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


DISCRETE STRUCTURE

Assignment 1

# Using Branch And Bound Algorithm To Solve The Travelling Saleman Problem

**Instructors:** Nguyen Van Minh Man, *Mahidol University*
Tran Tuan Anh, *CSE-HCMUT*

**Author:** Truong Gia Ky Nam
**ID:** 2352787
**Email:** nam.truonggiaky@hcmut.edu.vn

**Ho Chi Minh City, May 2024**

**Abstract**

# Contents

**1 Introduction** **1**

**2 The Traveling Salemen Problem** **1**
2.1 Overview . . . . . 1
2.2 Traveling Saleman Problem implementation . . . . . 1
2.3 Traveling Saleman Problem solving methods . . . . . 1

**3 Branch and Bound Algorithm** **2**
3.1 Overview . . . . . 2
3.2 Some basic concepts . . . . . 2
3.3 Applications of Branch and Bound . . . . . 2
3.4 Advantages and Disadvantages . . . . . 2

**4 Detailed Explanation of the Algorithm** **3**
4.1 Initialization . . . . . 3
4.2 Branching . . . . . 3
4.3 Bounding . . . . . 3
4.4 Pruning . . . . . 4
4.5 Termination . . . . . 4

**5 Implementation Details** **4**

**6 Complexity Analysis** **4**
6.1 Time Complexity . . . . . 4
6.2 Space complexity . . . . . 4

**7 Experimental Results** **4**

**8 Challenges and Improvements** **4**

**9 Conclusion** **4**

# 1 Introduction

This report aims to provide a comprehensive explaination of The Traveling Salemen Problem along with the Branch and Bound algorithm. By delving into the Branch and Bound algorithm and implement it using the C++ language, this report seek to highlights its effectiveness in solving one of the most challenging problems in optmization.

# 2 The Traveling Salemen Problem

## 2.1 Overview

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?"* or in other words, the problem is: *"Given a complete graph with weight edges, what is the Hamiltonian cycle of minimum cost?*. It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.

## 2.2 Traveling Saleman Problem implementation

Although TSP is a problem in computer science, it has a wide range of applications in various fields:

- **Logistics and Supply Chain Managment**: TSP can determine the most efficient routes for delivery transportation, reducing travel time and fuel consumption,which can help the companies to save lots of money.
- **Manufacturing**: In manaufacturing processes involving CNC machines, TSP helps in determining the optimal path for tools, reducing the time and cost of production.
- **Transportation**: TSP can help public transport such as buses or trains to minimize travel time and improve service efficiency.
- **Telecommunications**: TSP is used in the design of efficient telecommunication networks, ensuring that the data transmission routes are optimized to reduce latency and cost.

By solving the TSP, we can address real-world challenges more effectively, leading to substantial economic, operational, and environmental benefits.

## 2.3 Traveling Saleman Problem solving methods

Throught out the history, many methods have been made to find the optimal solution for TSP:

- **Brute-Force Search**: This is the straight-forward solution by computing all possible permutations of the cities until the shortest path is found.
- **Nearest Neighbor Algorithms**: Start from the origin city, repeatedly visit the nearest unvisited city until all cities are visited.
- **Held-Karp Algorithm**: This approach using dynamic programming to solve TSP. It is more efficient than brute force but still exponential.
- **Christofides's Algorithm**: This algorithm involves constructing a minimum spanning tree, finding a minimum matching, and combining these to form a tour.
- **Genetic Algorithms**:These use principles of natural selection and genetics to evolve a population of solutions over several generations, aiming to improve the quality of solutions.

And there are others algorithm and heuristics which can be used to solve TSP. Each method has its advantages and disadvantages, depending on the problem size, required solution quality, and computational resources. In my report, I will focusing on solving TSP using the **Branch and Bound** algorithm, a well used algorithm for optmization problems.

# 3 Branch and Bound Algorithm

## 3.1 Overview

Branch and bound is a method used in computer science to solve optimization problems. Branch and bound algorithms involves dividing the problem space into smaller sub-problems and applying constraints or bounds to eliminate certain sub-problems. This process continue until the optimal solution is found.

## 3.2 Some basic concepts

Before getting into the algorithms, first I need to declare some basic concepts:

- **Branching**: This is the process where the big problem are divided into smaller subproblems (branches) that easy to solve.
- **Bounding**: Calculating the lower bound (for minimization problems) or an upper bound (for maximization problems) on the optimal solution within a branch.
- **Pruning**: "Cutting" the branches that can not yield a better solution than the current best-known solution.
- **Search Tree**: The structure used to represent the branching of subproblems. Each node in the tree represents a subproblem.
- **Backtracking**: Return to the previous node to explore other branches after completing the exploration of a branch

## 3.3 Applications of Branch and Bound

Beside solving TSP, Branch and Bound algorithm has many applications. It includes:

- **Solving Knapsack Problem**: Branch and Bound is used to solve the Knapsack Problem, which involves finding the optimized combination of items to pack into a knapsack of limited capacity.
- **Resource Allocation**: Branch and Bound is used to solve resource allocation problems, like scheduling work on machines or assigning work to workers.
- **Network Optimization**: Branch and bound is used to solve network optimization problems, it helps in finding the optimized path or flow through a network.
- **Game Playing**: Branch and bound is used in some of the game-playing algorithms, like chess or tic-tac or 16 puzzle problem, to explore the various possible moves and find the optimized strategies.

## 3.4 Advantages and Disadvantages

There are advantages when using this algorithms:

- By pruning the search space using bounds, it can avoid exploring irrelevant sub-problems and find the optimal solution more quickly when compare to other algorithms.
- There is a guarantees that the optimal solution will be found, as long as the bounds are computed correctly. This makes it a reliable algorithm for solving optimization problems.
- Branch and bound can be applied to a wide range of optimization problems, including linear programming, integer programming, and combinatorial optimization problems.
- Branch and bound can be used to solve large-scale optimization problems, as it can be parallelized and distributed across multiple processors.

However, there are some drawbacks when using Branch and Bound algorithms

- Branch and bound can be a complex algorithm to implement, as it requires careful design of the bounds and branching strategy.

- The performance of branch and bound is highly dependent on the quality of the bounds used. If the bounds are too loose, the algorithm may explore too many irrelevant sub-problems, leading to poor performance.

# 4 Detailed Explanation of the Algorithm

## 4.1 Initialization

## 4.2 Branching

## 4.3 Bounding

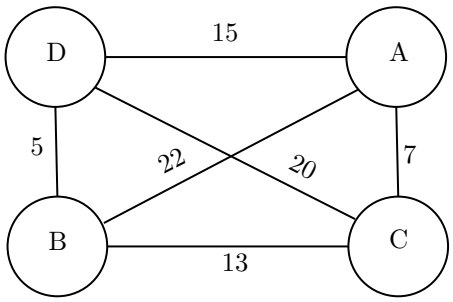
To use the Branch and Bound algorithms, first we need to compute the bound (in TSP is the lower bound) on the best solution. We can find the bound of any path by the following idea:

For any tour T, the cost of any tour can be written as:

$$\text{Tour cost} = \frac{1}{2} \times (\text{Sum of cost of two edges adjacent to u and in the tour T})$$

Because for every vertex u, there are two edges through it in T and sum their costs. The overall sum for all vertices would be twice of cost of tour T.

For a clearer explaination, I will create an small example of TSP with 4 cities




Consider the tour $T_1$: A - C - D - B - A, the toal cost of this tour is:

$$T_1 = 7 + 20 + 5 + 22 = 54$$

In tour $T_1$, the edges that adjacent to vertex A have the weight 7 and 22. Similarly for vertex B are 22 and 5; vertex C are 20 and 7; vertex D are 5 and 20. Using the above idea:

$$T_1 = \frac{1}{2} \times (7 + 22 + 22 + 5 + 20 + 7 + 5 + 20) = 54$$

Or consider tour $T_2$: B - C - A - D - B, the total cost of this tour is:

$$T_2 = 13 + 7 + 15 + 5 = 40$$

In tour $T_2$, the edges that adjacent to vertex B have the weight 13 and 5. Similarly for vertex C is 13 and 7; vertex A is 15 and 7; vertex D is 5 and 15. Using the above idea:

$$T_2 = \frac{1}{2} \times (13 + 5 + 13 + 7 + 15 + 7 + 15 + 5) = 40$$

For every vertex in the graph, we have:

$$(\text{Sum of two tour edges adjacent to u}) \geqslant (\text{Sum of minimum weight two edges adjacent to u})$$

From that, we have:

$$\text{Cost any tour} \geqslant \frac{1}{2} \times (\text{Sum of minimum weight two edges adjacent to u})$$

Consider the above graph, we have:

| Node | Minimum cost edges | Total cost |
|---|---|---|
| A | (A,D);(A,C) | 22 |
| B | (B,D);(B,C) | 18 |
| C | (C,B);(C,A) | 20 |
| D | (D,B);(D,A) | 20 |

The lower bound of the cost of any tour in the given graph is:

$$t = \frac{1}{2} \times (22 + 18 + 20 + 20) = 40$$

So now we need functions find the first and second minimum edge cost having an end at vertex u to calculate the lower bound of any node

### 4.4 Pruning

### 4.5 Termination

## 5 Implementation Details

## 6 Complexity Analysis

The Branch and Bound algorithm for solving the Traveling Salesman Problem (TSP) can be analyzed in terms of both time and space complexity.

### 6.1 Time Complexity

The time complexity of Branch and Bound algorithm to solving TSP is analyze as below

- Finding two minimum values for each city requires $O(n^2)$ time in total (each call to 'findTwoMin' function is $O(n)$, and it's called $n$ times)
- In the worst-case scenario, the time complexity is $O(n!)$ where n is the number of cities. This is because the algorithm have to explore all permutations of cities. However, due to the pruning procedure, the real performance is much better.
- Each node exploration in the search tree involves computing the bound and making decisions based on the bound, which is $O(n)$

Overall, the worst-case time complexity is factorial term $O(n!)$ but with effective pruning technique, the performance is often better.

### 6.2 Space complexity

## 7 Experimental Results

## 8 Challenges and Improvements

## 9 Conclusion

```
1  void findTwoMin(int G[20][20], int numCities, int i, int &first, int &second)
2  {
3      first = INT_MAX, second = INT_MAX;
4      for (int j = 0; j < numCities; j++)
5      {
6          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];
        }
    }
}
```

```
void branchAndBoundTSP(int G[20][20], int numCities, int (&curr_path)[21], int &curr_bound,
int curr_weight, int level, bool (&visited)[20], int &min_cost, int (&final_path)[21])
{
    if (level == numCities)
    {
        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 < min_cost)
            {
                for (int i = 0; i < numCities; i++)
                {
                    final_path[i] = curr_path[i];
                }
                final_path[numCities] = curr_path[0];
                min_cost = curr_res;
            }
        }
    }
    else
    {
        for (int i = 0; i < numCities; i++)
        {
            if (G[curr_path[level - 1]][i] != 0 &&
                visited[i] == false)
            {
                int temp = curr_bound;
                curr_weight += G[curr_path[level - 1]][i];

                int firstMinCurr, secondMinCurr;
                findTwoMin(G, numCities, curr_path[level - 1], firstMinCurr, secondMinCurr);
                int firstMinNext, secondMinNext;
                findTwoMin(G, numCities, i, firstMinNext, secondMinNext);

                curr_bound -= ((secondMinCurr + firstMinNext) / 2);

                // curr_bound -= ((secondMin(G, numCities, curr_path[level - 1]) + firstMin(G, numCities, i)) / 2);

                if (curr_bound + curr_weight < min_cost)
                {
                    curr_path[level] = i;
                    visited[i] = true;

```

```
                    branchAndBoundTSP(G, numCities, curr_path, curr_bound, curr_weight, level + 1, visited, min_cost, fir
                }

                curr_weight -= G[curr_path[level - 1]][i];
                curr_bound = temp;

                for (int i = 0; i < numCities + 1; i++)
                {
                    visited[i] = false;
                }

                for (int j = 0; j <= level - 1; j++)
                {
                    visited[curr_path[j]] = true;
                }
            }
        }
    }
}
```

```
string Traveling(int G[20][20], int numCities, char startVertex)
{

    int curr_path[21] = {-1};
    bool visited[20] = {false};
    int final_path[21] = {-1};
    int min_cost = INT_MAX;
    int curr_bound = 0;
    string ans;

    for (int i = 0; i < numCities; i++)
    {
        int first, second;
        findTwoMin(G, numCities, i, first, second);
        curr_bound += (first + second);
        // curr_bound += (firstMin(G,numCities,i) + secondMin(G,numCities,i));
    }

    if (curr_bound % 2 == 0)
    {
        curr_bound = curr_bound / 2;
    }
    else
    {
        curr_bound = curr_bound / 2 + 1;
    }

    visited[int(startVertex - 'A')] = true;
    curr_path[0] = int(startVertex - 'A');

    branchAndBoundTSP(G, numCities, curr_path, curr_bound, 0, 1, visited, min_cost, final_path);

    for (int i = 0; i <= numCities; i++)
    {
        ans += char(final_path[i] + 'A');
        ans += " ";
    }

```

```
39      return ans;
40  }
```

# References

[1] CVX Introduction "**link: http://cvxr.com/cvx/doc/intro.html/**", *What is CVX*.