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:

- Cost of reaching the node from the root (When we reach a node, we have this cost computed)
- 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

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

// Declare functions
void copy(int final_path[], int curr_path[], int N);
string PrintResult (int final_path[], char startVertex, int numberofVertices);
int firstMin (int G[][20], int i, int numberofVertices);
int secondMin (int G[][20], int i, int numberofVertices);
void TSPRec (int G[][20], int numberofVertices, int curr_bound, int curr_weight, int level, int & ↩
    final_res, int curr_path[], int final_path[], bool visited[]);
string Traveling (int G[][20],int numberofVertices, char startVertex);
```

Listing 1: C++ code in tsp.h

4.2 C++ file

```
// Define a constant for the maximum possible distance
const int MAX_N = 100000000;

// Function to copy an array (used for copying the final path)
void copy(int final_path[], int curr_path[], int N)
{
    for (int i = 0; i < N; i++)
        final_path[i] = curr_path[i];
    final_path[N] = curr_path[0];
}

// Function to format and return the final path as a string
string PrintResult (int final_path[], char startVertex, int numberofVertices)
{
    string ans = "";
    for (int i = 0; i < numberofVertices; i++) {
        char c = final_path[i] + 'A';
        ans = ans + c + " ";
    }
    ans = ans + startVertex;
    return ans;
}

// Function to find the minimum weight edge from a vertex (excluding itself)
int firstMin (int G[][20], int i, int numberofVertices)
{
    int min = MAX_N;
    for (int j = 0; j < numberofVertices; j++) {
        if (G[i][j] < min && i != j) {
            min = G[i][j];
        }

    }
    return min;
}

// Function to find the second minimum weight edge from a vertex (excluding itself)
int secondMin (int G[][20], int i, int numberofVertices)
{
    int first = MAX_N, second = MAX_N;
    for (int j = 0; j < numberofVertices; j++)
    {
        if (i == j)
            continue;

        if (G[i][j] <= first)
        {
```

```
            second = first;
            first = G[i][j];
        }
        else if (G[i][j] <= second && G[i][j] != first)
            second = G[i][j];
    }
    return second;
}

// Recursive function to solve the Traveling Salesman Problem (TSP)
void TSPRec (int G[][20], int numberofVertices, int curr_bound, int curr_weight, int level,
             int & final_res, int curr_path[], int final_path[], bool visited[])
{
    // If all vertices are visited and there's an edge back to the starting vertex
    if (level == numberofVertices) {
        if (G[curr_path[level-1]][curr_path[0]] != 0) {
            int curr_res = curr_weight + G[curr_path[level-1]][curr_path[0]];
            if (curr_res < final_res) {
                copy(final_path, curr_path, numberofVertices);
                final_res = curr_res;
            }
        }
        return;
    }

    // Loop through unvisited vertices
    for (int i = 0; i < numberofVertices; i++) {
        if (G[curr_path[level-1]][i] != 0 && visited[i] == false) {
            int temp = curr_bound;

            // Update current weight by adding the edge weight
            curr_weight += G[curr_path[level-1]][i];

            // Calculate a lower bound for the remaining path based on minimum edge weights
            if (level == 1)
                curr_bound -= ((firstMin(G,curr_path[level-1],numberofVertices)
                                + firstMin(G,i,numberofVertices))/2);
            else
                curr_bound -= ((secondMin(G,curr_path[level-1],numberofVertices)
                                + firstMin(G,i,numberofVertices))/2);

            // Prune branches if the current bound (estimate) + current weight is already higher than the best solution found so far
            if (curr_bound + curr_weight < final_res) {
                // Explore this path further as it's a promising candidate
                curr_path[level] = i;
                visited[i] = true;
                // Recursively explore the remaining path starting from vertex 'i'
                TSPRec(G, numberofVertices, curr_bound, curr_weight, level+1, final_res, curr_path, final_path , visited);
            }

            // Backtrack and undo changes if this path doesn't lead to a better solution
            curr_weight -= G[curr_path[level-1]][i]; // Remove the weight added from visiting vertex i
            curr_bound = temp; // Restore the previous bound
            for (int j = 0; j < numberofVertices; j++)
                visited[i] = false;

            for (int j = 0; j <= level - 1; j++)
                visited[curr_path[j]] = true; // Restore visited state for vertices in the current path
        }
    }
}

string Traveling (int G[][20],int numberofVertices, char startVertex)
{
    int curr_path[numberofVertices + 1] = {-1};
    int final_path[numberofVertices + 1];
    bool visited[numberofVertices] = {false};
    int final_res = MAX_N;
    int curr_bound = 0;

    // Calculate a lower bound for the entire journey based on minimum edge weights from each vertex
```

```
    for (int i = 0; i < numberofVertices; i++)
        curr_bound += (firstMin(G,i,numberofVertices)
                       + secondMin(G,i,numberofVertices));

    // Ensure the lower bound is always an integer (round up if odd)
    curr_bound = (curr_bound&1)? curr_bound/2 + 1 : curr_bound/2;
    visited[startVertex - 'A'] = true;
    curr_path[startVertex - 'A'] = 0;

    // Start the recursive TSP search from the starting vertex
    TSPRec(G, numberofVertices, curr_bound, 0, 1, final_res, curr_path, final_path ,visited);
    // Convert the final path array into a readable string representing the visited vertices
    return PrintResult(final_path, startVertex, numberofVertices);
}
```

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 Important Constants and Variables

MAX_N: This constant represents the maximum possible distance, set to a large value (100,000,000) to handle most cases.

5.3 Helper Functions

- The function *copy* (*int final_path*[], *int curr_path*[], *int N*): This function copies the elements of the curr_path array, which represents the current travel path, into the final_path array. It also adds the starting vertex at the end to complete the cycle.
- The function *PrintResult* (*int final_path*[], *char startVertex*, *int numberofVertices*): This function takes the final path array, starting vertex character, and number of vertices. It iterates through the final_path array, converting the integer values (representing vertex positions) to characters (A, B, C, etc.) and building a string representation of the visited vertices in the order of the path.
- The function $firstMin$ (*int G*[][20], *int i*, *int numberofVertices*): This function finds the minimum weight edge (distance) from a given vertex i to any other vertex in the graph G. It excludes the vertex itself (i != j) and iterates through all connected vertices, returning the minimum edge weight.
- The function $secondMin(intG[][20], inti, intnumberofVertices)$: This function finds the second minimum weight edge from a given vertex i to any other vertex, similar to firstMin. However, it keeps track of both the first and second minimum values during the iteration.

5.4 Recursive TSP Function

Variables

TSPRec (int G[][20], int numberofVertices, int curr_bound, int curr_weight, int level, int & final_res, int curr_path[], int final_path[] , visited[]): This is the core function that recursively solves the TSP problem. It takes several arguments:

- G: The adjacency matrix representing the graph, where G[i][j] represents the distance between vertex i and vertex j.
- numberofVertices: The total number of vertices in the graph.
- curr_bound: The current lower bound on the total path distance.
- curr_weight: The current weight (distance traveled) in the current path exploration.
- level: The current level in the recursion, representing the number of vertices visited so far (starts at 1).
- final_res (reference): A reference to the variable storing the minimum total distance found so far.
- curr_path: An array representing the current path being explored.
- final_path: An array to store the final shortest path found. visited: A boolean array to track visited vertices.

5.5 Base Case

The function checks if all vertices have been visited (level == numberofVertices). If yes, it also checks if there's an edge back to the starting vertex (G[curr_path[level-1]][curr_path[0]] != 0). If so, it calculates the total distance for this complete path and compares it with the current final_res. If it's a shorter path, it updates final_res and copies the current path to final_path.

5.6 Recursive Exploration

- The function iterates through all unvisited vertices (i) connected to the current vertex (curr_path[level-1]).
- It checks if the vertex i is not visited (visited[i] == false).
- If it's a valid vertex to visit:
  - A temporary variable temp stores the current curr_bound.
  - The current weight (curr_weight) is updated by adding the weight of the edge from the current vertex to vertex i.
  - Depending on the current level (level):
    + If it's the first level (starting vertex), a lower bound (curr_bound) is calculated using firstMin for both the current vertex and vertex i. This estimates the minimum possible distance by considering the two smallest edges from each vertex.
    + If it's not the first level, a lower bound is calculated using secondMin for the current vertex and firstMin for vertex i. This is because the first minimum edge from the current vertex has already been used in the path.
  - The function checks if the calculated lower bound (curr_bound + curr_weight) is less than the current final_res. This pruning step helps avoid exploring paths that cannot lead to a shorter solution.
- If the lower bound is promising (curr_bound + curr_weight < final_res), the function progresses further down the exploration path:
  - The vertex i is added to the current path (curr_path[level] = i).
  - The visited flag for vertex i is set to true to mark it as visited.
  - The TSPRec function is called recursively with updated parameters:
    + Same G, numberofVertices, and visited arrays. Updated curr_bound, curr_weight, and level (incremented by 1).
    + Reference to the same final_res to keep track of the minimum distance found so far.
    + Updated curr_path array reflecting the current exploration path.
    + Same final_path array (not modified within the recursive call).
- Backtracking:
  - After the recursive call explores a particular branch, it might not lead to the shortest path. So, the code backtracks
    + The weight of the edge from the current vertex to vertex i is removed from curr_weight.
    + The curr_bound is restored to its previous value stored in temp.
    + The visited flag for vertex i is set back to false to indicate it's unvisited again.
    + The visited flags for all vertices in the current path (curr_path[0] to curr_path[level-1]) are set back to true to maintain the visited state for the partially explored path.
- Main Traveling Function:
  - Initializes arrays and variables:
    + curr_path: An array to store the current path during exploration (size numberofVertices + 1).
    + final_path: An array to store the final shortest path found (size numberofVertices + 1).
    + visited: A boolean array to track visited vertices (size numberofVertices).
    + final_res: Set to MAX_N to represent a very high initial distance.
    + curr_bound: Set to 0 initially.
  - Calculates a lower bound (curr_bound) for the entire journey. It iterates through all vertices and finds the sum of the first and second minimum edges from each vertex using firstMin and secondMin. This provides an estimate of the shortest possible distance.
  - Ensures the lower bound is an integer (rounded up if odd) using bitwise operations.
  - Marks the starting vertex as visited (visited[startVertex - 'A'] = true) and sets its position in the curr_path array.
  - Calls the recursive TSPRec function to start the exploration from the starting vertex:
    + Provides G, numberofVertices, curr_bound, curr_weight (0), level (1), reference to final_res, curr_path, final_path, and visited arrays.
  - Finally, the function calls PrintResult to convert the final path stored in final_path into a human-readable string representing the visited vertices in the order of the shortest path.

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