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

The Traveling Salesman Problem is a classic optimization challenge in which the goal is to determine the shortest possible route that allows a salesman to visit each city exactly once and return to the origin city. This report explores the application of the Branch and Bound algorithm to solve the problem. The Branch and Bound algorithm is a combinatorial optimization method that systematically explores branches of a solution space, eliminating suboptimal solutions through bounding techniques to significantly reduce the search area. By searching information on the Internet, I am able to gather required information to make the programme and explaining it in this report.

Contents

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 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.

3.3 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.

Although there are still some drawbacks of Branch and Bound algorithm, but with carefully designation, these drawbacks can be removed.

4 Detailed Explanation of the Algorithm

To see the full source code, take a look at these files: *tsm.cpp* and *tsm.h*

4.1 Finding two minimum weight edges

The *findTwoMin* function is designed to identify the two smallest weights of edges that terminate at every vertex in the graph. These 2 values are crucial for calculating the bound for different vertex in the current consider path.

4.1.1 Function Signature

```
//Finding two minimum edges function having an end at every vertex
void findTwoMin(int G[20][20], int numCities, int firstMin[20], int secondMin[20]);
```

This function take in parameters:

- **int G[20][20]**: The adjacency matrix of the graph.
- **int numCities**: The number of vertices on the graph.
- **int firstMin[20]**: An array to store the minimum edge weight for each vertex. In the array *firstMin[i]* will store the smallest edge weight connected to node 'i'.
- **int secondMin[20]**: An array to store the second minimum edge weight for each vertex. In the array, *secondMin[i]* will store the second smallest edge weight connected to node 'i'.

4.1.2 Function Logic

First, initialize all of the first and second minimum weight of each vertex to be very large to ensure that all weights are smaller.

```
for(int i = 0; i < numCities; i++){
    firstMin[i] = 100000;
    secondMin[i] = 100000;

    /*The other parts */
}
```

Next, loop through all possible connected edges ending at vertex 'j' and update the 'first' with smallest edge weight.

```
//Find the first minimum value
for(int j = 0; j < numCities; j++){
    if(i == j) continue;
    if(firstMin[i] >= G[i][j]) firstMin[i] = G[i][j];
}
```

After finding the minimum value, loop through all edges again, update the 'second' variable with the smallest edge weight which is greater or equal than the 'first' variable.

```
//Find the second minimum value
for(int j = 0; j < numCities; j++){
    if(i == j) continue;
    if(secondMin[i] >= G[i][j] && G[i][j] >= firstMin[i]) secondMin[i] = G[i][j];
}
```

4.2 Exploration function

The *branchAndBoundTSP* function implements the Brand and Bound algorithm to solve TSP. This function explores all possible paths from the from the current node, recursively building a search tree to find the optimal path with minimum cost.

4.2.1 Function Signature

```
void branchAndBoundTSP(int G[20][20], int numCities, int (&currentPath)[21], int &currentBound, int currentCost
, int level, bool (&visited)[20], int &finalCost, int (&finalPath)[21], int firstMin[20], int secondMin[20]);
```

The function take in parameters:

- **int G[20][20]**: The adjacency matrix of the graph.
- **int numCities**: The number of vertices on the graph.
- **int (&currentPath[21])**: An array with size 21 representing the current path being explored.
- **int &currentBound**: The current lower bound of the cost for the path being explored.
- **int currentCost**: The current weight of the path being explored.
- **int level**: The current level in the search tree.
- **bool (&visited)[20]**: A boolean array indicating whether a vertex has been visited or not.
- **int &finalCost**: The minimum cost found so far for finding the optimal path
- **int (&finalPath)[21]**: An array reprenting the path with the minimum cost found so far.
- **int firstMin[20]**: An array storing the minimum weight edges for each vertex.
- **int secondMin[20]**: An array storing the second minimum weight edges for each vertex.

4.2.2 Function Logic

First, iterate through all vertices to explore all possible routes.

```
//Iterate for all vertices to build the tree recursively
for (int i = 0; i < numCities; i++)
```

At each considered vertex, check if that vertex can be visited or not. Specifically, check if the next vertex 'i' is not visited and there is an edge between the last vertex of the current path and the next vertex 'i'.

```
//Consider the next vertex
//if it is not visited
//and it connects to the last vertex within the current path
if (G[currentPath[level - 1]][i] != 0 && visited[i] == false)
```

Store the current bound in temporary variable for backtracking to explore other routes.

```
//Using a temporary to store the current bound for backtracking
int temp = currentBound;
```

As the connected edge is found, add the weight of that edge to the current path's cost.

```
//Add the weight of the consider edges to the current path's cost
currentCost += G[currentPath[level - 1]][i];
```

Also, reduce the current bound based on the two minimum weight edges for the current and next vertices.

```
//Different computation for level 1 from others
if(level == 1){
    currentBound -= (firstMin[currentPath[level - 1]] + firstMin[i])/2;
} else {
    currentBound -= (secondMin[currentPath[level - 1]] + firstMin[i])/2;
}
```

There is a different in calculation between level 1 and the others level. At the initial level of the search tree, the current bound subtract the average of the smallest edge costs connected to the starting node and the next node ‘i’. But in the others level, as the minimum weight has been used in the previous level, now the the current bound have to subtract the average of the second smallest edge of the current node and the smallest edge of the next node.

Now the actual lower bound is the sum of the current bound and the current cost. Check that bound is less than the minimum cost found so far or not.

```
//Check if the actual lower bound smaller than the minimum cost, explore the node further
if (currentBound + currentCost < finalCost)
```

If so, update the current path and mark the vertex as visited, then recursively call the function to explore this node further

```
//Add the current node to the current path
currentPath[level] = i;
visited[i] = true;

//Continue to explore recursively
branchAndBoundTSP(G, numCities, currentPath, currentBound, currentCost, level + 1, visited, finalCost, finalPath);
```

If the exploration can not proceed further, reset the current cost and bound and update the visited array to backtrack and explore other possible paths.

```
//Else prune the node by resetting all change to current cost and current bound
//Or if the exploration success, backtracking to explore other solutions
currentCost -= G[currentPath[level - 1]][i];
currentBound = temp;

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

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

When all vertices have been visited, the function checks if the current explore path is less than the minimum cost found so far. If so, updates the final path and continue to explore other possible routes.

```
//Base case: When the search tree has reached the level N
//which mean the algorithm has covered all nodes
if (level == numCities)
```

```
{
    //Check if there is a path from last vertex in the current path to the start vertex
    if (G[currentPath[level - 1]][currentPath[0]] != 0)
    {
        //The total weight of the current path
        int currentResult = currentCost + G[currentPath[level - 1]][currentPath[0]];

        //Update the final cost and final path
        //if the current path is more optimal
        if (currentResult < finalCost)
        {
            //Make the current path is the final path
            for (int i = 0; i < numCities; i++)
            {
                finalPath[i] = currentPath[i];
            }
            finalPath[numCities] = currentPath[0];
            finalCost = currentResult;
        }
    }
}
```

4.3 Traveling function

The *Traveling* function is designed to call the *branchAndBoundTSP* function to return the optimal solution for the TSP.

4.3.1 Function Signature

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

This function take in parameters:

- **int G[20][20]**: The adjacency matrix of the graph.
- **int numCities**: The number of vertices on the graph.
- **char startVertex**: The character representing the starting city

4.3.2 Function Logic

At first, check if the input graph only have 1 vertex or not. If so, the optimal path only have that vertex.

```
//Check for special case
if (numCities == 1){
    string ans;
    ans += startVertex;
    return ans;
}
```

First, initialize necessary variables and array

```
//Initialization
int currentPath[21] = {-1};
bool visited[20] = {false};
int finalPath[21] = {-1};
int finalCost = 100000;
int currentBound = 0;
string ans;
```

Where:

- The **'currentPath'** array stores the current path being explored. At first initialize it with all -1.
- The **'visited'** array keeps track which cities have been visited. At first, none of the cities are visited so initialize it with false value.
- The **'finalPath'** array stores the optimal path. At first initialize it with all -1.
- The **'finalCost'** to store the cost of the optimal path, at the beginning let it be infinity
- The **'currentBound'** to store the lower bound of the cost for the current node. At first initialize it with 0.
- The string **'ans'** to return the optimal path in string format.

Next, find the minimum weight edges and second minimum weight edges of each vertex using the *findTwoMin* function.

```
//Fill in the minimum and second minimum weight edge for each vertex
int firstMin[20] = {-1};
int secondMin[20] = {-1};
findTwoMin(G,numCities,firstMin,secondMin);
```

Next, calculate the lower bound for any possible tour in the graph

```
//Calculate the bound for any tour by formula:
//1/2 * (sum of first minimum and second minimum of all vertices)
for (int i = 0; i < numCities; i++)
{
    currentBound += (firstMin[i] + secondMin[i]);
}

//Rounding off the bound
if (currentBound % 2 == 0)
{
    currentBound = currentBound / 2;
}
else
{
    currentBound = currentBound / 2 + 1;
}
```

At the beginning, marked the starting vertex as visited and included it to the current path array

```
//Store the starting vertex to the beginning of the current path
visited[int(startVertex - 'A')] = true;
currentPath[0] = int(startVertex - 'A');
```

After initialize all of the necessary elements, call to the *branchAndBoundTSP* with the current weight of the current path is 0 and start to explore the search space tree from root node.

```
//Call to branchAndBound with current path cost is 0 and explore the tree from level 1.
branchAndBoundTSP(G, numCities, currentPath, currentBound, 0, 1, visited, finalCost, finalPath);
```

After the algorithm has explore all paths and construct the optimal final path , return the final path in string type.

```
//After the optimal solution is found
//Return it in the string form
for (int i = 0; i <= numCities; i++)
{
    ans += char(finalPath[i] + 'A');
    if (i == numCities) break;
    ans += " ";
}
return ans;
```

5 Conclusion

This report has explore the Branch and Bound algorithm to solve The Traveling Saleman Problem. Although this is not the most efficient algorithm to solve the problem as in some worst-case scenario, Branch and Bound algorithm time complexity will be the same as Brute Force technique. However, in most cases, the algorithm will perform better than Brute Force technique.

References

[1] Introduction to TSP “**link: https://en.wikipedia.org/wiki/Travelling_salesman_problem**”, *Travelling salesman problem.*

[2] Branch and Bound introduction “**link: https://www.geeksforgeeks.org/branch-and-bound-algorithm/**”, *Branch and Bound Algorithm*