Artificial Intelligence

BS (CS) \_Spring\_2025

# Lab\_04 Manual



Learning Objectives:

1. Uninformed searches (BFS,DFS,UCS)
2. Greedy Search

# **Lab Manual**

**Uninformed Search:**

Uninformed search is a class of general-purpose search algorithms. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called **blind** search. They operate in a brute force, meaning they try out every part of search space blindly.

**Types:**

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search

**1. Breadth First Search:**

Breadth-First Search (BFS) is an **uninformed search algorithm** that explores all the nodes at the current depth level before moving on to the next level. It guarantees finding the **shortest path in an unweighted graph**. BFS uses a **queue (FIFO structure)** to track nodes.

**Features:**

* Explores all neighbors before moving to the next depth level.
* Guarantees the shortest path in an unweighted graph.
* Uses a queue **(FIFO)** data structure to store nodes.
* Can be used for pathfinding and shortest path problems.

**Step-by-Step Guide:**

1. **Initialize** an empty queue and add the **start node**.
2. **Initialize** an empty set to track **visited nodes**.
3. While the **queue is not empty**:

* Remove a node from the **front** of the queue.
* If it is the **goal**, return the path.
* Otherwise, **add all unvisited neighbors** to the queue.
* Mark the node as **visited**.

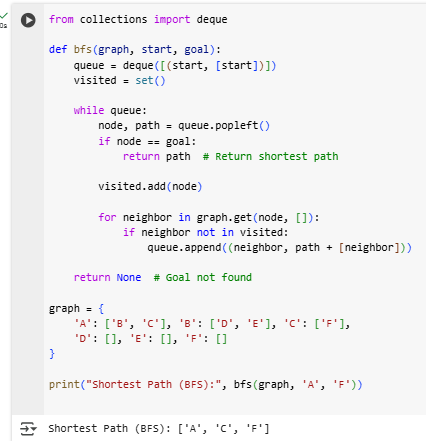
1. If the **goal is not found**, return failure.

**Example:**

**A diagram of a graph

AI-generated content may be incorrect.**

**Code:**



**2. Depth-First-Search:**

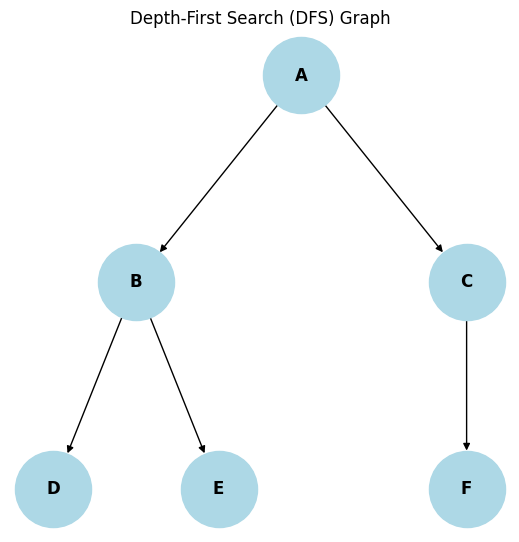
Depth-First Search (DFS) is an **uninformed search algorithm** that explores as far as possible along one branch before backtracking. Unlike BFS, DFS **does not guarantee the shortest path** but is useful for exploring large spaces with low memory usage. DFS uses a **stack (LIFO structure) or recursion**.

**Features:**

* Explores as deep as possible before backtracking.
* Does **not guarantee the shortest path**.
* Uses a **stack (LIFO) or recursion** to store nodes.
* Efficient for **searching large graphs with fewer edges**.

**Step-by-Step Guide:**

1. **Initialize** an empty stack and add the **start node**.
2. **Initialize** an empty set to track **visited nodes**.
3. While the **stack is not empty**:
   * Remove a node from the **top** of the stack.
   * If it is the **goal**, return the path.
   * Otherwise, **add all unvisited neighbors** to the stack.
   * Mark the node as **visited**.
4. If the **goal is not found**, return failure.

**Example:**

**Code:**

A screenshot of a computer program

AI-generated content may be incorrect.

**3. Depth Limited Search:**

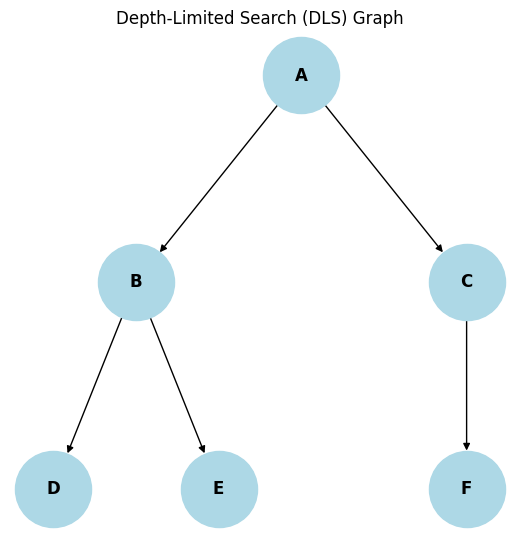
Depth-Limited Search (DLS) is a variation of Depth-First Search (DFS) that limits the depth of exploration to a predefined level to avoid infinite loops in deep or **infinite** search spaces.

**Features:**

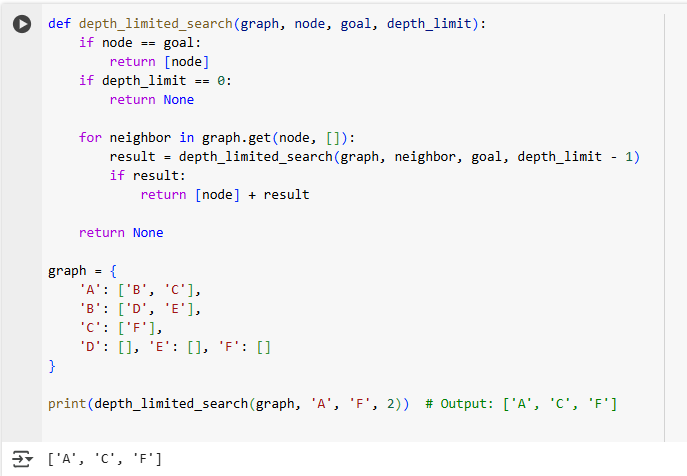
* Uses a depth limit to prevent infinite recursion.
* May not find a solution if the limit is too low.
* Uses a **stack** (like DFS) but stops at a given depth.

**Step-by-Step Guide:**

1. Set a **maximum depth limit** for the search.
2. Start from the **initial node** and explore as deep as possible until the **depth limit** is reached.
3. If the **goal is found**, return the path.
4. If the **depth limit is reached**, stop exploring that branch.
5. If no solution is found and the limit is reached, return failure.

**Example:**

**Code:**

****

**4.Iterative Deepening Search**

Iterative Deepening DFS (IDDFS) is a combination of Depth-First Search (DFS) and Breadth-First Search (BFS). It repeatedly runs **Depth-Limited Search** with increasing depth limits until the goal is found.

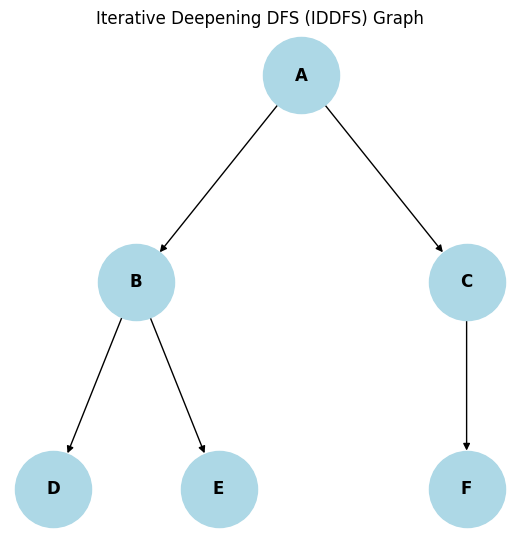
**Features:**

* Uses **less memory** than BFS.
* Finds the **shortest path** like BFS.
* Works well for **large and infinite search spaces**.

**Step-by-Step Guide:**

1. Start with **depth limit = 0**.
2. Perform **Depth-Limited Search (DLS)**.
3. If the goal is found, **return the path**.
4. If not, **increase depth limit by 1** and repeat.
5. Continue until the goal is found or the search space is exhausted.

**Example:**

****

**Code:**

****

**5. Uniform Cost Search**

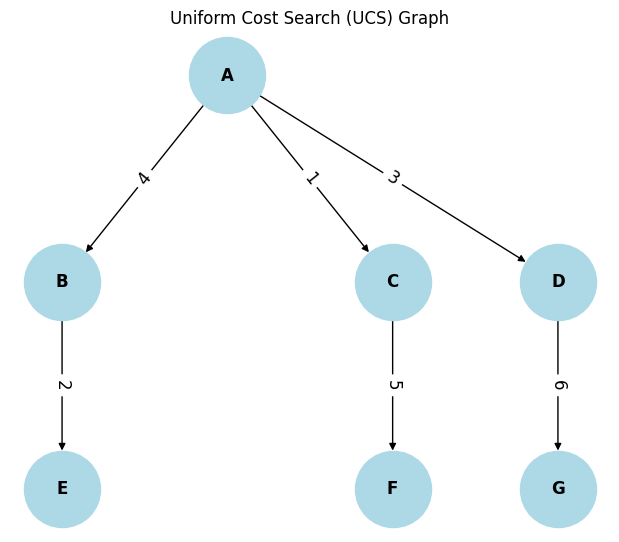
Uniform Cost Search (UCS) is a **variant of BFS** that considers **path cost** rather than depth. It always expands the **least-cost node** first, using a **priority queue**.

**Features:**

* Guarantees **optimal solution** if all costs are positive.
* Uses a **priority queue** to expand the lowest-cost node first.
* Works well in weighted graphs.

**Step-by-Step Guide:**

1. **Initialize** a priority queue with the **start node and cost = 0**.
2. Expand the **node with the least cost** from the priority queue.
3. If the **goal is found**, return the path.
4. Otherwise, **add neighbors to the queue with their updated costs**.
5. Repeat until the goal is found or the queue is empty.

**Example:**

**Code:**

1. Initialize an empty priority queue (min-heap).

2. Add the start node to the queue with a cost of 0.

   - Priority Queue: [(0, start\_node, [start\_node])]

     (Each element in the queue is a tuple: (cost, current\_node, path\_to\_node))

3. Create an empty set to track visited nodes.

4. While the priority queue is not empty:

    a. Dequeue the node with the lowest cost.

       - cost, current\_node, path = priority\_queue.pop()

    b. If the current\_node is the goal:

         Return the path and cost (solution found).

    c. If the current\_node has already been visited:

         Skip this node and continue.

    d. Mark the current\_node as visited.

    e. For each neighbor of the current\_node:

        i. Compute the new cost as:

           new\_cost = cost + cost\_to\_neighbor

        ii. If the neighbor is not visited:

           Add the neighbor to the priority queue:

           priority\_queue.push((new\_cost, neighbor, path + [neighbor]))

5. If the goal is not found and the queue is empty:

    Return "No path found!"

**Informed Search:**

Informed search algorithms, also known as **heuristic search algorithms**, use problem-specific knowledge to find solutions more efficiently than uninformed search algorithms. These algorithms make use of a heuristic function to estimate the cost of reaching the goal from a given state.

**Heuristics function:**

Heuristic is a function which is used in Informed Search, and it finds the most promising path. It provides an **estimate** of the cost from a given node to the goal node. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by and it calculates the cost of an optimal path between the pair of states. The value of the **heuristic function is always positive.**

Admissibility of the heuristic function is given as:

Here is heuristic cost, and is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost. Two popular informed search algorithms are **Greedy Search** and **A\* (A-star).**

**1. Greedy Search:**

Greedy Search is an informed search algorithm that always chooses the path that appears to be the best at the current moment. It evaluates each state based solely on the heuristic function, without considering the cost of reaching that state. While Greedy Search can find solutions quickly, it **may not** always find the **optimal solution.**

**Step by step Guide:**

1. **Initialize** the **open list** with the initial state and its **heuristic value**.
2. **Initialize** the **closed list** as empty.
3. **While the open list is not empty:**
   * Remove the state with the **lowest heuristic value** from the open list.
   * If the state is the **goal**, **return the path** to the goal.
   * Otherwise, **expand the state** and add its **neighbors** to the open list with their heuristic values.
   * Add the **current state** to the **closed list**.
4. If the open list is empty and the goal state **has not been found,** return failure**.**

1. Initialize an empty priority queue (min-heap).

2. Add the start node to the queue with its heuristic value.

   - Priority Queue: [(h(start\_node), start\_node, [start\_node])]

     (Each element is a tuple: (heuristic\_value, current\_node, path\_to\_node))

3. Create an empty set to track visited nodes.

4. While the priority queue is not empty:

    a. Dequeue the node with the lowest heuristic value.

       - \_, current\_node, path = priority\_queue.pop()

    b. If the current\_node is the goal:

         Return the path (solution found).

    c. If the current\_node has already been visited:

         Skip this node and continue.

    d. Mark the current\_node as visited.

    e. For each neighbor of the current\_node:

        i. If the neighbor is not visited:

           Add the neighbor to the priority queue with its heuristic value:

           priority\_queue.push((h(neighbor), neighbor, path + [neighbor]))

5. If the goal is not found and the queue is empty:

    Return "No path found!"