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

Discrete Structures for Computing Large Assignment Travelling Salesman Problem - Using Branch And Bound

Student Name	Student ID
Nguyen Duy Thanh	2353101

June, 2024

Contents

1 Introductory

The Traveling Salesman Problem (TSP) is a classic combinatorial optimization problem. It involves a salesman who must visit a set of cities exactly once, minimizing the total travel distance and returning to the starting city. This problem has numerous real-world applications, including logistics planning, delivery route optimization, and circuit board drilling.

While the TSP can be solved by brute force for small instances, it becomes computationally intractable for larger problems due to the exponential growth of possible routes. This necessitates the use of efficient algorithms like Branch and Bound.
There are two important things to be cleared about in this problem statement:

- Visit every city exactly once.
- Find the shortest path.

2 Ideas

There are several algorithms designed to tackle the Traveling Salesman Problem for example Exact Algorithms (Brute Force, Branch and Bound, ...), Heuristic Algorithms (Nearest Neighbor, Nearest Insertion, ...) , Meta-heuristics (Simulated Annealing, Ant Colony Optimization, ...)

The choice of algorithm for a specific TSP instance depends on factors like problem size, desired accuracy, and computational resources:

- For small problems, exact algorithms like Branch and Bound may be feasible to find the optimal solution.
- For larger problems, heuristic or meta-heuristic approaches often provide a good balance between solution quality and computational efficiency.

For my large assignment i chose Branch and Bound due to its fitting criteria for solving graph with size ≤ 20.

3 What is Branch and Bound?

Branch and Bound is a general-purpose optimization technique that explores the solution space systematically, discarding (bounding) sub-optimal branches that cannot lead to a better solution.

More specifically: For a current node in tree, we compute we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the sub-tree rooted with the node.

A node cost includes two costs:

1. Cost of reaching the node from the root (When we reach a node, we have this cost computed)
2. Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore sub-tree with this node or not).

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution.

3.1 Lower bound

A crucial aspect of Branch and Bound is calculating a lower bound on the cost of any complete tour. This bound guarantees that no solution within a particular branch can be better than the current best solution we have found.
A common lower bound for the TSP is the minimum spanning tree (MST) heuristic. The MST connects all cities with the shortest possible edges, but it doesn't guarantee visiting each city exactly once. However, the total distance of the MST serves as a lower bound on the true TSP solution.

3.2 Branching strategy

Branch and Bound builds a search tree, where each node represents a partial tour.
Branching essentially involves extending the current partial tour by selecting a city not yet visited. This creates new branches in the search tree, each representing a different possibility for completing the tour.

3.3 Bounding function

At each node, a bounding function estimates the minimum possible cost of completing the tour from that point. We use sub-problem Relaxation Techniques - a method similar to the Held-Karp algorithm.

3.4 Pruning Branches

If the lower bound of a partial tour (including the estimated cost to complete) is greater than the current best solution (shortest complete tour found so far), that entire branch can be pruned. This effectively eliminates entire sections of the search tree that cannot lead to a better solution.

3.5 Termination

The algorithm terminates when all branches have been explored or a predetermined time limit is reached. The best complete tour found (with the shortest distance) is considered the optimal solution.

4 Full Code

4.1 Header file

```cpp
#include <iostream>
#include <string>
using namespace std;

// Declarations of functions and variables
bool visited[20] = {false};
int curr_path[20] = {0};
int final_path[20] = {0};
int curr_dist = 0;
int final_dist = 100000;
int MIN = 100000;
void TSPRec(int level, int numberOfVertices, int G[20][20], char startVertex, int MIN);
string Traveling(int G[20][20], int numberOfVertices, char startVertex);
```

Listing 1: C++ code in tsp.h

4.2 C++ file

```cpp
#include "tsm.h"

// Recursive function to solve the Traveling Salesman Problem (TSP)
void TSPRec(int level, int numberOfVertices, int G[20][20], char startVertex, int MIN)
{
    // Loop through unvisited vertices
    for (int i = 0; i < numberOfVertices; i++) {
        // Check if vertex is not visited yet
        if (visited[i] == 0) {
            visited[i] = 1; // Mark the vertex as visited
            curr_path[level] = i; // Add the vertex to the current path

            // Update the current distance by adding the edge weight between the previous and current vertex
            curr_dist += G[curr_path[level - 1]] [curr_path[level]];

            // If all vertices are visited and there's an edge back to the starting vertex
            if (level == numberOfVertices - 1) {
                // Check if a shorter path is found
                if (final_dist > curr_dist + G[curr_path[numberOfVertices-1]][startVertex - 'A']) {
                    // Ensure there's an edge back to the starting vertex and from the previous vertex
                    if (G[curr_path[numberOfVertices-1]][startVertex - 'A'] != 0
                        && G[curr_path[level - 1]][curr_path[level]] != 0) {
                        final_dist = curr_dist + G[curr_path[numberOfVertices-1]][startVertex - 'A'];

                        for (int j = 0; j < numberOfVertices; j++) {
                            // Update the final path with the current path
                            final_path[j] = curr_path[j];
                        }
                    }

                }
            }
            else if (curr_dist + (numberOfVertices)*MIN  < final_dist) {
                // Prune branches if the current distance + lower bound is already higher than the best solution found so far
                if (G[curr_path[level-1]][curr_path[level]] != 0)
                    TSPRec(level + 1, numberOfVertices, G, startVertex, MIN);
            }

            // Backtrack after exploring this path
            visited[i] = 0;
            curr_dist -= G[curr_path[level - 1]][curr_path[level]];
        }
    }
}
```

```
45
46 string Traveling(int G[20][20], int numberOfVertices, char startVertex)
47 {
48     // Find the minimum weight edge from each vertex
49     for (int i = 0; i < numberOfVertices; i++) {
50         for (int j = 0; j < numberOfVertices; j++) {
51             if (G[i][j] != 0)
52                 MIN = min(MIN, G[i][j]);
53         }
54     }
55
56     // Initialize starting path and visited state
57     curr_path[0] = startVertex - 'A';
58     visited[startVertex - 'A'] = 1;
59
60     // Start the recursive TSP search from the starting vertex
61     TSPRec(1, numberOfVertices, G, startVertex, MIN);
62
63     // Print the final path string
64     string ans = "";
65     for (int i = 0; i < numberOfVertices; i++) {
66         ans += string(1,'A' + final_path[i]) + " ";
67     }
68     ans += string(1,startVertex);
69
70     // Resetting variables, starting path, final path and visited state
71     for (int i = 0; i < numberOfVertices; i++) {
72         curr_path[i] = 0;
73         final_path[i] = 0;
74         visited[i] = 0;
75     }
76     curr_dist = 0;
77     final_dist = 100000;
78     return ans;
79 }
```

Listing 2: C++ code in tsp.cpp

5 Code explanation

5.1 Overview

This code implements a Branch-and-Bound algorithm to solve the TSP. It explores possible paths recursively, keeping track of the current distance and a lower bound on the remaining distance. The lower bound helps prune unpromising paths that cannot lead to a shorter solution overall. The recursive calls explore different branches of the search tree, backtracking when necessary. Finally, the function returns the shortest path found during the exploration.

5.2 Recursive Function: TSPRec

This function (TSPRec) is the core of the algorithm and performs the recursive exploration to find the shortest path. Let's break down its arguments and functionality:
Arguments:

1. int level: The current level in the recursion, representing the number of vertices visited so far (starts at 1).
2. int numberOfVertices: The total number of vertices in the graph.
3. int G[20][20]: A 2D array representing the adjacency matrix of the graph. G[i][j] stores the distance between vertex i and vertex j.
4. char startVertex: The starting vertex (represented as a character).
5. int MIN: The minimum edge weight found in the graph (used for bounding).

Steps:

1. Iterate through Unvisited Vertices:
   - Loops through all vertices (i) in the graph using a loop from 0 to numberOfVertices - 1.
   - Checks if the current vertex i is not visited yet (visited[i] == 0).
2. Mark Visited and Update Path:
   - If i is not visited, mark it as visited (visited[i] = 1). Add vertex i to the current path (curr_path[level] = i).
   - Update the current distance (curr_dist) by adding the weight of the edge from the previous vertex (curr_path[level-1]) to the current vertex (i) using G[curr_path[level - 1]][curr_path[level]].
3. Base Case - Check for Complete Path:
   - If the current level (level) is equal to numberOfVertices - 1 (all vertices have been visited):
     - Check if there's an edge back to the starting vertex (G[curr_path[numberOfVertices-1]][startVertex - 'A'] != 0). This ensures a valid closed path returning to the starting point.
     - If there's an edge back and both edges leading to and from the current vertex exist (G[curr_path[level - 1]][curr_path[level]] != 0), calculate the total distance for this complete path (curr_dist + G[curr_path[numberOfVertices-1]][startVertex - 'A']).
     - Compare the total distance of the complete path with the current best distance (final_dist). If it's shorter, update final_dist and copy the current path (curr_path) to the final_path array.
4. Bounding and Recursion:
   - If it's not a complete path yet (level != numberOfVertices - 1), perform bounding:
     - Calculate a lower bound on the remaining distance using curr_dist + (numberOfVertices - level) * MIN. This assumes the minimum possible distance for the remaining unvisited vertices is MIN.
     - If the lower bound (curr_dist + (numberOfVertices - level) * MIN) is less than the current best distance (final_dist), there's a chance this path could lead to a shorter solution. So, explore further by calling the TSPRec function recursively.
     - Pass the updated arguments: level + 1 (next level), numberOfVertices, G, startVertex, and MIN.
5. Backtracking:
   - After exploring a branch (path) recursively, backtrack:
     - Mark the current vertex i as unvisited again (visited[i] = 0).
     - Remove the weight of the edge from the previous vertex to the current vertex from the current distance (curr_dist -= G[curr_path[level - 1]][curr_path[level]]).

5.3 Traveling function:

1. Find Minimum Edge Weight:
   - Iterates through the adjacency matrix (G) to find the minimum edge weight (MIN) in the entire graph.
2. Initialize Variables:
   - Sets the starting vertex position in the current path (curr_path[0] = startVertex - 'A').
   - Marks the starting vertex as visited (visited[startVertex - 'A'] = 1).
3. Start Recursive Exploration:
   - Call TSPRec: The Traveling function initiates the recursive exploration by calling the TSPRec function
4. Once the recursive exploration (TSPRec) finishes, the Traveling function prepares the output string (ans) containing the vertices in the shortest path.
5. Reset for Next Run:
   - Reset curr_path, final_path, and visited to 0 or false for all elements.
   - Reset curr_dist and final_dist to 0 and a high value, respectively.

6 Bibliography

References

[1] Ideas on how to approach the traveling salesman problem

[2] Articles on TSP

[3] Video goes in-depth about Pseudo code for TSP