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

DISCRETE STRUCTURE (CO1007)

Assignment

“Traveling Saleman Problem”

Instructor(s):	Nguyễn Văn Minh Mẫn, Mahidol University
	Nguyễn An Khương, CSE-HCMUT
	Trần Tuấn Anh,CSE-HCMUT
	Nguyễn Tiến Thịnh, CSE-HCMUT
	Trần Hồng Tài, CSE-HCMUT
	Mai Xuân Toàn, CSE-HCMUT
Class:	L05
Student:	Nguyễn Việt Anh - 2210116

Table of Contents

1 Introduction to the Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is one of the most extensively studied problems in the field of combinatorial optimization. It is defined as follows: 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?

TSP is significant both theoretically and practically. It is an NP-hard problem in computational complexity theory, meaning that there is no known polynomial-time solution for it. Despite this, TSP has numerous practical applications, including logistics, planning, and the manufacturing of microchips.

1.1 History and Importance

The Travelling Salesman Problem was first formulated in the 19th century by the Irish mathematician William Rowan Hamilton and the British mathematician Thomas Penyngton Kirkman. Its importance has grown over time due to its applicability in various fields and its role in the development of optimization algorithms and computational complexity theory.

1.2 Formal Definition

Formally, the TSP can be described as follows: let $G = (V, E)$ be a complete graph where V represents the set of vertices (cities) and E represents the set of edges (paths between the cities). Each edge $(i, j) \in E$ has an associated weight $d(i, j)$, which represents the distance between cities i and j. The goal is to find a Hamiltonian cycle (a cycle that visits each vertex exactly once and returns to the starting vertex) with the minimum possible total weight.

1.3 Applications

The TSP has a wide range of applications:

- **Logistics and Transportation:** Optimizing routes for delivery trucks, mail carriers, and sales representatives.
- **Manufacturing:** Planning the movement of robotic arms in automated assembly lines.
- **Genomics:** Sequencing DNA to minimize errors.
- **Telecommunications:** Optimizing the layout of circuits on a microchip.

1.4 Challenges and Complexity

The primary challenge of the TSP lies in its computational complexity. As the number of cities increases, possible routes grow factorially, making brute-force solutions impractical for large instances. This has led to the development of various heuristic and approximation algorithms, such as the nearest neighbor algorithm, genetic algorithms, and simulated annealing.

1.5 Conclusion

The Travelling Salesman Problem continues to be a central problem in operations research and computer science due to its theoretical significance and practical relevance. Advances in solving TSP have broad implications for optimization and computational theory, making it a fascinating and crucial area of study.

2 Methods for Solving the Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a classic optimization problem that has been approached using various methods. These methods can be broadly classified into exact algorithms, approximation algorithms, and heuristics. Each method has its own advantages and limitations depending on the problem size and specific requirements.

2.1 Exact Algorithms

Exact algorithms guarantee to find the optimal solution to the TSP but are generally computationally expensive for large instances. Some of the common exact algorithms include:

- **Dynamic Programming:** The Held-Karp algorithm uses dynamic programming to solve TSP with a time complexity of $O(n^2 \cdot 2^n)$, where n is the number of cities.
- **Integer Linear Programming (ILP):** Formulates TSP as a linear program with integer constraints. Solvers like CPLEX or Gurobi can be used to find the optimal solution.
- **Branch and Bound:** A systematic method of solving TSP by exploring all possible routes while pruning suboptimal routes to reduce computation.

2.2 Approximation Algorithms

Approximation algorithms provide solutions close to the optimal solution with a guarantee of the solution quality. These include:

- **Christofides' Algorithm:** Guarantees a solution within 1.5 times the optimal solution for metric TSP.
- **Nearest Neighbor Algorithm:** Builds a route by repeatedly visiting the nearest unvisited city, though it does not guarantee optimality.

2.3 Heuristics and Metaheuristics

Heuristics and metaheuristics provide good solutions within a reasonable time frame, often without guarantees of optimality. Common approaches include:

- **Genetic Algorithms:** Uses principles of natural selection to iteratively improve the solution.
- **Simulated Annealing:** Mimics the cooling process of metals to escape local minima and find a good approximation of the global minimum.
- **Ant Colony Optimization:** Simulates the behavior of ants to find optimal paths based on pheromone trails.

2.4 Why Use Branch and Bound?

The Branch and Bound method is a popular exact algorithm for solving the TSP due to several reasons:

- **Optimality:** It guarantees finding the optimal solution by systematically exploring and evaluating all possible routes.

- **Pruning:** By effectively pruning suboptimal routes early in the search process, it reduces the computational burden compared to brute-force methods.
- **Flexibility:** Branch and Bound can be combined with other techniques, such as heuristics, to improve its efficiency for large instances.
- **General Applicability:** It is not limited to TSP and can be applied to various other combinatorial optimization problems.

The Branch and Bound method involves dividing the problem into smaller subproblems (branching) and calculating a bound on the minimum possible cost of any solution within each subproblem. If the bound for a subproblem exceeds the current best-known solution, the subproblem is discarded (pruned). This process continues until the optimal solution is found.

In conclusion, while many methods exist for solving the TSP, the choice of method depends on the specific requirements and constraints of the problem. Branch and Bound stand out as a powerful method due to its balance between guaranteeing optimal solutions and reducing computational complexity through effective pruning.

3 Code and Explaination

3.1 Header File - tsm.h

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
#include <cstring>
#include <string>
#include <fstream>
#ifndef TSM_H
#define TSM_H

using namespace std;
const int MAXN = 20; // Maximum number of cities/nodes
const int MAXINT = -1;
extern int DEM;  // External declaration for the count of recursive calls
extern int COUNT;  // External declaration for the count of better solutions found

// Declarations for the Branch and Bound algorithm functions
void branchBoundHelp(int G[MAXN][MAXN], int n, int startCity, int i, int minCost, int& tmpRes, int& result,
int tempPath[], int path[], int visited[]);
std::string branchBound(int G[MAXN][MAXN], int n, char start);
std::string Traveling(int G[MAXN][MAXN], int n, char start);

#endif
```

3.2 Implementation File - tsm.cpp

```cpp
#include "tsm.h"

// Global variables
int DEM = 0;  // Counter for the number of recursive calls
int COUNT = 0;  // Counter for the number of times a better solution is found

// Helper function for the Branch and Bound algorithm
// G: adjacency matrix, n: number of cities, startCity: starting city index
// i: current depth in the search tree, minCost: minimum edge cost
// tmpRes: current path cost, result: best path cost so far
// tempPath: current path being explored, path: best path found so far
// visited: array to keep track of visited cities
void branchBoundHelp(int G[MAXN][MAXN], int n, int startCity, int i, int minCost, int& tmpRes, int& result,
int tempPath[], int path[], int visited[]) {
    ++DEM;  // Increment the recursive call counter

    for (int city = 0; city < n; city++) {
        if (visited[city] == 0) {  // If the city hasn't been visited yet
            visited[city] = 1;  // Mark the city as visited
            tempPath[i] = city;  // Add the city to the current path
            tmpRes += G[tempPath[i - 1]][tempPath[i]];  // Add the cost of the edge to the current path cost

            // If all cities are visited (last city)
            if (i == n - 1) {
                int candidate = tmpRes + G[tempPath[i]][startCity];  // Total cost including return to start
                // If this path is better than the best so far
                if (result > candidate) {
                    ++COUNT;  // Increment the counter for better solutions found
                    result = candidate;  // Update the best result
                    for (int j = 0; j < n; j++) {
                        path[j] = tempPath[j];  // Copy the current path as the best path
                    }
                }
            } else if (tmpRes + (n - i - 1) * minCost < result) {  // Pruning condition
                // If the current cost plus the minimum possible cost for remaining edges
                // is less than the best result, continue searching this branch
                branchBoundHelp(G, n, startCity, i + 1, minCost, tmpRes, result, tempPath, path, visited);
            }
            visited[city] = 0;  // Backtrack: mark the city as unvisited
            tmpRes -= G[tempPath[i - 1]][tempPath[i]];  // Backtrack: remove the edge cost
        }
    }
}

// Main function for the Branch and Bound algorithm
// G: adjacency matrix, n: number of cities, start: starting city character
std::string branchBound(int G[MAXN][MAXN], int n, char start) {
    int minCost = INT_MAX;  // Initialize minimum edge cost
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                minCost = min(minCost, G[i][j]);  // Find the minimum edge cost
            }
        }
    }

    int tmpRes = 0, Res = INT_MAX;  // Initialize current and best path costs
    int tempPath[MAXN];  // Array to store the current path
    int visited[MAXN] = { 0 };  // Array to mark visited cities
    int path[MAXN];  // Array to store the best path
    int startCity = start - 'A';  // Convert start city character to index (A=0, B=1, etc.)
    string sPath = "";  // String to store the result path

    visited[startCity] = 1;  // Mark the start city as visited
    tempPath[0] = startCity;  // Start the path with the start city

    // Call the helper function to find the best path
    branchBoundHelp(G, n, startCity, 1, minCost, tmpRes, Res, tempPath, path, visited);

    // Convert the best path (indices) to a string of city characters
    for (int i = 0; i < n; i++) {
        char c = (char)(path[i] + 'A');
        sPath += c;
        if (i != n - 1) {
            sPath += " ";
        }
    }
    sPath += " ";
    sPath += (char)(path[0] + 'A');  // Add the start city at the end to complete the cycle

    return sPath;
}
```

4 Explanation of the Branch and Bound Algorithm for the TSP

Key Components of the Implementation:

- **branchBoundHelp:** This recursive function explores all potential paths using a depth-first search approach. It features a pruning mechanism that discards paths that cannot lead to an optimal solution, thereby reducing the unnecessary expansion of the search space.
- **branchBound:** This function sets up the necessary initial conditions, such as calculating the minimum edge cost, which is essential for the pruning process. It then calls `branchBoundHelp` to compute the optimal path and formats the result as a string representation of the route.
- **Traveling:** Acting as the main function, it accepts the graph in the form of an adjacency matrix, the total number of cities, and the starting city. It preprocesses the graph by converting zero costs (indicating no direct paths) to a very high value (represented by `MAXINT`).

This adjustment ensures that non-existent paths are only considered when absolutely necessary, thus focusing the search on viable routes.

4.1 Workflow of the Algorithm

- The algorithm builds the solution incrementally, selecting one city at a time to add to the path.
- At each step, it calculates the cost associated with the current path and estimates the minimum possible cost for completing the tour. This estimate helps in deciding whether to continue exploring the current path or to abandon it.
- If the sum of the current path cost and the estimated cost to complete the tour is greater than or equal to the best solution found so far, the path is pruned from further consideration. This pruning significantly reduces the computational load compared to a brute-force approach.

4.2 Analysis Variables

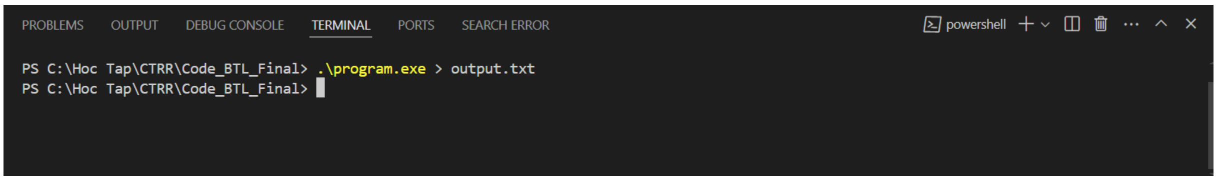
- **DEM:** This variable counts the number of recursive function calls, indicating how many partial solutions (paths) have been considered.
- **COUNT:** This tracks how many times the algorithm has found a solution that is better than previously identified solutions. This count can provide insights into the efficiency of the search process.

4.3 Performance Considerations

The presented implementation is effectively suited for small to medium-sized problem sets. For larger datasets, where the number of cities grows significantly, the algorithm might require enhancements to maintain efficiency. In such cases, more advanced techniques like dynamic programming or heuristic methods (for example, genetic algorithms) might be preferable. These methods can provide a good balance between runtime and solution quality for complex instances of the TSP.

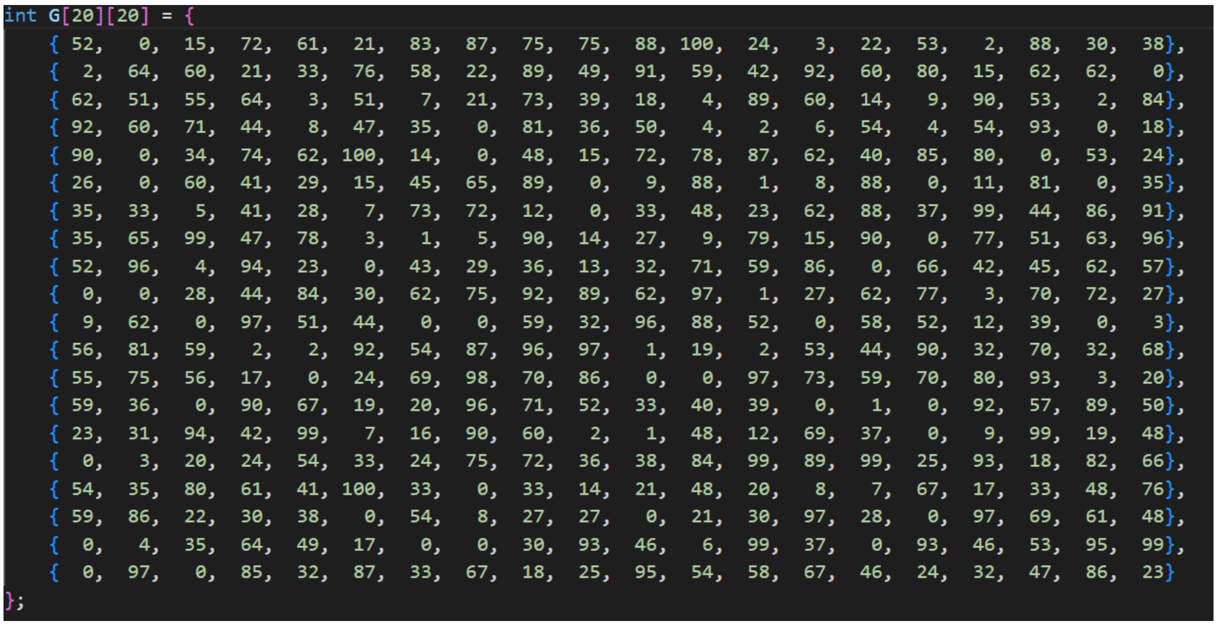
This theoretical background supports the detailed code explanations and provides a comprehensive understanding of the algorithm's workings and its application to the TSP.

5 Run Code



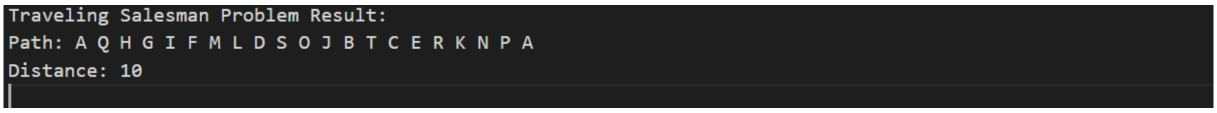
```
PS C:\Hoc Tap\CTRR\Code_BTL_Final> .\program.exe > output.txt
PS C:\Hoc Tap\CTRR\Code_BTL_Final>
```

Figure 1: terminal



```
int G[20][20] = {
    { 52,   0,  15,  72,  61,  21,  83,  87,  75,  75,  88, 100,  24,   3,  22,  53,   2,  88,  30,  38},
    {  2,  64,  60,  21,  33,  76,  58,  22,  89,  49,  91,  59,  42,  92,  60,  80,  15,  62,  62,   0},
    { 62,  51,  55,  64,   3,  51,   7,  21,  73,  39,  18,   4,  89,  60,  14,   9,  90,  53,   2,  84},
    { 92,  60,  71,  44,   8,  47,  35,   0,  81,  36,  50,   4,   2,   6,  54,   4,  54,  93,   0,  18},
    { 90,   0,  34,  74,  62, 100,  14,   0,  48,  15,  72,  78,  87,  62,  40,  85,  80,   0,  53,  24},
    { 26,   0,  60,  41,  29,  15,  45,  65,  89,   0,   9,  88,   1,   8,  88,   0,  11,  81,   0,  35},
    { 35,  33,   5,  41,  28,   7,  73,  72,  12,   0,  33,  48,  23,  62,  88,  37,  99,  44,  86,  91},
    { 35,  65,  99,  47,  78,   3,   1,   5,  90,  14,  27,   9,  79,  15,  90,   0,  77,  51,  63,  96},
    { 52,  96,   4,  94,  23,   0,  43,  29,  36,  13,  32,  71,  59,  86,   0,  66,  42,  45,  62,  57},
    {  0,   0,  28,  44,  84,  30,  62,  75,  92,  89,  62,  97,   1,  27,  62,  77,   3,  70,  72,  27},
    {  9,  62,   0,  97,  51,  44,   0,   0,  59,  32,  96,  88,  52,   0,  58,  52,  12,  39,   0,   3},
    { 56,  81,  59,   2,   2,  92,  54,  87,  96,  97,   1,  19,   2,  53,  44,  90,  32,  70,  32,  68},
    { 55,  75,  56,  17,   0,  24,  69,  98,  70,  86,   0,   0,  97,  73,  59,  70,  80,  93,   3,  20},
    { 59,  36,   0,  90,  67,  19,  20,  96,  71,  52,  33,  40,  39,   0,   1,   0,  92,  57,  89,  50},
    { 23,  31,  94,  42,  99,   7,  16,  90,  60,   2,   1,  48,  12,  69,  37,   0,   9,  99,  19,  48},
    {  0,   3,  20,  24,  54,  33,  24,  75,  72,  36,  38,  84,  99,  89,  99,  25,  93,  18,  82,  66},
    { 54,  35,  80,  61,  41, 100,  33,   0,  33,  14,  21,  48,  20,   8,   7,  67,  17,  33,  48,  76},
    { 59,  86,  22,  30,  38,   0,  54,   8,  27,  27,   0,  21,  30,  97,  28,   0,  97,  69,  61,  48},
    {  0,   4,  35,  64,  49,  17,   0,   0,  30,  93,  46,   6,  99,  37,   0,  93,  46,  53,  95,  99},
    {  0,  97,   0,  85,  32,  87,  33,  67,  18,  25,  95,  54,  58,  67,  46,  24,  32,  47,  86,  23}
};
```

Figure 2: Matrix input in file main.cpp



```
Traveling Salesman Problem Result:
Path: A Q H G I F M L D S O J B T C E R K N P A
Distance: 10
```

Figure 3: Output

6 Conclusion

This report presents the Branch and Bound method for solving the Traveling Salesman Problem (TSP). This method is a combinatorial optimization technique, notable for its ability to find optimal solutions without the inefficient trial of all possibilities.

6.1 Highlights

The Branch and Bound method has proven extremely effective in reducing the search space through its pruning mechanism, which helps eliminate routes that cannot provide an optimal solution. This is particularly important when dealing with problems involving a large number of cities, where the number of permutations can increase exponentially.

6.2 Applications and Limitations

Branch and Bound is not only useful for the TSP but can also be applied to many other combinatorial optimization problems. However, it is important to note that the effectiveness of this method can significantly decrease when the input size increases to a certain threshold—this requires more advanced techniques or integration with heuristics to maintain applicability.

6.3 Future Directions

Combining Branch and Bound with other advanced algorithms such as genetic algorithms or simulated annealing could extend its capability to solve larger and more complex problems. Continued research in this field not only improves the efficiency of the algorithm but also contributes to the overall theory of optimization.

Conclusion: Through the study and application of the Branch and Bound method, we find that it is a powerful tool, especially when used in suitable problems. Continued development and adjustment of this method will be key to addressing future optimization challenges.

References

1. Nhánh và Cận - Branch and Bound, Viblo. Available at: https://viblo.asia/p/nhanh-va-can-branch-and-bound-Qbq5QBPEKD8.

2. Introduction to Branch and Bound, Geeks for Geeks. Available at: https://www.geeksforgeeks.org/introduction-to-branch-and-bound-data-structures-and-algorithms-tutorial/.