Travelling Salesman Problem

Aim:
The aim is to solve the Traveling Salesman Problem (TSP), which involves finding the shortest possible route that visits each city exactly once and returns to the original city. The objective is to optimize the tour length, minimizing the total distance traveled while visiting all cities in a given graph.

Theory:
The Traveling Salesman Problem (TSP) is a classic problem in combinatorial optimization and graph theory. It revolves around finding the shortest possible route that visits each city exactly once and returns to the starting city. This problem is significant in logistics, transportation, and route optimization, where minimizing the total distance traveled is crucial for efficiency and cost-effectiveness.

In the context of a weighted graph representing cities and distances between them, the goal is to find a Hamiltonian cycle with the minimum total weight. The problem is known to be NP-hard, meaning that it becomes computationally challenging as the number of cities increases.

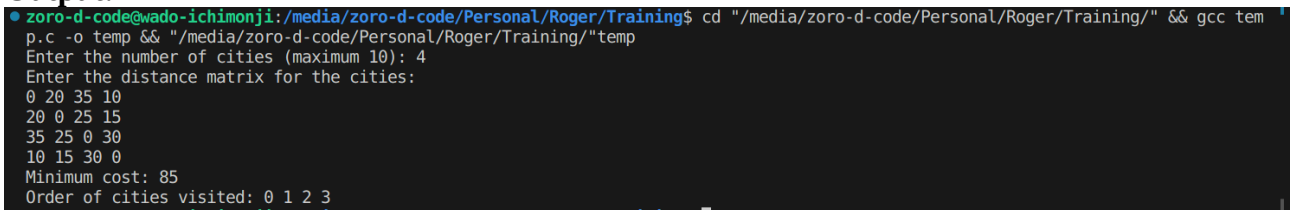
Various algorithms and techniques are used to tackle the TSP, ranging from brute-force methods that explore all possible permutations to heuristic approaches like nearest neighbor and genetic algorithms that offer faster but approximate solutions. Branch and bound is another method that intelligently explores the solution space, pruning branches that cannot lead to an optimal solution, thus improving efficiency in finding near-optimal solutions for moderately sized instances of the problem.

Code:
```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <limits.h>

#define MAX_N 10

int graph[MAX_N][MAX_N];
int n;
bool visited[MAX_N];
int path[MAX_N];
int min_cost = INT_MAX;
int final_path[MAX_N];

// Calculate lower bound for pruning branches
int lowerBound()
{
   int lb = 0;
   for (int i = 0; i < n; i++)
   {
      if (!visited[i])
      {
         int min_edge = INT_MAX;
         for (int j = 0; j < n; j++)
         {
```

```
            if (i != j && !visited[j] && graph[i][j] < min_edge)
            {
                min_edge = graph[i][j];
            }
        }
        lb += min_edge;
    }
  }
  return lb;
}

// Branch and bound algorithm
void branchAndBound(int curr_cost, int level)
{
  if (level == n)
  {
    if (graph[path[level - 1]][path[0]] != 0)
    {
      curr_cost += graph[path[level - 1]][path[0]];
      if (curr_cost < min_cost)
      {
        min_cost = curr_cost;
        for (int i = 0; i < n; i++)
        {
          final_path[i] = path[i];
        }
      }
    }
    return;
  }
  int bound = lowerBound();
  if (curr_cost + bound >= min_cost)
  {
    return;
  }
  for (int i = 0; i < n; i++)
  {
    if (!visited[i] && graph[path[level - 1]][i] != 0)
    {
      path[level] = i;
      visited[i] = true;
      branchAndBound(curr_cost + graph[path[level - 1]][i], level + 1);
      visited[i] = false;
    }
  }
}

// Solve TSP using branch and bound
void solveTSP()
{
  printf("Enter the number of cities (maximum %d): ", MAX_N);
  scanf("%d", &n);
```

```
    // Input distances between cities
    printf("Enter the distance matrix for the cities:\n");
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            scanf("%d", &graph[i][j]);
        }
    }

    path[0] = 0;
    visited[0] = true;
    branchAndBound(0, 1);
}

int main()
{
    solveTSP();

    // Output minimum cost and order of cities visited
    printf("Minimum cost: %d\n", min_cost);
    printf("Order of cities visited: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", final_path[i]);
    }
    printf("\n");

    return 0;
}
```

Output:



Algorithm Steps:

1. Input: Obtain the number of cities (n) and the distance matrix representing the distances between cities.

2. Initialization:
- Initialize an empty path array to store the current tour.
- Initialize a boolean array to track visited cities.
- Set the minimum cost to infinity (or a very large value).

3. Calculate Lower Bound:
- Implement a function to calculate the lower bound for pruning branches in the search tree. This involves finding the minimum edge for each unvisited city and summing these minimum edges.

4. Branch and Bound:
- Implement a recursive function to perform the branch and bound algorithm.
- The function takes parameters including the current cost, current level (depth of the search tree), and other necessary variables.
- At each level of the tree, explore all possible paths by considering unvisited cities and updating the current tour.
- Prune branches if the current cost plus the lower bound exceeds the minimum cost found so far.
- Continue exploring until all cities are visited, updating the minimum cost and final tour as needed.

5. Solve TSP:
- Prompt the user for input (number of cities and distance matrix).
- Call the branch and bound function with initial parameters to solve the TSP.

6. Output:
- Display the minimum cost found and the order of cities visited to form the optimal tour.

7. Termination:
- The algorithm terminates when all branches of the search tree have been explored, and the optimal tour with minimum cost is found.

Time complexity:

1. Permutation Generation:
- Generating all permutations of cities has a time complexity of (O(n!)), where (n) is the number of cities. This is because there are (n!) possible permutations to consider.

2. Branching:
- The branching factor in the search tree depends on the number of unvisited cities at each level, which ranges from (n-1) to 1.
- The worst-case time complexity for exploring all branches can be (O(n!)) when the search tree is fully expanded.

3. Bounding:
- Calculating the lower bound and performing pruning operations has a time complexity of (O(n^2)), as it involves iterating over the distance matrix to find minimum edges.

4. Overall:
- Combining these factors, the overall time complexity of the branch and bound algorithm for TSP is typically considered to be (O(n!)) in the worst case.
- However, the algorithm's actual performance may vary depending on the specific problem instance, the efficiency of the lower bound calculation, and the effectiveness of pruning.

Space Complexity:

1. Input Space:
- The input space includes the number of cities (n) and the distance matrix representing the distances between cities.
- This space requirement is typically (O(n^2)) for storing the distance matrix.

2. Auxiliary Space:

- The branch and bound algorithm requires additional space for data structures such as arrays, boolean flags for visited cities, and variables to track the current tour and minimum cost.
- The auxiliary space complexity is (O(n)) for storing the visited array, path array, and other variables.

3. Recursion Stack:

- As the algorithm uses recursion to explore branches of the search tree, the space complexity is influenced by the maximum recursion depth.
- In the worst case, the recursion depth can be (O(n)) when all cities are visited in a branch before pruning occurs.
- This contributes to the overall space complexity as (O(n)).

4. Overall:

- Combining these factors, the overall space complexity of the branch and bound algorithm for TSP is (O(n^2 + n)), which can be simplified to (O(n^2)) asymptotically.
- The space required for storing the distance matrix dominates the space usage, while the additional auxiliary space and recursion stack add a linear component.