Vietnam National University, Ho Chi Minh City
University of Technology
Faculty of Computer Science and Engineering

DISCRETE STRUCTURE REPORT

TRAVELLING SALESMAN PROBLEM

NGUYỄN TẤT KIÊN
STUDENT ID: 2311735

Ho Chi Minh City, May 2024

Table of Contents

1 Travelling Salesman Problem

"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" - from Wikipedia.

Solve Travelling Salesman problem with input being a graph represented as weighted matrices (0 is not connected) with maximum of 26 vertices always be named by using the uppercase alphabet in order where vertex 0_{th} is 'A', 1_{st} is 'B', and so on.

2 Intuition

Initially, think about the backtracking algorithm, which efficiently explores all solutions. This algorithm allows us to backtrack when faced with a dead-end or after trying all feasible options. However, the time complexity of this approach grows rapidly with the number of cities, resulting in $O(n!)$ time complexity (try all permutation of all cities and find the shortest path).

To efficient algorithm, we can see that during backtracking, we often revisit previous shortest paths. For optimizing, we define a two-dimensional array to store these paths. The first dimension corresponds to the current vertex, while the second dimension corresponds a bit mask to represent the set of visited vertices. Through bitwise manipulation, where 1's indicates a visited vertex and 0's indicates an unvisited one. By storing previous shortest path values in array, we don't need recalculations from that optimize the algorithm's performance.

3 Approach

3.1 Why Bitmasking Works: Unlocking TSP Efficiency

Bitmasking is a technique that is used to represent subsets of items or to check, set, clear, and toggle bits in an efficient manner. In the context of the Travelling Salesman Problem (TSP), bitmasking is used to represent the set of visited vertices.

Here's why bitmasking is efficient:

1. Power of bits: During backtracking, we must check that adjacency vertex has visited or not, because it only has two situations (has visited, has not visited), that is why bits come to.

2. Space Efficient: A bit mask uses only one bit per element, making it a very space-efficient way to represent a set. For example, a set of 26 vertices can be represented using just 26 bits. Coincidentally, the problem is limited to a maximum of 26 cities.

3. Time Efficient: Bitwise operations (like AND, OR, XOR, NOT, shift left, and shift right) are very fast, often taking only one CPU cycle. This makes bitmasking a very time-efficient way to manipulate sets.

4. Simplicity: Bitmasking simplifies the code. For example, to check if a vertex i is in the set, you can simply check if the i_{th} bit is 1. To add a vertex i to the set, you can simply set the i_{th} bit to 1. To remove a vertex i from the set, you can simply set the i_{th} bit to 0.

3.2 Dynamic Programming Approach with Bitmasking

1. Definition: The state of dynamic programming solution is defined by two variables (curr and mask), where curr is current visited vertex and mask is a bitmask representing the set of visited vertices. The function calculates and returns the shortest path that starts from the start vertex,visit all vertices in mask and ends at curr.

2. Base Case: As in the previous part, approach is just efficient backtracking algorithm, so we will use recursion for this solution with base case is when all vertices have been visited (all bits in mask are 1's). In this case, the function return the weight of the edge from curr to the start vertex if such an edge exists. If it doesn't, it is not a valid path.

3. Memoization: The function uses a 2D array to store the shortest path for each state. If the shortest path for a state has already been calculated, the function returns the stored value.

4. Iteration and Recursion: The function iterates over all vertices. For each vertex i_{th} that hasn't been visited and is reachable from curr, use recursion and mark that vertex i_{th} has been visited in mask.

5. Update Shortest Path: If a new shortest path has found after recursive call, update shortest path and update previous vertex for each state.

3.3 Time Complexity Analysis

This approach effectively explores all possible paths that start from the start vertex and visit each vertex exactly once, and it uses memoization to avoid redundant calculations. The time complexity is $O(n^2 2^n)$, where n is the number of vertices.

Here is the reason why we have above time complexity:

1. 2^n: There are 2^n possible subsets of the set of all vertices.

2. n: For each subset, function use a loop for all vertices in graph.

3. n: Additionally, for each subset of vertices, the function make a recursive calls.

Multiplying above factors together give time complexity for this solution.

3.4 Difference between only backtracking and backtracking with dynamic programming

Let's call $\Delta u_n = n! - n^2.2^n$ is difference time complexity between only backtracking and backtracking with dynamic programming. We have $\Delta u_{n+1} - \Delta u_n = n.n! - (2n+1).2^n$.

We know that $n!$ increases faster than 2^n when n raise, and $\frac{n}{2n+1}$ has lower bound $\frac{1}{2}$.

So we can see that when n large enough, the difference is absolutely huge.

4 References

[1] A little bit of classics: dynamic programming over subsets and paths in graphs: https://codeforces.com/blog/entry/337