4. Uninformed Searching in AI

1. Introduction to Search in AI

- **Search in AI:**
  The process of finding a sequence of actions (a plan) that transforms an initial state into a goal state within a defined state space.
- **Key Components:**
  - **Search Space:** Set of all possible states.
  - **Start State:** The initial state where the search begins.
  - **Goal Test:** A function that checks if the current state is a goal state.
  - **Path Cost:** A function that assigns a cost to each path; the optimal solution minimizes this cost.
- **Search Tree:**
  A tree representation of the search process, where the root represents the start state and the branches represent possible state transitions via actions.

---

2. Search Problem Representation

- **Formal Representation:**
  A search problem is defined as {S, s_0, A, G, P}, where:
  - **S:** Set of states.
  - **s_0:** Initial state.
  - **A:** Set of actions (operators).
  - **G:** Goal state.
  - **P:** Path cost function.
- **Transition Model:**
  Describes the effect of an action on a state.

---

3. Basic Search Algorithms

- **State Space Tree Search:**
  Generates a search tree without checking for duplicate states (can lead to inefficiency).
- **State Space Graph Search:**
  Uses
  OPEN (generated but unexplored nodes) and **CLOSED** (explored nodes) lists to avoid revisiting states, thereby improving efficiency.

- OPEN = List of generated but unexplored states; initially containing initial state
- CLOSED = List of explored nodes; initially empty
- **Algorithm:**

Loop until OPEN is empty or success is returned.
  (N,P) $\leftarrow$ remove-head(OPEN) and add it to CLOSED
    If N is goal node then return success and construct path from initial state to N.
    Else generate successors of node N and add it to open i.e. OPEN $\leftarrow$ OPEN $\cup$ {MOVEGEN(N)}
    End if
End Loop

State Space tree search algo

Basic Search Algorithm 2
State Space Graph Search Algorithm

- OPEN = List of generated but unexplored states; initially containing initial state
- CLOSED = List of explored nodes; initially empty
- **Algorithm:**

Loop until OPEN is empty or success is returned.

(N,P)←remove-head(OPEN) and add it to CLOSED

If N is goal node then return success and construct path from initial state to N.

Else generate successors of node N and add the nodes to OPEN that are not already generated or expanded

i.e. OPEN ← OPEN ∪ {MOVEGEN(N)\{OPEN ∪ CLOSED}}

End if

End Loop

- The algorithm does not add any node that has already been generated or explored. Hence it is a better algorithm as it can deal with all kinds of problems.

---

4. Uninformed (Blind) Search Methods

- **Overview:**
  Uninformed search algorithms operate without additional problem-specific knowledge. They explore the search space systematically.
- **Examples:**
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
  - Depth-Limited Search
  - Iterative Deepening DFS (DFS-ID)
  - Uniform Cost Search (UCS)
  - Bidirectional Search

---

5. Breadth-First Search (BFS)

- **Method:**
  Explores nodes level-by-level using a
  queue.
- **Key Properties:**
  - **Completeness:** Complete in finite search spaces. (Always)
  - **Optimality:** Optimal when each step cost is equal.
  - **Time Complexity:** $O(b^{(d+1)})$ (exponential with branching factor *b* and depth *d*).
  - **Space Complexity:** High, since it stores all nodes at the current level. $O(b^d)$

Breadth-First Search Algorithm

- OPEN = List of generated but unexplored states; initially containing initial state (OPEN is used as a QUEUE in BFS).
- CLOSED = List of explored nodes; initially empty
- **Algorithm:**

Loop until OPEN is empty or success is returned.

(N,P)←remove-head(OPEN) and add it to CLOSED

If N is goal node then return success and construct path from initial state to N.

Else generate successors of node N and add the nodes **to end of OPEN** that are not already generated or expanded.

i.e. $OPEN \leftarrow OPEN \cup \{MOVEGEN(N) \setminus \{OPEN \cup CLOSED\}\}$ **(to end)**

End if

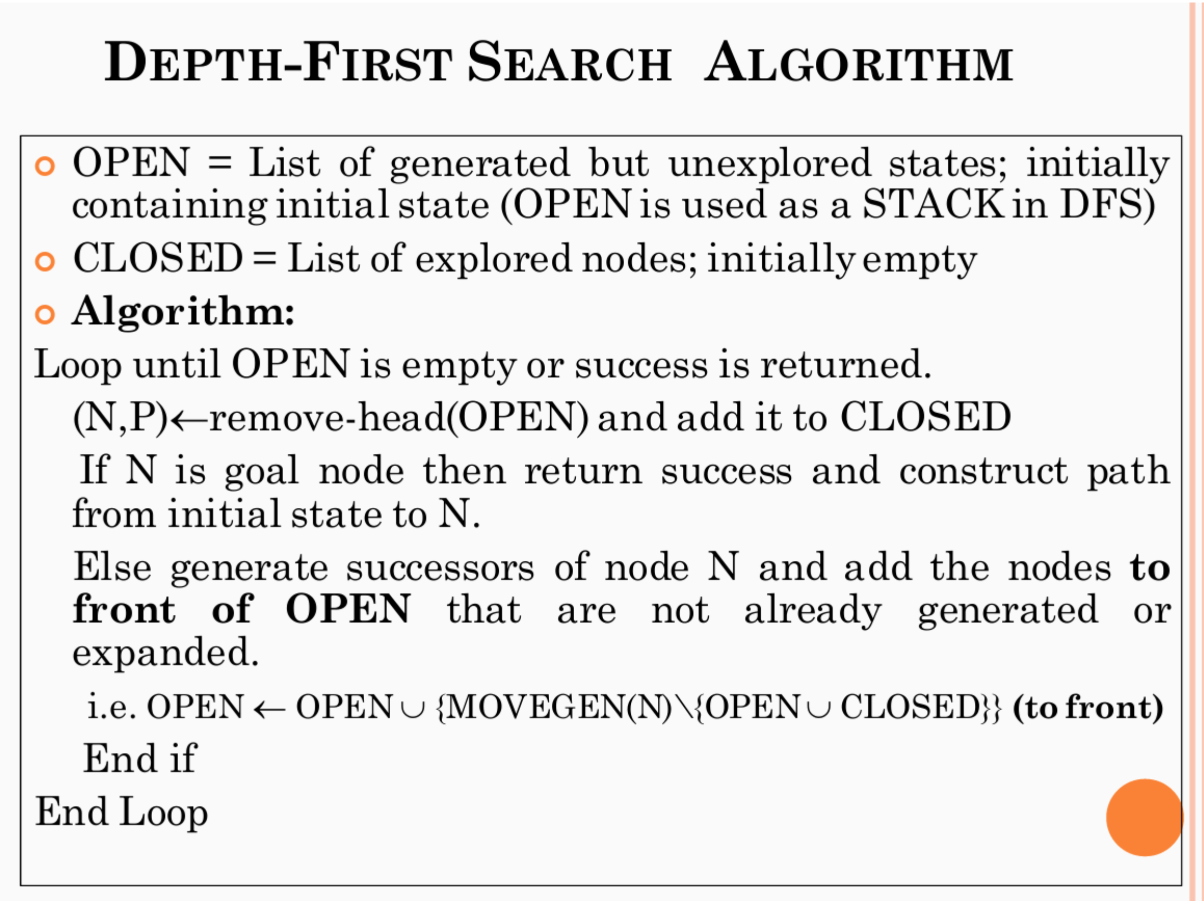
End Loop

---

6. Depth-First Search (DFS)

- **Method:**
  Explores as deep as possible along a branch before backtracking using a **stack**.

- **Key Properties:**
  - **Completeness:** May fail in infinite-depth spaces.
  - **Optimality:** Not guaranteed (may not find the shortest path).
  - **Time Complexity:** O(b^(d+1)) in the worst case.
  - **Space Complexity:** O(b*d), significantly lower than BFS.
  - **Note:** Risk of getting stuck in a "blind alley" (deep, unproductive paths).



---

7. Depth-Limited Search

- **Method:**
  A variant of DFS that imposes a maximum depth limit (cutoff) to avoid infinite descent.
- **Key Properties:**
  - **Completeness:** Incomplete if the solution lies beyond the cutoff.
  - **Optimality:** Not guaranteed.

- **Time Complexity:** O(b^l), l = depth limit (Similar to DFS)
- **Space:** O(b*l)

---

8. Depth-First Search with Iterative Deepening (DFS-ID)

- **Method:**
  Repeatedly applies depth-limited search with increasing depth limits, combining the space efficiency of DFS with the optimality of BFS.
- **Key Properties:**
  - **Completeness:** Complete for finite search spaces. (Always)
  - **Optimality:** Optimal if path costs are uniform.
  - **Time Complexity:** O(b^d) (similar to BFS in the worst case).
  - **Space Complexity:** O(b*d), matching DFS.

DFS-ID ALGORITHM

- DFS-ID (problem) returns success or failure
- for depth= 0 to ∞

    result = DEPTH-LIMITED-SEARCH(problem,depth) return result

  end for

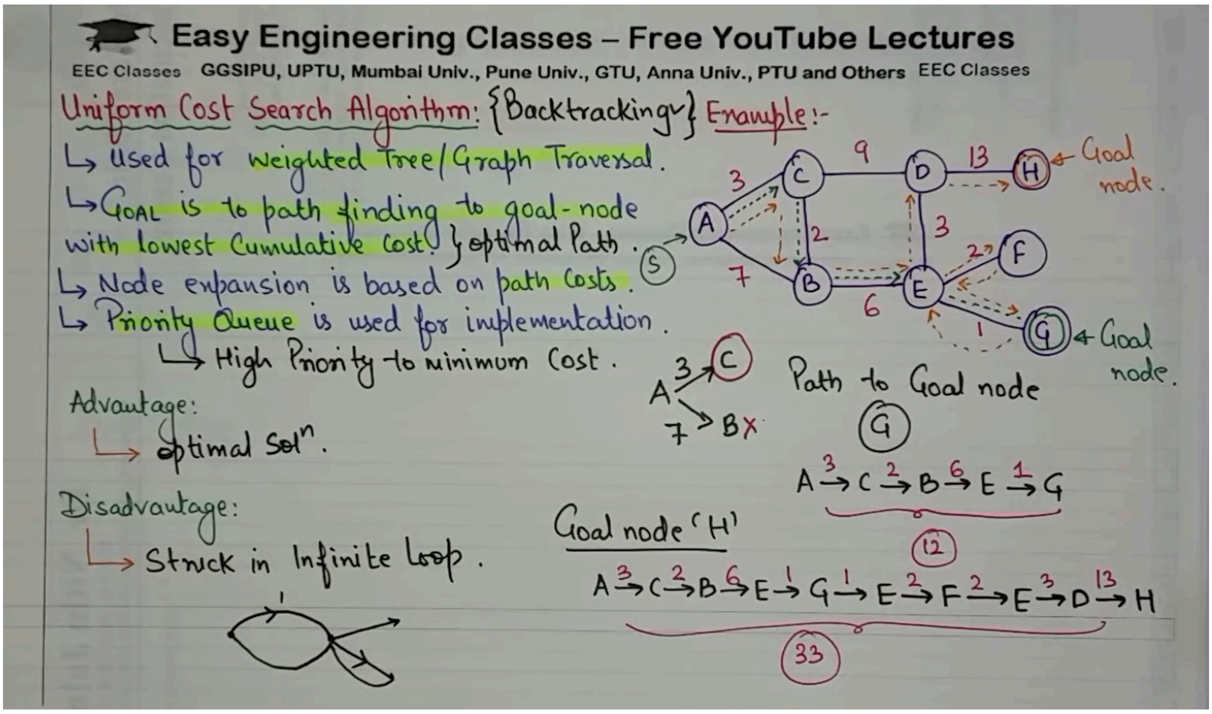
---

9. Uniform Cost Search (UCS)

- **Method:**
  Expands the node with the lowest cumulative cost (g(n)) using a **priority queue**.
- **Key Properties:**
  - **Completeness:** Complete when step costs are non-negative.
  - **Optimality:** Optimal (finds the least cost path).

- **Time Complexity:** O(b^(c*/m)), where c* is the optimal path cost and m is the minimum edge cost.
- **Space Complexity:** Similar to BFS. O(b^(d+1))
- Backtracking possible
- Node expansion based on path cost
- Can stuck in infinite loop

- g(N) = g(N’s parent) + cost(N,P)



---

10. Bidirectional Search

- **Method:**
  Simultaneously searches forward from the start state and backward from the goal state until the two search frontiers meet.
- **Key Benefits:**
  - **Efficiency:** Reduces search complexity from O(b^d) to approximately O(2*b^(d/2)).
  - Bidirectional search would be complete and optimal if BFS is used during both forward and backward search.

- **Challenges:**
  - Defining the backward search (predecessors).
  - Handling multiple goal states.
  - Efficiently detecting the intersection of the two search trees.

---

11. Performance Evaluation of Search Algorithms

- **Completeness:**
  Whether the algorithm is guaranteed to find a solution if one exists.
- **Optimality:**
  Whether the algorithm finds the best (least-cost) solution.
- **Time Complexity:**
  How the number of nodes expanded grows with the size of the search space.
- **Space Complexity:**
  The maximum number of nodes stored in memory during the search.
- **Trade-offs:**
  - **BFS:** Complete and optimal (with uniform costs) but high space usage.
  - **DFS:** Low space usage but may not be complete or optimal.
  - **DFS-ID:** Combines the benefits of BFS and DFS.
  - **UCS:** Optimal for non-uniform cost problems.
  - **Bidirectional Search:** Can drastically reduce search effort if implemented effectively.

---

12. Key Differences Among Algorithms

- **BFS vs. DFS:**
  - BFS explores level-by-level; DFS dives deep into one branch.
  - BFS is optimal (if uniform cost) but uses more memory.
  - DFS uses less memory but may miss the optimal solution.
- **Iterative Deepening (DFS-ID):**

    - Provides the optimality of BFS with the low memory footprint of DFS.
- **Uniform Cost Search:**
    - Prioritizes nodes based on path cost, ideal for varying step costs.
- **Bidirectional Search:**
    - Splits the search into two smaller searches, reducing overall complexity.

Important Highlights:

- **Search Space vs. Search Tree:**
  Understand that the search space includes all possible states, while the search tree is the actual exploration process.
- **OPEN and CLOSED Lists:**
  These lists manage node expansion and help avoid repeated state exploration.
- **Trade-offs in Search:**
  Balancing time and space complexity is crucial when selecting a search algorithm.
- **Algorithm Selection:**
  Consider completeness, optimality, and complexity based on the specific problem's requirements.

Difference between DFS and BFS

Depth First Search	Breadth First Search
1. Explores all the nodes in the depth first fashion.	1. Explores all the nodes in the breadth first fashion
2. May caught in blind alley i.e. searching at a greater depth in the left path of the search tree while the solution may lie on the right part.	2. Never caught at blind alley because it explores all the nodes at the lower depth before moving to the next node.
3. It uses less space because only current nodes at the current path need to be stored.	3. It uses more space as all the nodes of the current level needs to be stored. The space increases with increase in branching factor.
4. It may find the first solution which may not be optimal.	4. It always finds the optimal solution (if the path cost is uniform).