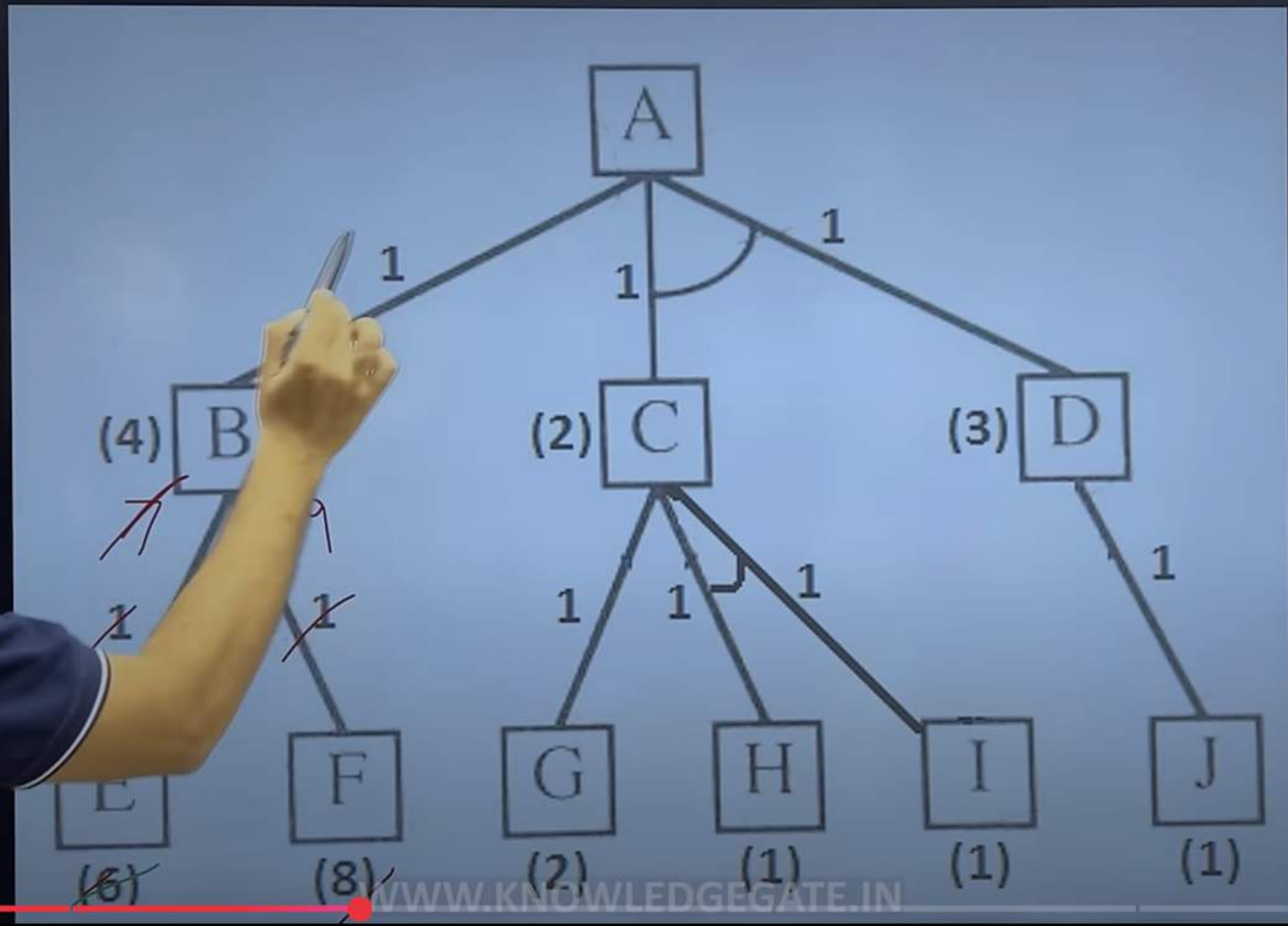
The pseudocode of AO* is as follows:

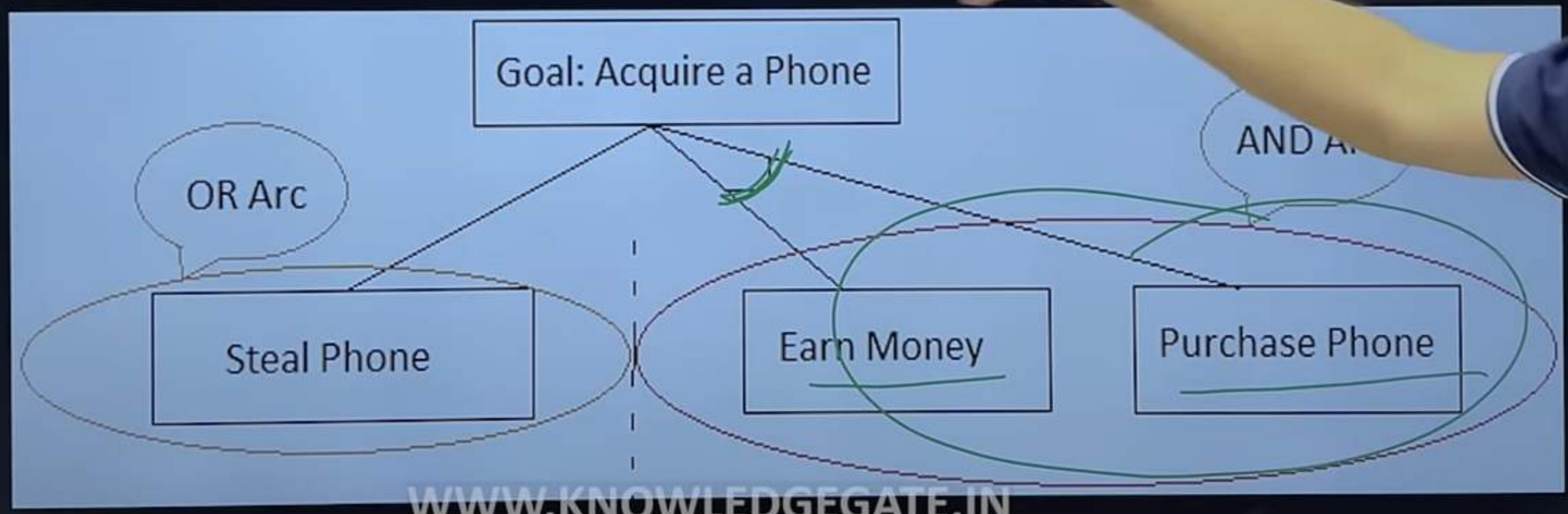
1. Initialize the graph with a single node (the start node).
2. While the solution graph contains non-terminal nodes (nodes that have successors not in the graph):
    1. Choose a non-terminal node for expansion based on a given strategy.
    2. Expand the node (add successors to the graph and update the costs of nodes).
3. The process continues until the start node is labeled as a terminal node.

AO Search Algorithm*



Problem with A* Algorithm

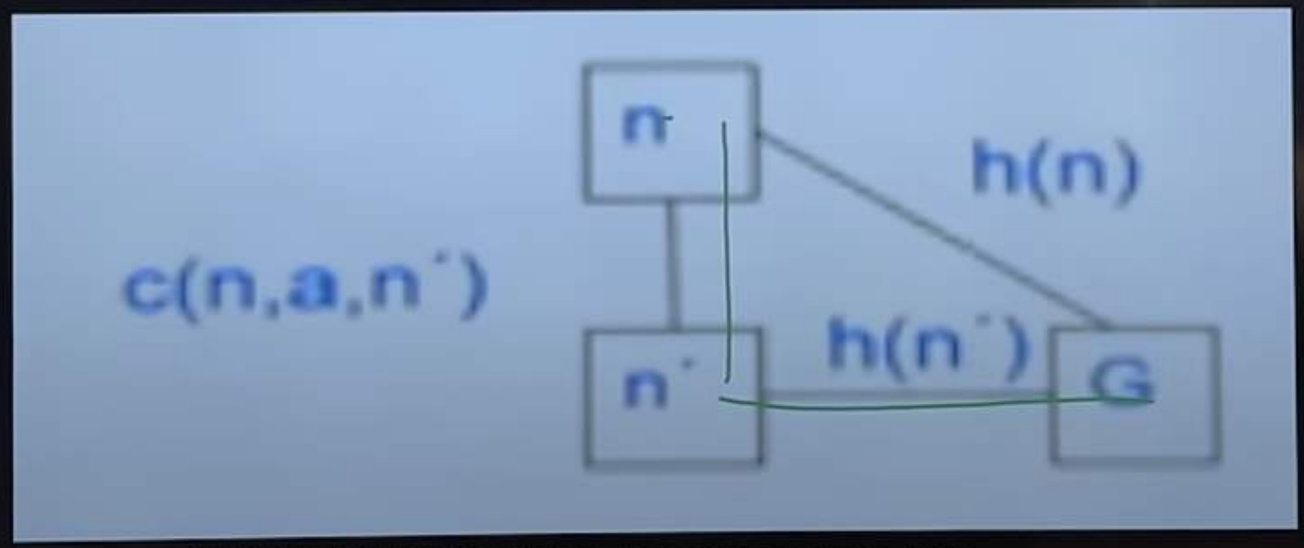
- So far we have considered search strategies for OR graphs through which we want to find a single path to a goal.
- The AND-OR GRAPH (or tree) is useful for representing the solution of problems that can
- solved by decomposing them into a set of smaller problems, all of which must then be solve
- This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may
- point to any number of successor nodes, all of which must be solved in order for the arc to
- point to a solution.



- A second, slightly stronger condition called **consistency** (or **monotonicity**) is required only for applications of A* to graph search.

- A heuristic h(n) is **consistent** if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

$$h(n) \leq c(n, a, n') + h(n')$$



Conditions for optimality: Admissibility and consistency

- The first condition we require for optimality is that h(n) be an **admissible heuristic.**

- An admissible heuristic is one that ***never overestimates*** the cost to reach the goal. Admissible heuristics are by nature **optimistic** because they think the cost of solving the problem is less than it actually is.

- For example, Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

Completeness:

- A* Search is complete if the search space is finite and the heuristic function is admissible (never overestimates the actual cost).

Optimality:

- A* Search is optimal if the heuristic function is admissible and consistent (also known as monotonic).

Time Complexity:

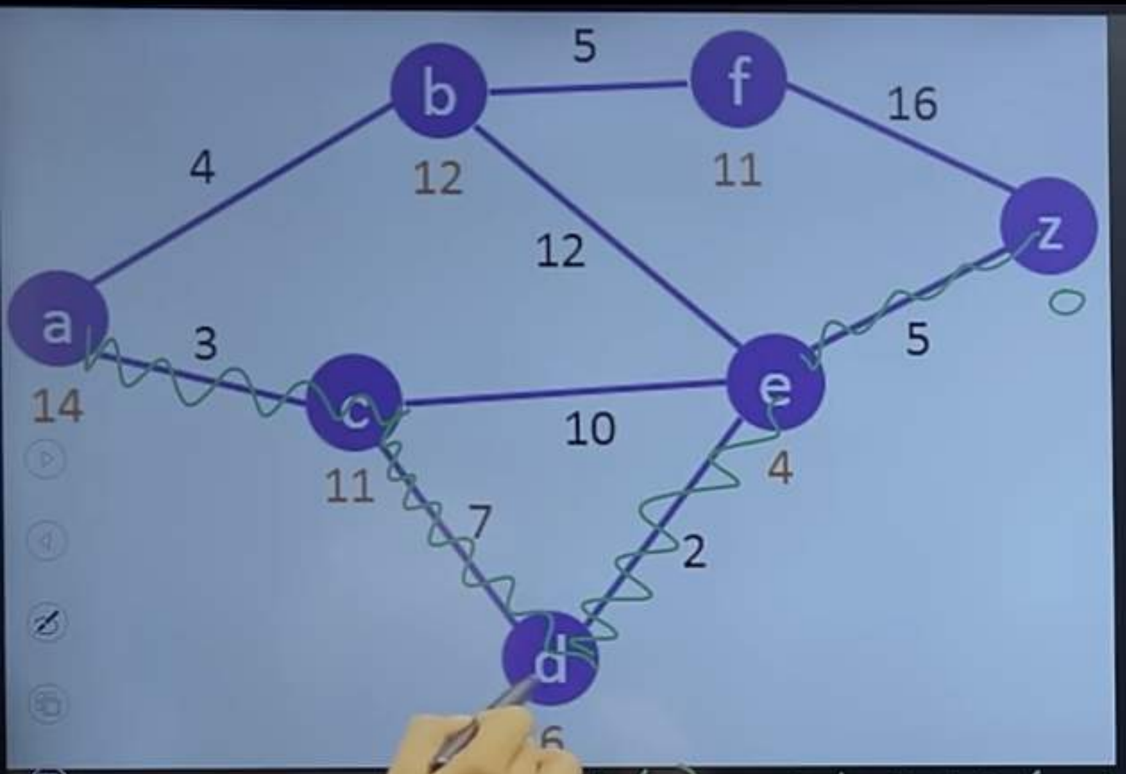
- The time complexity of A* Search depends on the heuristic function, the branching factor, and the structure of the search space. In the worst case, it can be exponential.

Space Complexity:

- The space complexity of A* Search depends on the size of the priority queue and the number of nodes stored in memory. In the worst case, it can be exponential.

Algorithm:

1. Start with the initial state as the root node.
2. Create an evaluation function that combines the cost of the path and a heuristic estimate.
3. Initialize an empty priority queue or priority-based data structure.
4. Enqueue the initial state into the priority queue based on the evaluation function.
5. While the priority queue is not empty, do the following:
    1. Dequeue the node with the highest priority from the priority queue.
    2. If the dequeued node is the goal state, terminate the search and return the solution.
    3. Otherwise, expand the node and enqueue its unvisited neighboring nodes into the priority queue based on the evaluation function.
6. Repeat steps 5 until a solution is found or the priority queue is empty.



$f(d) = g(d) + h(d)$
$= 10 + 6 = 16$ ✓

$f(e) = g(c) + h(c)$
$= 13 + 4 = 17$

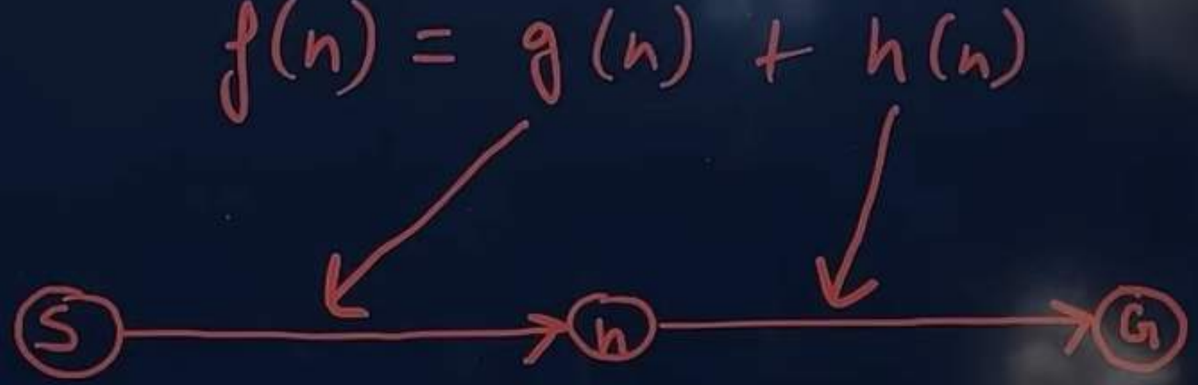
$8 + 7 + 2 + 5$
$10 \quad 7 = (17)$

$f(b) = g(b) + h(b)$
$4 + 12$
$= 16$

$f(c) = g(c) + h(c)$
$3 + 11 = 14$

A* search: Minimizing the total estimated solution cost

- A* Search is an informed search algorithm that combines the advantages of both Uniform Cost Search and Greedy Best-First Search. It evaluates a n
- a combination of the cost of the path from the start node to that
- estimated heuristic function that estimates the cost to reach the current node.



- A* search evaluates nodes by combining g(n), the cost to reach th
  h(n), the estimated cost to get from the node to the goal: f(n) = g(

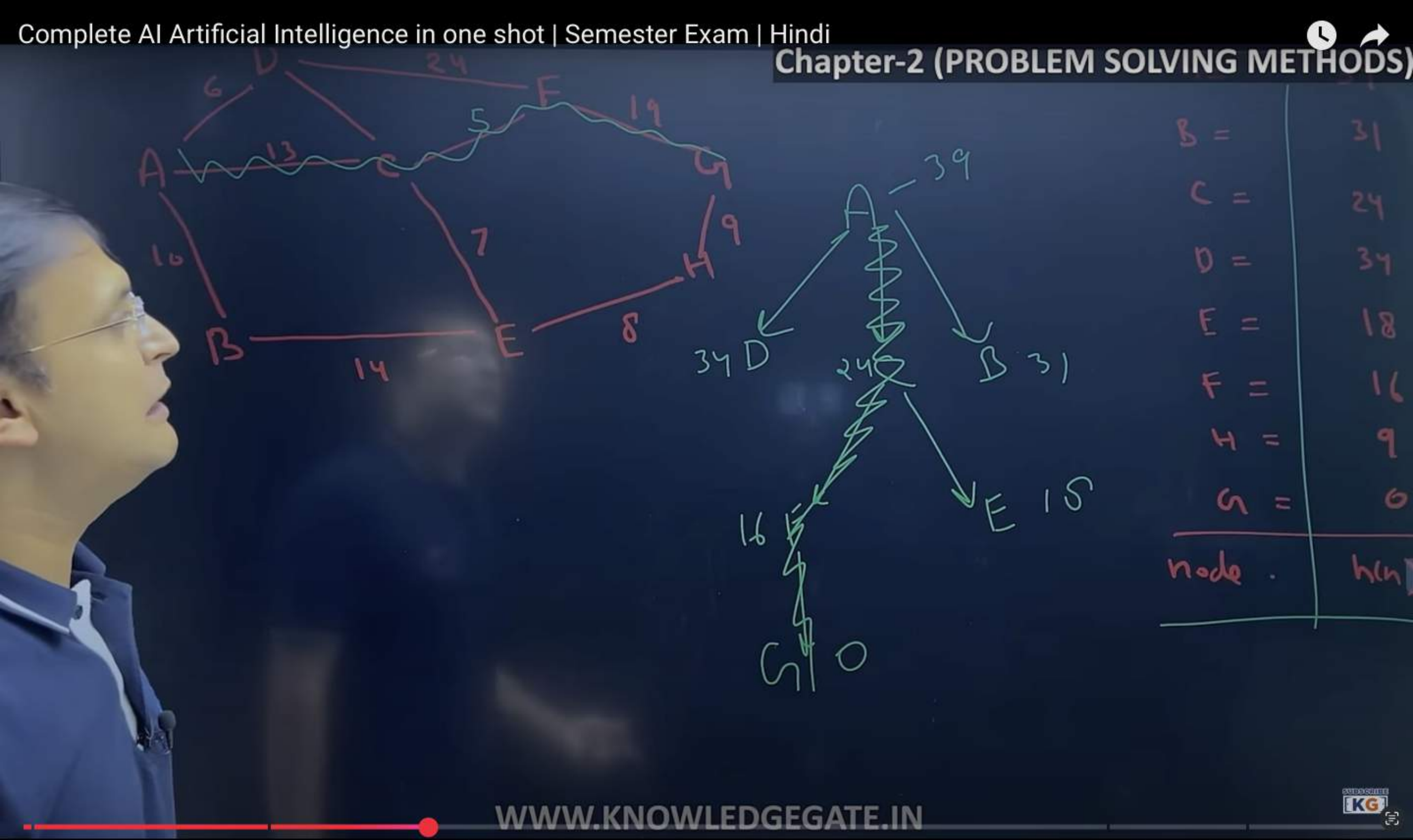
- **Time and Space Complexity**
  - The time and space complexity of Best-First Search are both $O(b^d)$, where b is the branching factor (number of successors per state), and d is the depth of the shallowest solution.
  - These complexities can be very high in problems with a large number of st and actions. In the worst case, the algorithm will need to visit all nodes, an since each node is stored, the space complexity is also proportional to the number of nodes.

Advantages of Best-First Search

- It can find a solution without exploring much of the state space.
- It uses less memory than other informed search methods like A* as it does not store all the generated nodes.

Disadvantages of Best-First Search

- It is not complete. In some cases, it may get stuck in an infinite loop.
- It is not optimal. It does not guarantee the shortest path will be found.
- It heavily depends on the accuracy of the heuristic.



Best-first Search

- Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule. It's called "best-first" because it greedily chooses the node that seems most promising according to the heuristic.

- Best-first search takes the advantage of both BFS and DFS. The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first.

- ***h(n) = estimated cost of the cheapest path from the state at node n to a goal state.***

Advantages:

- **Faster Exploration**: Bidirectional Search can be faster than traditional searches by exploring the search space simultaneously from both ends, potentially reducing the search effort.
- **Reduces Effective Branching Factor**: As the search progresses from both directions, the effective branching factor is reduced, leading to a more efficient search.

Disadvantages:

- **Increased Memory Requirement**: Bidirectional Search requires storing visited nodes from both directions, leading to increased memory consumption compared to unidirectional searches.
- **Additional Overhead**: The coordination and synchronization between the two searches introduce additional overhead in terms of implementation complexity.

Completeness:

- Bidirectional Search is complete if both the forward and backward searches are complete in a finite search space.

Optimality:

- Bidirectional Search is optimal if both the forward and backward searches are optimal.

Time Complexity:

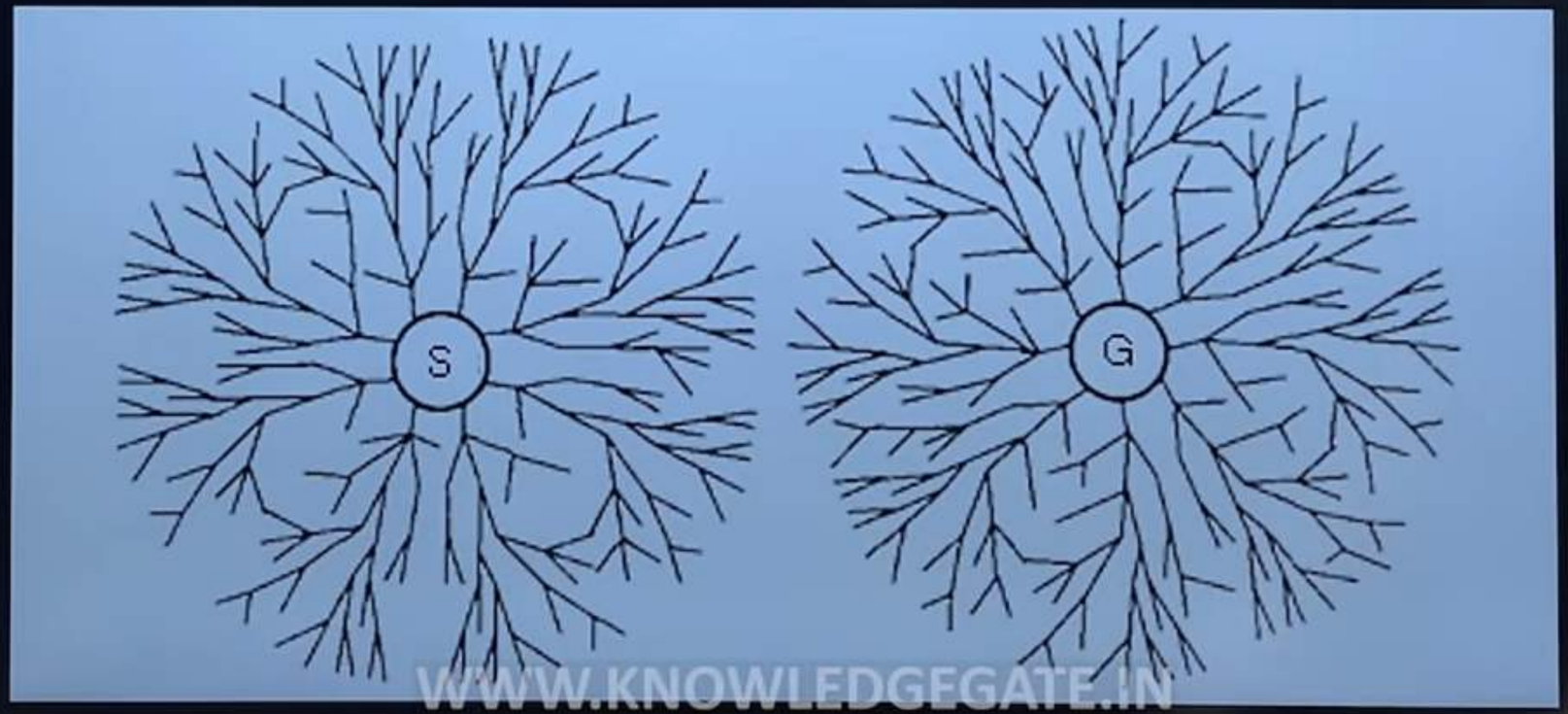
- The time complexity of Bidirectional Search depends on the branching factor, the depth of the shallowest goal state, and the meeting point of the two searches. In the best case, it can be $O(b^{d/2})$, where b is the branching factor and d is the depth of the goal state.

Space Complexity:

- The space complexity of Bidirectional Search depends on the memory required to store visited nodes from both directions. In the best case, it can be $O(b^{d/2})$, where b is the branching factor and d is the depth of the goal state.

Bidirectional Search

- Bidirectional Search is a search algorithm that simultaneously performs two separate searches, one forward from the initial state and one backward from the goal state. It aims to meet in the middle by searching for a common node reached from both directions.
- The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d. Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

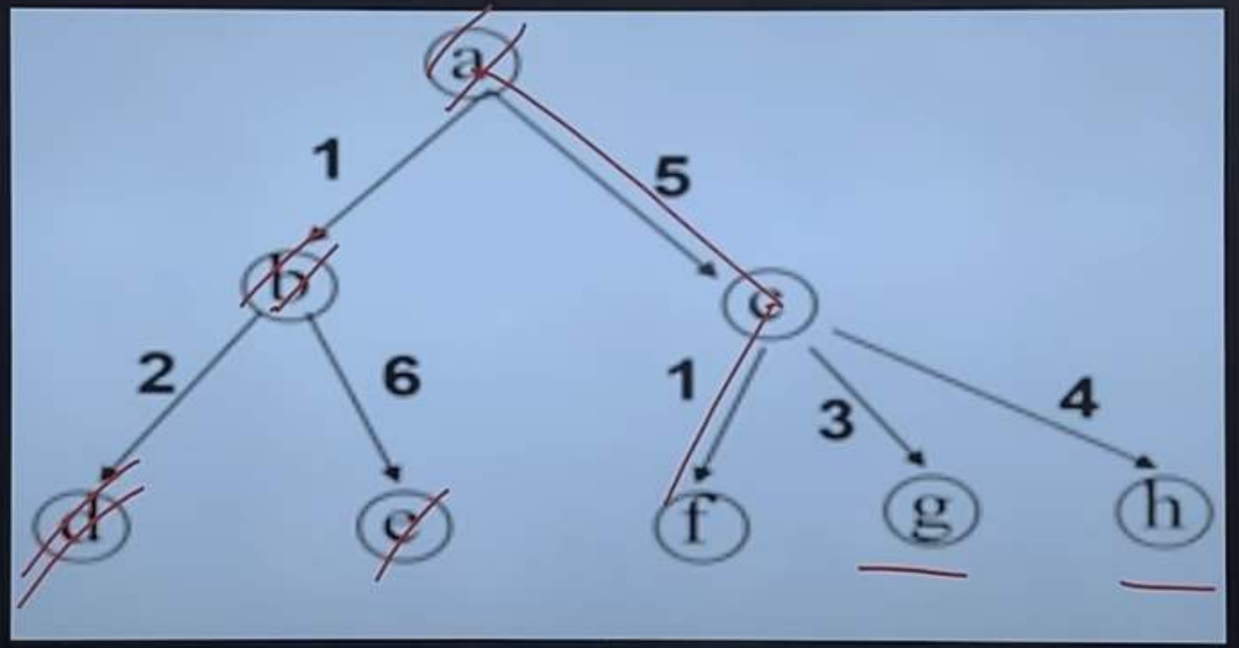


Algorithm:

1. Start with the initial state as the root node.
2. Maintain a priority queue or a priority-based data structure to store nodes based on their path cost.
3. Enqueue the root node with a path cost of zero.
4. While the priority queue is not empty, do the following:
    1. Dequeue the node with the lowest path cost from the priority queue.
    2. If the dequeued node is the goal state, terminate the search and return the solution.
    3. Otherwise, expand the node and enqueue its unvisited neighboring nodes with their updated path costs.
5. Repeat steps 4 until the goal state is found or the priority queue is empty.

Depth-limited search

- It is an extension of Breadth-First Search (BFS) that takes into account the cost of reaching each node to find the lowest-cost path to the goal. **Example:** Consider the graph below.



- **Based on the cost we will expand the graph in order: a→ b → d → c → f → e → g → h**

Completeness:

- DFS is not complete if the search space is infinite or contains cycles. If depth is finite than it is complete.

Optimality:

- DFS does not guarantee finding the optimal solution, as it may find a solution at a greater depth before finding a shorter path.

Time Complexity:

- The time complexity of DFS can vary depending on the search space structure. In the worst case, it can be $O(b^m)$, where b is the branching factor and m is the maximum depth of the search tree.

Space Complexity:

- The space complexity of DFS is O(bm), where b is the branching factor and m is the maximum depth of the search tree. It stores nodes along the current path in memory.

Advantages:

- **Memory Efficiency**: Depth-First Search stores fewer nodes in memory compared to breadth-first search, as it explores deeper paths first.

Disadvantages:

- **Completeness Not Guaranteed**: DFS may get trapped in infinite loops or infinite branches without reaching the goal state if there are cycles in the search space.
- Non-Optimal Solution: DFS does not guarantee finding the optimal solution, as it may find a solution at a greater depth before finding a shorter path.