelling Salesman Problem (TSP) is a classic problem in computer science and operations research. It involves finding the shortest possible route that visits each city (node) exactly once and returns to the original city, where the distance between each pair of cities is known. Despite its simple formulation, solving the TSP efficiently is known to be NP-hard, making it challenging for large problem instances.

2.2 Objective

This report aims to explore various approaches to solving the TSP efficiently. We will discuss the fundamental concepts behind the TSP and present a step-by-step guide to implementing a solution using dynamic programming.

2.3 Understanding the Problem

The TSP can be represented as a graph, where cities are nodes and the distances between them are edges. The objective is to find a Hamiltonian circuit (a circuit that visits each node exactly once) with minimum total weight.

2.4 Common approaches

2.4.1 Exact Algorithms:

- **Branch and Bound:** This algorithm systematically explores the search space by branching at each node and pruning branches that cannot lead to an optimal solution.
- **Integer Linear Programming (ILP):** Formulate the TSP as an ILP problem and use optimization solvers to find the optimal solution.

2.4.2 Heuristic Algorithms:

- **Nearest Neighbor:** Start from an arbitrary city and repeatedly visit the nearest unvisited city until all cities are visited, forming a Hamiltonian cycle.
- **Insertion Heuristics:** Begin with a partial tour and iteratively insert new cities into the tour using heuristics such as nearest insertion or farthest insertion.

2.4.3 Metaheuristic Algorithms:

- **Genetic Algorithms:** These algorithms simulate the process of natural selection and evolution to search for good solutions to the TSP.
- **Ant Colony Optimization (ACO):** Inspired by the foraging behavior of ants, ACO algorithms use pheromone trails to guide the search for the optimal tour.

2.4.4 Approximation Algorithms:

- **Minimum Spanning Tree (MST):** Construct a minimum spanning tree of the graph and then find a Eulerian tour by traversing each edge exactly once.
- **Farthest Insertion:** Start with any city and iteratively insert the farthest city into the tour until all cities are visited.

However, among these approaches, I've chosen to utilize dynamic programming with bitmasking. This involves breaking down the problem into smaller subproblems. Each subproblem focuses on a subset of cities to visit, with bitmasking efficiently representing these subsets. By finding the shortest tour for each subset that concludes at a particular city, we build towards solving the overall problem.

2.5 Bitmasking

Bitmasking is chosen over other approaches for solving the Travelling Salesman Problem (TSP) in certain scenarios because of its efficiency and simplicity, especially when dealing with small to medium-sized instances of the problem(Since this assignment graph is maximum 20x20, which is a medium size). While other approaches such as exact algorithms, heuristic algorithms, metaheuristic algorithms, and approximation algorithms offer different trade-offs in terms of solution quality, computational complexity, and implementation complexity, making it's not a suitable choice.

- **Efficiency:** Bitmasking allows for efficient representation of subsets of cities. In the TSP, we need to consider all possible subsets of cities to find the optimal solution. Bitwise operations provide a compact and efficient way to represent and manipulate these subsets.

- **Dynamic Programming:** Bitmasking is often used in conjunction with dynamic programming to solve the TSP efficiently. Dynamic programming breaks down the problem into smaller subproblems, and bitmasking helps in efficiently representing and accessing these subproblems. This approach results in a time complexity of $O(2^n \times n^2)$, which is manageable for small to medium-sized instances of the TSP.

- **Simplicity:** Implementing the bitmasking approach is relatively straightforward compared to some other algorithms like genetic algorithms, ant colony optimization, or simulated annealing. It involves iterating over all possible subsets of cities and computing the optimal tour length for each subset, which can be easily implemented in code.

Cons of Bitmasking While bitmasking is effective for small to medium-sized instances of the TSP, it may not scale well to very large instances due to its exponential time complexity. In such cases, other approaches like heuristic algorithms (e.g., nearest neighbor), metaheuristic algorithms (e.g., genetic algorithms), or approximation algorithms (e.g., minimum spanning tree) may be more suitable. These approaches offer trade-offs between solution quality and computational complexity. Therefore, we have chosen bitmasking for its correctness in handling small to medium-sized graphs, which aligns well with the requirements of this assignment.

2.6 Algorithm for Traveling salesman problem

Pseudocode: algorithm described[1]

```

int[][] dp = new int[1 << n][n];
// Some initialization of dp, possibly.
for (int mask = 1; mask <= (1<<n); mask++) {
 for (int last = 0; last<n; last++) {
 if (((mask >> last) & 1) == 0) continue;
 int prev = mask - (1 << last);
 dp[mask][last] = Integer.MAX_VALUE;
 // v is the last item visited in prev.
 for (int v=0; v<n; v++) {
 if (((prev >> v) & 1) == 0) continue;
 int curScore = dp[prev][v] + cost(v, last);
 dp[mask][last] = Math.min(dp[mask][last], curScore);
 }
 }
}
```

Explanation of pseudocode:

- The first two loops go through the total search space.
- The first if statement screens out impossible sub-cases, where the last vertex isn't in the subset specified by mask.
- prev represents the subset of vertices visited BEFORE arriving at vertex last.
- v represents the possible last locations we could visit in visiting each vertex in prev.
- The inner most if does the same task as the previous if, screening out impossible cases.
- The relevant score for this path (curScore) is the sum of the best score of visiting everything in prev, ending in vertex v, added to the cost/edge from vertex v to vertex last, which is where we are trying to end up. I denote the edge weight simply by a cost function that the programmer is free to define.

- The last line in the inner-most loop simply updates our answer to be the best of all possibilities where we visit all the vertices in prev followed by traveling to vertex last.

2.7 My implementation

Inspiration from https://zhongquan789.gitbook.io/ [2].

```cpp
string Traveling(int G[20][20], int n, char startNode) {
int startIndex = vertexIndex(startNode);
int maxs = 1 << n; // Shift left, maximum number of subsets
    (2^n)

// Initialize dynamic programming table dp with appropriate
    initial values
vector<vector<pair<int, string>>> dp(maxs, vector<pair<int,
    string>>(n, {INF, ""}));
dp[1 << startIndex][startIndex] = {0, string(1, startNode)};
    // Path containing just the start node

for (int s = 1; s < maxs; s++) {
    // Skip subsets that do not include the chosen start node
    if (!(s & (1 << startIndex))) continue;

    for (int cur = 0; cur < n; cur++) {
        // Skip if cur is already in subset s
        if (s & (1 << cur)) continue;

        for (int prev = 0; prev < n; prev++) {
            if (prev == cur || !(s & (1 << prev)) || G[prev][
                cur] == 0) continue;

            auto& mindis = dp[s | (1 << cur)][cur];
            if (dp[s][prev].first + G[prev][cur] < mindis.
                first) {
                mindis = {dp[s][prev].first + G[prev][cur],
                    dp[s][prev].second + " " + (char)('A' +
                    cur)};
            }
        }
    }
}

// Complete the cycle by returning to the chosen startNode
int minCost = INF;
string minPath = "";
for (int end = 0; end < n; end++) {
    if (dp[maxs - 1][end].first + G[end][startIndex] <
       minCost && G[end][startIndex] != 0) {
        minCost = dp[maxs - 1][end].first + G[end][startIndex
           ];
        minPath = dp[maxs - 1][end].second + " " + startNode;
    }
}

if (minCost == INF) {
    return "No valid path exists";
}
cout << "The cost of shortest path(TSM): ";
cout << minCost << endl;
return minPath;
```

Core Logic and Loop Explanation:

- **Bitmask Representation:**Each subset of nodes is represented as an integer where the bits correspond to whether a node is included in the subset.
- **Dynamic Programming Table (dp):**dp[s][i] stores a pair where the first element is the minimum cost to visit all nodes in subset s ending at node i, and the second element is the path taken to achieve this cost.

Main Loop:

```
for (int s = 1; s < maxs; s++) {
if (!(s & (1 << startIndex))) continue;
```

```

for (int cur = 0; cur < n; cur++) {
    if (s & (1 << cur)) continue;

    for (int prev = 0; prev < n; prev++) {
        if (prev == cur || !(s & (1 << prev)) || G[prev][cur]
            == 0) continue;

        auto& mindis = dp[s | (1 << cur)][cur];
        if (dp[s][prev].first + G[prev][cur] < mindis.first)
           {
            mindis = {dp[s][prev].first + G[prev][cur], dp[s
               ][prev].second + " " + (char)('A' + cur)};
        }
    }
}
```

Explanation of the main loop:

- Iterate over all subsets s.
- Skip subsets that do not include the start node (If a subset does not include the start node, it is not a valid partial solution for the TSM.).
- For Each Node cur Not in Subset s, iterate over all possible prev nodes in s (To extend the subset s to include cur, we need to consider all nodes prev already in s. This helps in finding the minimum cost path to reach cur.)
- Update dp if a shorter path to cur through all prev nodes is found

```
int minCost = INF;
string minPath = "";
for (int end = 0; end < n; end++) {
    if (dp[maxs - 1][end].first + G[end][startIndex] <
       minCost && G[end][startIndex] != 0) {
        minCost = dp[maxs - 1][end].first + G[end][startIndex
           ];
        minPath = dp[maxs - 1][end].second + " " + startNode;
    }
}

if (minCost == INF) {
    return "No valid path exists";
}
cout << "The cost of shortest path(TSM): ";
cout << minCost << endl;
return minPath;
```

Explanation

- After filling the DP table, find the minimum cost path that returns to the start node.
- Iterate over all possible end nodes, checking the cost of returning to the start node.
- If minCost remains INF, it means no valid path exists that visits all nodes and returns to the start node.

3 Results and Discussion

- This Dynamic programming (*Bitmasking*) approach efficiently solves the TSP for a moderate number of cities.
- This approach guarantees finding the optimal solution by exploring all possible subsets of paths by an exact number
- However, for larger instances of TSP, more advanced techniques or heuristics may be required due to the exponential growth of subsets.

4 Conclusion

The Traveling Salesman Problem (TSP) represents a fundamental challenge in the field of computer science and operations research. This report has provided a comprehensive overview of various approaches to solving the TSP, emphasizing the strengths and limitations of each method.

The implementation of the dynamic programming approach with bitmasking demonstrated the effectiveness of this method for solving the TSP for a moderate number of cities. This approach guarantees finding the optimal solution by systematically exploring all possible subsets of paths. However, as the number of cities increases, the computational cost grows exponentially, highlighting the need for more advanced techniques or heuristics to handle larger instances efficiently.

In conclusion, while the TSP remains a challenging problem, the continuous development of algorithms and computational techniques provides promising avenues for effectively tackling increasingly complex instances. Future research and advancements in computational power are expected to further enhance our ability to solve the TSP and apply these solutions to a wide range of practical problems.

References

[1] Algorithm for traveling salesman problem. http://www.cs.ucf.edu/~dmarino/progcontests/modules/dptsp/DP-TSP-Notes.pdf.

[2] Travelling salesman problem. https://zhongquan789.gitbook.io/leetcode/unsensored/travelling-salesman-problem?fbclid=IwAR0SETFSHtYw4GOUJZtE6ZqkHmB9RWsF5r6SUk3o8ITe_FbG51sCf6ZKwzE. Accessed: 2024-06-01.