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

**Definition:**

Depth First Search (DFS) is a **graph traversal algorithm** used to explore all the nodes and edges of a **graph** or **tree** data structure. It starts at a **starting vertex (source)** and explores as far as possible along each branch before backtracking.

DFS can be applied to **both directed and undirected graphs** and is useful for exploring **connected components** and **solving complex problems like cycle detection, pathfinding, and more**.

**🔹 Key Concepts:**

* DFS uses **stack-based traversal**. This stack can be **explicit** (user-defined stack) or **implicit** (handled by recursive function calls).
* It marks each visited vertex to avoid revisiting nodes and getting stuck in an infinite loop.
* DFS explores **deep** into the graph before exploring sibling nodes.

**🔹 Applications of DFS:**

* Solving puzzles and games (like Sudoku, maze solving)
* Pathfinding problems

**🔹 Algorithm Steps (Recursive DFS):**

1. Start from the **source vertex**.
2. Mark the current vertex as **visited**.
3. For each **unvisited adjacent vertex**, **recursively** perform DFS.
4. Continue the process until all reachable vertices are visited.

**🔹 DFS Algorithm (Pseudocode):**

DFS(vertex):

mark vertex as visited

for each neighbor of vertex:

if neighbor is not visited:

DFS(neighbor)

**🔹 Time Complexity:**

* **O(V + E)**  
  Where:
  + V = number of vertices
  + E = number of edges  
    This is because each vertex and each edge is explored once.

**🔹 Space Complexity:**

* **O(V)** in the worst case (due to recursive call stack or visited list).

2. Implement Breadth First Search algorithm, use an undirected graph and develop an algorithm for searching all the vertices of a graph or tree data structure.

**Definition:**

Breadth First Search (BFS) is a **graph traversal algorithm** that explores the vertices of a graph in the order of their **distance from the starting point**, meaning it visits all neighbors at the current level before moving on to nodes at the next level.

BFS is particularly useful for finding the **shortest path** (minimum number of edges) between nodes in an **unweighted graph**.

**🔹 Key Concepts:**

* BFS uses a **queue** data structure to maintain the order of traversal.
* It explores all the neighbors of the current node before moving on to their neighbors.
* Each vertex is marked **visited** once it's enqueued to avoid repeated processing.

**🔹 Applications of BFS:**

* Finding the shortest path in an unweighted graph
* GPS navigation systems
* Detecting connected components

**🔹 Algorithm Steps (for BFS in an Undirected Graph):**

1. Create a queue and enqueue the **starting vertex**.
2. Mark the starting vertex as **visited**.
3. While the queue is not empty:
   * Dequeue a vertex from the queue.
   * Visit all **unvisited neighbors** of that vertex.
   * Mark each as visited and enqueue them.

**🔹 BFS Algorithm (Pseudocode):**

BFS(start\_vertex):

create an empty queue

mark start\_vertex as visited

enqueue start\_vertex

while queue is not empty:

current = dequeue()

for each neighbor of current:

if neighbor is not visited:

mark neighbor as visited

enqueue(neighbor)

**🔹 Time Complexity:**

* **O(V + E)**  
  Where:
  + V = number of vertices
  + E = number of edges  
    This is because each vertex and edge is processed once.

**🔹 Space Complexity:**

* **O(V)** due to the visited list and queue storage.

3. Implement A\* Algorithm for any game search problem.

**Definition:**

The A\* algorithm is a **graph traversal and pathfinding algorithm** widely used in game development and artificial intelligence. It is designed to find the **shortest path** from a **start node** to a **goal node**, while using **heuristic estimation** to optimize the search.

A\* combines the advantages of:

* **Dijkstra’s Algorithm** (which guarantees the shortest path),
* and **Greedy Best-First Search** (which uses heuristics to guide the search faster).

**🔹 Core Idea:**

A\* evaluates nodes using the function:

**✅ f(n) = g(n) + h(n)**

Where:

* **f(n)** = total cost of the node n
* **g(n)** = cost from the start node to node n (actual cost so far)
* **h(n)** = estimated cost from node n to the goal (heuristic)

The **heuristic function h(n)** is problem-specific and must be **admissible**, i.e., it should **never overestimate** the actual cost to reach the goal.

**🔹 *Applications of A Algorithm:*\***

* Game AI (e.g., NPC pathfinding)
* Robot navigation
* Solving puzzles (like 8-puzzle, 15-puzzle)

**🔹 *A Algorithm Steps:*\***

1. Add the **start node** to an **open list** (nodes to be evaluated).
2. Initialize an empty **closed list** (nodes already evaluated).
3. While the open list is not empty:
   * Pick the node with the **lowest f(n)** from the open list.
   * If it's the **goal node**, stop and reconstruct the path.
   * Move the node to the **closed list**.
   * For each **neighbor** of the current node:
     + If it is in the closed list, skip it.
     + Calculate g(n), h(n), and f(n) for the neighbor.
     + If it is **not in the open list**, add it.
     + If it is already in the open list with a higher f(n), update its values.

**🔹 *Pseudocode for A Algorithm:*\***

function A\*(start, goal):

open\_set = {start}

came\_from = empty map

g\_score[start] = 0

f\_score[start] = heuristic(start, goal)

while open\_set is not empty:

current = node in open\_set with lowest f\_score

if current == goal:

return reconstruct\_path(came\_from, current)

remove current from open\_set

for each neighbor of current:

tentative\_g\_score = g\_score[current] + cost(current, neighbor)

if tentative\_g\_score < g\_score[neighbor]:

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = g\_score[neighbor] + heuristic(neighbor, goal)

if neighbor not in open\_set:

add neighbor to open\_set

**🔹 Time and Space Complexity:**

* **Time Complexity:** Depends on the heuristic, but in the worst case it is **O(b^d)**  
  where b is branching factor, and d is depth of the solution.
* **Space Complexity:** **O(b^d)**, for storing all nodes in memory.

**🔹 Heuristic Examples:**

* **Manhattan Distance:** For grid-based movement (up/down/left/right).
* **Euclidean Distance:** For diagonal or free movement.

**Heuristic – Simple Theory**

A **heuristic** is a technique used to make a **smart guess** or **estimate** about the cost to reach the goal from a current position.

In search algorithms like **A\***, the heuristic helps to **guide the search** in the right direction by estimating the **remaining distance** to the target.

4. Implement Greedy search algorithm for any of the following application:

I. Selection Sort

II. Minimum Spanning Tree

III. Single-Source Shortest Path Problem

IV. Job Scheduling Problem

V. Prim's Minimal Spanning Tree Algorithm

VI. Kruskal's Minimal Spanning Tree Algorithm

VII. Dijkstra's Minimal Spanning Tree Algorithm

5. Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem.

**N-Queens Problem – Theory**

The **N-Queens problem** is a classic **Constraint Satisfaction Problem (CSP)** in which the goal is to place **N queens** on an **N×N chessboard** such that:

* No two queens attack each other.
* This means **no two queens** share the same **row**, **column**, or **diagonal**.

**🔄 Backtracking Approach:**

* Backtracking is a general method to solve CSPs.
* It builds the solution **step-by-step** and **removes** (backtracks) wrong choices as soon as a conflict is found.
* It tries placing a queen row by row and checks for **safety** before moving to the next row.

**✏️ Algorithm Steps: (Backtracking)**

1. Start with the first row.
2. For each column in the current row:
   * Check if placing a queen is **safe** (no conflicts).
   * If safe, place the queen and **recurse** for the next row.
   * If a solution is found, return.
   * If not, **remove** the queen (backtrack) and try next column.
3. Repeat until all queens are placed or no solution is possible.

**🔍 Branch and Bound Optimization:**

Branch and Bound is an improvement over backtracking that **cuts off branches early** using extra constraints.

**✏️ Key Concepts Used in Branch and Bound:**

* Maintain:
  + A boolean array for **columns** (column[ ])
  + A boolean array for **main diagonals** (diag1[ ])
  + A boolean array for **anti-diagonals** (diag2[ ])
* Before placing a queen, **check if the column and diagonals are free**.
* This avoids checking the entire board and **speeds up the process**.

**✅ Time Complexity:**

* Worst-case: **O(N!)**
* With optimizations (like branch and bound), it becomes **much faster in practice**.

**🧠 Summary:**

* The N-Queens problem is solved using **Backtracking** by placing one queen at a time and undoing steps when necessary.
* **Branch and Bound** improves it by eliminating invalid options early, using constraints on **columns** and **diagonals**.

6. Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for graph coloring problem.

**Definition:**

The **Graph Coloring Problem** is a type of **Constraint Satisfaction Problem (CSP)** where each vertex of a graph must be assigned a color such that **no two adjacent vertices** share the same color.

The goal is to **color the graph using the minimum number of colors** (or with a given number of colors m) while satisfying the constraint.

**🔄 Backtracking Approach:**

* This method tries to assign colors to vertices **one by one**.
* At each step, it checks if the current color assignment is **valid**.
* If a conflict occurs, it **backtracks** and tries a different color.
* It continues recursively until all vertices are colored or a conflict is unavoidable.

**🚫 Branch and Bound Optimization:**

Branch and Bound enhances backtracking by **cutting off** branches (partial solutions) that **cannot possibly lead to a valid or better solution**.

**✏️ How It Works:**

* A **bound** is used to **limit the number of colors** or prevent invalid assignments early.
* Before assigning a color, it checks:
  + If the color is **not used by adjacent vertices**
  + If the current partial coloring can lead to a valid full coloring

If any of these checks fail, the branch is **pruned**.

**🧮 Algorithm Steps (Backtracking + Branch and Bound):**

1. Start with the first vertex.
2. For each color from 1 to m:
   * Check if the color is **safe** (not used by any adjacent vertex).
   * If yes, assign the color.
   * Recurse to assign color to the next vertex.
   * If coloring leads to a solution, return success.
   * If not, **remove** the color (backtrack) and try next.
3. If no color can be assigned, backtrack further.
4. If all vertices are colored, return success.

**🧠 Example Use Case:**

* Coloring a map where no two neighboring regions can have the same color.

**✅ Time Complexity:**

* Worst-case: **O(m^V)**  
  Where:
  + m = number of colors
  + V = number of vertices

Optimized with Branch and Bound, many invalid colorings are skipped early.

**📌 Conclusion:**

The **Graph Coloring Problem** can be efficiently solved using **Backtracking** by exploring all valid color combinations, and **Branch and Bound** helps to prune unnecessary paths by applying additional constraints.

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

**What is a Chatbot?**

A **chatbot** is a computer program that simulates a **conversation with a human user** using text or voice. It is commonly used in **customer support**, **online shopping**, **banking**, and **web applications**.

An **elementary chatbot** is a **rule-based or keyword-matching system** that provides **predefined responses** to user queries.

**🔹 Use Case Example:**

Answer common questions like:

* + “What are your working hours?”
  + “How do I track my order?”
  + “How can I return a product?”
* Give basic guidance or redirect to a human agent

**🔹 Components of an Elementary Chatbot:**

1. **User Input:** The message typed by the customer.
2. **Keyword Matching / Rule Base:** The chatbot looks for specific **keywords** or **phrases**.
3. **Response Generator:** Based on the matched keyword, the bot gives a **predefined reply**.
4. **Output:** The response is shown back to the user.

**🧠 Working Logic:**

* If the input contains **"track"**, respond with tracking instructions.
* If no keyword is matched, respond with:  
  *"I'm sorry, I didn't understand that. Please try again or contact support."*

**✏️ Algorithm Steps:**

1. Take input from the user.
2. Convert input to lowercase and tokenize.
3. Search for keywords in the input.
4. If a match is found, return the predefined response.
5. If no match, return a default message.

**🔸 Advantages of an Elementary Chatbot:**

* Simple and easy to build
* Saves time for human agents

8. Case study on Amazon EC2 and learn about Amazon EC2 web services.

9. Installation and configure Google App Engine.

10. Creating an Application in SalesForce.com using Apex programming Language.