LP2

1. Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

DFS

Depth First Search (DFS) is a fundamental graph traversal algorithm used to explore or search through all the vertices of a graph or tree data structure.

It starts at a chosen vertex and explores as far as possible along each branch before backtracking. Here's a more detailed explanation:

Basic Concept:

- **Traversal Technique:** DFS traverses a graph or tree in a depthward motion rather than in breadth. It explores as far as possible along each branch before backtracking.
- **Stack or Recursion:** DFS can be implemented using either a stack data structure or recursion. In the recursive implementation, the call stack implicitly acts as the stack.

Process:

1. **Start at a Vertex:** Begin the traversal by starting at a chosen vertex.
2. **Explore Neighbors:** Visit the neighbors of the current vertex one by one.
3. **Recursion or Stack:** If using recursion, call the DFS function recursively for each unvisited neighbor. If using a stack, push unvisited neighbors onto the stack.
4. **Backtrack:** After exploring all neighbors, backtrack to the previous vertex.
5. **Continue Exploration:** Continue the process until all vertices have been visited.

Key Points:

- **Depth-First:** The name "Depth First" signifies that the traversal explores as deeply as possible along each branch before backtracking.
- **Visited Vertices:** DFS maintains a list of visited vertices to prevent revisiting already explored vertices and to avoid infinite loops in graphs with cycles.
- **Applications:** DFS is widely used in various applications such as topological sorting, cycle detection, connected components, pathfinding, and solving puzzles.

Code

```
import java.util.*;

class Graph {
   private int V;
   private LinkedList < Integer > [] adj;

   Graph(int v) {
      V = v;
      adj = new LinkedList[v];
      for (int i = 0; i < v; i++)
         adj[i] = new LinkedList();
   }

   void addEdge(int v, int w) {
      adj[v].add(w);
      adj[w].add(v);
   }

   void DFSUtil(int v, boolean[] visited) {
      visited[v] = true;
      System.out.println(v + " ");

      for (int n: adj[v]) {
         if (!visited[n]) {
            DFSUtil(n, visited);
         }
      }
   }

   void DFS(int v) {
      boolean[] visited = new boolean[V];
      DFSUtil(v, visited);
   }
}

class Main {
   public static void main(String args[]) {
      Graph g = new Graph(4);

      g.addEdge(0, 1);
      g.addEdge(0, 2);
      g.addEdge(1, 3);

      g.DFS(0);

   }
}
```

Complexity:

- **Time Complexity:** The time complexity of DFS is O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity:** The space complexity of DFS is O(V), where V is the number of vertices. This space is primarily used for the visited array or stack in the recursive implementation.

Depth First Search (DFS) algorithm has several advantages and disadvantages:

Advantages:

1. **Simplicity:** DFS is relatively simple to implement and understand, making it an accessible algorithm for beginners.
2. **Memory Efficiency:** DFS uses less memory compared to breadth-first search (BFS) as it explores as deeply as possible along each branch before backtracking, which requires maintaining only a small amount of additional data.
3. **Solves Problems:** DFS is a fundamental algorithm used in various graph-related problems such as finding connected components, determining if a graph contains cycles, and generating topological orderings.
4. **Space Efficiency (for Tree Traversal):** In tree structures, DFS can be implemented without using additional data structures like queues, making it memory-efficient.
5. **Finding Paths:** DFS can be used to find paths between two vertices, and it can generate all paths between two vertices if needed.

Disadvantages:

1. **Completeness:** DFS may not find a solution if there are infinitely deep branches or cycles in the graph. It can get stuck in infinite loops if not properly implemented with cycle detection mechanisms.
2. **Non-Optimal Solutions:** DFS does not guarantee finding the shortest path in weighted graphs. It may find a solution, but it may not be the shortest one.
3. **Memory Consumption (for Graph Traversal):** In graphs with a large number of vertices and edges, DFS can consume a lot of memory due to the need to maintain a visited array or stack, especially if implemented recursively.
4. **Vulnerability to Stack Overflow (Recursive Implementation):** If implemented recursively, DFS can encounter stack overflow errors for very deep graphs, especially in languages with limited stack size.
5. **Lack of Breadth-First Characteristics:** Since DFS explores as deeply as possible along each branch before backtracking, it may take longer to find a solution in graphs where the solution is located far from the starting vertex.

BFS

Breadth First Search (BFS) is a fundamental graph traversal algorithm used to explore or search through all the vertices of a graph or tree data structure. It systematically visits the vertices of a graph in a breadthward motion, visiting all the vertices at the present depth level before moving on to the vertices at the next depth level. Here's a detailed explanation of BFS:

Basic Concept:

- **Traversal Technique:** BFS traverses a graph or tree in a breadthward motion, exploring all the vertices at the present depth level before moving on to the vertices at the next depth level.
- **Queue:** BFS uses a queue data structure to keep track of vertices to be visited. The first-in, first-out (FIFO) nature of the queue ensures that vertices are visited in the order they were discovered.

Process:

1. **Start at a Vertex:** Begin the traversal by starting at a chosen vertex.
2. **Visit and Enqueue:** Visit the current vertex and enqueue its neighbors that have not been visited yet.
3. **Dequeue and Repeat:** Dequeue a vertex from the front of the queue and repeat the process for its unvisited neighbors.
4. **Level-by-Level:** BFS explores vertices level-by-level, ensuring that all vertices at one level are visited before moving to the next level.
5. **Termination:** Continue the process until the queue is empty, indicating that all vertices have been visited.

Key Points:

- **Breadth-First:** The name "Breadth First" signifies that the traversal explores vertices at each level of the graph before moving to the next level.
- **Visited Vertices:** BFS maintains a list of visited vertices to prevent revisiting already explored vertices and to avoid infinite loops in graphs with cycles.
- **Shortest Paths (for Unweighted Graphs):** BFS can be used to find the shortest path between two vertices in an unweighted graph. Since BFS explores vertices level-by-level, the first path found between two vertices is guaranteed to be the shortest.
- **Connected Components:** BFS can identify connected components in a graph efficiently.
- **Applications:** BFS is widely used in various applications such as shortest path finding, minimum spanning tree, network flow, and bipartite graph detection.

Complexity:

- **Time Complexity:** The time complexity of BFS is O(V + E), where V is the number of vertices and E is the number of edges.
- **Space Complexity:** The space complexity of BFS is O(V), where V is the number of vertices. This space is primarily used for the queue and the visited set.

Advantages:

1. **Shortest Path Finding (for Unweighted Graphs):** BFS guarantees to find the shortest path between two vertices in an unweighted graph. Since BFS explores vertices level by level, the first path found between two vertices is guaranteed to be the shortest.
2. **Optimality:** BFS provides an optimal solution for many problems such as finding the shortest path, minimum spanning tree, and bipartite graph detection.
3. **Completeness:** BFS is complete for finite graphs, meaning it will always terminate and explore all reachable vertices.
4. **Connected Components:** BFS efficiently identifies connected components in a graph, making it useful for analyzing the connectivity of the graph.
5. **Applications:** BFS is widely used in various applications such as network flow, minimum spanning tree, cycle detection, and solving puzzles like the 15-puzzle or maze traversal.

Disadvantages:

1. **Memory Consumption:** BFS can consume a significant amount of memory, especially for graphs with a large number of vertices and edges. This is because BFS typically requires maintaining a queue and a visited set/array to keep track of visited vertices.
2. **Time Complexity (for Dense Graphs):** In dense graphs with many edges, BFS may have a higher time complexity compared to other algorithms due to the need to explore all vertices level by level.
3. **Non-Optimal Solutions (for Weighted Graphs):** BFS does not guarantee finding the shortest path in weighted graphs. While it can find a solution, it may not be the shortest one.
4. **Not Suitable for Some Problems:** BFS may not be suitable for certain problems where depth information is important, or where exploring deeper levels of the graph is necessary.
5. **Space Complexity:** The space complexity of BFS is proportional to the number of vertices in the graph, which can be inefficient for large graphs.

Code

```
import java.util.*;
class Graph {
   private int V;
   private LinkedList < Integer > [] adj;

   Graph(int v) {
      V = v;
      adj = new LinkedList[v];
      for (int i = 0; i < v; i++) {
         adj[i] = new LinkedList < > ();
      }
   }

   void addEdge(int v, int w) {
      adj[v].add(w);
      adj[w].add(v);
   }


   void BFSUtil(Queue < Integer > queue, boolean[] visited) {
      if (queue.isEmpty())
         return;

      int v = queue.poll();
      System.out.print(v + " ");

      for (int neighbor: adj[v]) {
         if (!visited[neighbor]) {
            visited[neighbor] = true;
            queue.add(neighbor);
         }
      }

      BFSUtil(queue, visited);
   }

   void BFS(int start) {
      boolean[] visited = new boolean[V];
      Queue < Integer > queue = new LinkedList < > ();
      visited[start] = true;
      queue.add(start);
      BFSUtil(queue, visited);
   }
}


class Main {
   public static void main(String args[]) {
      Graph g = new Graph(4);
```

```
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Breadth First Traversal (BFS): ");
        g.BFS(0);

    }
}
```

2. Implement A star Algorithm for any game search problem.

here's an explanation of the theory behind implementing the A* algorithm for a game search problem:

1. Introduction to A* Algorithm:

The A* algorithm is a widely used pathfinding and graph traversal algorithm. It's an extension of Dijkstra's algorithm with enhancements that allow it to find the shortest path efficiently by using a heuristic to guide the search. A* is widely used in various applications including game development, robotics, and route planning.

2. Problem Statement:

In a game search problem, the goal is to find the shortest path from a starting point to a goal state while navigating through a graph or grid. This is a common problem in game development where characters or agents need to move through a game world efficiently.

3. Components of A* Algorithm:

- **Nodes**: Represent points or states in the game world. Each node has a position and may have additional properties.
- **Edges**: Represent connections between nodes. They define the possible transitions from one state to another.
- **Costs**: Each edge has a cost associated with it, which represents the effort or distance required to move from one node to another.
- **Heuristic Function (h)**: Estimates the cost from a node to the goal. It provides a rough estimate of the remaining cost to reach the goal from a given node.
- **g-score (g)**: Represents the cost of getting from the start node to the current node.
- **f-score (f)**: Represents the total estimated cost of the cheapest path from the start node to the goal node that goes through the current node. It is calculated as f = g + h.

4. Steps of A* Algorithm:

1. **Initialization**: Initialize the open set with the starting node and set its g-score to 0. Initialize the f-score with the heuristic estimate from the start node to the goal.
2. **Main Loop**: While the open set is not empty, do the following:
    - Select the node with the lowest f-score from the open set.
    - If the selected node is the goal node, reconstruct and return the path.
    - Expand the selected node by considering its neighbors.
    - For each neighbor, calculate its tentative g-score and update it if it's better than the previous one.
    - Calculate the f-score for each neighbor and add them to the open set if they are not already there.
3. **Termination**: If the open set becomes empty, then no path exists from the start node to the goal node.

5. Heuristic Function:

- The effectiveness of the A* algorithm heavily depends on the heuristic function. It should be admissible (never overestimating the true cost) and consistent (satisfy the triangle inequality).
- In game search problems, heuristics can be based on factors such as Euclidean distance, Manhattan distance, or any domain-specific knowledge that estimates the remaining cost to reach the goal.

6. Application in Game Development:

- In game development, A* is commonly used for pathfinding of non-player characters (NPCs) or game agents.
- It helps NPCs navigate complex environments efficiently, avoiding obstacles and finding optimal paths to reach their goals.
- A* can be integrated into the game engine to provide real-time pathfinding capabilities, enhancing the overall gameplay experience.

```
import java.util.*;

public class Test {

    // Define a static nested class Node
    static class Node {
        char name;
        int heuristic;
        int f, g, h; // f, g, h scores for A* algorithm
        Node parent;
        Map<Character, Integer> neighbors; // Neighbors of the node

        // Constructor for Node class
        Node(char name, int heuristic) {
            this.name = name;
            this.heuristic = heuristic;
            this.neighbors = new HashMap<>();
        }
    }

    // Method to calculate heuristic between two nodes
    static int heuristic(Node a, Node b) {
        return Math.abs(a.heuristic - b.heuristic);
    }

    // Method to find the shortest path using A* algorithm
    static List<Node> findPath(Map<Character, Node> graph, Node start, Node end) {
        // Priority queue to store nodes based on their f scores
        PriorityQueue<Node> openSet = new PriorityQueue<>((a, b) -> a.f - b.f);

        // Set to store visited nodes
        Set<Node> closedSet = new HashSet<>();

        openSet.add(start); // Add start node to the open set

        while (!openSet.isEmpty()) {
            Node current = openSet.poll(); // Get the node with lowest f score

            if (current.name == end.name) { // If reached the goal node
                List<Node> path = new ArrayList<>();
                Node temp = current;
                // Reconstruct the path from goal node to start node
                while (temp != null) {
                    path.add(temp);
                    temp = temp.parent;
                }
                Collections.reverse(path);
                return path; // Return the shortest path
            }

            closedSet.add(current); // Mark current node as visited
        // Iterate through neighbors of current node
        for (char neighborName : current.neighbors.keySet()) {
            Node neighbor = graph.get(neighborName);
            if (closedSet.contains(neighbor)) {
                continue; // Skip if neighbor is already visited
            }

            // Calculate tentative g score
            int tentativeGScore = current.g + current.neighbors.get(neighborName);

            // Update neighbor if it is not in open set or new path is shorter
            if (!openSet.contains(neighbor) || tentativeGScore < neighbor.g) {
                neighbor.parent = current;
                neighbor.g = tentativeGScore;
                neighbor.h = heuristic(neighbor, end);
                neighbor.f = neighbor.g + neighbor.h;

                if (!openSet.contains(neighbor)) {
                    openSet.add(neighbor); // Add neighbor to open set
                }
            }
        }
    }

    return null; // No path found
}

public static void main(String[] args) {
    // Create a graph with nodes and their heuristics
    Map<Character, Node> graph = new HashMap<>();
    graph.put('A', new Node('A', 14));
    graph.put('B', new Node('B', 12));
    graph.put('C', new Node('C', 11));
    graph.put('D', new Node('D', 6));
    graph.put('E', new Node('E', 4));
    graph.put('F', new Node('F', 11));
    graph.put('G', new Node('G', 0));

    // Define neighbors for each node
    graph.get('A').neighbors.put('B', 4);
    graph.get('A').neighbors.put('C', 3);
    graph.get('B').neighbors.put('F', 5);
    graph.get('B').neighbors.put('E', 12);
    graph.get('C').neighbors.put('D', 7);
    graph.get('C').neighbors.put('E', 10);
    graph.get('D').neighbors.put('E', 2);
    graph.get('E').neighbors.put('G', 5);
    graph.get('F').neighbors.put('G', 16);

    Node start = graph.get('A');
    Node end = graph.get('G');
```

```
        // Find the shortest path from start to end node
        List<Node> path = findPath(graph, start, end);

        // Print the shortest path and total cost if path exists
        if (path != null) {
            System.out.println("Shortest path found:");
            int totalCost = 0;
            for (int i = 0; i < path.size() - 1; i++) {
                Node currentNode = path.get(i);
                Node nextNode = path.get(i + 1);
                int costToNextNode = currentNode.neighbors.get(nextNode.name);
                System.out.print(currentNode.name + " -> ");
                totalCost += costToNextNode;
            }
            System.out.println(end.name + " (Total cost: " + totalCost + ")");
        } else {
            System.out.println("No path found.");
        }
    }
}
```

3. Implement Greedy search algorithm for Job scheduling problem

The Job Scheduling problem is a classic optimization problem where a set of jobs with associated deadlines and profits needs to be scheduled on a single machine to maximize the total profit. Each job has a deadline by which it needs to be completed, and there is only one instance of each job available. The goal is to select a subset of jobs to maximize the total profit while meeting all deadlines.

Greedy Approach:

A Greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. In the context of the Job Scheduling problem, the Greedy approach involves making a series of locally optimal choices with the hope that these choices will lead to a globally optimal solution.

Greedy Strategy for Job Scheduling:

The Greedy strategy for the Job Scheduling problem involves sorting the jobs based on some criteria and then selecting jobs iteratively, making locally optimal choices at each step.

Here's a typical Greedy strategy for Job Scheduling:

1. **Sort jobs:** Initially, sort the jobs based on some criteria. Commonly used criteria include deadlines or profits.
2. **Iterate through sorted jobs:** Iterate through the sorted list of jobs. At each step, select the job that maximizes the profit and can still be completed within its deadline without conflicting with previously selected jobs.
3. **Update schedule and profit:** After selecting a job, update the schedule to reflect the job's completion and add its profit to the total profit.
4. **Repeat:** Repeat the process until all jobs are processed or no more jobs can be scheduled.

Pseudocode

```
Function jobScheduling(jobs):
    Sort jobs based on some criteria (e.g., deadlines or profits)
    Initialize an empty list for selected jobs
    Initialize totalProfit to 0
    Initialize an array or list to track job slots

    For each job in sorted jobs:
        Iterate through the deadline slots from job's deadline to the beginning:
            If the slot is available:
                Add the job to selected jobs
                Update totalProfit
                Mark the slot as filled
                Break

    Return selected jobs and totalProfit
```

Analysis:

The Greedy approach for the Job Scheduling problem has a time complexity of O(n^2) because it involves sorting the jobs and then iterating through each job. However, it provides a simple and efficient solution for small to moderate-sized instances of the problem.

Complexity:

The overall time complexity of the Greedy algorithm for Job Scheduling is dominated by the sorting step, which is O(n log n). The selection and scheduling of jobs contribute an additional linear time complexity of O(n).

```
import java.util.*;

class Job implements Comparable < Job > {
   String jobName;
   int deadline;
   int profit;

   public Job(String jobName, int deadline, int profit) {
      this.jobName = jobName;
      this.deadline = deadline;
      this.profit = profit;
   }

   // Sort jobs by their deadlines
   @Override
   public int compareTo(Job other) {
      return this.deadline - other.deadline;
   }
}

public class Jobs {

   public static List < Job > jobScheduling(List < Job > jobList) {
      // Sort jobs by their deadlines
      /*In the provided Job class, the natural ordering is determined by the compareTo
method, which compares Job objects based on their deadlines.
      Therefore, when you call Collections.sort(jobList);
      it sorts the jobList in ascending order of deadlines. */

      Collections.sort(jobList);



      List < Job > selectedJobs = new ArrayList < > ();
      int totalProfit = 0;

      // Greedy selection
      boolean[] slot = new boolean[jobList.size()];
      Arrays.fill(slot, false);
```

```
        for (int i = 0; i < jobList.size(); i++) {
            for (int j = Math.min(jobList.get(i).deadline - 1, jobList.size() - 1); j >= 0; j--) {
                if (!slot[j]) {
                    selectedJobs.add(jobList.get(i));
                    totalProfit += jobList.get(i).profit;
                    slot[j] = true;
                    break;
                }
            }
        }

        // Display selected jobs and total profit
        System.out.println("Selected Jobs:");
        for (Job job: selectedJobs) {
            System.out.println("Job Name: " + job.jobName + ", Deadline: " + job.deadline + ", Profit: 
" + job.profit);
        }
        System.out.println("Total Profit: " + totalProfit);

        return selectedJobs;
    }

    public static void main(String[] args) {
        List < Job > jobList = new ArrayList < > ();
        jobList.add(new Job("Job1", 2, 50));
        jobList.add(new Job("Job2", 1, 15));
        jobList.add(new Job("Job3", 2, 10));
        jobList.add(new Job("Job4", 1, 25));

        // Sort jobs by their profit in descending order
        Collections.sort(jobList, (a, b) - > b.profit - a.profit);

        // Run job scheduling algorithm
        jobScheduling(jobList);
    }
}
```

4. Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queensproblem

Certainly! The N-Queens problem is a classic example of a Constraint Satisfaction Problem (CSP), where the goal is to place N queens on an $N \times N$ chessboard such that no two queens threaten each other. This problem can be efficiently solved using both Backtracking and Branch and Bound algorithms.

Backtracking:

- Backtracking is a systematic way to explore all possible configurations of a solution space by trying partial solutions and incrementally building towards a complete solution.
- In the context of the N-Queens problem, backtracking involves trying to place queens on the chessboard one by one, column by column. At each step, the algorithm checks if the current placement is valid. If it is, it proceeds to the next column; if not, it backtracks and tries a different placement.
- Backtracking is guided by constraints. In the case of N-Queens, the constraint is that no two queens can threaten each other, i.e., no two queens can share the same row, column, or diagonal.

Branch and Bound:

- Branch and Bound is a more advanced technique that enhances backtracking by intelligently pruning branches of the search tree that cannot lead to a solution.
- In the context of the N-Queens problem, Branch and Bound involves assigning a cost or score to each partial solution and prioritizing the exploration of more promising branches of the search tree while avoiding unpromising ones.
- The idea is to use bounds (upper and lower) to determine whether it is worth exploring a particular branch further. If the current partial solution cannot lead to a better solution than the best solution found so far, that branch is pruned.
- Branch and Bound often employs heuristics to estimate the potential of a partial solution and to guide the search towards the most promising areas of the solution space.

Combining Backtracking and Branch and Bound for N-Queens:

- In the N-Queens problem, Backtracking is the primary method used to explore the solution space, systematically trying different configurations until a solution is found.
- Branch and Bound can be applied to optimize Backtracking by intelligently pruning branches that are guaranteed to lead to suboptimal solutions or no solution at all.
- Heuristics such as the "minimum remaining values" heuristic or the "degree heuristic" can be used to prioritize the exploration of more promising branches.

- The key to combining Backtracking and Branch and Bound effectively is to maintain an efficient representation of the current state of the search, along with upper and lower bounds on the quality of partial solutions.

In summary, Backtracking and Branch and Bound are powerful techniques for solving Constraint Satisfaction Problems like the N-Queens problem. By systematically exploring the solution space and intelligently pruning branches, these algorithms can efficiently find optimal solutions or prove that no solution exists.

Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is a class of computational problems where the goal is to find a solution that satisfies a set of constraints. In a CSP, variables are associated with objects or entities, and the goal is to find values for these variables that meet certain criteria or conditions, subject to specified constraints. CSPs are prevalent in various fields, including artificial intelligence, operations research, scheduling, and planning.

Here are the key components of a Constraint Satisfaction Problem:

1. **Variables**: Variables represent the objects or entities whose values need to be determined. Each variable typically has a domain of possible values from which a value must be chosen to satisfy the constraints.
2. **Domains**: The domain of a variable consists of all possible values that the variable can take. The domain can be discrete, finite, or infinite, depending on the problem domain. For example, in the N-Queens problem, each variable represents the position of a queen on the chessboard, and the domain for each variable is the set of all possible positions within the board.
3. **Constraints**: Constraints specify the relationships or restrictions among the variables. They define which combinations of variable assignments are allowed and which are not. Constraints can be unary (restrictions on individual variables), binary (restrictions between pairs of variables), or higher-order (involving more than two variables). Constraints can be specified using logical expressions, equations, or other formalisms depending on the problem.
4. **Objective Function (optional)**: In some CSPs, there may be an objective function that needs to be optimized while satisfying the constraints. The objective function assigns a value to each possible solution, and the goal is to find the solution that maximizes or minimizes this value.
5. **Solution**: A solution to a CSP is an assignment of values to all variables such that all constraints are satisfied. If an objective function is present, the solution is typically chosen to optimize the objective function.

Example of Constraint Satisfaction Problems include:

- **N-Queens Problem**: Place N queens on an $N \times N$ chessboard such that no two queens threaten each other.

- **Sudoku**: Fill a 9×9 grid with digits such that each row, each column, and each of the nine 3×3 subgrids contain all of the digits from 1 to 9.
- **Graph Coloring**: Color the vertices of a graph such that no two adjacent vertices share the same color.
- **Job Scheduling**: Assign tasks to workers over time subject to constraints on task dependencies and resource availability.

Solving a CSP involves finding a suitable algorithm or method to explore the solution space efficiently while ensuring that all constraints are satisfied. Common techniques for solving CSPs include Backtracking, Constraint Propagation, Local Search, Genetic Algorithms, and Mixed Integer Programming.

```
public class queen {

    // Function to solve N-Queens problem
    public static void solveNQueens(int n) {
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = '.';
            }
        }
        solveNQueensUtil(board, 0);
    }

    // Backtracking function
    private static boolean solveNQueensUtil(char[][] board, int col) {
        int n = board.length;
        if (col == n) {
            // Solution found, print the board
            printBoard(board);
            return true;
        }

        boolean res = false;
        for (int i = 0; i < n; i++) {
            if (isSafe(board, i, col)) {
                board[i][col] = 'Q';
                if (solveNQueensUtil(board, col + 1)) {
                    res = true;
                }
                // Backtrack
                board[i][col] = '.';
            }
        }
        return res;
    }
    // Check if placing a queen at board[row][col] is safe
    private static boolean isSafe(char[][] board, int row, int col) {
        int i, j;
        int n = board.length;

        // Check this row on left side
        for (i = 0; i < col; i++) {
            if (board[row][i] == 'Q') return false;
        }

        // Check upper diagonal on left side
        for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') return false;
        }

        // Check lower diagonal on left side
        for (i = row, j = col; j >= 0 && i < n; i++, j--) {
            if (board[i][j] == 'Q') return false;
        }

        return true;
    }

    // Print the board
    private static void printBoard(char[][] board) {
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[i].length; j++) {
                System.out.print(board[i][j] + " ");
            }
            System.out.println();
        }
        System.out.println();
    }

    public static void main(String[] args) {
        int n = 4; // Change n for different board sizes
        solveNQueens(n);
    }
}
```

5. Develop an elementary chatbot for any suitable customer interaction application.

Let's delve into the theory of developing an elementary chatbot for a customer interaction application.

Understanding Chatbots:

Chatbots are computer programs designed to simulate human conversation through text or voice interactions. They use natural language processing (NLP) techniques to understand and generate responses to user queries. Chatbots can be used for various purposes, including customer service, information retrieval, entertainment, and more.

Components of a Chatbot:

1. **Input Processing**: The chatbot receives user input, which can be in the form of text, voice, or other media. This input is then processed to extract relevant information and determine the user's intent.
2. **Natural Language Understanding (NLU)**: NLU is the component responsible for interpreting and understanding the user's input. It analyzes the input to extract entities, intents, and context, which are used to generate an appropriate response.
3. **Knowledge Base**: The chatbot needs access to a knowledge base containing information or responses to various queries. This knowledge base can be predefined or dynamically generated based on user interactions.
4. **Response Generation**: Based on the user's input and the information stored in the knowledge base, the chatbot generates a response. This response should be relevant, informative, and tailored to the user's query.
5. **Output Rendering**: Finally, the chatbot presents the generated response to the user in a suitable format, such as text, voice, or graphical interface.

Development Process:

1. **Define Use Case**: Determine the purpose and scope of the chatbot. Identify the target audience and the types of interactions the chatbot will handle.
2. **Select Platform**: Choose a platform or framework for building the chatbot. Common options include Python libraries like ChatterBot, Dialogflow, or platforms like Microsoft Bot Framework, Facebook Messenger, etc.
3. **Data Collection**: Collect relevant data or information that the chatbot will use to respond to user queries. This could include FAQs, product information, support documentation, etc.
4. **Training**: Train the chatbot using the collected data or pre-existing datasets. For machine learning-based approaches, this involves training models to understand user inputs and generate appropriate responses.

5. **Integration**: Integrate the chatbot into the chosen platform or application where users will interact with it. This may involve deploying the chatbot on a website, messaging platform, or voice assistant.
6. **Testing and Evaluation**: Test the chatbot to ensure that it functions correctly and provides accurate responses. Gather feedback from users and iterate on the design to improve performance.
7. **Maintenance and Updates**: Regularly maintain and update the chatbot to keep it relevant and effective. Monitor user interactions and make adjustments as needed to improve the user experience.

Conclusion:

Developing an elementary chatbot for customer interaction involves understanding the components involved, selecting appropriate tools and platforms, and following a structured development process. By focusing on user needs, collecting relevant data, and continuously improving the chatbot's performance, you can create a valuable and effective customer interaction tool.

```
import streamlit as st

bot_name = "College Buddy"

knowledge = {
    "what is your name": ["My name is {bot_name}, How are you!"],
    "list some colleges": ["Here are some colleges Harvard, Boston, Mit"],
    "how are you": ["I am doing real good"],
}

st.header("College Enquiry")



def respond(input_text: str):
    if input_text in knowledge:
        for value in knowledge[input_text]:
            st.write(value)
    else:
        st.subheader("Cannot understand")



if __name__ == "__main__":
    input_text = st.text_input("Enter your text").lower()
    ask_button = st.button("Ask")
    if ask_button:
        respond(input_text)
```

6. Implement Expert System

What is an Expert System?

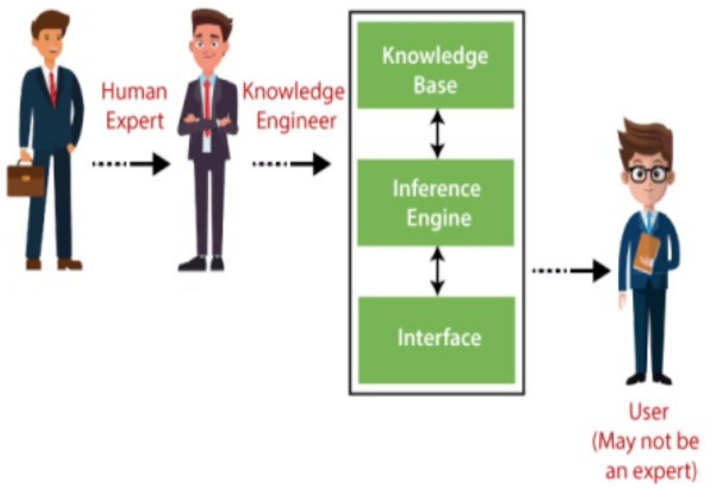
An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert. It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.The expert system is a part of AI, and the first ES was developed in the year 1970, which was the first successful approach of artificial intelligence. It solves the most complex issue as an expert by extracting the knowledge stored in its knowledge base. The system helps in decision making for compsex problems using both facts and heuristics like a human expert. It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that particular domain. These systems are designed for a specific domain, such as medicine, science, etc.
The performance of an expert system is based on the expert's knowledge stored in its knowledge base. The more knowledge stored in the KB, the more that system improves its performance. One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.

Below are some popular examples of the Expert System:

- **DENDRAL**: It was an artificial intelligence project that was made as a chemical analysis expert system. It was used in organic chemistry to detect unknown organic molecules with the help of their mass spectra and knowledge base of chemistry.
- **MYCIN**: It was one of the earliest backward chaining expert systems that was designed to find the bacteria causing infections like bacteraemia and meningitis. It was also used for the recommendation of antibiotics and the diagnosis of blood clotting diseases.
- **PXDES**: It is an expert system that is used to determine the type and level of lung cancer. To determine the disease, it takes a picture from the upper body, which looks like the shadow. This shadow identifies the type and degree of harm.
- **CaDeT**: The CaDet expert system is a diagnostic support system that can detect cancer a early stages.

Characteristics of Expert System

- **High Performance**: The expert system provides high performance for solving any type of complex problem of a specific domain with high efficiency and accuracy.
- **Understandable**: It responds in a way that can be easily understandable by the user. It can takeinput in human language and provides the output in the same way.
- **Reliable**: It is much reliable for generating an efficient and accurate output.
- **Highly responsive**: ES provides the result for any complex query within a very short period of time.



Components of Expert System

An expert system mainly consists of three components:

o User Interface

o Inference Engine

o Knowledge Base

Advantages of Expert System

- These systems are highly reproducible.o They can be used for risky places where the human presence is not safe.
- Error possibilities are less if the KB contains correct knowledge.
- The performance of these systems remains steady as it is not affected by emotions, tension, or fatigue.
- They provide a very high speed to respond to a particular query.

Limitations of Expert System

- The response of the expert system may get wrong if the knowledge base contains the wrong information.
- Like a human being, it cannot produce a creative output for different scenarios.
- Its maintenance and development costs are very high.
- Knowledge acquisition for designing is much difficult.
- For each domain, we require a specific ES, which is one of the big limitations.
- It cannot learn from itself and hence requires manual updates.

Applications of Expert System

In designing and manufacturing domain

It can be broadly used for designing and manufacturing physical devices such as camera lenses and automobiles.

In the knowledge domain

These systems are primarily used for publishing the relevant knowledge to the users. The two popular ES used for this domain is an advisor and a tax advisor.

In the finance domain

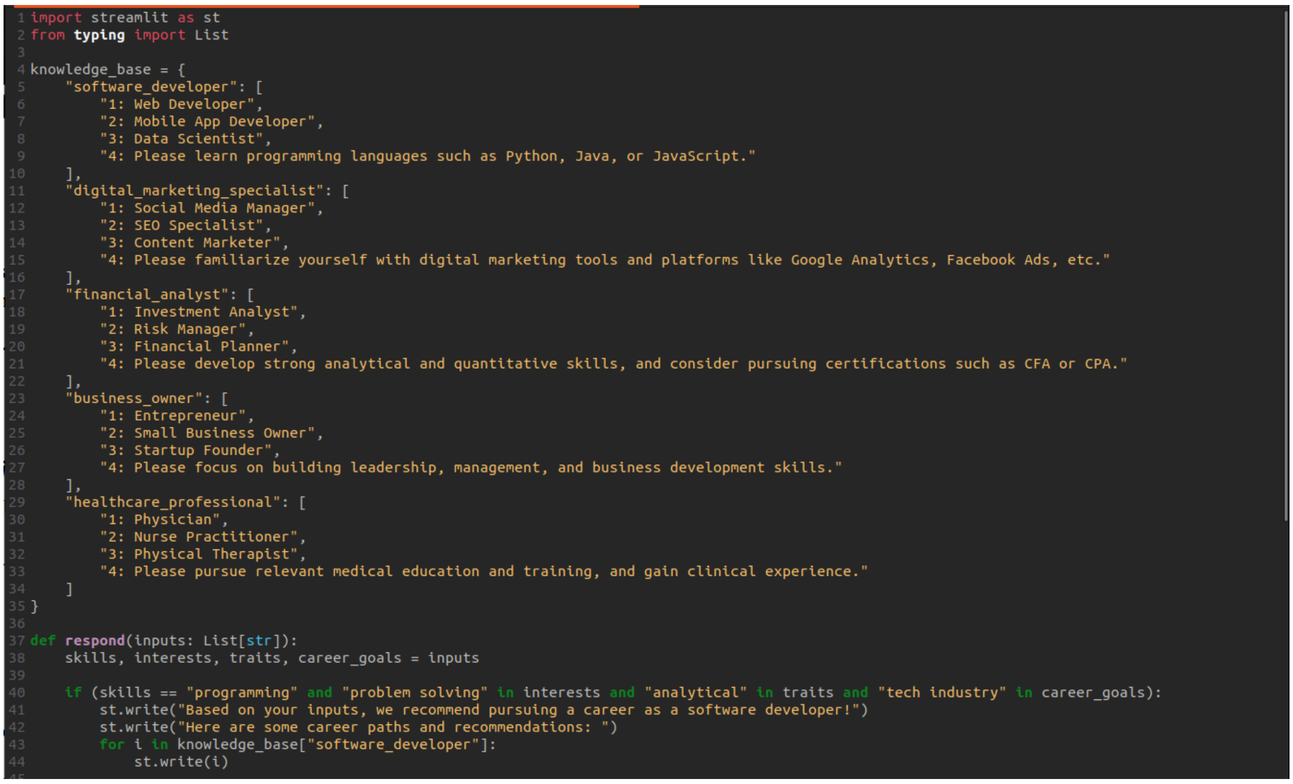
In the finance industries, it is used to detect any type of possible fraud, suspicious activity, and advise bankers that if they should provide loans for business or not.

In the diagnosis and troubleshooting of devices

In medical diagnosis, the ES system is used, and it was the first area where these systems were used.

Planning and Scheduling

The expert systems can also be used for planning and scheduling some particular tasks for achieving the goal of that task.

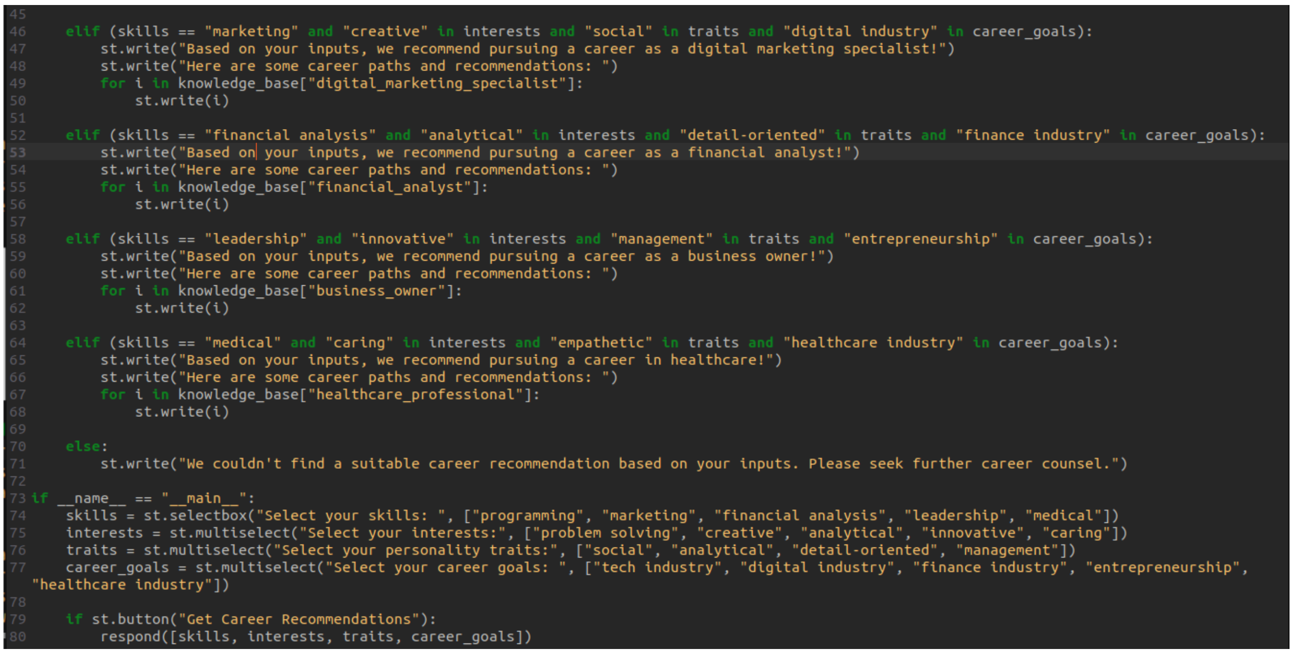


```python
import streamlit as st
from typing import List

knowledge_base = {
    "software_developer": [
        "1: Web Developer",
        "2: Mobile App Developer",
        "3: Data Scientist",
        "4: Please learn programming languages such as Python, Java, or JavaScript."
    ],
    "digital_marketing_specialist": [
        "1: Social Media Manager",
        "2: SEO Specialist",
        "3: Content Marketer",
        "4: Please familiarize yourself with digital marketing tools and platforms like Google Analytics, Facebook Ads, etc."
    ],
    "financial_analyst": [
        "1: Investment Analyst",
        "2: Risk Manager",
        "3: Financial Planner",
        "4: Please develop strong analytical and quantitative skills, and consider pursuing certifications such as CFA or CPA."
    ],
    "business_owner": [
        "1: Entrepreneur",
        "2: Small Business Owner",
        "3: Startup Founder",
        "4: Please focus on building leadership, management, and business development skills."
    ],
    "healthcare_professional": [
        "1: Physician",
        "2: Nurse Practitioner",
        "3: Physical Therapist",
        "4: Please pursue relevant medical education and training, and gain clinical experience."
    ]
}

def respond(inputs: List[str]):
    skills, interests, traits, career_goals = inputs

    if (skills == "programming" and "problem solving" in interests and "analytical" in traits and "tech industry" in career_goals):
        st.write("Based on your inputs, we recommend pursuing a career as a software developer!")
        st.write("Here are some career paths and recommendations: ")
        for i in knowledge_base["software_developer"]:
            st.write(i)
```



```python

    elif (skills == "marketing" and "creative" in interests and "social" in traits and "digital industry" in career_goals):
        st.write("Based on your inputs, we recommend pursuing a career as a digital marketing specialist!")
        st.write("Here are some career paths and recommendations: ")
        for i in knowledge_base["digital_marketing_specialist"]:
            st.write(i)

    elif (skills == "financial analysis" and "analytical" in interests and "detail-oriented" in traits and "finance industry" in career_goals):
        st.write("Based on your inputs, we recommend pursuing a career as a financial analyst!")
        st.write("Here are some career paths and recommendations: ")
        for i in knowledge_base["financial_analyst"]:
            st.write(i)

    elif (skills == "leadership" and "innovative" in interests and "management" in traits and "entrepreneurship" in career_goals):
        st.write("Based on your inputs, we recommend pursuing a career as a business owner!")
        st.write("Here are some career paths and recommendations: ")
        for i in knowledge_base["business_owner"]:
            st.write(i)

    elif (skills == "medical" and "caring" in interests and "empathetic" in traits and "healthcare industry" in career_goals):
        st.write("Based on your inputs, we recommend pursuing a career in healthcare!")
        st.write("Here are some career paths and recommendations: ")
        for i in knowledge_base["healthcare_professional"]:
            st.write(i)

    else:
        st.write("We couldn't find a suitable career recommendation based on your inputs. Please seek further career counsel.")

if __name__ == "__main__":
    skills = st.selectbox("Select your skills: ", ["programming", "marketing", "financial analysis", "leadership", "medical"])
    interests = st.multiselect("Select your interests:", ["problem solving", "creative", "analytical", "innovative", "caring"])
    traits = st.multiselect("Select your personality traits:", ["social", "analytical", "detail-oriented", "management"])
    career_goals = st.multiselect("Select your career goals: ", ["tech industry", "digital industry", "finance industry", "entrepreneurship",
"healthcare industry"])

    if st.button("Get Career Recommendations"):
        respond([skills, interests, traits, career_goals])
```