CONTENTS

## S. NO TITLE

1. Abstract
2. Acknowledgement
3. Introduction
4. Problem Statement
5. Dry-Run for Divide and Conquer
6. Dry-Run for Branch and bound Approach
7. Analysis for Divide and Conquer
8. Analysis for Branch and Bound Approach
9. Comparison
10. Code for Divide and Conquer approach
11. Code for Branch and Bound Approach
12. Applications of Branch and bound Approach
13. References
14. Conclusion

### ABSTRACT

Finding theVechile Routing problem a common problem in computer science and is used in many applications such as data analysis and sorting. In this project, we explore two algorithms for finding the

vechile routing problem: Divide and Conquer and Branch and bound

Apprach. We compare the time complexity and efficiency of these two algorithms for different array sizes and analyse the results. Our findings show that the branch and Bound is faster and more efficient than the

Divide and Conquer approach for finding the Vechile Routing problem ACKNOWLEDGMENT

I would like to express my gratitude to all those who have helped me

complete this project. Firstly, I would like to thank my professor for

providing me with this opportunity to explore the Divide and Conquer and Branch and Bound Approach. Their guidance and support have been invaluable throughout the project.

I would also like to thank my peers for their support and feedback during the development of this project. Their insights and perspectives have been critical in shaping the final results.

Finally, I would like to acknowledge the resources and references that I have used throughout this project, including online tutorials, textbooks, and academic papers. These resources have provided me with the

necessary knowledge and skills to complete this project successfully.

INTRODUCTION:

The Vehicle Routing Problem (VRP) is a combinatorial optimization problem that involves finding the optimal set of routes for a fleet of vehicles to service a set of customers, while minimizing the total distance traveled and satisfying certain constraints.

The VRP is a complex problem that arises in many real-world applications, such as transportation, logistics, and distribution. The problem can be formulated in different ways depending on the specific requirements of the application. However, the basic VRP formulation involves the following components:

* A set of customers, each with a demand for a certain quantity of goods or services.
* A set of depots, each with a fixed capacity for carrying goods and a fixed number of vehicles available for servicing customers.
* A distance or travel time matrix that specifies the distance or travel time between each pair of customers and depots.
* A set of constraints that must be satisfied, such as the maximum capacity of each vehicle and the maximum travel time or distance allowed for each vehicle.

.

There are various algorithms and heuris cs that can be used to solve the VRP, including exact algorithms such as branch and bound and heuris cs such as the Nearest Neighbor algorithm, the Clarke and Wright algorithm, and the Sweep algorithm. Addi onally, advanced techniques such as metaheuris cs and simula onbased op miza on can be used to obtain high-quality solu ons for larger and more complex instances of the VRP.

PROBLEM STATEMENT:

The VRP is a generalisation of the travelling salesperson problem, a classic optimisation problem that has been examined since the 1930s. It is based on a scenario where a salesperson must visit several cities. Their journey must finish back where they started, and they can visit each location only once. They can visit the cities in any order, but they need to travel the least possible distance. The problem is modelled as a network, with the cities represented by nodes and connected by paths weighted in terms of the time or distance required to travel between two cities.

DESIGN APPROACHES:

The vehicle rou ng problem (VRP) is a well-known combinatorial op miza on problem that involves finding op mal routes for a fleet of vehicles to visit a set of customers, subject to capacity and distance constraints.

Divide and conquer and back and bounce are two different algorithms that can be used to solve VRP. The me complexity of these algorithms depends on various factors such as the size of the input, the structure of the problem, and the specific implementa on of the algorithm.

DIVIDE AND CONQUER:

1. Divide the problem into a number of subproblems that are smaller instances of the same problem.
2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. Combine the solutions to the subproblems into the solution for the original problem.

PSEUDO CODE:

Pseudo code for vehicle routing problem using divide and conquer approach.

Function VRP\_Divide\_Conquer(D,C,K):

//D:Distance matrix,C:customer demand ,K: number of vehicles

//Returns a solution for VRP

//If there is only one vehicle,solve TSP for all customers If K ==1:

Return Solve\_TSP(D,C)

//Divide customers into two groups

M=len(c)//2

C1 = c[:m]

C2=c[m:]

//solve sub-problems recursively

Solution1 = VRP\_Divide\_Conquer(D, C1, K//2)

Solution2 = VRP\_Divide\_Conquer(D, C2 , K-K//2)

//Merge Solutions

merged\_solution = Merge\_Solutions(solution1 , solution2)

//Return merged solution

return merged\_solution

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 100

#define INF 9999999

// Structure to store the matrix typedef struct { int cost[MAX][MAX]; int nodes;

} matrix;

matrix adj; // Adjacency matrix bool visited[MAX]; // Array to keep track of visited nodes int best\_cost = INF; // Best cost found so far

void input\_matrix() { printf("Enter the number of nodes: "); scanf("%d", &adj.nodes); printf("Enter the adjacency matrix:\n"); for (int i = 0; i < adj.nodes; i++) { for (int j = 0; j < adj.nodes; j++) { scanf("%d",

&adj.cost[i][j]);

}

}

}

int calculate\_cost(int path[], int n) { int cost = 0; for (int i = 0; i < n - 1; i++) { cost += adj.cost[path[i]][path[i+1]];

}

return cost;

}

void print\_path(int path[], int n) { printf("Path: "); for (int i = 0; i < n; i++) { printf("%d ", path[i]);

}

printf("%d\n", path[0]);

}

void divide\_and\_conquer(int path[], int n) {

if (n == 2) {

int cost = calculate\_cost(path, n); if (cost < best\_cost) { best\_cost = cost; print\_path(path, n);

}

return;

} for (int i = 1; i < n - 1; i++) { if (!visited[i]) { visited[i] = true; int temp = path[i]; path[i] = path[n-1]; path[n-1] = temp; divide\_and\_conquer(path, n-1);

visited[i] = false; temp = path[i]; path[i]

= path[n-1]; path[n-1]

= temp;

}

}

}

int main() { input\_matrix();

int path[MAX]; for (int i = 0; i < adj.nodes; i++) { path[i] = i; visited[i] = false;

}

visited[0] = true;

divide\_and\_conquer(path, adj.nodes); printf("Best cost: %d\n", best\_cost); return 0;

}

TIME COMPLEXITY AND ANAlYSIS:

Divide and Conquer Approach: The time complexity of the Divide and Conquer algorithm depends on the number of recursive calls made and the time taken to merge the sub-solutions. In the worst case, the algorithm must make n recursive calls, resulting in a time complexity of O(n log n), where n is the number of cities. However, with efficient merging techniques and heuristics, the actual running time can be much less than the worst-case complexity.

BRANCH AND BOUND APPROACH:

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

PSEUDO CODE:

Vehicle Routing Problem using branch and bound pseudocode:

1. Initialize the search tree with the root node
2. While there are nodes to explore:
   1. Choose the most promising node to explore
   2. Generate child nodes by adding one city to the current path
   3. Calculate the lower bound of each child node
   4. Add the child nodes to the list of nodes to explore
3. Return the best solution found

Here is a more detailed pseudocode for the above algorithm:

1. Initialize the search tree with the root node
   1. Create a node with an empty path and level 0
   2. Set the lower bound of the node to 0
   3. Add the node to the list of nodes to explore

1. While there are nodes to explore:
   1. Choose the most promising node to explore
      1. Select the node with the highest priority (lowest bound) ii.

Remove the node from the list of nodes to explore

* 1. Generate child nodes by adding one city to the current path
     1. For each city not in the current path:
        1. Create a new node with the new city added to the path
        2. Set the level of the node to the level of its parent + 1
        3. Calculate the lower bound of the node
  2. Calculate the lower bound of each child node
     1. For each child node:
        1. Calculate the cost of adding the next city to the path
        2. Calculate the minimum cost of adding the remaining cities using a greedy algorithm
        3. Set the lower bound of the node to the sum of these costs
  3. Add the child nodes to the list of nodes to explore
     1. Add each child node to the list of nodes to explore, sorted by

priority (lowest bound)

1. Return the best solution found
   1. When there are no more nodes to explore, return the best solution found (the node with the lowest bound)

CODE:

Here is an example implementation of the Vehicle Routing Problem using Branch and Bound in C:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <stdbool.h>

#define MAX 100

#define INF 9999999

// Structure to store the matrix typedef struct { int cost[MAX][MAX]; int nodes;

} matrix;

// Structure to store the partial solution typedef struct { int path[MAX]; int level; int cost; } partial; matrix adj; // Adjacency matrix partial best; // Best solution found so far bool visited[MAX]; // Array to keep track of visited nodes void input\_matrix() { printf("Enter the number of nodes: "); scanf("%d", &adj.nodes); printf("Enter the adjacency matrix:\n"); for (int i = 0; i < adj.nodes; i++) { for (int j = 0; j < adj.nodes; j++) { scanf("%d",

&adj.cost[i][j]);

}

} } void initialize() { for (int i = 0; i < MAX; i++) { visited[i] = false;

}

best.cost = INF; best.level = -1;

}

void print\_path(partial p) { printf("Path: "); for (int i = 0; i <= p.level; i++) { printf("%d ", p.path[i]);

}

printf("%d\n", p.path[0]); printf("Cost: %d\n", p.cost);

}

void copy\_solution(partial src, partial \*dst) { dst->level = src.level; dst->cost = src.cost; memcpy(dst->path, src.path, sizeof(src.path));

}

void add\_to\_path(int node, partial \*p) { p->level++; p->path[p->level] = node; p->cost += adj.cost[p->path[p->level - 1]][node];

}

void remove\_from\_path(partial \*p) { int last = p->path[p->level]; p->cost -= adj.cost[p->path[p->level - 1]][last]; p-

>level--;

}

void branch\_and\_bound(partial p) { if (p.level == adj.nodes - 1) {

p.cost += adj.cost[p.path[p.level]][p.path[0]]; if (p.cost < best.cost) { copy\_solution(p,

&best);

print\_path(best);

}

return;

}

for (int i = 0; i < adj.nodes; i++) {

if (!visited[i]) { add\_to\_path(i, &p); visited[i] = true;

if (p.cost + adj.cost[p.path[p.level]][i] < best.cost) {

branch\_and\_bound(p);

}

remove\_from\_path(&p); visited[i] = false;

}

}

} int main() { input\_matrix(); initialize(); partial start; start.level

= 0; start.path[0] = 0; visited[0] = true; branch\_and\_bound

(start); printf("Best solution:\n"); print\_path(best); return 0;

}

This implementation uses a recursive function called branch\_and\_bound() to explore all possible solutions using a depth-first search strategy. The function takes a partial solution as input and generates all possible children by adding unvisited nodes to the path. It then recursively calls itself on each child and prunes branches that have a cost greater than the best solution found so far. The best solution is stored in the best variable.

TIME COMPLEXITY:

In the back and bounce approach, the problem is solved by iteratively improving an initial solution. The time complexity of this approach depends on the number of iterations required to find a good solution, the size of the search space, and the efficiency of the heuristics used to guide the search. The time complexity of the back and bounce approach for VRP is typically O(n^2), where n is the number of customers.

APPLICATIONS:

Take Tesco, a grocery and general merchandise retailer, as an example. They use over-the-road vehicles for good distribution. They transport the goods on pallets and each vehicle can accommodate only a limited number of pallets, while each business unit (BU) demands a different number of pallets

COMPARISON:

Both divide and conquer and branch and bound are common techniques used to solve optimization problems like the vehicle routing problem (VRP). However, the best approach for VRP depends on the specific problem instance and the desired outcome.

Divide and conquer is a method that involves breaking down a problem into smaller, more manageable subproblems that can be solved independently. In the context of VRP, this could involve dividing the problem into smaller subproblems, such as dividing the routes into smaller segments or breaking the problem into smaller geographical areas. The advantage of this approach is that it can lead to faster computation times, as the smaller subproblems can be solved more efficiently. However, it may not be suitable for all instances of the VRP, as dividing the problem may not always lead to a feasible solution.

On the other hand, branch and bound is a method that involves systematically exploring the space of possible solutions by branching out from the current solution and pruning branches that cannot lead to an optimal solution. In the context of VRP, this approach involves generating and exploring a large number of possible solutions, and eliminating those that are infeasible or suboptimal. The advantage of this approach is that it can lead to more accurate solutions, as it systematically explores the entire solution space. However, it may be computationally expensive and may not be suitable for larger VRP instances.

In summary, both divide and conquer and branch and bound are viable approaches for solving the VRP, and the best approach depends on the specific problem instance and desired outcome. In practice, a combination of these approaches, along with other heuristics and optimization techniques, may be used to solve VRP efficiently.