**EX - 10: Branch and Bound**

**Aim:** Implementation of different real time problems using Branch and Bound approach.

 1.Implement 0/1 knapsack problem using FIFO BB

2. Implement 0/1 knapsack problem using LIFO BB

3. Implement 0/1 knapsack problem using LC BB

4. Implement Travelling sales person problem using LCBB

**Program Description:**

Branch and Bound is a fundamental algorithmic method used for solving optimization problems, particularly those involving integer programming and combinatorial optimization. The idea is to systematically explore branches of decision trees to find optimal solutions while bounding the search space to discard suboptimal solutions early.

**Branch and Bound Framework**

1. **Branching**: The problem is divided into smaller subproblems (branches) which are easier to solve.

2. **Bounding**: For each subproblem, a bound on the best possible solution is calculated. If this bound is worse than the best solution already found, the subproblem is discarded.

3. **Pruning**: Using the bounds, entire branches of the decision tree that cannot possibly contain the optimal solution are pruned away, reducing the search space.



**Steps in Branch and Bound**

1. **Initialization**: Start with the initial problem and calculate an initial bound.

2. **Branching**: Split the problem into subproblems.

3. **Bounding**: Compute the bounds for each subproblem.

4. **Pruning**: Prune subproblems that cannot yield a better solution than the current best. 5. **Selection**: Select the next subproblem to explore.

6. **Termination**: When all subproblems are explored or pruned, the algorithm terminates with the optimal solution.

**Applications**

• **Traveling Salesman Problem (TSP)**: Finding the shortest possible route that visits each city exactly once and returns to the origin city.

• **Knapsack Problem**: Determining the most valuable subset of items that fit within a given weight capacity.

• **Integer Linear Programming**: Solving linear programs where some or all variables are constrained to take integer values.

Branch and Bound is powerful but can be computationally intensive for large problems. However, its ability to prune vast areas of the search space makes it more efficient than brute force methods in many cases.

**1.Implement 0/1 knapsack problem using FIFo BB theory**

The 0/1 Knapsack Problem is a classic optimization problem. The goal is to maximize the total value of items that can be included in a knapsack without exceeding its capacity. Each item can either be included (1) or excluded (0), hence the name 0/1 Knapsack.

FIFO (First In, First Out) Approach: In the FIFO approach, nodes are explored in the order they are generated, using a queue data structure.

Steps of the FIFO Branch and Bound Algorithm:

1. Initialization:

o Initialize the queue with the root node representing the initial state ( value = 0, weight = 0). o Calculate the upper bound for the root node.

2. Branching:

o At each step, remove the node at the front of the queue.

o Create two child nodes:

▪ One representing the inclusion of the next item (if it doesn't exceed capacity).

▪ One representing the exclusion of the next item.

3. Bounding:

o For each child node, calculate the bound (upper limit of possible value).

o If the bound is less than the current best known solution, prune that node (do not add it to the queue).

4. Selection and Pruning:

o Add valid child nodes to the back of the queue.

o Update the best known solution if a better one is found.

5. Termination:

o The algorithm terminates when the queue is empty, meaning all nodes have been explored or pruned.

Bounding Function:

The bounding function estimates the maximum value that can be obtained from a node. It adds the current node’s value and the maximum possible additional value based on the remaining capacity. This often involves sorting items by value-to-weight ratio to prioritize high-value items.

**Advantages and Disadvantages:**

• Advantages:

o Can provide optimal solutions by systematically exploring possible solutions.

o Pruning significantly reduces the search space.

• Disadvantages:

o Can be computationally expensive for large problems.

o Performance depends on the effectiveness of the bounding function.

**Applications:**

• Resource allocation problems

• Budget management

• Project selection

• Any scenario requiring optimal selection of discrete items under constraints

 **PROGRAM:**

import java.util.\*;

class Node {

 int level,value,weight,bound;

 boolean include;

 public Node(int level, int value, int weight, int bound) {

 this.level = level;

 this.value = value;

 this.weight = weight;

 this.bound = bound;

 this.include = false;

 }

}

class Main {

 public static int knapsackBound(Node node, int n, int W, int[] weights, int[] values) { if (node.weight >= W) {

 return 0;

 }

 int bound = node.value;

 int totalWeight = node.weight;

 int i = node.level + 1;

 while (i < n && totalWeight + weights[i] <= W) {

 totalWeight += weights[i];

bound += values[i];

 i++;

 }if (i < n) {

 bound += (W - totalWeight) \* values[i] / weights[i];

 }

 return bound;

 }

 public static int knapsack01BranchBound(int n, int W, int[] weights, int[] values) { Node root = new Node(-1, 0, 0, 0);

 root.bound = knapsackBound(root, n, W, weights, values);

 Queue<Node> queue = new LinkedList<>();

 queue.add(root);

 int maxValue = 0;

 while (!queue.isEmpty()) {

 Node node = queue.poll();

 if (node.value > maxValue) {

 maxValue = node.value;

 }if (node.bound > maxValue) {

 Node left = new Node(node.level + 1, node.value + values[node.level + 1], node.weight + weights[node.level + 1], 0);

 if (left.weight <= W) {

 left.bound = knapsackBound(left, n, W, weights, values);

 if (left.bound > maxValue) {

 queue.add(left);

 }

 }

 Node right = new Node(node.level + 1, node.value, node.weight, 0); right.bound = knapsackBound(right, n, W, weights, values);

 if (right.bound > maxValue) {

 queue.add(right);

 }

 }

 }

return maxValue;

 }

 public static void main(String[] args) {

 Scanner sc = new Scanner(System.in);

 System.out.print("Enter number of items: ");

 int n = sc.nextInt();

 System.out.print("Enter knapsack capacity: ");

 int W = sc.nextInt();

 int[] weights = new int[n];

 int[] values = new int[n];

 System.out.println("Enter weights of the items: ");

 for (int i = 0; i < n; i++) {

 weights[i] = sc.nextInt();

 }

 System.out.println("Enter values of the items: ");

 for (int i = 0; i < n; i++) {

 values[i] = sc.nextInt();

 }

 int maxValue = knapsack01BranchBound(n, W, weights, values);

 System.out.println("Maximum value in the knapsack: " + maxValue);

 }

}

**OUTPUT:**

****

**Time Complexity:** O(n⋅2n)

**Space Complexity:** O(2n)

**2. Implement 0/1 knapsack problem using LIFO BB**

In the LIFO approach, nodes are processed in a stack manner. This means the most recently generated nodes are explored first, delving deeper into the decision tree before backtracking.

**Steps of the LIFO Branch and Bound Algorithm:**

1. **Initialization**:

o Begin with the root node, which represents the state where no items are included (level = 0, value = 0, weight = 0).

o Calculate an initial upper bound for the root node.

2. **Branching**:

o For the current node, generate two child nodes:

▪ One representing the inclusion of the next item.

▪ One representing the exclusion of the next item.

3. **Bounding**:

o Compute the bound (upper limit of possible value) for each child node.

o If the bound is lower than the current best known solution, prune that node by not adding it to the stack.

4. **Selection and Pruning**:

o Add valid child nodes to the stack.

o Update the best known solution if a better one is found.

5. **Termination**:

o The algorithm terminates when the stack is empty, indicating all nodes have been explored or pruned.

**Bounding Function:**

The bounding function estimates the maximum value obtainable from the current state. This is done by adding the value of the current node and the maximum possible additional value, based on the remaining capacity of the knapsack and the value-to-weight ratio of the remaining items.

**Advantages and Disadvantages:**

• **Advantages**:

o Can find the optimal solution by exhaustively exploring possibilities.

o Pruning reduces the number of nodes to be explored, improving efficiency.

• **Disadvantages**:

o Can be computationally demanding for large problems.

o Performance is heavily dependent on the effectiveness of the bounding function. **Applications:**

• Resource allocation problems.

• Budget management.

• Project selection.

• Situations requiring optimal selection of discrete items within constraints. **PROGRAM:**

import java.util.\*;

class Node {

 int level, value, weight, bound;

 boolean include;

 public Node(int level, int value, int weight, int bound) {

 this.level = level;

 this.value = value;

 this.weight = weight;

 this.bound = bound;

 this.include = false;

 }

}

class Main {

 public static int knapsackBound(Node node, int n, int W, int[] weights, int[] values) { if (node.weight >= W) {

 return 0;

 }

 int bound = node.value;

 int totalWeight = node.weight;

 int i = node.level + 1;

 while (i < n && totalWeight + weights[i] <= W) {

 totalWeight += weights[i];

 bound += values[i];

 i++;

 }if (i < n) {

 bound += (W - totalWeight) \* values[i] / weights[i];

 }

 return bound;

 }

public static int knapsack01BranchBound(int n, int W, int[] weights, int[] values) { Node root = new Node(-1, 0, 0, 0);

 root.bound = knapsackBound(root, n, W, weights, values);

 Stack<Node> stack = new Stack<>();

 stack.push(root);

 int maxValue = 0;

 while (!stack.isEmpty()) {

 Node node = stack.pop();

 if (node.value > maxValue) {

 maxValue = node.value;

 }if (node.bound > maxValue) {

 Node left = new Node(node.level + 1, node.value + values[node.level + 1], node.weight + weights[node.level + 1], 0);

 if (left.weight <= W) {

 left.bound = knapsackBound(left, n, W, weights, values);

 if (left.bound > maxValue) {

 stack.push(left);

 }

 }

 Node right = new Node(node.level + 1, node.value, node.weight, 0); right.bound = knapsackBound(right, n, W, weights, values);

 if (right.bound > maxValue) {

 stack.push(right);

 }

 }

 }

 return maxValue;

 }

 public static void main(String[] args) {

 Scanner sc = new Scanner(System.in);

 System.out.print("Enter number of items: ");

 int n = sc.nextInt();

 System.out.print("Enter knapsack capacity: ");

int W = sc.nextInt();

 int[] weights = new int[n];

 int[] values = new int[n];

 System.out.println("Enter weights of the items: ");

 for (int i = 0; i < n; i++) {

 weights[i] = sc.nextInt();

 }

 System.out.println("Enter values of the items: ");

 for (int i = 0; i < n; i++) {

 values[i] = sc.nextInt();

 }

 int maxValue = knapsack01BranchBound(n, W, weights, values);

 System.out.println("Maximum value in the knapsack: " + maxValue);

 }

}

**OUTPUT:**

****

**Time Complexity**: O(2n⋅n)

**Space Complexity**: O(2n)

**3. Implement 0/1 knapsack problem using LC BB**

LC (Least Cost) Approach: In the LC approach, nodes are processed based on their bound values using a priority queue. The nodes with the highest bound are explored first, ensuring that the most promising paths are evaluated early.

**Steps of the LC Branch and Bound Algorithm:**

1. **Initialization:**

o Begin with the root node representing the initial state (no items included, level = -1, value = 0, weight = 0).

o Calculate the upper bound for the root node and initialize the priority queue with this node. 2. **Branching:**

o For the current node, generate two child nodes:

▪ One representing the inclusion of the next item.

▪ One representing the exclusion of the next item.

3. **Bounding:**

o Compute the bound (upper limit of possible value) for each child node.

o If the bound is lower than the current best known solution, prune that node by not adding it to the priority queue.

4. **Selection and Pruning:**

o Add valid child nodes to the priority queue.

o Update the best known solution if a better one is found.

o Always explore the node with the highest bound (least cost) first from the priority queue. 5. **Termination:**

o The algorithm terminates when the priority queue is empty, indicating all nodes have been explored or pruned.

**Advantages and Disadvantages:**

• **Advantages:**

o Can find the optimal solution by exhaustively exploring possibilities.

o Pruning significantly reduces the number of nodes to be explored, improving efficiency. • **Disadvantages:**

o Can be computationally expensive for large problems.

o Performance is heavily dependent on the effectiveness of the bounding function. **Applications:**

• Resource allocation problems.

• Budget management.

• Project selection.

• Situations requiring optimal selection of discrete items within constraints.

**PROGRAM:**

import java.util.\*;

class Node {

 int level, value, weight, bound;

 public Node(int level, int value, int weight, int bound) {

 this.level = level;

 this.value = value;

 this.weight = weight;

this.bound = bound;

 }

}

class NodeComparator implements Comparator<Node> {

 public int compare(Node n1, Node n2) {

 return Integer.compare(n1.bound, n2.bound);

 }

}

class Main {

 public static int knapsackBound(Node node, int n, int W, int[] weights, int[] values) { if (node.weight >= W) {

 return 0;

 }

 int bound = node.value;

 int totalWeight = node.weight;

 int i = node.level + 1;

 while (i < n && totalWeight + weights[i] <= W) {

 totalWeight += weights[i];

 bound += values[i];

 i++;

 }if (i < n) {

 bound += (W - totalWeight) \* values[i] / weights[i];

 }

 return bound;

 }

 public static int knapsack01BranchBound(int n, int W, int[] weights, int[] values) { Node root = new Node(-1, 0, 0, 0);

 root.bound = knapsackBound(root, n, W, weights, values);

 PriorityQueue<Node> pq = new PriorityQueue<>(new NodeComparator()); pq.add(root);

 int maxValue = 0;

 while (!pq.isEmpty()) {

Node node = pq.poll();

 if (node.value > maxValue) {

 maxValue = node.value;

 }if (node.bound > maxValue) {

 Node left = new Node(node.level + 1, node.value + values[node.level + 1], node.weight + weights[node.level + 1], 0);

 if (left.weight <= W) {

 left.bound = knapsackBound(left, n, W, weights, values);

 if (left.bound > maxValue) {

 pq.add(left);

 }

 }

 Node right = new Node(node.level + 1, node.value, node.weight, 0); right.bound = knapsackBound(right, n, W, weights, values);

 if (right.bound > maxValue) {

 pq.add(right);

 }

 }

 }

 return maxValue;

 }

 public static void main(String[] args) {

 Scanner sc = new Scanner(System.in);

 System.out.print("Enter number of items: ");

 int n = sc.nextInt();

 System.out.print("Enter knapsack capacity: ");

 int W = sc.nextInt();

 int[] weights = new int[n];

 int[] values = new int[n];

 System.out.println("Enter weights of the items: ");

 for (int i = 0; i < n; i++) {

 weights[i] = sc.nextInt();

 }

System.out.println("Enter values of the items: ");

 for (int i = 0; i < n; i++) {

 values[i] = sc.nextInt();

 }

 int maxValue = knapsack01BranchBound(n, W, weights, values);

 System.out.println("Maximum value in the knapsack: " + maxValue);

 }

}

**OUTPUT:**

****

**Time Complexity**: O(n2⋅2n)

**Space Complexity**: O(2n)

**4. Implement Travelling sales person problem using LCBB**

The Traveling Salesperson Problem (TSP) is a well-known optimization problem that involves finding the shortest possible route that visits each city exactly once and returns to the starting city. The objective is to minimize the total travel cost or distance. The LC (Least Cost) Branch and Bound method provides an efficient approach to solve this problem by exploring the solution space in a structured manner.

**Branch and Bound Technique:** Branch and Bound (B&B) is a method for solving combinatorial optimization problems. It involves systematically exploring the solution space by branching into smaller subproblems and using bounds to prune suboptimal solutions early.

**LC (Least Cost) Approach:** In the LC approach, nodes are processed based on their bound values using a priority queue. This ensures that nodes with the least cost (highest potential for an optimal solution) are explored first.

**Steps of the LC Branch and Bound Algorithm for TSP:**

1. **Initialization**:

o Start with the root node, representing the initial state (only the starting city is included in the tour, level = 0, path = [starting city], bound = initial bound).

o Calculate the initial bound using a heuristic, such as the minimum cost to each unvisited city. 2. **Branching**:

o For the current node, generate child nodes by including each unvisited city in the path. o Each child node represents a new partial tour.

3. **Bounding**:

o Calculate the bound for each child node. The bound is an estimate of the minimum possible cost of completing the tour from the current state.

o If the bound of a node is greater than the current best known solution, prune that node by not adding it to the priority queue.

4. **Selection and Pruning**:

o Add valid child nodes to the priority queue.

o Always expand the node with the least bound value (least cost) first from the priority queue. o Update the best known solution if a complete tour with a lower cost is found.

5. **Termination**:

o The algorithm terminates when the priority queue is empty, meaning all nodes have been explored or pruned.

**Bounding Function:**

The bounding function estimates the cost of completing the tour from the current state. It typically includes: • The cost of the current partial tour.

• A heuristic estimate of the minimum cost to complete the tour from the current state. This might involve the minimum outgoing edge costs for unvisited cities.

**Advantages and Disadvantages:**

• **Advantages**:

o Can provide optimal solutions by systematically exploring possible routes.

o Pruning significantly reduces the number of nodes to be explored, improving efficiency. • **Disadvantages**:

o Can be computationally expensive for large problems.

o Performance depends on the effectiveness of the bounding function and heuristic used. **Applications:**

• Logistics and delivery route optimization.

• Network design and analysis.

• Scheduling and planning.

• Any scenario requiring optimal routing among multiple locations.

**PROGRAM:**

import java.util.\*;

class Node {

 int[] path;

 int level,cost,bound;

 public Node(int n) {

path = new int[n];

 Arrays.fill(path, -1);

 level = 0;

 cost = 0;

 bound = 0;

 }

}

class Main {

 public static int calculateBound(int[][] dist, Node node, int n) { int bound = 0;

 boolean[] visited = new boolean[n];

 for (int i = 0; i < node.level - 1; i++) {

 visited[node.path[i]] = true;

 }

 for (int i = 0; i < node.level - 1; i++) {

 bound += dist[node.path[i]][node.path[i + 1]]; }

 for (int i = 0; i < n; i++) {

 if (!visited[i]) {

 int min = Integer.MAX\_VALUE;

 for (int j = 0; j < n; j++) {

 if (!visited[j] && dist[i][j] < min) {

 min = dist[i][j];

 }

 }

 bound += min;

 }

 }

 return bound;

 }

 public static int tsp(int[][] dist, int n) {

 Node root = new Node(n);

root.path[0] = 0;

 root.level = 1;

 Queue<Node> queue = new LinkedList<>();

 queue.add(root);

 int minCost = Integer.MAX\_VALUE;

 while (!queue.isEmpty()) {

 Node node = queue.poll();

 if (node.level == n) {

 if (dist[node.path[node.level - 1]][0] > 0) {

 int totalCost = node.cost + dist[node.path[node.level - 1]][0]; minCost = Math.min(minCost, totalCost);

 }

 } else {

 for (int i = 0; i < n; i++) {

 if (isSafe(i, node, n, dist)) {

 Node child = new Node(n);

 System.arraycopy(node.path, 0, child.path, 0, n); child.path[node.level] = i;

 child.cost = node.cost + dist[node.path[node.level - 1]][i]; child.level = node.level + 1;

 child.bound = calculateBound(dist, child, n); if (child.bound < minCost) {

 queue.add(child);

 }

 }

 }

 }

 }

 return minCost;

 }

 public static boolean isSafe(int city, Node node, int n, int[][] dist) { for (int i = 0; i < node.level; i++) {

if (node.path[i] == city) {

 return false;

 }

 }

 return true;

 }

 public static void main(String[] args) {

 Scanner sc = new Scanner(System.in);

 System.out.print("Enter the number of cities: ");

 int n = sc.nextInt();

 int[][] dist = new int[n][n];

 System.out.println("Enter the distance matrix:");

 for (int i = 0; i < n; i++) {

 for (int j = 0; j < n; j++) {

 dist[i][j] = sc.nextInt();

 }

 }

 int result = tsp(dist, n);

 System.out.println("Minimum cost of the TSP tour: " + result); }

}

**OUTPUT:**

****

**Time Complexity**: O(n!⋅n2)

**Space Complexity**: O(n!⋅n)