COLLEGE OF ENGINEERING TRIVANDRUM

DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

CST 306
ALGORITHM ANALYSIS AND DESIGN
Traveling Salesman Problem using Dynamic Programming
Assignment - 3

Demel C John
CSE 24 Batch 2
Roll no 22
TVE20CS042

Traveling Salesman Problem using Dynamic Programming

1 Task

Solving Traveling Salesman Problem using Dynamic Programming Algorithm Design Strategy
The Traveling Salesman Problem (TSP) is an optimization problem where a salesman needs to visit a given set of cities and return to the starting city, minimizing the total distance traveled. In class, we saw greedy algorithm design strategy don't work for TSP. Design and implement an algorithm using dynamic programming to solve the TSP with the following requirements:

(a) Problem Definition:
i. Define the problem of the TSP, where a salesman needs to visit a set of cities and return to the starting city, finding the shortest possible route. ii. Specify the input format, such as the number of cities and the distances between them.

(b) Recursive Formulation:
i. Formulate the TSP problem recursively, identifying the subproblems and their relationships.
ii. Define the base cases or boundary conditions that terminate the recursion.

(c) Dynamic Programming Approach:
i. Implement the TSP algorithm using dynamic programming techniques, break- ing down the problem into overlapping subproblems and storing the results for future reference.
ii. Explain the data structures, such as arrays or matrices, used to store the in- termediate results.

(d) Time and Space Complexity Analysis:
i. Analyze the time complexity of your algorithm, considering the number of subproblems solved and any additional computations.
ii. Analyze the space complexity, taking into account the storage requirements for the dynamic programming table or other data structures.

(e) Optimized Solution Reconstruction:
i. Describe how the algorithm can reconstruct the optimal route based on the dynamic programming table or stored results.

(f) Performance Evaluation:
i. Test the algorithm on different-sized instances of the TSP problem to evaluate its efficiency and scalability.
ii. Discuss any trade-offs or optimizations that can be made to improve the algo- rithm's performance.

(g) Visualization:
i. Provide a visualization of the optimal route obtained by your algorithm on a map or graph.
Write a detailed algorithm description that outlines the steps involved in solving the TSP using dynamic programming. Provide clear pseudocode or code snippets illustrating the key components of your algorithm. Explain any design decisions or optimizations made during the implementation process.

In addition to the algorithm description, provide a working implementation of the dy- namic programming algorithm for the TSP in a programming language of your choice.

Test the algorithm with various scenarios to demonstrate its correctness, efficiency, and scalability.
Submit your algorithm description, implementation code, and a report discussing the efficiency, performance, and any challenges encountered during the implementation process.

2 Travelling Salesman Problem

The traveling salesman problem (TSP) is a classic problem in combinatorial optimization. It 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?
TSP is an NP-hard problem, which means that there is no known polynomial-time algorithm to solve it. However, there are many heuristics that can be used to find good solutions to TSP. These heuristics are often based on the following general approach:
Start with a random solution.
Evaluate the cost of the solution.
Generate a set of neighboring solutions.
Select the neighbor with the lowest cost.
Repeat steps 3 and 4 until no further improvements can be found.
The traveling salesman problem has many applications in real-world problems, such as planning delivery routes, scheduling manufacturing operations, and designing communication networks.
Here are some of the challenges of solving the traveling salesman problem:
The problem is very sensitive to the input data. Even small changes in the distance between two cities can lead to large changes in the optimal solution. The problem is often very large. For example, the problem of finding the shortest route that visits all 50 states in the United States has over `10^20` possible solutions.
Despite these challenges, the traveling salesman problem is an important problem that has many applications in real-world problems. There are many heuristics that can be used to find good solutions to TSP, and research into new and improved heuristics is ongoing.

2.1 Using dynamic programming

Dynamic programming is a powerful problem-solving technique that involves breaking down complex problems into smaller, more manageable subproblems. By utilizing the solutions to these subproblems, dynamic programming allows for efficient and optimal solutions to be obtained. One classic problem that can be solved using dynamic programming is the traveling salesman problem (TSP).
To tackle TSP using dynamic programming, it is divided into a series of subproblems. The first step is to find the shortest path from one city, let's say city A, to the next city, city B. This involves considering all possible paths from city A to city B and selecting the shortest one.
Once the shortest path from city A to city B is determined, the next subproblem involves finding the shortest path from city B to the next city, city C. This process continues until the final subproblem is reached, which is finding the shortest path from the last city, city N, back to the

starting city, city A.
By solving each of these subproblems independently and storing their solutions, we can then calculate the overall solution to the TSP by summing up the shortest paths obtained. This approach ensures that we consider all possible paths and select the shortest one for each subproblem, ultimately leading to an optimal solution for the TSP.
Dynamic programming offers a systematic and efficient way to solve the traveling salesman problem, allowing for the exploration of all possible routes and the identification of the shortest path. By utilizing the principle of breaking down problems into smaller subproblems and utilizing their solutions, dynamic programming provides a powerful tool for solving a wide range of optimization problems, including the TSP.

2.2 Data structures

Several data structures can be used to solve the Traveling Salesman Problem (TSP) using dynamic programming.The main data structures used here include:

- Distance Matrix: A distance matrix is a two-dimensional array that stores the distances between pairs of cities. It represents the graph of cities and their connections. The matrix allows quick access to the distance between any two cities, which is crucial for calculating the shortest paths.
- State Table or cache: A state table is a dynamic programming table that stores intermediate solutions to subproblems. It helps in memoization, where the solutions to subproblems are stored to avoid redundant computations. The state table can be implemented using a two-dimensional array or a hash table, with the cities and their corresponding states as the keys.
- Bitmasking: Bitmasking is a technique commonly used in dynamic programming to represent the state of visited cities. A bitmask is a binary number where each bit represents whether a particular city has been visited or not. By using bitmasks, the state of visited cities can be efficiently represented, and operations like bitwise OR, AND, and XOR can be applied to manipulate the bitmask.
- Stack: A stack is used to keep track of the path followed in going through optimal path.

3 Algorithm

- Initialize the necessary variables:
  - dist: A distance matrix representing the distances between cities.
  - minCost: A variable to store the minimum cost found so far.
  - minStack: A list to store the order of nodes for the most efficient tour.
- Iterate over the range from 1 to n (assuming n is the number of cities):
- Reset the stack for each iteration.
- Call the tspSolver function with the current city i, a bitmask representing all unvisited cities, and an empty stack.

- Append the current city i to the stack.
- Append the starting city (0) to the stack.
- Calculate the cost from the last city back to the starting city and add it to cost.
- If the calculated cost is less than the current minCost, update minCost and minStack.
- Print the minimum cost of the most efficient tour and the order of nodes in minStack.

tspSolver function:

- Check the base condition: if all cities have been visited (mask & (1 « i)) == 1), return the distance from the current city i back to the starting city (0) and the stack.
- Check if the result for the current city i and mask is already memoized. If so, return the memoized result.
- Initialize the minimum result (resMin) to a large value (`10^10`) and create a copy of the current stack.
- Iterate over all cities (j) to find the next unvisited city:
- If city j is unvisited (mask & (1 « j) != 0) and not the starting city or the current city i:
- Create a copy of the stack.
- Recursively call the tspSolver function with city j, removing the current city i from the mask.
- Calculate the cost from the current city i to city j and add it to the result (res).
- Create a copy of the stack and append city j to it.
- If the calculated res is less than resMin, update resMin and the main stack (stack) with the current stack (stack3).
- Memoize the result by storing resMin and the main stack (stack) for the current city i and mask.
- Return resMin and the main stack (stack). It's worth noting that the code assumes the presence of the memo data structure for memoization, which is not included in the provided code snippet.

4 Analysis

The time complexity of the algorithm is primarily determined by the dynamic programming approach used to solve the TSP. The main part of the algorithm is the tspSolver function, which is called recursively for each city and mask combination.The number of subproblems for the TSP is determined by the number of cities (n) and the number of possible subsets of cities. Since the bitmask is used to represent the state of visited cities, the number of possible subsets is `n2^n`. Therefore, the time complexity of the algorithm can be approximated as O(n * (`2^n`)). In the given code, the range of the main loop is limited to 1 to n, which implies that it considers a fixed

number of cities. Hence, if n is a constant, the time complexity can be considered as O(2^n).
The space complexity of the algorithm is primarily determined by the storage required for the distance matrix, the memoization table (memo), and the stack.
Distance Matrix: The distance matrix requires O(n^2) space since it stores the distances between all pairs of cities.
Memoization Table: The memoization table (memo) is used to store the results of subproblems to avoid redundant calculations. It has a size of n * 2^n, representing all possible combinations of cities and their states. Therefore, the space complexity for the memoization table is O(n * 2^n).
Stack: The stack is used to store the order of nodes for the most efficient tour. The maximum size of the stack is equal to the number of cities plus 2 (start and end cities), which is O(n).
Hence, the overall space complexity of the algorithm can be approximated as O(n^2 + n *2^n), considering the dominant terms. It's important to note that the space complexity can be optimized by using techniques such as iterative dynamic programming or bitmask compression, which can reduce the space requirements to O(2^n). However, the provided code snippet does not demonstrate such optimizations.

5 Result

The approach employed in this implementation generates and solves the Traveling Salesman Problem (TSP) by leveraging various techniques. Although it may encounter certain issues with execution time and looping, it offers accurate solutions and provides a visually intuitive representation.
The algorithm begins by generating the TSP maze, considering the given set of cities. It then proceeds to solve the TSP by employing an optimization approach. The primary objective is to find the shortest path that visits all cities exactly once and returns to the starting city. During the solving process, the algorithm may encounter certain challenges related to execution time, especially when dealing with large numbers of cities. However, it strives to mitigate these issues by employing dynamic programming principles and efficient data structures.
Despite these potential time issues, the algorithm still manages to produce accurate solutions to the TSP. It explores all possible paths and determines the most efficient tour by considering the distances between cities and minimizing the overall cost.
Furthermore, the implementation provides a visual representation of the TSP maze and the resulting tour. This visual aspect helps in understanding and analyzing the problem, allowing for intuitive comprehension and evaluation of the generated solution.
While the implementation may have certain limitations but it still offers valuable insights into the TSP and delivers accurate solutions. The visual representation enhances the overall understanding of the problem and aids in assessing the quality of the obtained tour.

6 Test

```
*****************
0 9 8 10 5 8 6
9 0 1 1 2 2 3
8 1 0 1 7 4 2
10 1 1 0 7 7 9
5 2 7 7 0 0 1
8 2 4 7 0 0 3
6 3 2 9 1 3 0
*****************
The cost of most efficient tour = 19
The order of nodes :  [6, 4, 5, 1, 3, 2, 0]
PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-3>
```

Figure 6.1: Output

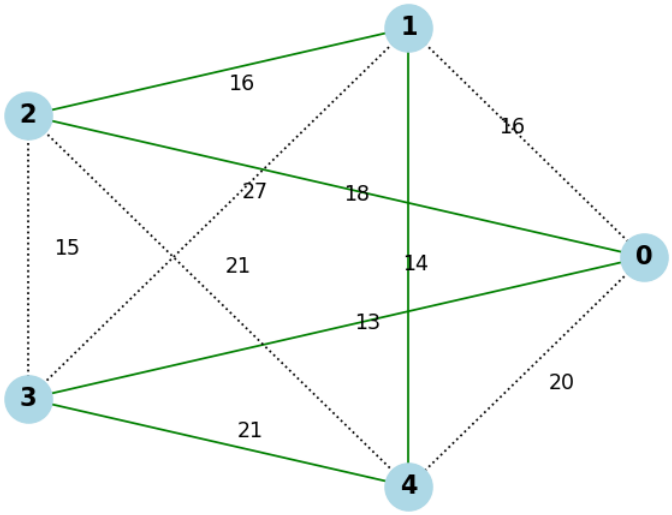

Figure 6.2: Output

```
*****************
0 16 18 13 20
21 0 16 27 14
12 14 0 15 21
11 18 19 0 21
16 14 17 12 0
*****************
The cost of most efficient tour = 69
The order of nodes :  [3, 4, 1, 2, 0]
PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-3>
```

Figure 6.3: Output

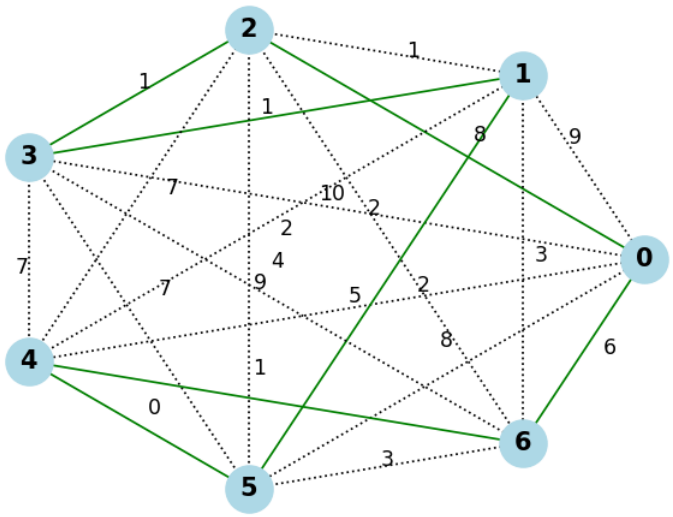

Figure 6.4: Output

```
****************
0 9 3 0 6 10 9 7 9 5 3 6
9 0 2 3 3 3 3 10 1 6 3 8
3 2 0 1 5 10 0 0 0 0 3 1 10
0 3 1 0 1 10 0 2 8 9 8 9
6 3 5 1 0 5 6 6 10 3 8 6
10 3 10 10 5 0 4 1 1 6 0 9
9 3 0 0 6 4 0 10 6 6 9 3
7 10 0 2 6 1 10 0 9 8 7 0
9 1 0 8 10 1 6 9 0 9 10 8
5 6 3 9 3 6 6 8 9 0 3 6
3 3 1 8 8 0 9 7 10 3 0 1
6 8 10 9 6 9 3 0 8 6 1 0
****************
The cost of most efficient tour = 17
The order of nodes :  [3, 4, 9, 10, 11, 7, 5, 8, 1, 6, 2, 0]
PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-3>
```

Close

Figure 6.5: Output

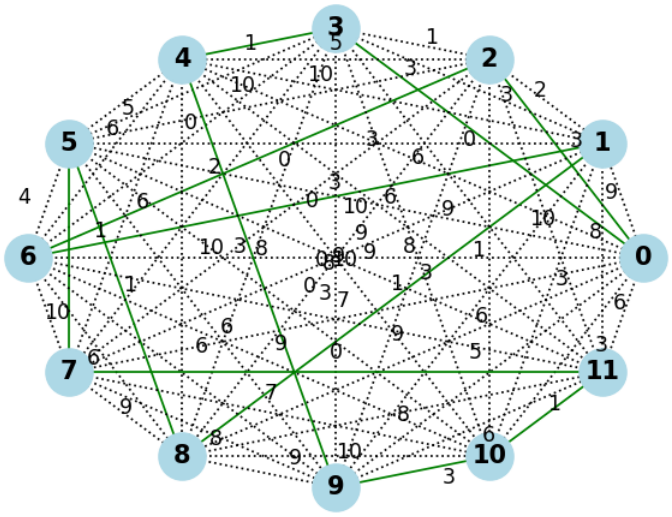

Figure 6.6: Output