Practical Class Notes: Search Algorithms

Note By: Himanshi Liyanage
Faculty of Mathematics and Computer Science
University of ŁÓDZ, Poland

January 14, 2025

Introduction to Search Algorithms

Searching algorithms are fundamental in computer science, designed to find specific elements or solutions in a dataset or a problem space. These algorithms navigate data structures like arrays, graphs, or trees to locate the required information efficiently. They play a critical role in artificial intelligence, database querying, web search engines, and game theory.

There are two primary types of search algorithms:

- **Uninformed Search (Blind Search):** These algorithms do not have any additional information about the goal state other than the problem definition. They explore the search space blindly.
- **Informed Search (Heuristic Search):** These algorithms utilize problem-specific knowledge to guide the search, making them more efficient in many cases.

Uninformed Search Algorithms

1. Breadth-First Search (BFS)

Description: Explores all nodes at the current depth level before moving to the next level.

Mechanism: Utilizes a **queue (FIFO)** to keep track of nodes to visit.

Advantages:

- Guaranteed to find the shortest path in an unweighted graph.

Disadvantages:

- Memory-intensive due to storage of all child nodes.

Example: Traversing a tree level by level.

Pseudocode:

Listing 1: BFS Pseudocode

```
def BFS(graph, start):
    visited = []
    queue = [start]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)
            queue.extend(graph[node])
    return visited
```

2. Depth-First Search (DFS)

Description: Explores as far as possible along each branch before backtracking.

Mechanism: Uses a **stack (LIFO)** or recursion.

Advantages:

- Requires less memory compared to BFS.

Disadvantages:

- May get stuck in infinite loops if not properly managed.
- Not guaranteed to find the shortest path.

Example: Solving mazes or puzzles.

Pseudocode:

Listing 2: DFS Pseudocode

```
def DFS(graph, start, visited=None):
    if visited is None:
        visited = []

    visited.append(start)
    for neighbor in graph[start]:
        if neighbor not in visited:
            DFS(graph, neighbor, visited)
    return visited
```

Informed Search Algorithms

3. Best-First Search

Description: Uses a heuristic function to prioritize nodes that are most likely to lead to the goal.

Mechanism: A **priority queue** is used, where nodes are sorted based on the heuristic value.

Advantages:

- Faster than uninformed searches.

Disadvantages:

- Not guaranteed to find the optimal solution.

Example: Pathfinding algorithms like Greedy Search.

4. A* Search

Description: Combines the cost from the start node to the current node ($g(n)$) with the estimated cost to the goal ($h(n)$).

Formula: $f(n) = g(n) + h(n)$

Mechanism: Uses a **priority queue**.

Advantages:

- Guaranteed to find the optimal solution if the heuristic is admissible (never overestimates).

Disadvantages:

- Can be slow if the heuristic is not well-designed.

Example: Google Maps shortest path search.

5. AO* Search

Description: Used for AND-OR graphs where the solution may involve a combination of actions or paths.

Mechanism: Evaluates and expands nodes based on a cost function and supports subgoal combinations.

Advantages:

- Effective for problems with multiple possible solutions.

Disadvantages:

- Complex implementation.

Example: Game decision trees.

6. Hill Climbing

Description: Continuously moves towards the node with the highest heuristic value.

Mechanism: A greedy approach focusing on local optimization.

Advantages:

- Simple and memory-efficient.

Disadvantages:

- May get stuck in local maxima, plateaus, or ridges.

Example: Solving optimization problems like the Travelling Salesman Problem.

7. Min/Max Search

Description: Used in decision-making scenarios like games to minimize loss and maximize gain.

Mechanism: Alternates between minimizing the opponent's score and maximizing the player's score.

Advantages:

- Ensures an optimal strategy in a perfect information setting.

Disadvantages:

- Computationally expensive for large trees.

Example: Chess AI algorithms.

Example: Chess AI.

Shortest Path Guarantees in Search Algorithms

This section explains how different search algorithms (BFS, DFS, Best-First Search, A*, and AO*) guarantee or fail to guarantee the shortest path. Each algorithm works under specific conditions, and their guarantees depend on the type of graph and the properties of the heuristic used (if applicable).

1 Breadth-First Search (BFS)

Does BFS guarantee the shortest path?
Yes, BFS guarantees the shortest path, but **only under specific conditions**:

- **Condition:** The graph must be **unweighted**. BFS explores all nodes at the same distance (level) from the source before moving to nodes further away.
- **Exception:** In a **weighted graph**, BFS treats all edges as having the same weight, so it may not find the shortest path.

Example:

- In an unweighted graph, BFS will find the shortest path in terms of the **number of edges**.
- In a weighted graph, BFS may fail because it does not consider edge weights.

2 Depth-First Search (DFS)

Does DFS guarantee the shortest path?
No, DFS does not guarantee the shortest path. It explores one branch (path) fully before backtracking, which may lead to longer or unnecessary paths.

Why?

- DFS prioritizes depth over proximity to the destination, so it can miss shorter paths if they lie along another branch.

3 Best-First Search

Does Best-First Search guarantee the shortest path?
No, Best-First Search does not guarantee the shortest path. It uses a **heuristic function** to decide which node to explore next.

Why?

- Best-First Search prioritizes nodes that appear closer to the goal (based on the heuristic) but may ignore actual path costs.

Exception:

- If the heuristic is perfectly accurate, it might find the shortest path, but this is rare in practical scenarios.

4 A* Search Algorithm

Does A* guarantee the shortest path?
Yes, A* guarantees the shortest path, but only under the following conditions:

- The heuristic function $h(n)$ must be **admissible**, meaning it never overestimates the actual cost to reach the goal.
- The heuristic function $h(n)$ must be **consistent** (or monotonic), meaning it satisfies the triangle inequality:
$$h(n) \leq c(n, m) + h(m)$$
where $c(n, m)$ is the cost of the edge between nodes n and m.

How A* Works:

- A* combines the **cost-so-far** ($g(n)$) with the **estimated cost-to-go** ($h(n)$):
$$f(n) = g(n) + h(n)$$
- Nodes are explored based on the smallest $f(n)$, ensuring the shortest path is found first.

5 AO* Algorithm

Does AO* guarantee the shortest path?
Yes, but AO* is designed for **And-Or graphs**, not typical shortest-path problems. It guarantees the optimal solution under the following conditions:

- The heuristic is **admissible** and **consistent**.
- The problem involves subgoal dependencies (AND/OR nodes).

Key Features:

- **AND nodes:** Tasks requiring multiple subgoals to be completed together.
- **OR nodes:** Tasks where only one subgoal needs to be completed.

Summary Table

Breadth-First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores all nodes level by level. BFS is commonly used to find the shortest path in unweighted graphs because it visits nodes in order of increasing distance from the source. This document provides a detailed explanation of BFS, practical examples, and an application to the given tree diagram.

Algorithm	**Shortest Path Guarantee**	**Condition/Exception**
BFS	Yes (in unweighted graphs)	Fails in weighted graphs.
DFS	No	Explores longer/unnecessary paths first.
Best-First Search	No	Depends on heuristic; may ignore actual costs.
A*	Yes (with valid heuristic)	Heuristic must be admissible and consistent.
AO*	Yes (for And-Or graphs)	Solves problems with AND/OR dependencies.

Table 1: Comparison of Search Algorithms

How BFS Works

The BFS algorithm works as follows:

1. Initialize a queue and add the starting node (source) to it.
2. Visit the node at the front of the queue, mark it as visited, and add its unvisited neighbors to the queue.
3. Repeat the process until the queue is empty.

Key Properties of BFS:

- BFS guarantees the shortest path in terms of the number of edges in **unweighted graphs**.
- It explores all nodes at the same distance from the source before moving further.

Practical Applications of BFS

BFS has a wide range of applications, including:

- **Social Networks:** Finding the shortest connection path between two people.
- **Web Crawling:** Exploring web pages level by level starting from a root URL.
- **Maze Solving:** Finding the shortest route from the entrance to the exit in a maze.
- **Networking:** Broadcasting messages across all nodes in a network.

Application to the Tree Diagram

The provided tree diagram consists of nodes labeled from 1 to 12, with Node 1 as the root.

Tree Diagram Placeholder:

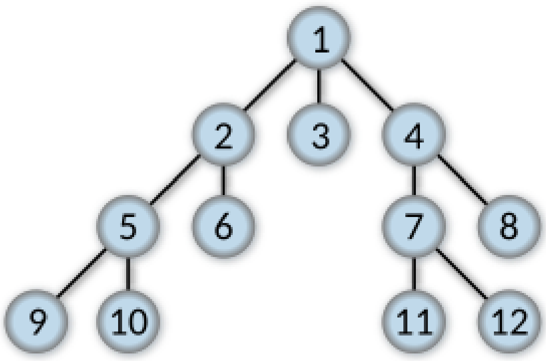


Figure 1: Tree diagram with nodes 1 to 12.

BFS Traversal of the Tree

To perform BFS on this tree:

1. Start at Node 1 (root).
2. Visit all neighbors of Node 1: $2, 3, 4$.
3. Next, visit the children of Node 2: $5, 6$.
4. Then visit the children of Node 4: $7, 8$.
5. Finally, visit the children of $5, 7, 8$: $9, 10, 11, 12$.

BFS Traversal Order:

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10 \rightarrow 11 \rightarrow 12$$

Finding the Shortest Path Using BFS

BFS can also be used to find the shortest path between a source node and a destination node. In this example, we calculate the shortest path from Node 1 to Node 12.

Steps to Find the Shortest Path

1. Start BFS at Node 1.
2. Track the parent of each node as BFS progresses:
   - Node 1 (source) has no parent.
   - Node 4 is reached from Node 1.
   - Node 8 is reached from Node 4.
   - Node 12 is reached from Node 8.
3. Trace back the path from Node 12 to Node 1 using the recorded parent relationships.

Shortest Path from Node 1 **to Node** 12

The shortest path is:

$$1 \to 4 \to 8 \to 12$$

This path has a total of 3 edges.

Conclusion

Breadth-First Search (BFS) is a powerful algorithm for both traversal and finding the shortest path in unweighted graphs. In the given tree, the BFS traversal order is:

$$1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10 \to 11 \to 12$$

However, the shortest path from Node 1 to Node 12 is:

$$1 \to 4 \to 8 \to 12$$

This demonstrates how BFS can systematically explore nodes and calculate shortest paths.

Depth-First Search (DFS)

Depth-First Search (DFS) is a traversal technique that explores as far as possible along each branch before backtracking. For the given tree, the DFS traversal proceeds as follows:

- Start at the root node: 1.
- Traverse left branches fully before backtracking.
- Visit each node exactly once, following the order: 1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 7.

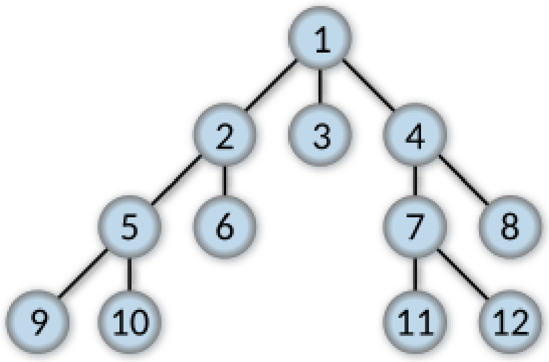


Figure 2: Tree diagram with nodes 1 to 12. DFS traverses the nodes in the order: 1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 7.

DFS explores a tree by following one branch to its deepest node before backtracking to explore other branches. Here's the step-by-step process:

1. Start at the root node: 1.
2. Move to the left child: 2, then to its left child: 4.
3. Continue to the leftmost leaf node: 8 (no further children).
4. Backtrack to node 4 and visit its right child: 9.
5. Backtrack to node 2 and explore its right subtree: visit 5, then its children: 10 and 11.
6. Backtrack to the root node (1) and move to the right subtree.
7. Visit node 3, explore its left child: 6, then its left child: 12.
8. Backtrack to node 3 and visit its right child: 7.

Final DFS Order: 1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 7.

Best-First Search (BFS)

Best-First Search (BFS) is an informed search algorithm that uses a heuristic to decide the next node to visit. It explores nodes with the lowest heuristic value first, prioritizing the most promising path.

Steps of BFS

1. Start at the source node (S). Explore its neighbors and add them to a priority queue.
2. Expand the node with the lowest heuristic value.
3. Continue exploring and updating the priority queue until the goal node is reached.

Graph Traversal Example

For the given graph:

- Start at node S.
- Traverse nodes based on their heuristic values.
- Final order of traversal: **S** $\rightarrow$ **A** $\rightarrow$ **C** $\rightarrow$ **H** $\rightarrow$ **I** $\rightarrow$ **K**.

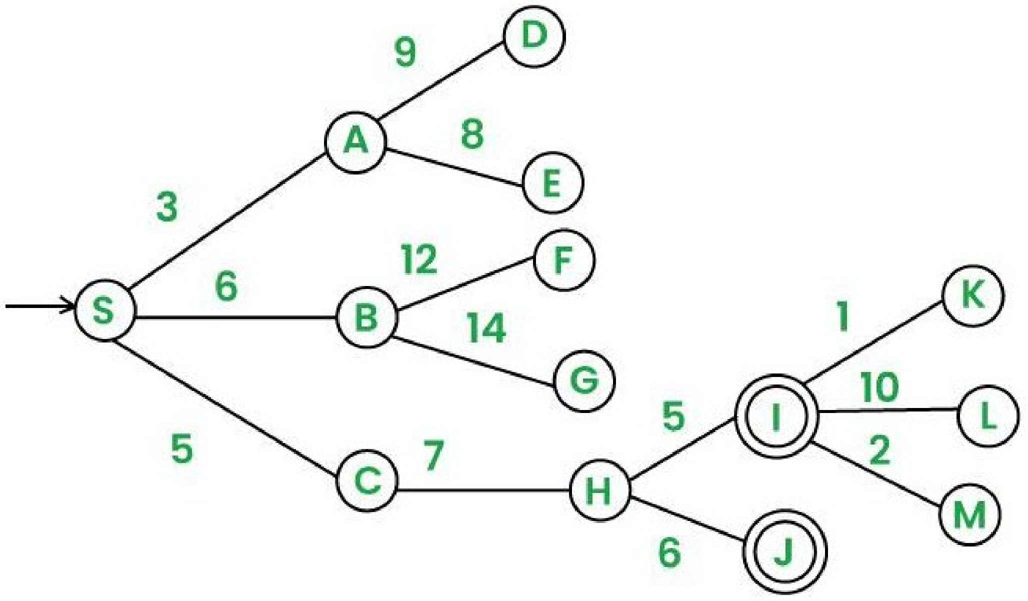


Figure 3: Graph for Best-First Search. Nodes are expanded based on their heuristic values, starting at S and ending at K.

How Best-First Search Works

Best-First Search (BFS) finds the goal node by using a heuristic function to determine the most promising path. The steps to determine the answer are as follows:

1. Start at the root node (S). Add its neighbors to the priority queue based on their heuristic values.
2. Expand the node with the lowest heuristic value in the queue.
3. Add the neighbors of the expanded node to the queue, ensuring the queue remains sorted by heuristic values.
4. Continue expanding nodes until the goal node is reached.

Example Execution

Graph Traversal Steps:

1. Start at S. Expand its neighbors: A (3), B (6), C (5). Priority Queue: {A (3), C (5), B (6)}.
2. Expand A (lowest heuristic = 3). Add its neighbors: D (9), E (8). Priority Queue: {C (5), B (6), E (8), D (9)}.
3. Expand C (lowest heuristic = 5). Add its neighbor: H (5). Priority Queue: {H (5), B (6), E (8), D (9)}.
4. Expand H (lowest heuristic = 5). Add its neighbors: I (1), J (6). Priority Queue:

Step-by-Step Explanation

(a) **Start from the Initial Node (S):**
   - The algorithm starts at the root node **S**.
   - All neighbors of **S** (**A**, **B**, **C**) are examined, and their heuristic values are compared.
   - These neighbors are added to the priority queue, sorted by their heuristic values.
   - *Priority Queue after step 1:* {A (3), C (5), B (6)}.

(b) **Expand the Node with the Lowest Heuristic Value:**
   - Node **A** has the smallest heuristic value (**3**) and is expanded first.
   - Neighbors of **A** (**D (9)** and **E (8)**) are added to the priority queue.
   - *Priority Queue after step 2:* {C (5), B (6), E (8), D (9)}.

(c) **Expand the Next Node (C):**
   - Node **C** has the smallest heuristic value (**5**) and is expanded next.
   - Neighbor of **C** (**H (5)**) is added to the priority queue.
   - *Priority Queue after step 3:* {H (5), B (6), E (8), D (9)}.

(d) **Expand the Next Node (H):**
   - Node **H** has the smallest heuristic value (**5**) and is expanded.
   - Neighbors of **H** (**I (1)** and **J (6)**) are added to the priority queue.
   - *Priority Queue after step 4:* {I (1), B (6), J (6), E (8), D (9)}.

(e) **Expand the Next Node (I):**
   - Node **I** has the smallest heuristic value (**1**) and is expanded.
   - Neighbors of **I** (**K (1)**, **L (10)**, and **M (2)**) are added to the priority queue.
   - *Priority Queue after step 5:* {K (1), M (2), B (6), J (6), E (8), D (9), L (10)}.

(f) **Expand the Next Node (K):**
   - Node **K** has the smallest heuristic value (**1**) and is expanded. If **K** is the goal node, the algorithm terminates.

Final Answer

The algorithm outputs the traversal path:

$$\mathbf{S \rightarrow A \rightarrow C \rightarrow H \rightarrow I \rightarrow K}$$

Explanation of the Algorithm

- **Heuristic Values:** At each step, the node with the lowest heuristic value is selected, ensuring the most promising node is explored first.
- **Priority Queue:** The priority queue maintains the sorted order of nodes based on their heuristic values.
- **Informed Search:** The heuristic values guide the search toward the goal, minimizing unnecessary exploration.

A* Algorithm

A* is an informed search algorithm that evaluates nodes based on the formula:

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

where:

- $g(n)$: Cost to reach the node from the start.
- $h(n)$: Heuristic value (estimated cost to the goal).
- $f(n)$: Total estimated cost of the path through the node.

Steps of A* Algorithm

(a) Start at the initial node (S).
(b) Compute $f(n) = g(n) + h(n)$ for all neighbors.
(c) Expand the node with the smallest $f(n)$.
(d) Repeat until the goal node is reached.

Example

Graph and Problem Statement

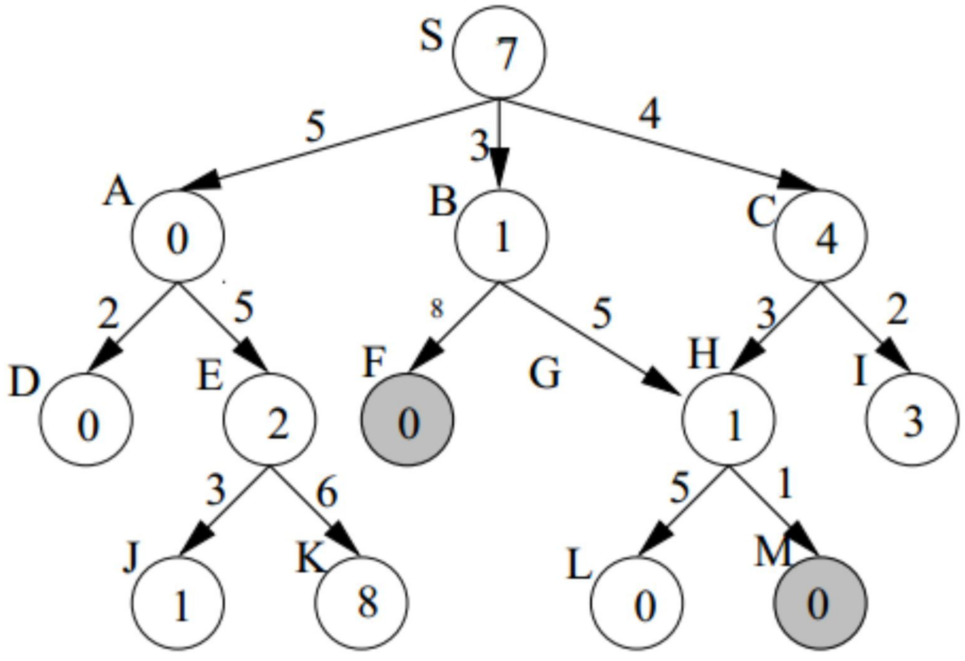


Figure 4: Graph for A* Algorithm Example. Each node has a heuristic value, and edges are labeled with their respective costs.

The A* algorithm combines two factors to decide which node to expand:

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

where:

- $g(n)$: The cost to reach the node from the start.
- $h(n)$: The heuristic value (estimated cost to the goal).
- $f(n)$: The total estimated cost to reach the goal through the node.

—

—

Step-by-Step Execution

1. Start at Node S:

- $g(S) = 0$ (cost to reach S).
- $h(S) = 7$ (heuristic value from S to the goal).
- $f(S) = g(S) + h(S) = 0 + 7 = 7$.

Expand S and calculate $f(n)$ for its neighbors (A, B, C).

2. Calculate $f(n)$ for Neighbors of S:

- For A:

  $$g(A) = g(S)+\text{cost}(S \to A) = 0+5 = 5, \quad h(A) = 0, \quad f(A) = g(A)+h(A) = 5+0 = 5.$$

- For B:

  $$g(B) = g(S)+\text{cost}(S \to B) = 0+3 = 3, \quad h(B) = 1, \quad f(B) = g(B)+h(B) = 3+1 = 4.$$

- For C:

  $$g(C) = g(S)+\text{cost}(S \to C) = 0+4 = 4, \quad h(C) = 4, \quad f(C) = g(C)+h(C) = 4+4 = 8.$$

Priority Queue: $B(4), A(5), C(8)$.

—

3. Expand Node B (Lowest $f(n) = 4$):

- Expand B and calculate $f(n)$ for its neighbors (F, G):
  - For F:
    $$g(F) = g(B) + \text{cost}(B \to F) = 3 + 8 = 11,$$
    $$h(F) = 0, \quad f(F) = g(F) + h(F) = 11 + 0 = 11.$$
  - For G:
    $$g(G) = g(B) + \text{cost}(B \to G) = 3 + 5 = 8,$$
    $$h(G) = 0, \quad f(G) = g(G) + h(G) = 8 + 0 = 8.$$

Priority Queue: $A(5), C(8), G(8), F(11)$.

4. Expand Node A (Lowest $f(n) = 5$):

- Expand A and calculate $f(n)$ for its neighbors (D, E):
  - For D:
$$g(D) = g(A) + \text{cost}(A \to D) = 5 + 2 = 7,$$
$$h(D) = 0, \quad f(D) = g(D) + h(D) = 7 + 0 = 7.$$
  - For E:
$$g(E) = g(A) + \text{cost}(A \to E) = 5 + 5 = 10,$$
$$h(E) = 2, \quad f(E) = g(E) + h(E) = 10 + 2 = 12.$$

Priority Queue: $D(7), C(8), G(8), F(11), E(12)$.

—

Final Answer

The shortest path is:
$$S \to A \to D$$
With a total cost:
$$f(D) = 7$$

—

Why A* Works

- **Heuristic Guidance:** The heuristic ($h(n)$) directs the search towards the goal.
- **Cost Accuracy:** The actual cost ($g(n)$) ensures the algorithm finds the shortest path.
- **Optimality:** A* guarantees the shortest path as long as $h(n)$ is admissible (never overestimates the true cost).

AO* Algorithm

The AO* algorithm is a heuristic-based search algorithm designed for graphs containing AND-OR structures. It is particularly useful in solving problems where the solution involves decomposable sub-problems with dependencies.

Key Features of AO* Algorithm:

- **AND Nodes:** These represent tasks where all child nodes must be completed to proceed. The total cost of an AND node is the sum of the costs of its children.
- **OR Nodes:** These represent choices. Only one of the child nodes needs to be selected, and the cost is the minimum among all children.
- **Optimal Search:** The AO* algorithm uses heuristic cost estimates to prioritize the most promising paths and prunes suboptimal branches.

Steps of AO* Algorithm:

(a) Start at the root node of the graph.

(b) Expand the most promising node, guided by the lowest estimated cost.

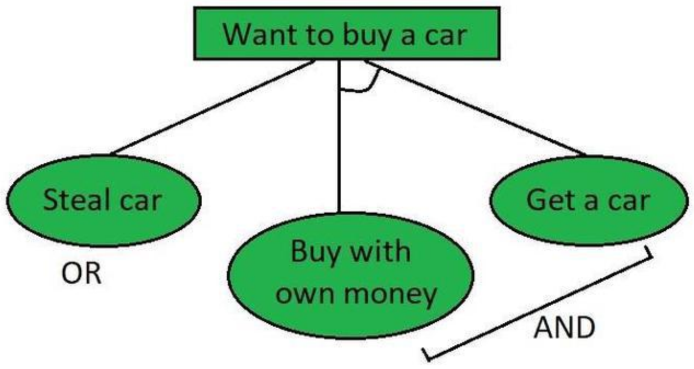
(c) For each expanded node:
   - Calculate the cost of child nodes.
   - For AND nodes, sum the costs of all children.
   - For OR nodes, select the child with the minimum cost.

(d) Backtrack to update the cost of the parent nodes.

(e) Repeat until the optimal path to the goal is determined.

Illustration of AND-OR Structures:

Below is an example of an AND-OR graph structure:



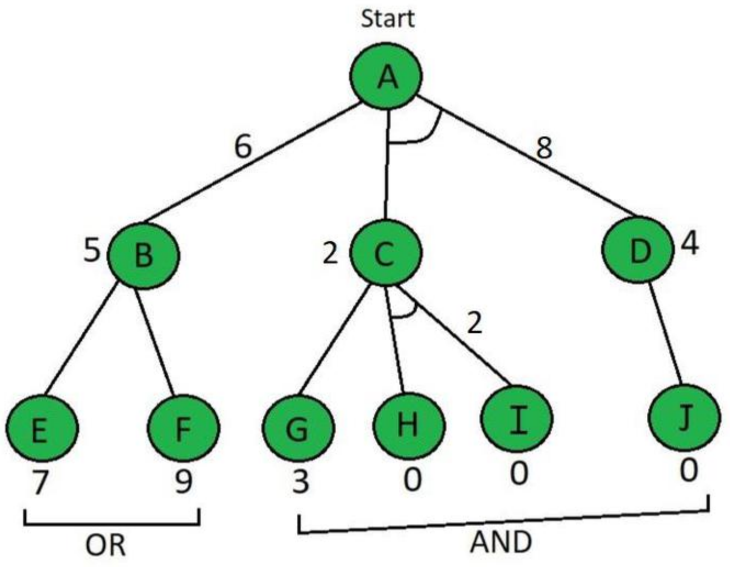
Advantages of AO* Algorithm:

- Efficiently identifies the optimal path in graphs with AND-OR relationships.
- Reduces computational overhead by pruning suboptimal branches.
- Provides a systematic approach for solving problems with dependencies.

The AO* algorithm is widely used in artificial intelligence for tasks such as game tree searches, decision-making, and problem-solving in hierarchical domains.

Finding the Optimal Path using AO* Algorithm

The following steps demonstrate how to find the optimal path using the AO* algorithm, referring to the graph below:



Steps to Find the Path

1. **Start at the Root Node (A):** Node A is the starting point. It has three branches leading to nodes B, C, and D.

2. **Check the Type of Nodes:**

- **OR Nodes:** Choose the child with the minimum cost.
- **AND Nodes:** Include all children and sum their costs.

3. **Expand the Children:** From A, expand its children:

$$\text{Cost to } B = 6 + f(B), \quad \text{Cost to } C = 2 + f(C), \quad \text{Cost to } D = 8 + f(D).$$

4. **Calculate Costs for Each Branch:**

- **Branch to B:** B is an OR node. Choose the minimum cost between E (7) and F (9):

$$f(B) = 7 \quad \text{(minimum cost)}, \quad \text{Total cost of path } A \to B = 6 + 7 = 13.$$

- **Branch to C:** C is an AND node. Add the costs of all children (G, H, I):

$$f(C) = 3 + 0 + 0 = 3, \quad \text{Total cost of path } A \to C = 2 + 3 = 5.$$

- **Branch to D:** D is an AND node. Add the costs of all children (J):

$$f(D) = 0, \quad \text{Total cost of path } A \to D = 8 + 0 = 8.$$

5. **Choose the Optimal Path:** Compare the total costs:

$$\text{Path } A \to B = 13, \quad \text{Path } A \to C = 5, \quad \text{Path } A \to D = 8.$$

The optimal path is:

$$\mathbf{A} \to \mathbf{C} \to \mathbf{G}, \mathbf{H}, \mathbf{I} \quad \text{with a total cost of 5.}$$

Key Points:

- **OR Nodes:** Select the child with the minimum cost.
- **AND Nodes:** Add the costs of all connected children.
- Use backtracking to update costs at each level and ensure the minimum cost path is chosen.

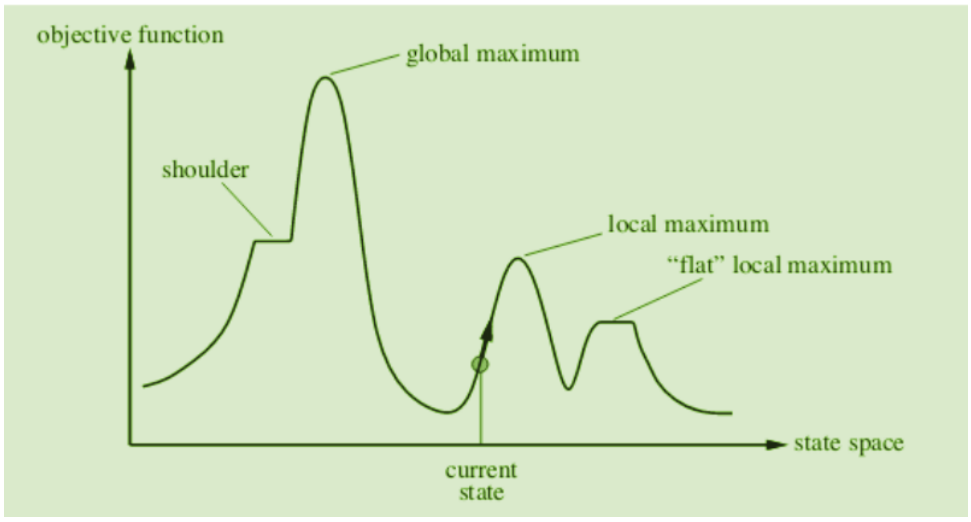
Hill Climbing Algorithm

The Hill Climbing algorithm is a heuristic search method used to find the optimal solution in problems with a single objective. It iteratively improves the current state by moving towards neighboring states with higher objective function values. The process continues until a peak (maximum value) is reached.

State-Space Diagram in Hill Climbing

The state-space diagram represents all possible states the algorithm can explore, plotted against the values of the objective function.

- **X-axis:** Represents the state space, which includes all the possible states or configurations.
- **Y-axis:** Represents the objective function values corresponding to each state.
- The **optimal solution** is the state where the objective function achieves its **global maximum**.



Key Regions in the State-Space Diagram

- **Local Maximum:** A local maximum is a state that is better than its neighbors but not the best overall. The algorithm might mistakenly stop here, thinking it has reached the optimal solution.
- **Global Maximum:** The global maximum is the best state, where the objective function achieves its highest value. This is the optimal solution the algorithm seeks.
- **Plateau/Flat Local Maximum:** A plateau is a flat region where neighboring states have the same objective function value. This makes it difficult for the algorithm to decide on the best direction to move.
- **Ridge:** A ridge is a higher region with a slope that may look like a peak, causing the algorithm to stop prematurely, missing better solutions nearby.
- **Current State:** The current state refers to the algorithm's position in the state-space diagram during its search for the optimal solution.
- **Shoulder:** A shoulder is a plateau with an uphill edge, allowing the algorithm to move toward better solutions if it continues searching beyond the plateau.

Challenges in Hill Climbing

- **Local Maxima:** The algorithm may get stuck in a local maximum and fail to find the global maximum.
- **Plateaus:** The algorithm may stop progressing if it encounters a flat region in the state space.
- **Ridges:** The algorithm may stop prematurely on ridges, missing better solutions nearby.

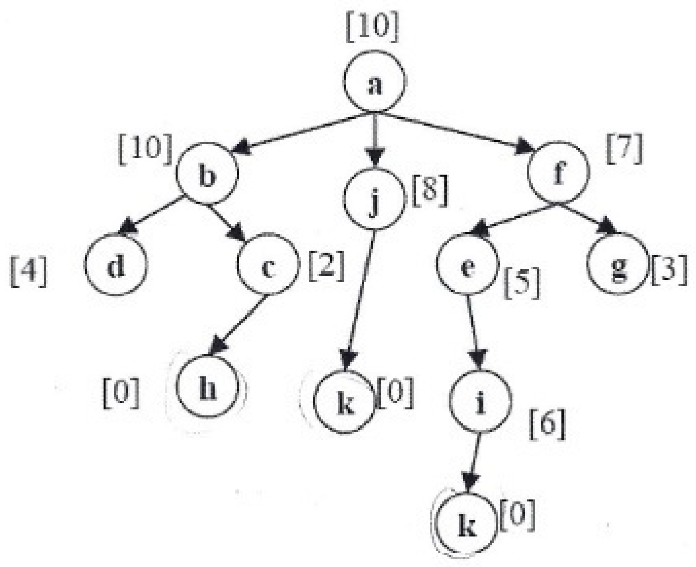
Hill Climbing is a simple yet effective algorithm for optimization tasks. However, its performance depends heavily on the problem's landscape, as shown in the state-space diagram.

Solving the Problem Using Hill Climbing Algorithm

The Hill Climbing algorithm is used to find the optimal path by iteratively moving towards the neighbor with the highest objective function value. Below is a step-by-step solution to the given problem.

Tree Diagram

The provided tree diagram is shown below:



Each node is annotated with its heuristic value in brackets, and the goal is to maximize this value starting from the root node (a).

Steps to Solve Using Hill Climbing Algorithm

1. **Start at the Root Node (a):** - The heuristic value of the root node a is 10. - The algorithm evaluates all children of a: b (10), j (8), and f (7).

2. **Choose the Child with the Highest Value:** - Among the children of a, b has the highest heuristic value (10). - Move to node b.

3. **Evaluate the Children of Node b:** - The children of b are d (4) and c (2). - The node b itself has a value of 10, which is greater than any of its children. - As b is a local maximum, the algorithm stops here.

4. **Alternative Exploration (Optional):** - If the algorithm allows backtracking or restarting, it can explore other branches: - From a, consider j (8) and evaluate its children: e (5) and k (0). - From a, consider f (7) and its children: g (3).

5. **Final Result:** - The Hill Climbing algorithm finds the path $a \to b$ as the optimal solution, stopping at b (local maximum) with a heuristic value of 10.

Challenges and Observations

- The algorithm stops at the local maximum (b) without exploring other potential paths to find a better global maximum.
- Nodes j and f could be explored with other strategies (e.g., simulated annealing) to find alternate solutions.

Key Notes:

- Hill Climbing only considers the immediate neighbors, moving greedily to the highest value. - This algorithm may fail to find the global maximum due to getting stuck in local maxima.

Min-Max Algorithm

The Min-Max algorithm is a decision-making algorithm used in game theory and artificial intelligence to find the optimal move for a player, assuming that the opponent also plays optimally. It is widely used in two-player, zero-sum games like chess or tic-tac-toe.

How Min-Max Works

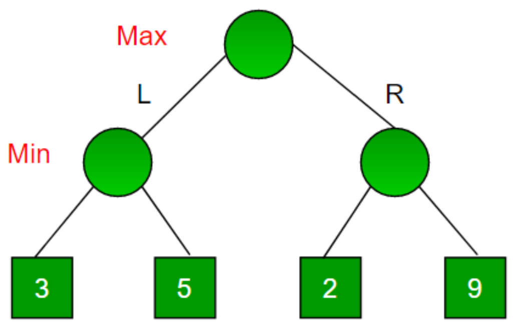
The algorithm evaluates the game tree by recursively analyzing all possible moves:

- **Maximizing Player (Max):** Attempts to maximize their score by choosing the highest value among child nodes.
- **Minimizing Player (Min):** Attempts to minimize the maximizing player's score by choosing the lowest value among child nodes.

The evaluation starts from the terminal nodes (leaf nodes) of the tree and backtracks to the root, calculating the optimal value at each step.

Example with Diagram

Consider the following game tree:



Steps to Solve Using Min-Max:

(a) Start from the leaf nodes:

$$\text{Leaf nodes: } 3, 5, 2, 9.$$

(b) Evaluate the Min level (L and R nodes):

- For L, the Min player chooses the smallest value:

$$\text{Min}(3, 5) = 3.$$

- For R, the Min player chooses the smallest value:

$$\text{Min}(2, 9) = 2.$$

(c) Evaluate the Max level (Root node):

   - The Max player chooses the largest value between L and R:

$$\text{Max}(3,2) = 3.$$

Optimal Value: The optimal value for the Max player is 3, and the optimal path is Root $\rightarrow L \rightarrow 3$.

Key Points

- Min-Max assumes that both players play optimally.
- The algorithm explores all possible moves, making it computationally expensive for large game trees.
- Pruning techniques like Alpha-Beta Pruning can be used to reduce the number of nodes evaluated.

Real-World Scenarios for Search Algorithms

The following table outlines real-world applications of various search algorithms, along with brief explanations.

Algorithm	**Real-World Application**	**Explanation**
Breadth-First Search (BFS)	Social Network Analysis	Used to find the shortest path or discover connections between users in platforms like Facebook or LinkedIn.
Depth-First Search (DFS)	Maze Solving	Explores all possible paths to find a solution by diving deep into one path at a time.
Best-First Search	GPS Navigation Systems	Finds the shortest or fastest route based on a heuristic like distance or travel time.
A* Algorithm	Route Optimization in Delivery Systems	Combines path cost and heuristic to determine the most cost-effective route for package deliveries.
AO* Algorithm	Expert Systems	Used in decision-making tasks like medical diagnosis, where decisions follow AND-OR relationships.
Hill Climbing Algorithm	Machine Learning Hyperparameter Tuning	Iteratively adjusts parameters to maximize model performance.
Min-Max Algorithm	Chess and Board Games	Helps find the best move in games by simulating both player and opponent moves optimally.

Modeling a Shooting Game Using Different Search Algorithms

The scenario involves a person navigating through rooms, killing lions, and progressing to the final goal. Each algorithm will approach the problem differently.

Game Setup

- The player starts at the entrance of a maze-like series of rooms.
- Each room may contain lions, weapons, or rewards.
- The goal is to reach the final room while minimizing damage, maximizing rewards, or completing the mission efficiently.

How Each Algorithm Handles the Game

1. Breadth-First Search (BFS)

- **Strategy:** Explore all rooms at the current depth before moving deeper into the maze.
- **Game Impact:** BFS guarantees that the shortest path (in terms of the number of rooms visited) to the final room is found, but it does not account for dangers (lions) or rewards in rooms.
- **Example:** The player goes room by room, exploring all rooms on the same floor before going to the next floor.

2. Depth-First Search (DFS)

- **Strategy:** Fully explore one path (room to room) until reaching a dead-end or the final room, then backtrack.
- **Game Impact:** The player may take longer paths to the goal, as DFS explores deeply before considering alternatives.
- **Example:** The player might explore a dead-end path full of lions before realizing a safer path exists.

3. Best-First Search

- **Strategy:** Use a heuristic to prioritize the most promising rooms (e.g., rooms with fewer lions or higher rewards).
- **Game Impact:** The player chooses paths that seem best based on the heuristic but may not find the optimal path.
- **Example:** The player prefers rooms with visible weapons or avoids rooms with roaring lions.

4. A* Algorithm

- **Strategy:** Combine path cost (rooms traveled) and a heuristic (e.g., danger level or rewards) to find the optimal path.
- **Game Impact:** The player balances between minimizing distance and avoiding danger, finding the most efficient and safest route.
- **Example:** The player moves strategically, avoiding dangerous rooms while progressing efficiently to the goal.

5. AO* Algorithm

- **Strategy:** Break the game into AND-OR dependencies. For example, certain rooms require completing tasks (killing lions) before progressing.
- **Game Impact:** The player must fulfill prerequisites (kill lions in some rooms) before accessing others, ensuring task dependencies are respected.

- **Example:** The player clears a room of lions (AND task) and then chooses which adjacent room to enter (OR choice).

6. Hill Climbing Algorithm

- **Strategy:** Move to the neighboring room with the highest immediate reward or lowest danger.
- **Game Impact:** The player may get stuck in a local maximum, failing to find the global optimal path.
- **Example:** The player keeps choosing rooms with visible rewards but gets stuck in a small safe area surrounded by lions.

7. Min-Max Algorithm

- **Strategy:** Simulate the opponent (lions) playing optimally to minimize the player's success while maximizing the player's reward.
- **Game Impact:** The player anticipates lion behavior and plans moves to counteract threats, ensuring optimal survival and success.
- **Example:** The player predicts lion movements in adjacent rooms and chooses paths to avoid being cornered.

Solving the 8-Puzzle Problem Using A* Algorithm

The A* algorithm is a heuristic-based search method used to find the shortest path to a goal state. In this example, we solve an 8-puzzle problem where the goal is to move tiles to match the desired configuration.

Problem Description

The initial and goal states for the puzzle are shown below:

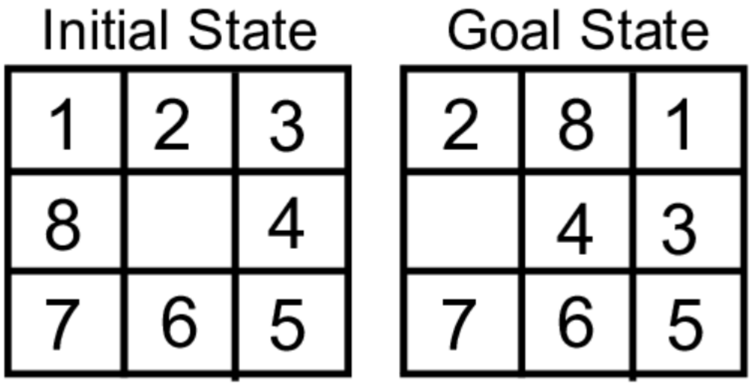


Figure 5: Initial State and Goal State

Key Definitions in A* Algorithm

1. $g(n)$: The cost to reach a node n (number of moves made so far). 2. $h(n)$: The heuristic estimate of the cost to reach the goal from node n (e.g., number of misplaced tiles). 3. $f(n)$: The total cost, calculated as:

$$f(n) = g(n) + h(n).$$

Steps to Solve the Puzzle

The algorithm explores nodes (puzzle configurations) based on the lowest $f(n)$ value.

(a) Start with the **initial state** and calculate $f(n) = g(n) + h(n)$: - $g = 0$ (no moves yet). - $h = 3$ (number of misplaced tiles: 1, 8, and 2 are in the wrong positions). - $f = g + h = 0 + 3 = 3$.

(b) Generate all possible moves (children nodes) from the initial state and calculate $f(n)$ for each: - Example moves include swapping blank space with adjacent tiles.

(c) Select the node with the lowest $f(n)$ value and expand further until the goal state is reached.

Visualization of the Solution Process

The following figure illustrates the exploration of nodes using A*:

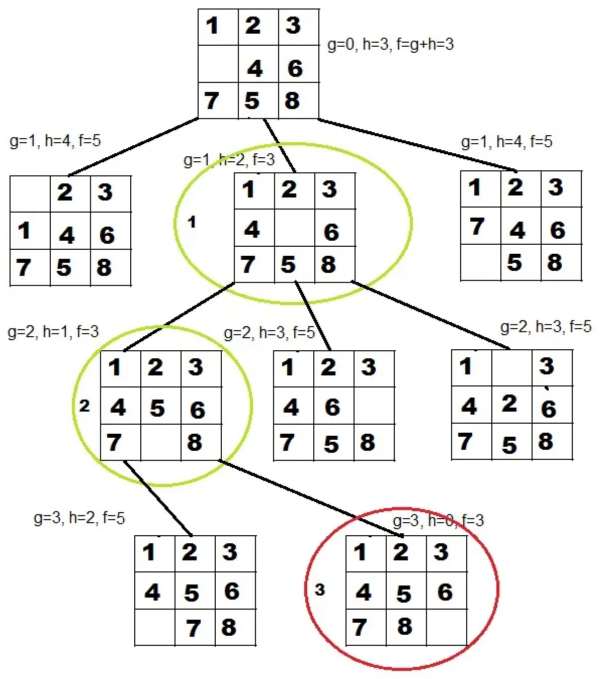


Figure 6: Solution Tree Using A* Algorithm

Example Calculations

1. From the initial state:

   - Swap blank with tile 2:

   $$g = 1, \quad h = 2 \text{ (misplaced tiles)}, \quad f = g + h = 3.$$

   - Swap blank with tile 8:

   $$g = 1, \quad h = 4 \text{ (misplaced tiles)}, \quad f = g + h = 5.$$

2. Choose the node with the smallest $f(n)$ (e.g., $f = 3$) and continue expanding.

Advantages of A* Algorithm

- Finds the shortest path to the goal efficiently.
- Combines path cost (g) and heuristic (h) to focus on the most promising paths.

Result

Using A*, the algorithm successfully reaches the goal state by systematically exploring nodes based on $f(n)$.