**Ass 1 (DFS AND BFS)**

**Deapth first search(DFS)**

* DFS is a graph traversal algorithm that starts from a source node and explores as far as possible along each branch before backtracking.
* It uses recursion or a stack (explicit or implicit).

**DFS Algorithm in Words (Iterative using Stack):**

1. Use a **stack** to keep track of nodes.
2. Push the starting node onto the stack.
3. While the stack is not empty:
   * Pop the top node.
   * If it hasn’t been visited, mark it and process it.
   * Push all **unvisited neighbors** onto the stack.
4. Repeat until stack is empty.

**Applications of DFS**

* Cycle detection in a graph
* Topological sorting
* Solving puzzles (like mazes or Sudoku)
* Finding connected components
* Pathfinding in AI (e.g. depth-first in games)

**Breath first Search(BFS)**

* BFS is a **graph traversal** algorithm that explores all neighbors of a node before moving to the next level.
* It uses a queue (FIFO).
* **BFS Algorithm in Words:**

1. Start from the selected node (root or source).
2. Mark it as **visited** and place it in a **queue**.
3. While the queue is not empty:
   * Dequeue a node from the front.
   * Process it.
   * For each **unvisited neighbor**, mark it as visited and enqueue it.
4. Repeat until all nodes are visited.

* **💼 Applications of BFS:**
* Finding the **shortest path** in an unweighted graph.
* **Web crawling** (e.g., Google bots visiting pages layer by layer).
* **Social networks** (finding levels of friendship).
* **Broadcasting** in computer networks.
* **Solving puzzles** (like sliding puzzles where you explore all possible moves).

**✅ Difference Between DFS and BFS**

| **Feature** | **DFS (Depth First Search)** | **BFS (Breadth First Search)** |
| --- | --- | --- |
| **Traversal Order** | Explores as deep as possible before backtracking | Explores all neighbors level by level |
| **Data Structure** | Stack (or Recursion) | Queue |
| **Path Discovery** | May not give shortest path | Guarantees shortest path in unweighted graphs |
| **Space Complexity** | O(V) in recursion stack or visited list | O(V) for visited list and queue |
| **Used For** | Cycle detection, topological sorting, puzzles, component finding | Shortest path, level order traversal, peer-to-peer networks |
| **Implementation** | Recursive or Stack-based | Queue-based |

**✅ Time Complexity of BFS and DFS**

Let:

* V = number of vertices
* E = number of edges

| **Algorithm** | **Time Complexity** | **Why?** |
| --- | --- | --- |
| **DFS** | O(V + E) | Visits every vertex and edge once |
| **BFS** | O(V + E) | Visits every vertex and edge once |

**Ass 2(A\*)**

A\* is a graph/tree search algorithm used to find the shortest path between a start and a goal node.  
It is widely used in pathfinding and graph traversal, especially in games, maps, and AI.

**A\* uses both:**

* **The actual cost to reach a node from the start (called g(n))**
* **A heuristic estimate of the cost from that node to the goal (called h(n))**

**It combines them into one formula:**

**Copy code**

**f(n) = g(n) + h(n)**

**Where:**

* **f(n) is the total cost of the path through node n**
* **g(n) is the cost from the start node to n**
* **h(n) is the heuristic function — an estimated cost from n to the goal**

**A\* Algorithm in Simple Steps:**

1. **Put the start node in a priority queue (open list).**
2. **While the queue is not empty:**
   * **Remove the node n with the lowest f(n)**
   * **If n is the goal, return the path**
   * **Otherwise:**
     + **For each neighbor m of n:**
       - **Calculate g(m) = g(n) + cost(n, m)**
       - **Calculate f(m) = g(m) + h(m)**
       - **If m was not visited or has a lower cost now, update and add to queue.**
3. **Keep track of visited nodes (closed list) to avoid re-checking.**

**Applications of A\* Algorithm:**

* **Game AI (finding the shortest path in a game map)**
* **Google Maps / GPS (pathfinding on roads)**
* **Robot Navigation**
* **Puzzle solvers (like 8-puzzle, sliding tiles)**
* **Network routing**

| **Aspect** | **Informed Search** | **Uninformed Search** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Heuristics** | **Uses heuristics to guide the search (e.g., A\*)** | **Does not use heuristics** |

|  |  |  |
| --- | --- | --- |
| **Efficiency** | **More efficient in many cases** | **Less efficient, explores blindly** |

|  |  |  |
| --- | --- | --- |
| **Optimality** | **Can be optimal if the heuristic is admissible** | **Can be optimal (BFS, UCS)** |

|  |  |  |
| --- | --- | --- |
| **Search Direction** | **Goal-directed (prioritizes promising paths)** | **Blind search (explores all paths equally)** |

|  |  |  |
| --- | --- | --- |
| **Examples** | **A\*, Greedy Best-First, IDA\*** | **BFS, DFS, UCS, Dijkstra** |

|  |  |  |
| --- | --- | --- |
| **Space Complexity** | **Can be high (depends on the heuristic used)** | **Can also be high, especially for BFS and DFS** |

**ASS3**

**1 selection sort**

**📊 Selection Sort:**

**Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest) element from an unsorted part of the list and moving it to the sorted portion of the list.**

**Algorithm for Selection Sort:**

**Steps:**

1. **Start with the first element of the array.**
2. **Find the minimum element in the unsorted part of the array (from the current element to the end of the array).**
3. **Swap the found minimum element with the first element of the unsorted part.**
4. **Move the boundary of the sorted and unsorted part one element to the right.**
5. **Repeat the process for the remaining unsorted portion of the array.**
6. **Continue until the entire array is sorted.**

**Time complexity**

* **Best-case time complexity: O(n^2)**
  + **This occurs if the input array is already sorted. Even in this case, the algorithm will still go through all comparisons.**
* **Average-case time complexity: O(n^2)**
  + **This is the general case, where the algorithm will have to go through all comparisons and swaps.**
* **Worst-case time complexity: O(n^2)**
  + **This happens when the array is sorted in reverse order, requiring the maximum number of comparisons and swaps.**

**2) prims algorithm**

**Prim's Algorithm:**

Prim's Algorithm is a greedy algorithm that is used to find the minimum spanning tree (MST) of a weighted, connected graph. The minimum spanning tree is a subset of the edges that connects all the vertices of the graph, without forming any cycle, and with the minimum possible total edge weight.

**Working of Prim's Algorithm:**

Prim’s algorithm starts with an arbitrary node (vertex) and grows the MST by adding the shortest possible edge that connects a vertex in the MST to a vertex outside it, until all vertices are included.

Steps of Prim’s Algorithm:

1. Initialize the MST:
   * Start with an arbitrary node, mark it as part of the MST, and set its key value to 0 (indicating the smallest weight).
   * Set the key values for all other vertices to infinity.
2. Select the Minimum Edge:
   * From the set of vertices not yet in the MST, pick the vertex that has the smallest key value (this vertex has the minimum weight edge connecting it to the MST).
3. Update the Key Values:
   * For each neighbor of the selected vertex, check if the weight of the edge from the vertex to the neighbor is smaller than the current key value of the neighbor. If it is, update the key value of the neighbor and set the parent of that vertex to the current vertex.
4. Repeat:
   * Continue adding vertices to the MST by picking the vertex with the smallest key value (among those not yet in the MST).
5. Stop when all vertices are included in the MST.

**Time Complexity of Prim's Algorithm:**

* **Using an adjacency matrix**:
  + The time complexity of Prim’s algorithm using an adjacency matrix is **O(V^2)**, where V is the number of vertices in the graph.
  + This is because, for each vertex, we need to check all the edges to find the minimum weight edge.

**Advantages of Prim's Algorithm:**

1. **Simplicity**: The algorithm is simple to understand and implement.
2. **Efficient for Dense Graphs**: Works well for graphs where the number of edges (E) is large, especially with an adjacency matrix representation.
3. **Greedy Approach**: It always makes the locally optimal choice, which leads to the global optimal solution in the case of the minimum spanning tree.

**Disadvantages of Prim's Algorithm:**

1. **Not Suitable for Sparse Graphs (with Adjacency Matrix)**: In graphs with very few edges, Prim’s algorithm (using an adjacency matrix) can be inefficient.
2. **Memory Consumption**: For very large graphs, the adjacency matrix representation may consume too much memory.
3. **Requires a Connected Graph**: The algorithm assumes that the graph is connected. If the graph is disconnected, it will not work correctly.

3) **Dijkstra's Algorithm:**

**Dijkstra's Algorithm** is a **shortest-path algorithm** used to find the shortest path from a source node to all other nodes in a graph. It works with graphs that have non-negative edge weights. It is often used in routing and navigation systems.

**Working of Dijkstra's Algorithm:**

1. **Initialization**:
   * Set the distance to the source node as 0 and all other nodes as infinity.
   * Create a set to keep track of unvisited nodes.
   * Set the parent of each node as undefined (null).
2. **Relaxation Process**:
   * For the current node, examine all of its unvisited neighbors.
   * Calculate their tentative distances through the current node. If the newly calculated tentative distance is smaller than the current assigned value, update the shortest distance.
   * Mark the current node as visited. A visited node will not be checked again.
3. **Repeat** the process for all nodes in the graph until all nodes have been visited or the shortest path to all nodes is found.
4. **Result**: The shortest path from the source node to every other node.

**Steps in Dijkstra’s Algorithm:**

1. **Create a set** of unvisited nodes and initialize the distance of the source node as 0 and all others as infinity.
2. **Pick the unvisited node** with the smallest tentative distance.
3. **Update the tentative distances** of the unvisited neighbors of the chosen node.
4. **Mark the node as visited** and remove it from the unvisited set.
5. **Repeat** the process until all nodes have been visited.

**Time Complexity**

* **Time Complexity of Dijkstra’s Algorithm** depends on the data structure used:
  + **Using Adjacency Matrix (Naive Approach)**: **O(V²)** where V is the number of vertices.
  + **Using Adjacency List with Binary Heap (Priority Queue)**: **O(E log V)** where E is the number of edges and V is the number of vertices.

**📌 Applications:**

* **GPS Navigation Systems** – to find the shortest route
* **Network Routing Protocols** – like OSPF (Open Shortest Path First)
* **Flight Reservation Systems**
* **Robot path planning**
* **Social network analysis**

**👍 Advantages:**

* Efficient for finding shortest paths in large graphs
* Can be optimized using priority queues (e.g., with a min-heap)
* Guarantees the shortest path with non-negative weights

**👎 Disadvantages:**

* Cannot handle **negative weights**
* Less efficient for sparse graphs compared to A\*
* Requires all nodes to be known in advance (not dynamic)

4) **Kruskal’s Algorithm:**

**Kruskal's Algorithm** is a **minimum spanning tree (MST)** algorithm that finds the minimum spanning tree (MST) of a graph. An MST is a subset of the edges in a graph that connects all the vertices with the minimum total edge weight, and without any cycles. It’s typically used in various network design problems, like designing a computer network, laying out electrical circuits, and in other applications where a minimum spanning tree is required.

**Working of Kruskal’s Algorithm:**

Kruskal’s algorithm is a **greedy algorithm**, which means it makes locally optimal choices at each step in order to reach a global optimum.

**Steps:**

1. **Sort all edges** in the graph by their weights in non-decreasing order.
2. **Initialize the MST** as an empty set.
3. **Iterate through the sorted edges** and for each edge, check if adding that edge to the MST will create a cycle.
   * If adding the edge doesn’t form a cycle, include it in the MST.
   * If adding the edge creates a cycle, discard it.
4. **Repeat** until there are V−1V-1V−1 edges in the MST (where VVV is the number of vertices in the graph).

**Steps of Kruskal's Algorithm:**

1. **Sort all edges**: Sort all the edges in the graph in ascending order based on their weights.
2. **Initialize the MST**: Initially, the MST is an empty set.
3. **Iterate through edges**: Start with the smallest edge, check if including it will form a cycle, and add it to the MST if it doesn't.
4. **Cycle detection**: To detect cycles, we can use a **Union-Find** (Disjoint Set) data structure, which helps in efficiently checking and merging sets of nodes.
5. **Repeat** until the MST has V−1V - 1V−1 edges.

**Time Complexity:**

* **Sorting the edges**: Sorting the edges takes **O(E log E)** time, where E is the number of edges.
* **Union-Find operations**: Each find and union operation takes **O(α(V))**, where α is the inverse Ackermann function (which is very small for practical inputs).
* **Overall time complexity**: The time complexity of Kruskal’s algorithm is **O(E log E)** or **O(E log V)** since in a connected graph EEE is at least V−1V-1V−1.

**📌 Applications:**

* **Network design** (e.g., laying cables, pipes, roads)
* **Clustering algorithms** (e.g., in image segmentation)
* **Electrical grid optimization**
* **Computer network topology design**
* **Minimum-cost infrastructure development**

**👍 Advantages:**

* Works well on **sparse graphs**
* Easier to implement using Disjoint Set (Union-Find)
* Can handle disconnected graphs (returns a Minimum Spanning Forest)

**👎 Disadvantages:**

* Requires edge list to be **sorted**, which can be time-consuming
* Not suitable for **dynamic graph changes**
* Does not find shortest path – only minimum spanning connection

**Ass 4) N queen**

The N-Queens Problem is to place N chess queens on an N×N chessboard so that no two queens attack each other.  
This means no two queens can be in the same row, column, or diagonal.

**Algorithm Used: Backtracking**

Backtracking is a **depth-first search** approach where we build the solution incrementally and **backtrack** when we find it invalid

**Algorithm Steps**

1. Start in the leftmost column.

2. Try placing a queen in each row of that column.

3. For each placement:

a. Check if it is safe (no queen in same row, or diagonals).

b. If safe, place queen and recurse to next column.

c. If placing in next columns leads to no solution, backtrack (remove the queen).

4. Continue until all queens are placed or all options are exhausted.

**Time Complexity**

* **Worst-case Time Complexity**: O(N!)
  + Because you may need to explore every possible arrangement in worst case.
* **Space Complexity**: O(N^2) for the board, or O(N) if optimized with arrays (row/diag tracking).

**💡 Applications**

* Constraint satisfaction problems.
* Optimization algorithms (like in AI and robotics).
* Puzzle-solving.
* Understanding backtracking techniques for interview prep.