CAP 378
ARTIFICIAL INTELIGENCE

Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University

UNIT – II
Heuristic Search and Game Playing

Tanzeela Javid Kaloo (32638)
Assistant Professor
System And Architecture
Lovely Professional University

Content

- Introduction To Heuristic And Blind Search
- Heuristic Search Strategies -
  - Best First Search
  - A* Algorithm
  - Iterative Deepening A*(IDA)
  - Small Memory A*(SMA)
- Heuristic Techniques:
  - Generate And Test
  - Hill Climbing
  - State Space Search
  - Constraint Satisfaction Problem
- Introduction To Game Playing
- Applications Of Game Playing
- Minimax Algorithm And Alphabeta Pruning
- Perfect Decision Game And Imperfect Decision Game

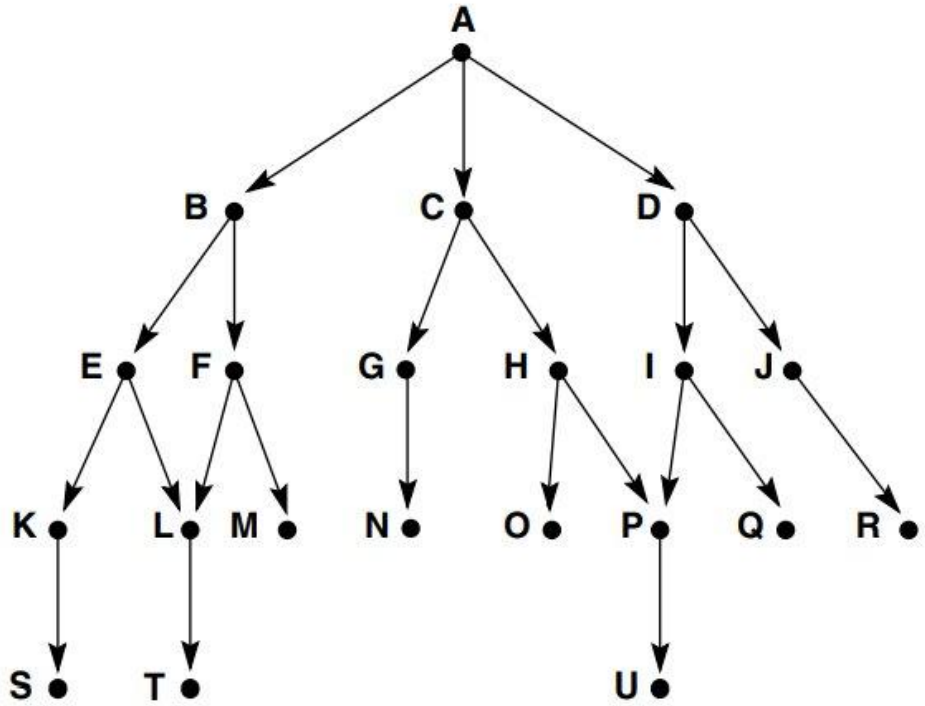
Introduction to Heuristic and Blind Search

- Search strategies are foundational in AI, enabling systems to find solutions efficiently. There are two broad categories of these search strategies:
  - **Blind Search:** Searches without additional information, relying only on the problem structure.
  - **Heuristic Search:** Utilizes problem-specific knowledge to guide the search towards solutions more effectively.

Blind Search

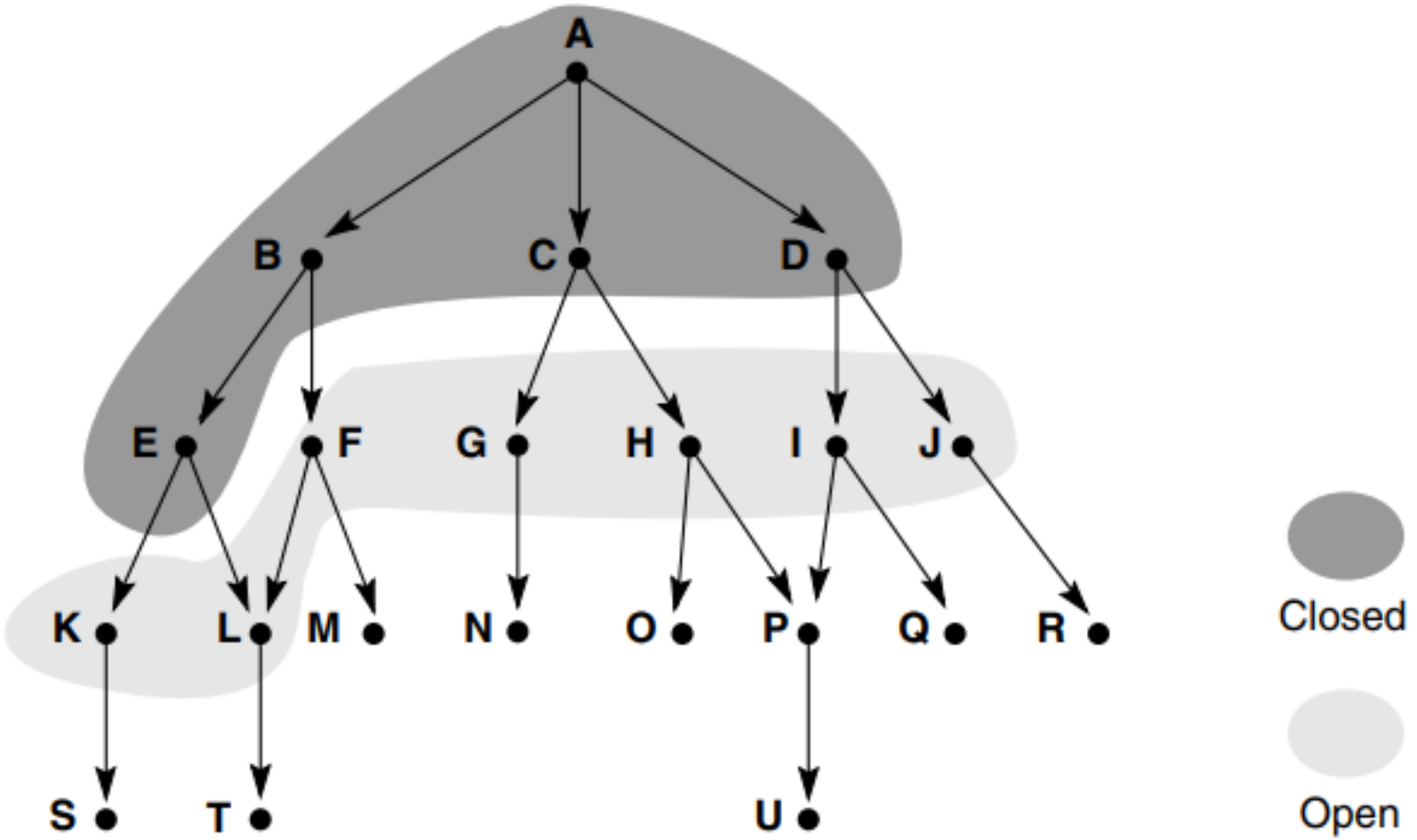
- Blind search methods explore the problem space systematically but lack any insight into the goal's location. Example:
  - **Breadth-First Search (BFS):** Explores all possible nodes at one depth level before moving to the next.
  - **Depth-First Search (DFS):** Follows one path to its deepest point before backtracking.
- While these methods are exhaustive, they are inefficient for large problem spaces. This limitation necessitates the use of more intelligent strategies.

Breadth First Search

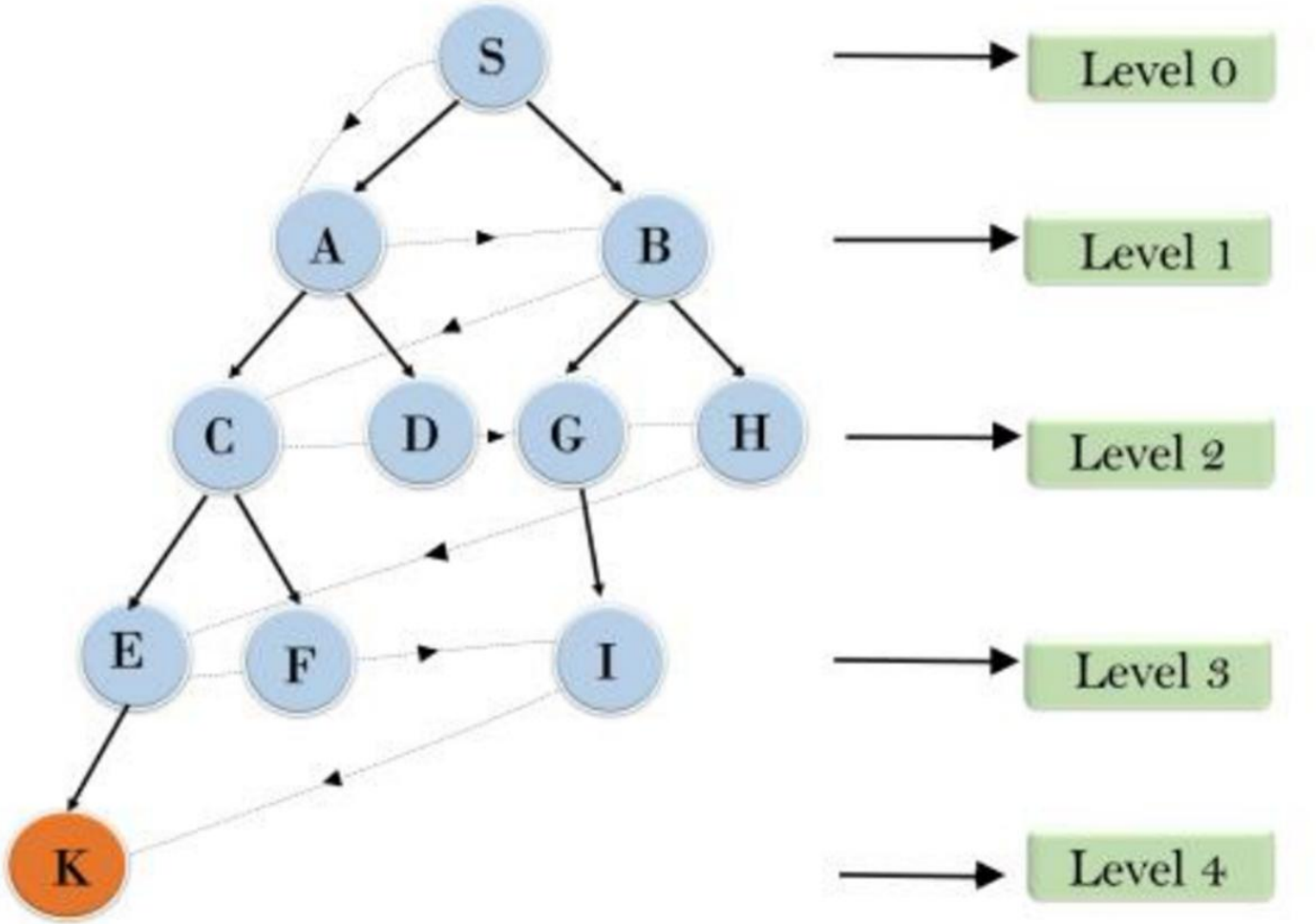


1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [C,D,E,F]; closed = [B,A]
4. open = [D,E,F,G,H]; closed = [C,B,A]
5. open = [E,F,G,H,I,J]; closed = [D,C,B,A]
6. open = [F,G,H,I,J,K,L]; closed = [E,D,C,B,A]
7. open = [G,H,I,J,K,L,M] (as L is already on open); closed = [F,E,D,C,B,A]
8. open = [H,I,J,K,L,M,N]; closed = [G,F,E,D,C,B,A]
9. and so on until either U is found or open = [ ].

open = [(D,A), (E,B), (F,B), (G,C), (H,C)]; closed = [(C,A), (B,A), (A,nil)]



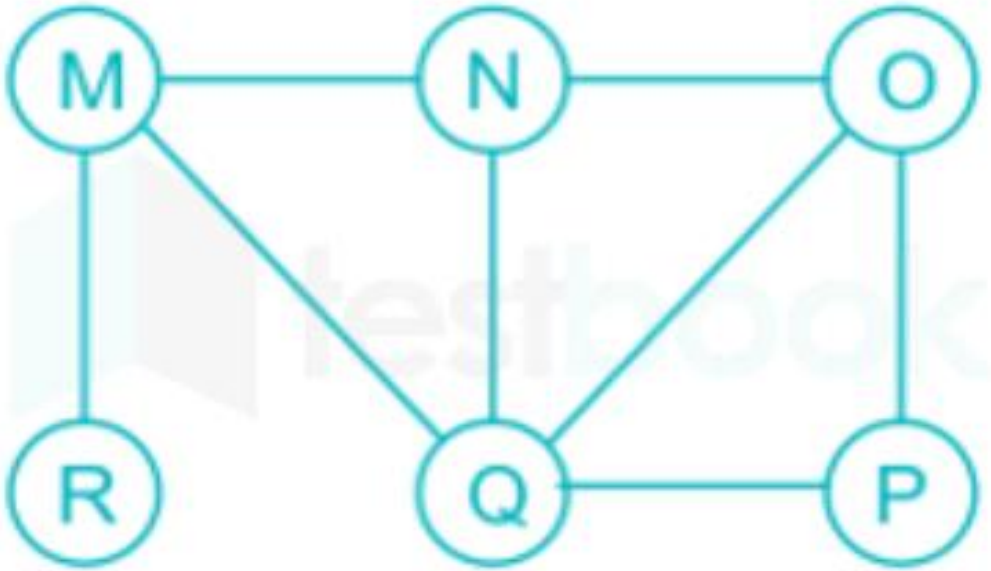
Breadth First Search (DFS)



Practice Questions

Which of the following is the possible order of visiting the nodes in the graph below using the Breadth First Search (BFS) algorithm?

1. MNOPQR
2. NQMPOR
3. QMNROP
4. POQNMR
5. PORNMQ



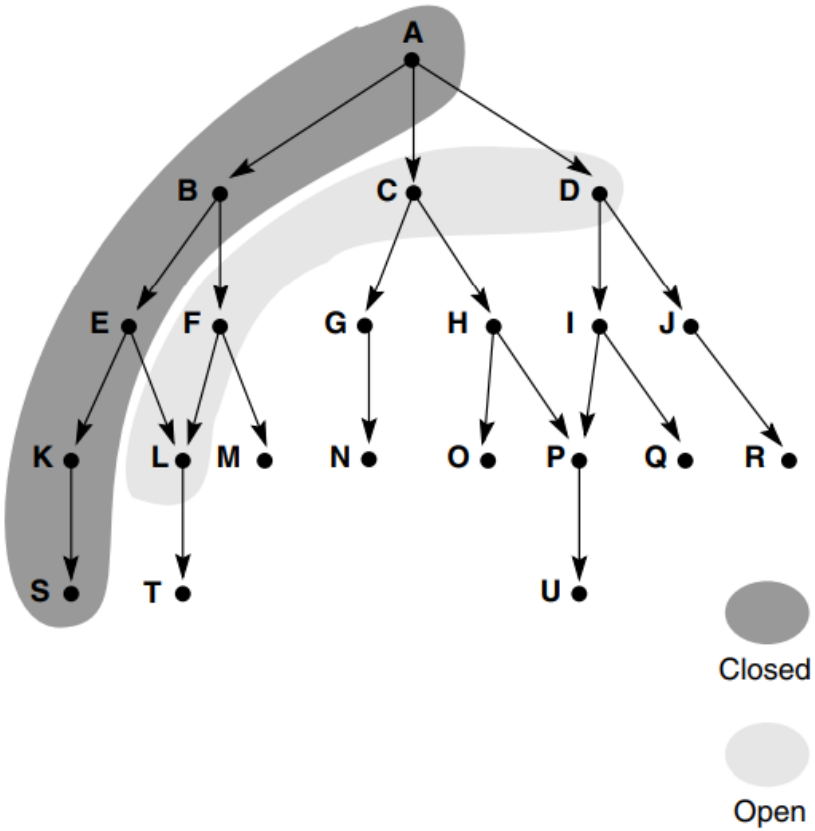
Practice Questions

Q: Breadth-First Search (BFS) is started on a binary tree beginning from the root vertex. There is a vertex t at a distance 4 from the root. If t is the n^{th} vertex in this BFS traversal, then the maximum possible value of n is _____?

Hint: Height of tree = N

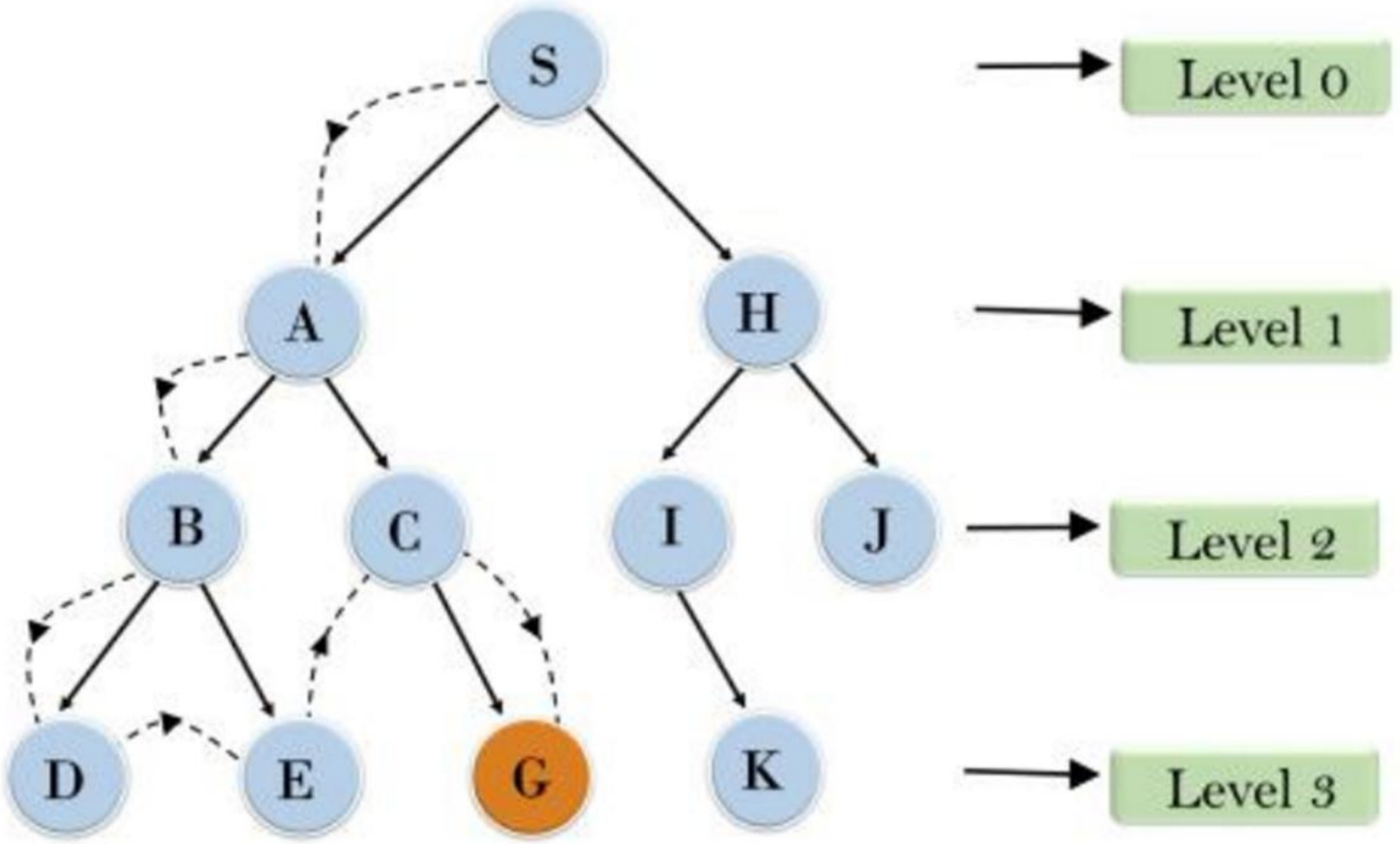
Maximum Nodes = B^{N+1} -1

Depth First Search



1. open = [A]; closed = [ ]
2. open = [B,C,D]; closed = [A]
3. open = [E,F,C,D]; closed = [B,A]
4. open = [K,L,F,C,D]; closed = [E,B,A]
5. open = [S,L,F,C,D]; closed = [K,E,B,A]
6. open = [L,F,C,D]; closed = [S,K,E,B,A]
7. open = [T,F,C,D]; closed = [L,S,K,E,B,A]
8. open = [F,C,D]; closed = [T,L,S,K,E,B,A]
9. open = [M,C,D], (as L is already on closed); closed = [F,T,L,S,K,E,B,A]
10. open = [C,D]; closed = [M,F,T,L,S,K,E,B,A]
11. open = [G,H,D]; closed = [C,M,F,T,L,S,K,E,B,A]

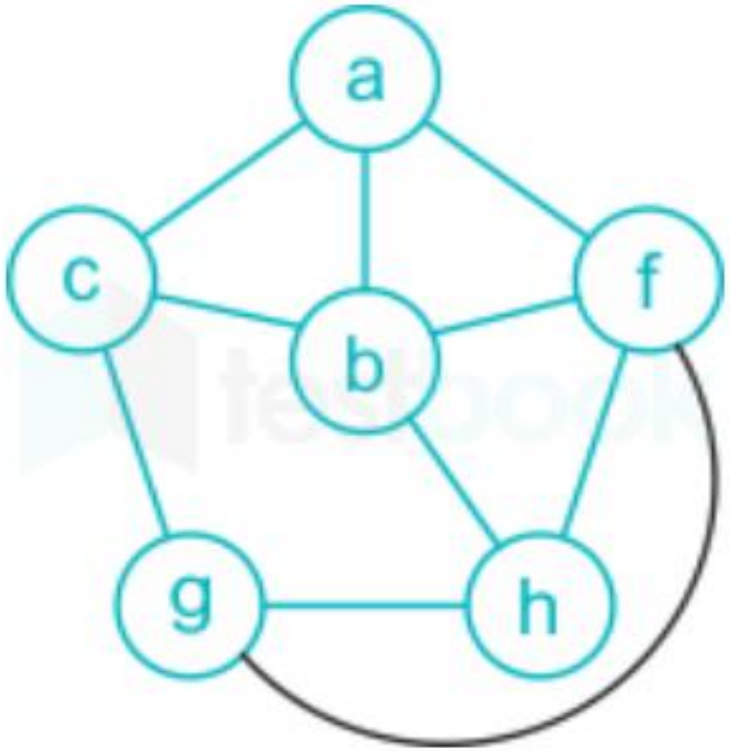
Depth-First Search (DFS)



Practice Questions

Q: Consider the following graph:
For the graph; the following sequences of Depth-First Search (DFS) are given
(A) abcghf
(B) abcfchg
(C) abfhgc
(D) afghbc
Which of the following is correct?

1. A, B and D only
2. A, B, C and D only
3. B, C and D only
4. A, C and D only



Practice Questions

Q: Consider the following graph:

For the graph; the following sequences of Depth-First Search (DFS) are given

(A) abeghf

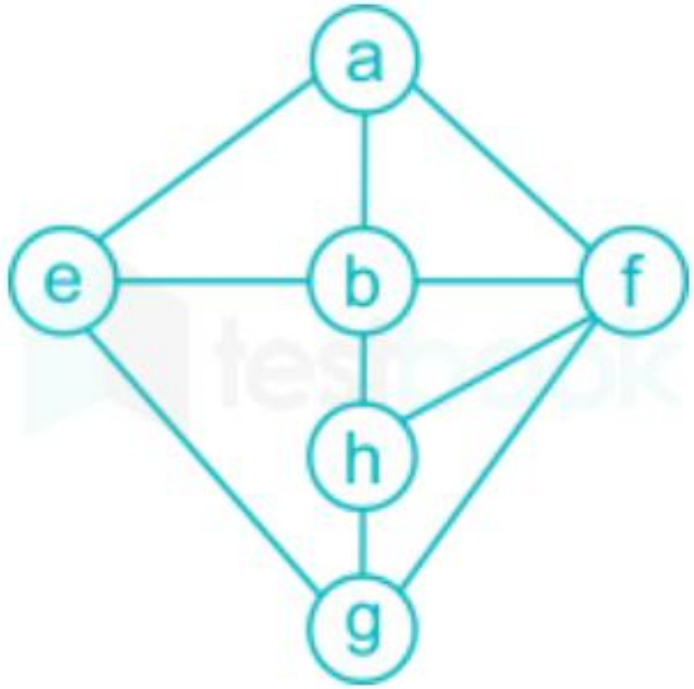
(B) abfehg

(C) abfhge

(D) afghbe

Which of the following is correct?

1. A, B and D only
2. A, C and D only
3. A, and D only
4. B, C and D only



Limitations of DFS and BFS

- **DFS:**
  - Can get stuck in deep paths without progress.
  - High memory usage.
- **BFS:**
  - Explores all paths at the same depth, leading to inefficiency.

- **Solution:**

  Use a smarter algorithm like Branch and Bound.

To address inefficiencies, we incorporate heuristics—rules of thumb or guiding principles. These help prioritize paths that seem more promising, saving time and resources.

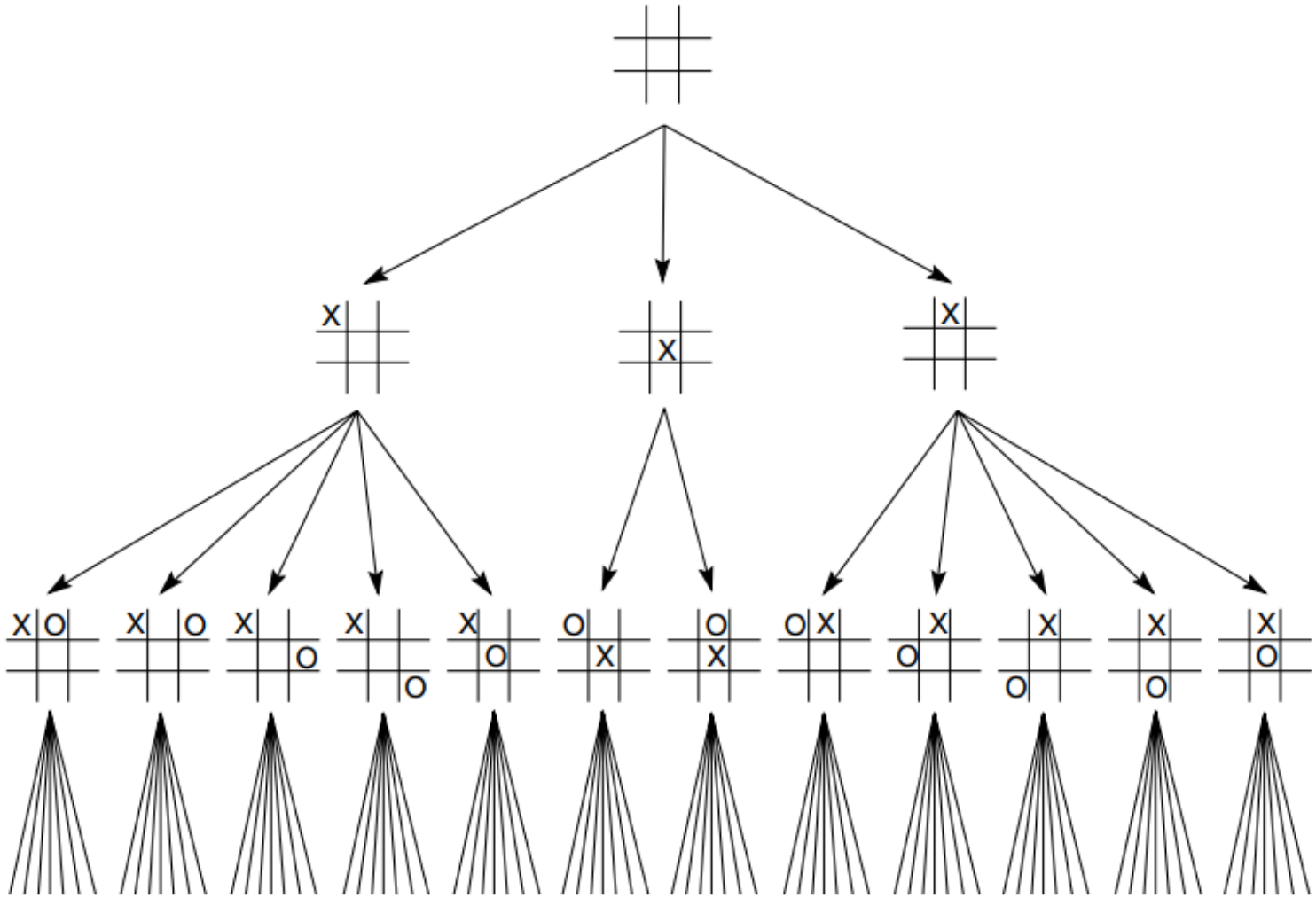
Heuristic Search Strategies

1. **Best-First Search / Algorithm A**
2. **Algorithm A***
3. **Iterative Deepening A* *(IDA)***
4. **Small Memory A* *(SMA)***

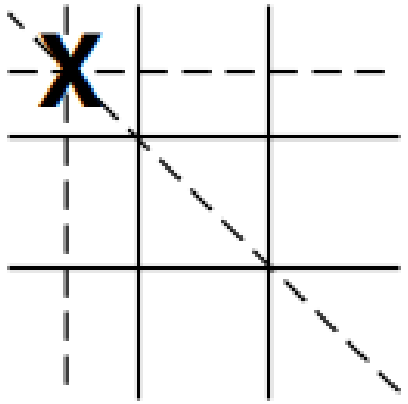
Heuristics

Heuristic is a function.

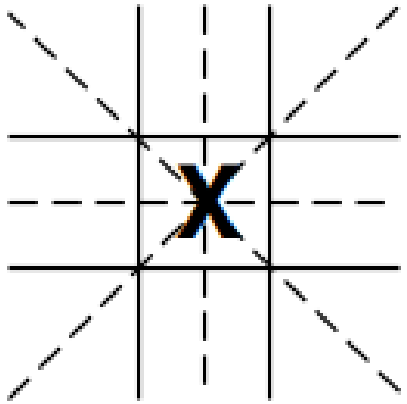
First three levels of the tic-tac-toe state space reduced by symmetry



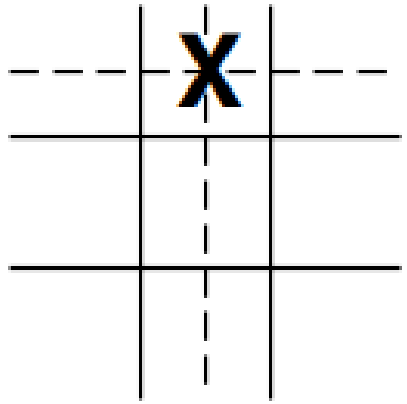
The most wins heuristic applied to the first children in tic-tac-toe



Three wins through a corner square

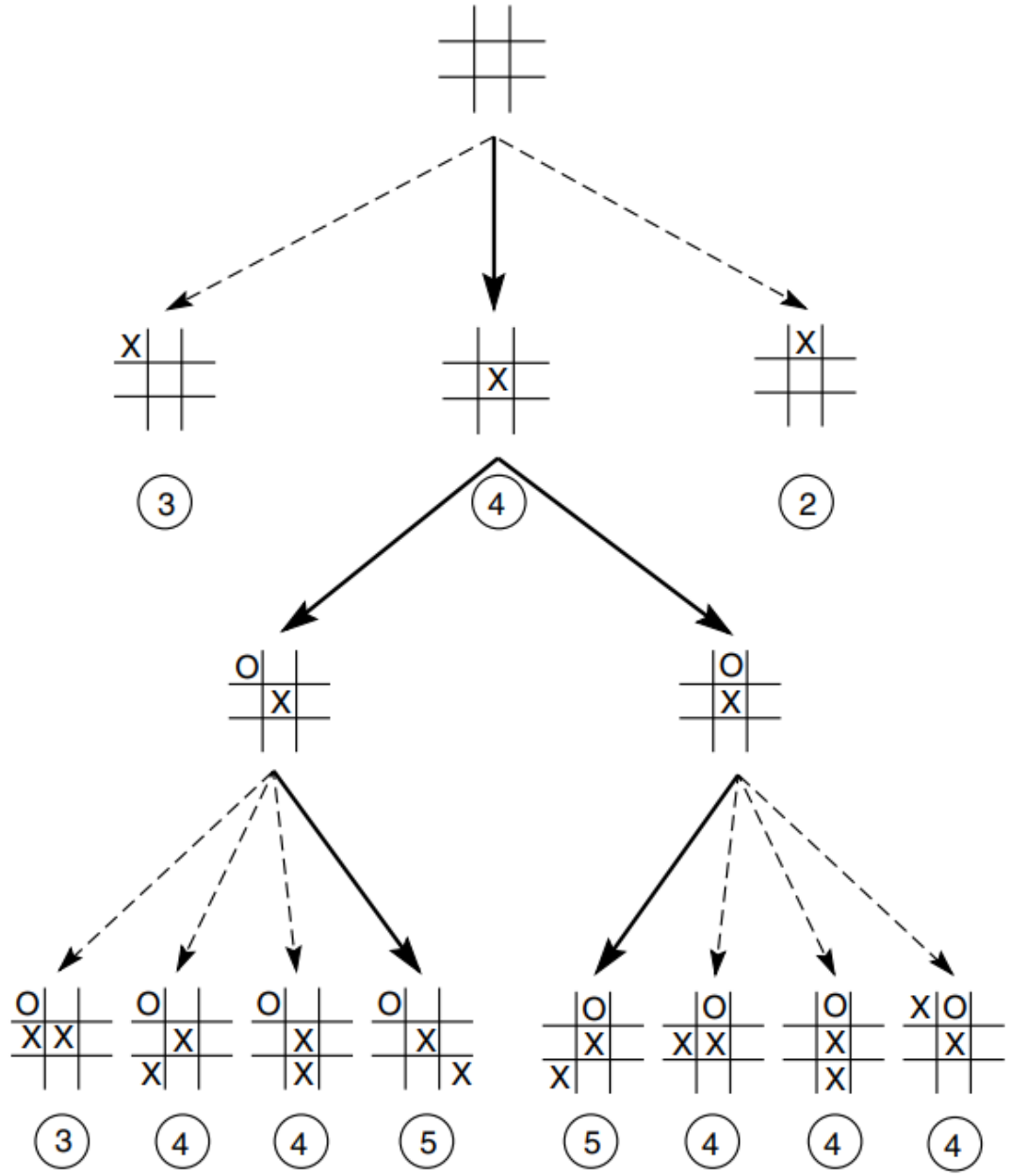


Four wins through the center square



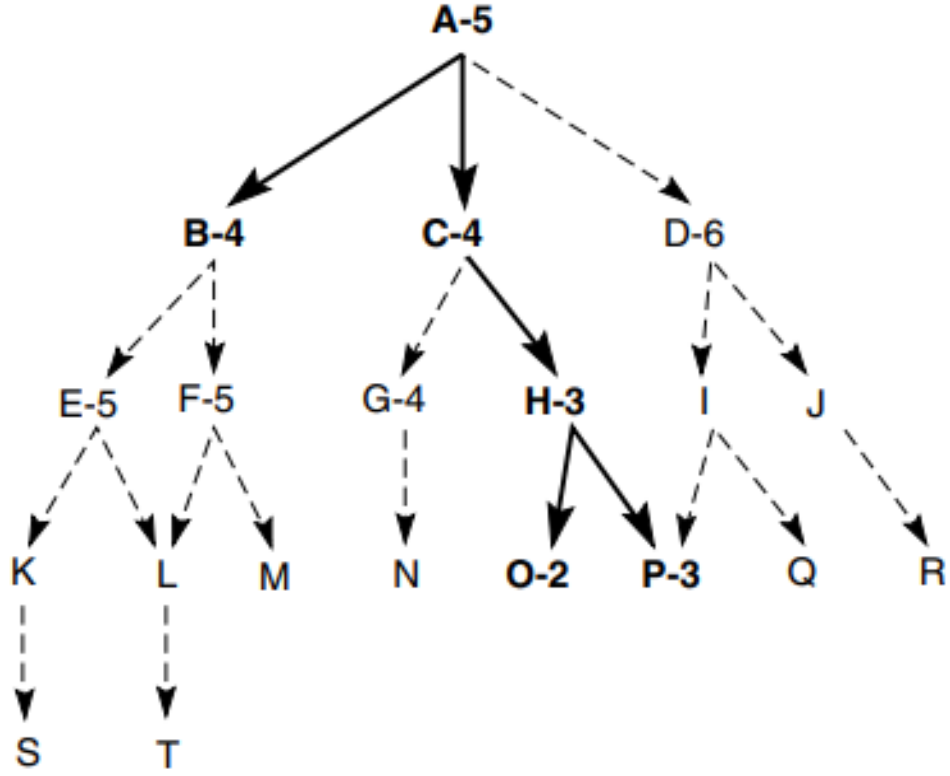
Two wins through a side square

Heuristically reduced state space for tic-tac-toe.



Best-First Search / Algorithm A

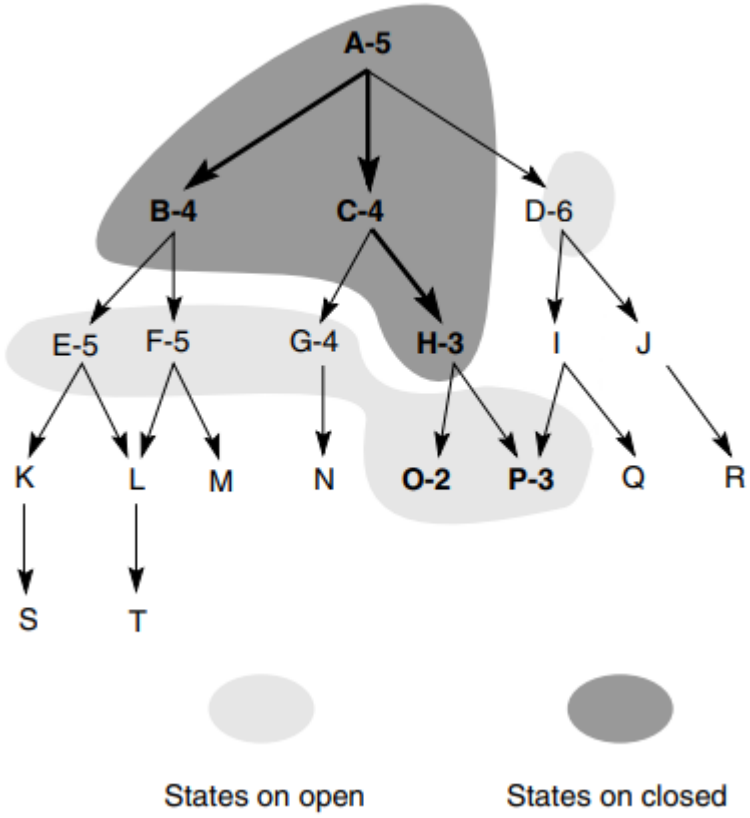
- **Heuristic-based** search algorithm
- Uses a **priority queue** for node selection
- **Expands most promising** nodes first
- More **efficient than uninformed search**
- **Minimizes unnecessary** exploration
- **Recovers from errors** via backtracking
- **Uses open & closed lists** to track states
- **Balances breadth & depth** in search



Heuristic search of a hypothetical state space.

Best-First Search / Algorithm A

1. open = [A5]; closed = [ ]
2. evaluate A5; open = [B4,C4,D6]; closed = [A5]
3. evaluate B4; open = [C4,E5,F5,D6]; closed = [B4,A5]
4. evaluate C4; open = [H3,G4,E5,F5,D6]; closed = [C4,B4,A5]
5. evaluate H3; open = [O2,P3,G4,E5,F5,D6]; closed = [H3,C4,B4,A5]
6. evaluate O2; open = [P3,G4,E5,F5,D6]; closed = [O2,H3,C4,B4,A5]
7. evaluate P3; the solution is found!



Heuristic search of a hypothetical state space with open and closed states highlighted.

Best-First Search / Algorithm A

- Algorithm A does **not search the entire state space**, avoiding unnecessary expansions.
- The **priority queue ensures the best heuristic states** are always considered first.
- The algorithm **recovers from incorrect heuristic choices** (e.g., exploring O-2 before P-3).
- The **open list maintains all frontier states**, allowing backtracking if necessary.

Best-First Search / Algorithm A Advantages

- **More efficient** than uninformed search strategies.
- **Adapts to heuristic errors** by revisiting promising alternative paths.
- **Can operate on varying depth levels**, unlike depth-first or breadth-first searches.

Best-First Search/ Algorithm A Limitations

- **Heuristic dependency**: Poor heuristic choices can lead to inefficiency.
- **Not always optimal**: Unless combined with path-cost strategies like A*.

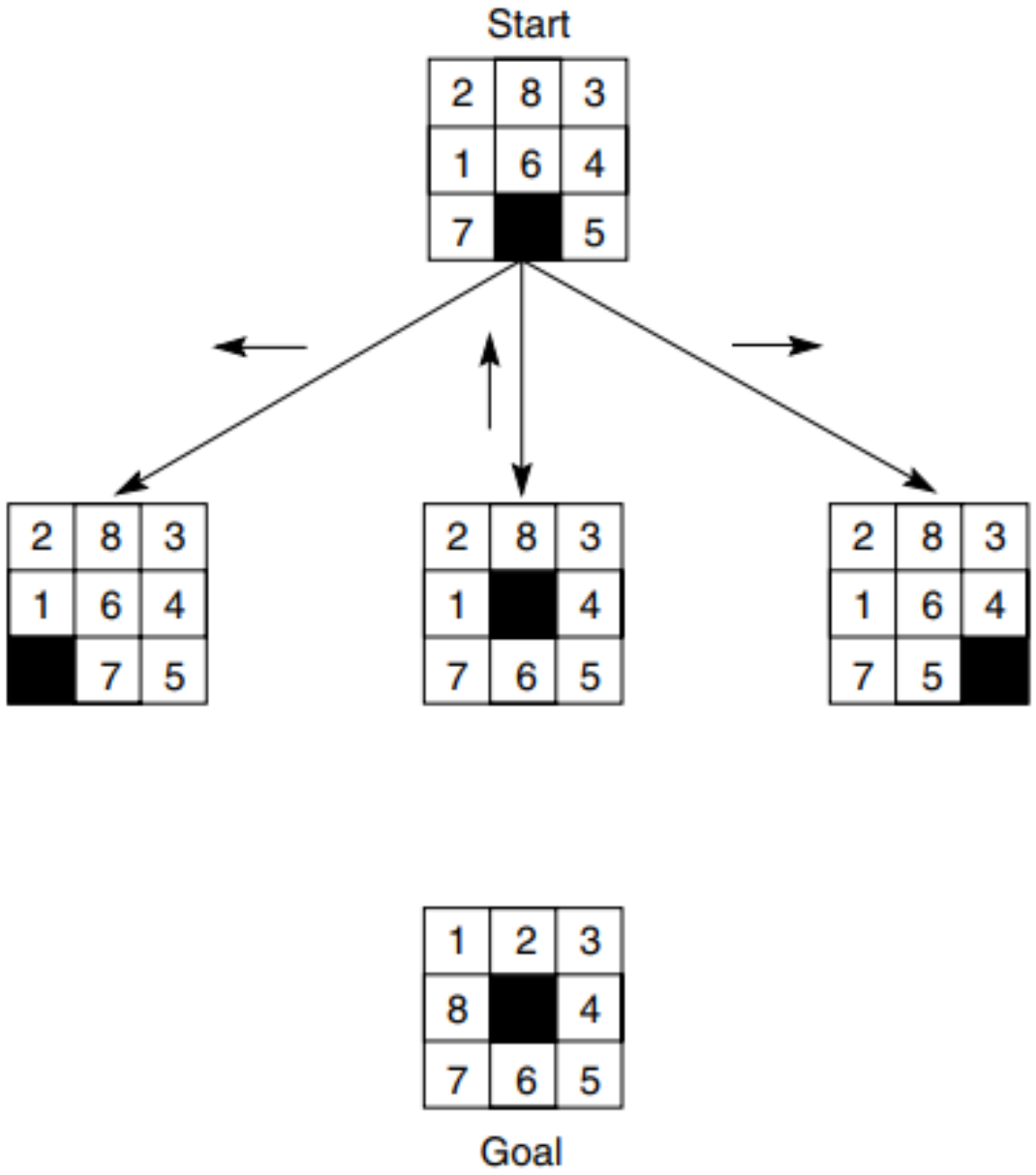
Heuristic Search in the 8-Puzzle

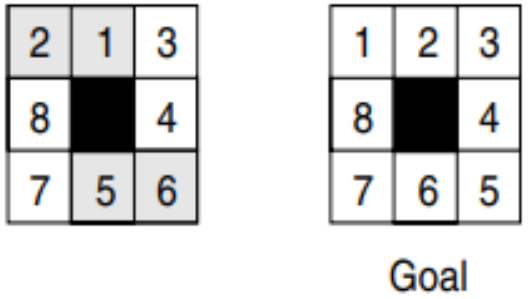
8-Puzzle Problem using Best-First Search / A Algorithm

- **8-puzzle:** Move tiles on a **3x3 grid** to reach the goal state.
- **Heuristic search:** Improves efficiency in solving the puzzle.
- **Evaluation functions:** Estimate the cost to reach the goal.
- **Guides search:** Helps prioritize better board configurations.

Understanding the Initial Problem

- The **start state** of the puzzle is given, along with the **goal state**.
- Three possible moves are depicted, showing the first step of the search process.
- Each move results in a different configuration of the board.





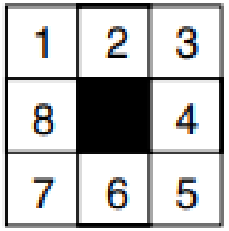
An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.

Evaluating Heuristic Functions

Three heuristic functions are applied to evaluate the effectiveness of different search strategies:

1. **Tiles Out of Place (h_1)**
   - Counts the number of tiles not in their correct positions.
   - Provides a rough estimate but does not consider movement cost.
   - Example: The first state has 5 misplaced tiles.
2. **Sum of Manhattan Distances (h_2)**
   - Sums the distances each tile must be moved to reach its correct position.
   - A more accurate heuristic as it accounts for movement steps.
   - Example: The first state has a total movement cost of 6.
3. **Tile Reversal Heuristic (h_3)**
   - Identifies tiles that must be swapped directly.
   - Less effective since it does not distinguish among states where no direct reversals exist.
   - Example: All states in Figure have a reversal score of 0.

2 8 3 1 6 4 ■ 7 5	**5**	**6**	**0**
2 8 3 1 ■ 4 7 6 5	**3**	**4**	**0**
2 8 3 1 6 4 7 5 ■	**5**	**6**	**0**
	Tiles out of place	Sum of distances out of place	2 x the number of direct tile reversals



Goal

Three heuristics applied to states in the 8-puzzle.

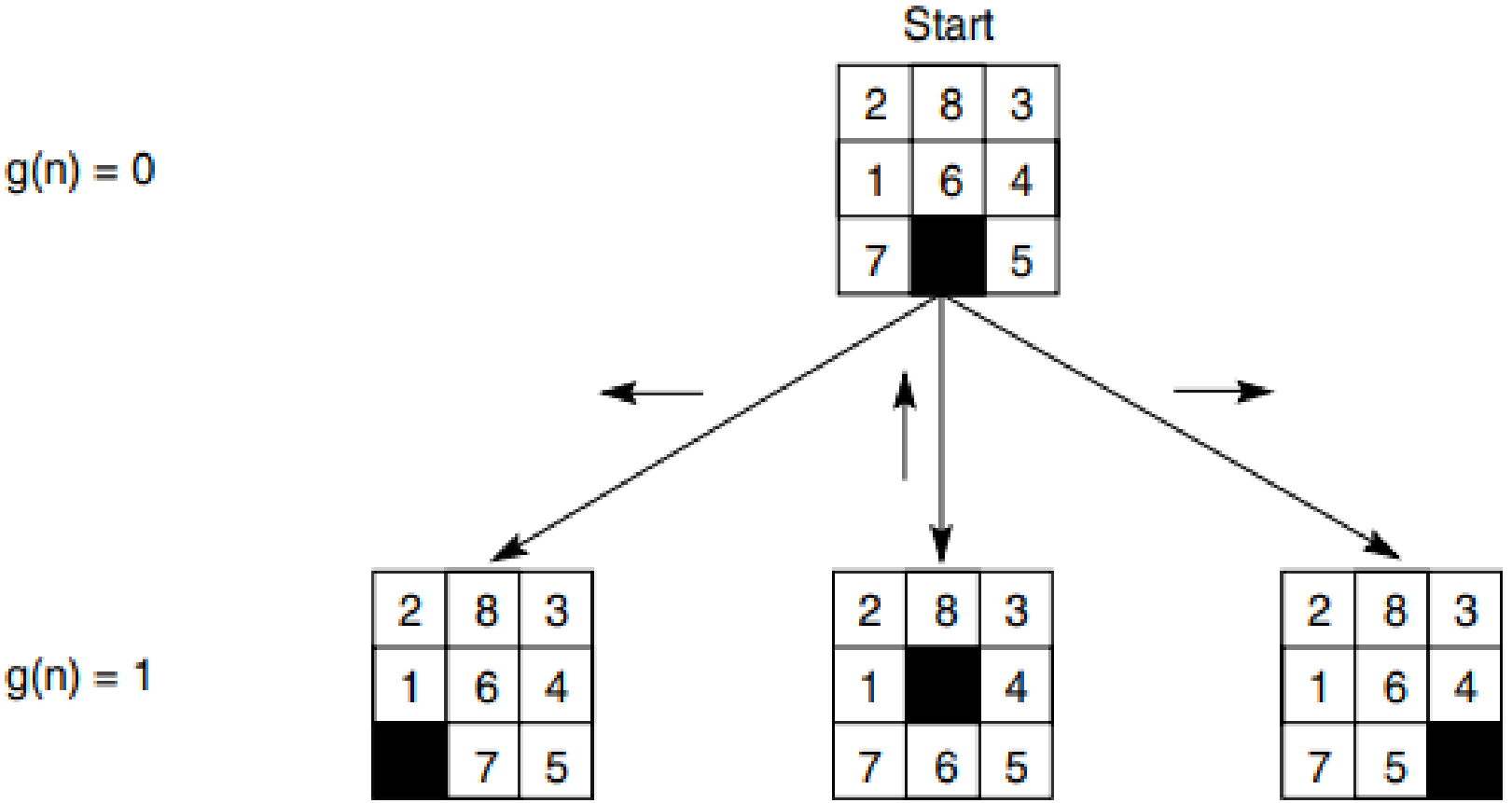
The Heuristic Cost Function f(n) in Search

- The evaluation function **f(n) = h(n)** is applied in Best First Search where,

  h(n) = Heuristic estimate of distance to goal.

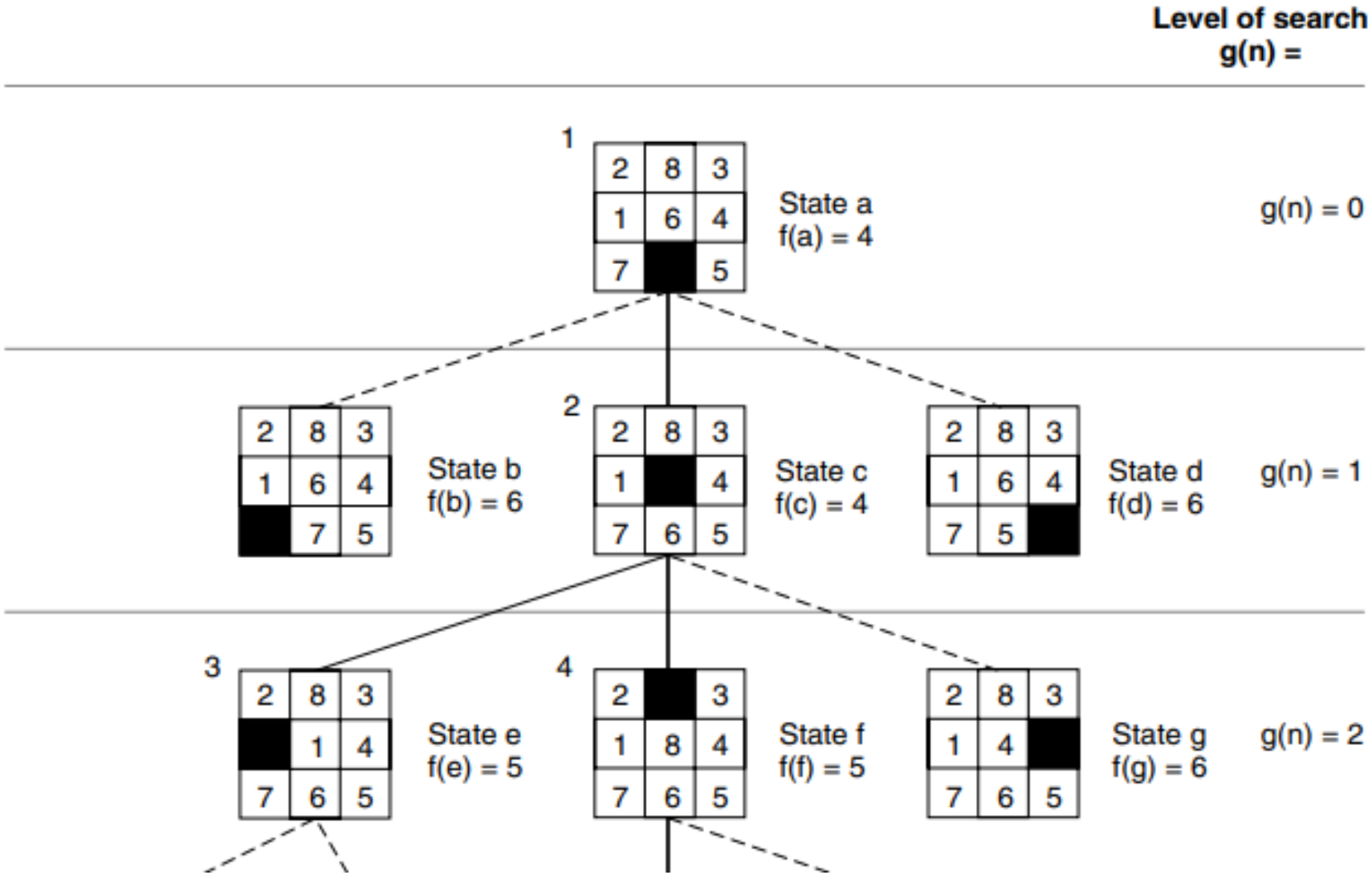
- The heuristic cost function guides the search, preventing infinite loops and improving efficiency.
- The lowest f-value (4) is chosen for expansion in this step.

- Let us choose ***Sum of Manhattan Distances h2*** as heuristic function

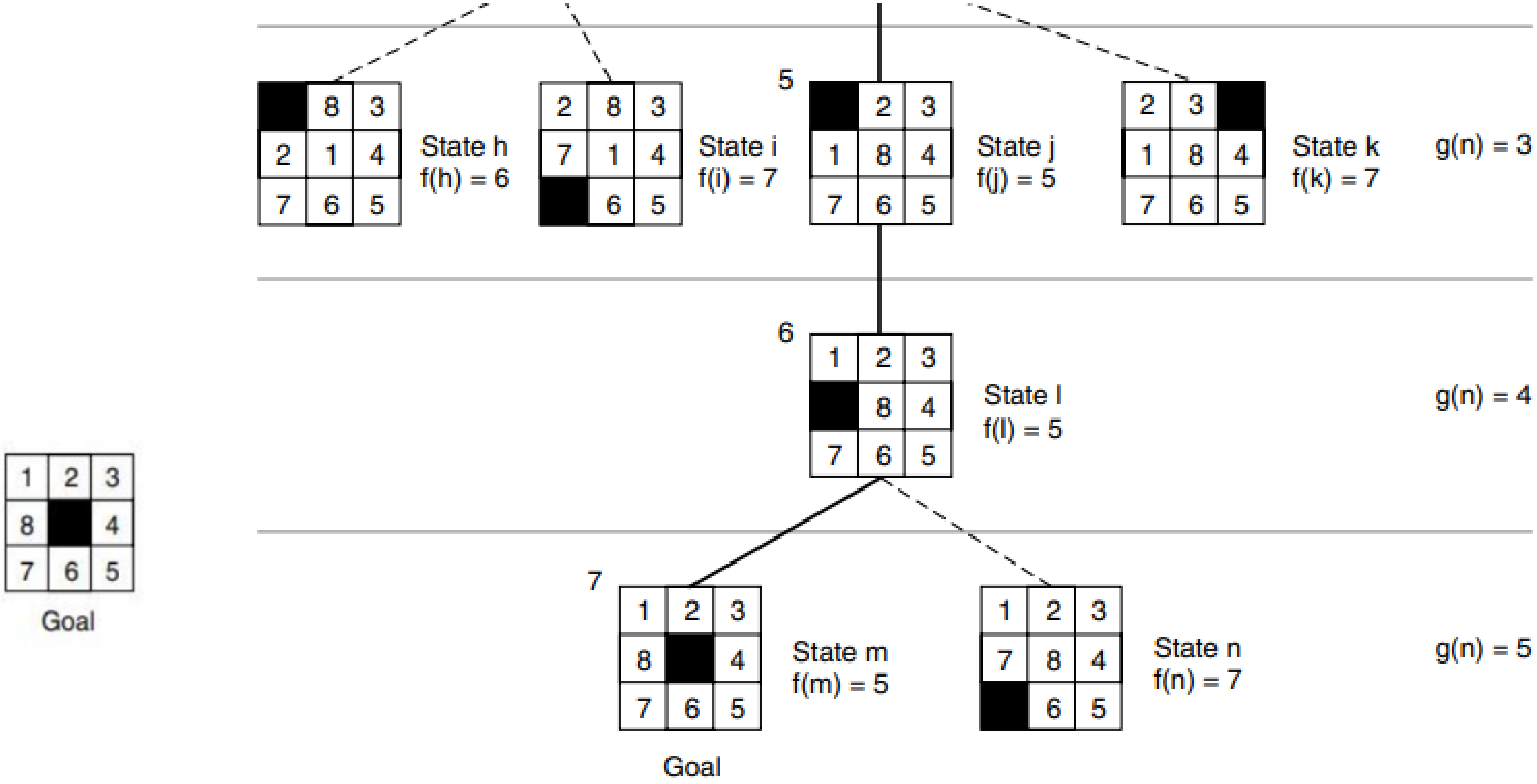


Values of f(n) for each state, **6** **4** **6**

The heuristic f applied to states in the 8-puzzle.



State space generated in heuristic search of the 8-puzzle graph.

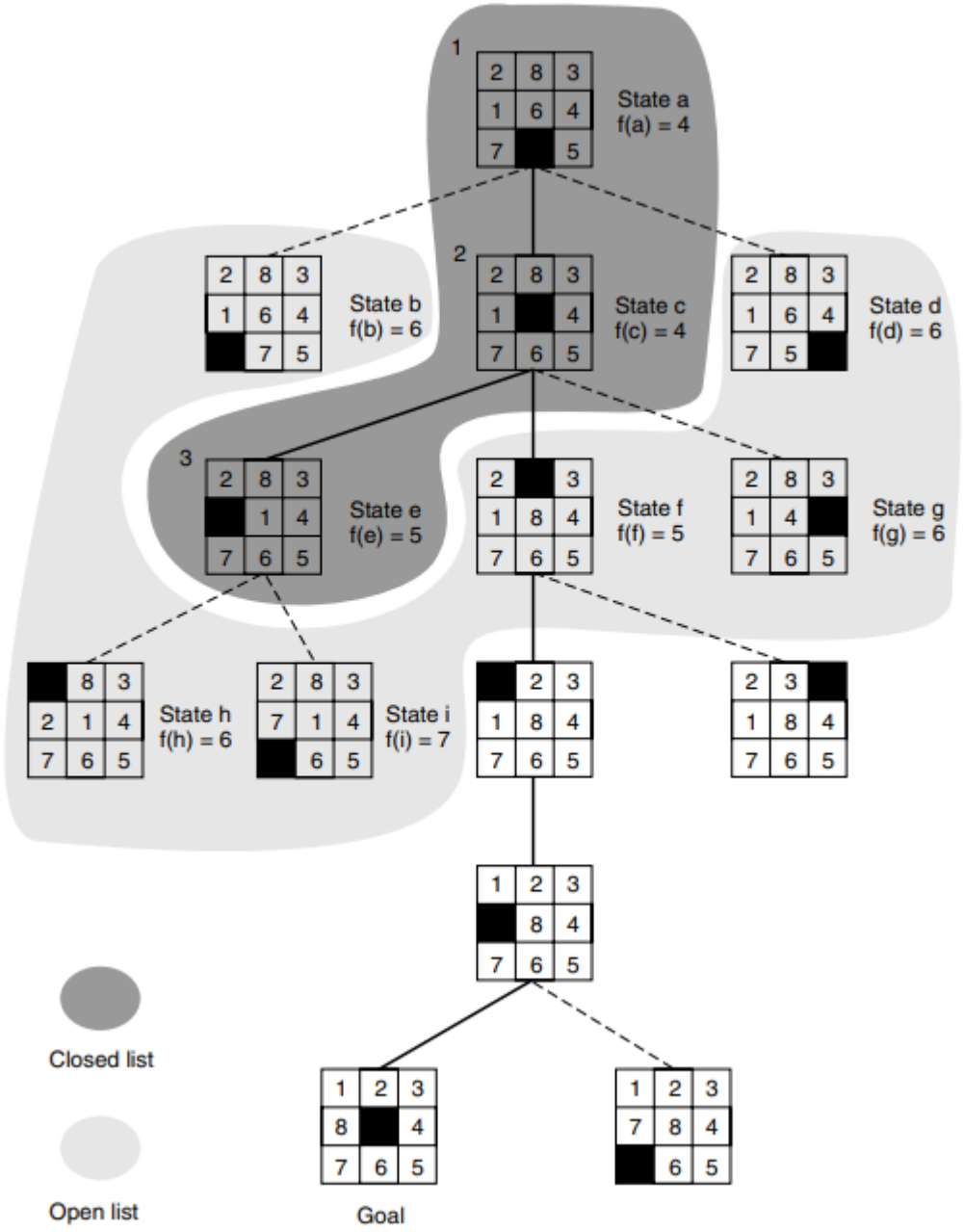


State space generated in heuristic search of the 8-puzzle graph.

The successive stages of open and closed that generate this graph are:

1. open = [a4];
   closed = [ ]
2. open = [c4, b6, d6];
   closed = [a4]
3. open = [e5, f5, b6, d6, g6];
   closed = [a4, c4]
4. open = [f5, h6, b6, d6, g6, i7];
   closed = [a4, c4, e5]
5. open = [j5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5]
6. open = [l5, h6, b6, d6, g6, k7, i7];
   closed = [a4, c4, e5, f5, j5]
7. open = [m5, h6, b6, d6, g6, n7, k7, i7];
   closed = [a4, c4, e5, f5, j5, l5]
8. success, m = goal!

- **Manhattan distance heuristic (h_2) is more effective** than simple misplaced tile counting.
- **Best-first search prioritizes states with lower f-values**, avoiding unnecessary explorations.
- **The g(n) component prevents infinite loops** by forcing backtracking if heuristics lead down incorrect paths.
- **Choosing a good heuristic is crucial** in optimizing search algorithms for problems like the 8-puzzle.



Best-First Search Using Heuristics

- The algorithm explores states based on their heuristic values.
- It maintains **open and closed lists** to track explored and unexplored states.
- If two states have the same heuristic value, the one **closest to the root** is preferred.
- The evaluation function helps avoid inefficient paths by balancing depth and heuristic cost.

*A** Algorithm

A* (**A-Star**) is a heuristic search algorithm that **finds the shortest path** from a start state to a goal state by using the evaluation function:

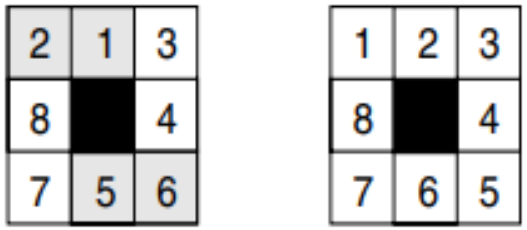
$$f(n) = g(n) + h(n)$$

Where,

f(n) = Total estimated cost of the path through

g(n) = Depth of the node (distance from start state).

h(n) = Heuristic estimate of distance to goal.

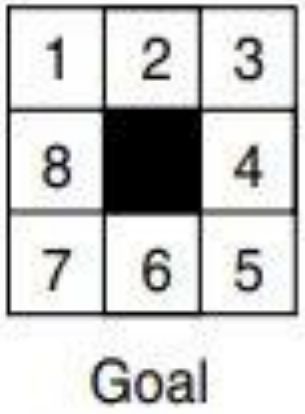


where:

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

g(n) = actual distance from n to the start state, and

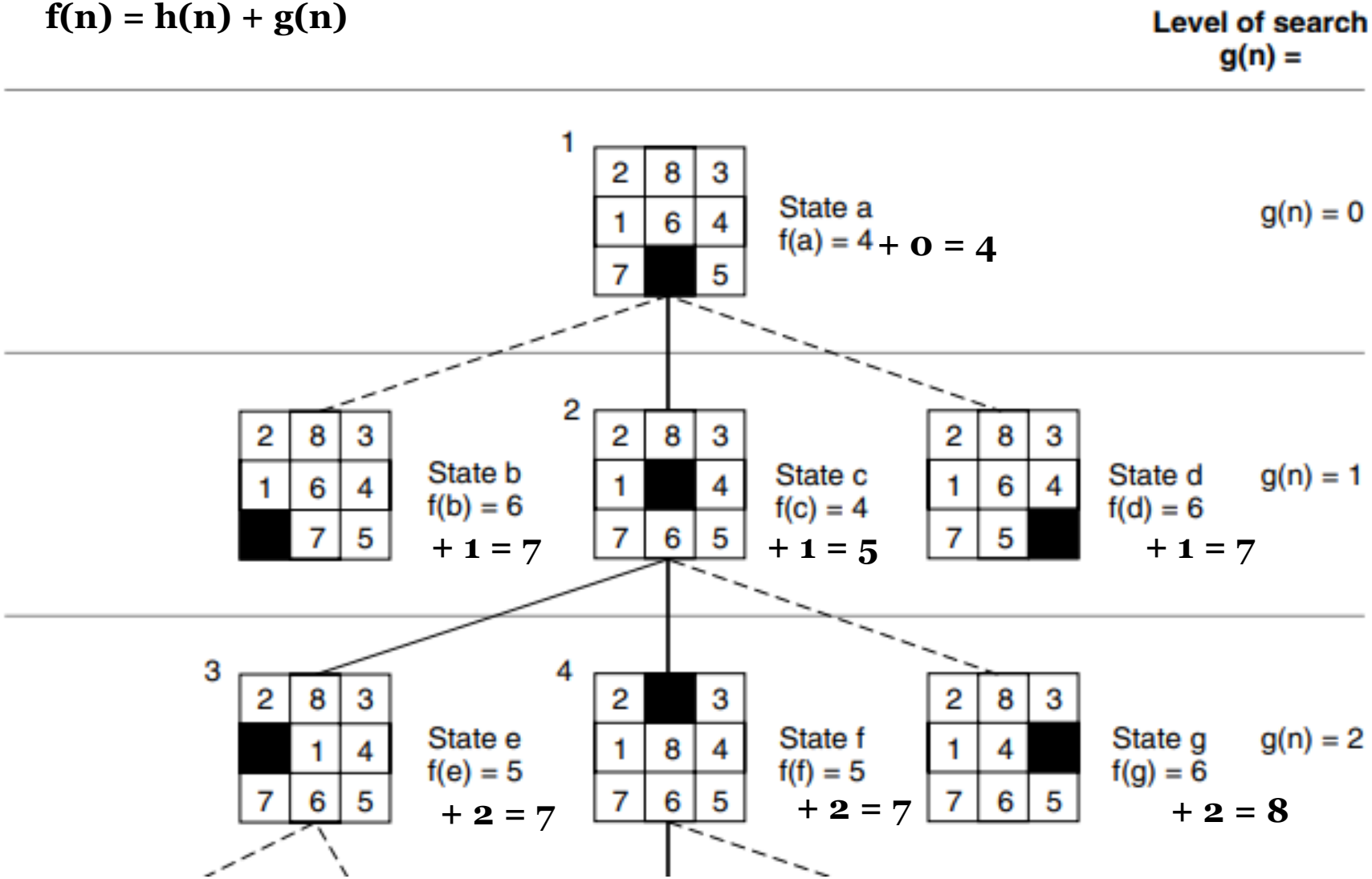
h(n) = number of tiles out of place.



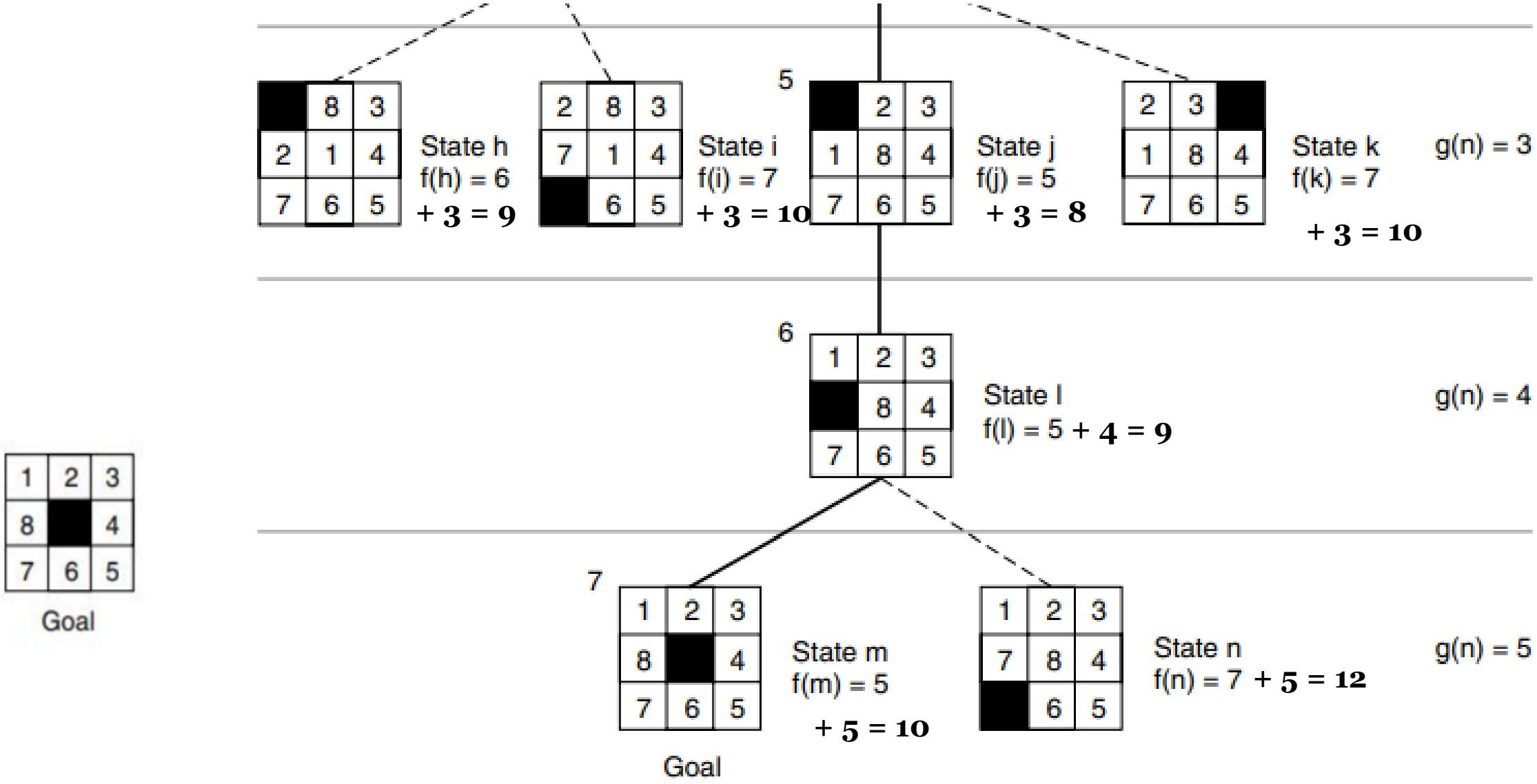
8-Puzzle Problem using A* Algorithm

- The A* algorithm **searches for the optimal solution** by prioritizing board configurations with the lowest **f(n)**.
- A* evaluates multiple moves, considering both the number of tiles out of place (**h(n)**) and the depth of the move (**g(n)**).

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



State space generated in heuristic search of the 8-puzzle graph.



State space generated in heuristic search of the 8-puzzle graph.

Evaluating Heuristic Behavior

- Heuristics help find solutions efficiently while minimizing cost.
- **Admissible heuristics** guarantee finding the shortest path to the goal.
- **Informedness** measures how well a heuristic estimates the shortest path.
- **Monotonicity** ensures that once a state is found at a certain cost, it won't be found later at a lower cost.

Admissibility Measures

- **A search algorithm is admissible** if it always finds the optimal path.
- **Breadth-first search** is admissible but inefficient.
- *A** Algorithm (A-Star): Uses **f(n) = g(n) + h(n)** for better efficiency.
- If $h(n) \leq h(n)^*$ for all states, A* is guaranteed to find the shortest path.

Monotonicity in Heuristics

- Ensures that heuristic values do not decrease along a path.
- Prevents redundant state evaluations and speeds up the search.
- A monotonic heuristic guarantees admissibility.

Comparing Heuristic Informedness

- More informed heuristics evaluate fewer states while still finding the optimal path.
- Example: **Manhattan distance heuristic is more informed** than simply counting misplaced tiles.
- A more informed heuristic reduces search space but may increase computation time.

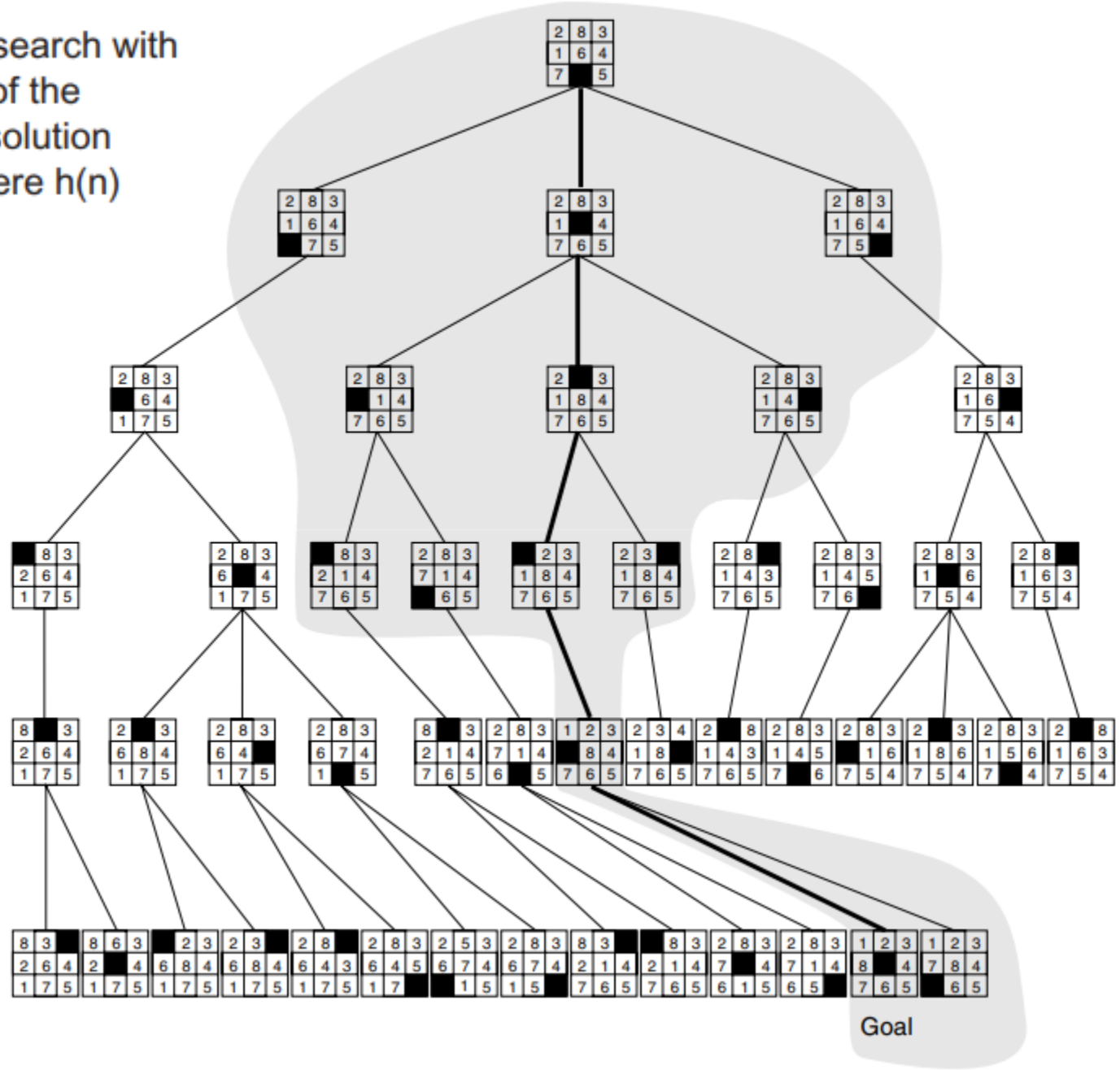
Trade-offs in Heuristic Search

- Some problems require balancing between heuristic accuracy and computational efficiency.
- **Example:** Chess programs may use either simple heuristics with deep searches or sophisticated heuristics with fewer expanded states.
- Finding the optimal trade-off remains an open research question.

Quick Recap

- **A*** *is both admissible and optimal* if heuristics are properly bounded.
- **More informed heuristics lead to fewer expanded nodes** in search.
- **Monotonic heuristics ensure efficient and correct solutions** in search algorithms.
- **Trade-offs between heuristic complexity and search depth** must be optimized based on the problem.

Comparison of state space searched using heuristic search with space searched by breadth-first search. The portion of the graph searched heuristically is shaded. The optimal solution path is in bold. Heuristic used is f(n) = g(n) + h(n) where h(n) is tiles out of place.



Main Difference Between Best-First Search and A* Algorithm

Feature	Best-First Search (BFS)	A* Algorithm
Evaluation Function	Uses only heuristic value: f(n) = h(n)	Uses both cost and heuristic: f(n) = g(n) + h(n)
Search Strategy	Expands the node with the lowest heuristic value (h(n))	Expands the node with the lowest estimated total cost (f(n))
Optimality	Not guaranteed to find the shortest path	Guaranteed to find the optimal path (if h(n) is admissible)
Completeness	Not necessarily complete (may get stuck in loops if heuristics are poor)	Complete (always finds a solution if one exists)
Admissibility	Not necessarily admissible (may not find the shortest path)	Admissible if $h(n) \leq h^*(n)$
Efficiency	Faster but can be misleading if heuristic is inaccurate	More efficient because it considers both cost and heuristic
Example in Context of 8-Puzzle	Could get trapped exploring non-optimal states (e.g., picking a misleading move)	Finds the shortest number of moves to the goal by balancing g(n) (cost) and h(n) (heuristic)

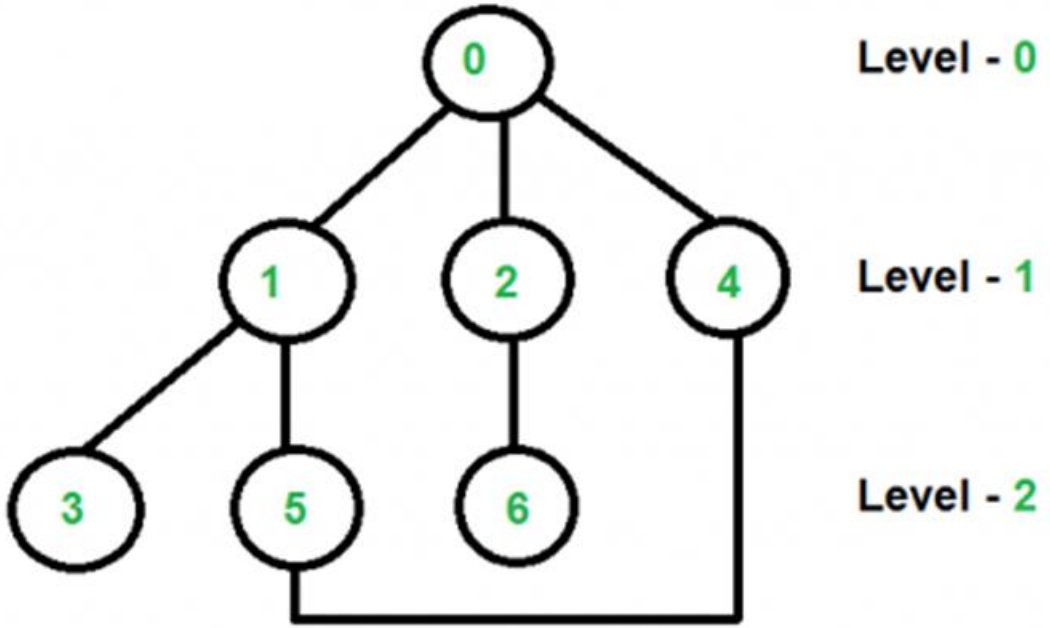
Depth First Search with Depth Bound

- Limits how deep the search goes before forcing failure.
- Simulates **a breadth-first sweep at each depth level**.
- Useful when the **solution is known to exist within a certain depth**.

Iterative Deepening Depth-First Search (DFS-ID)

- Runs **DFS with increasing depth limits (1, 2, 3, ...)**.
- **Each iteration performs a full DFS** to the current depth.
- No information from previous iterations is retained.

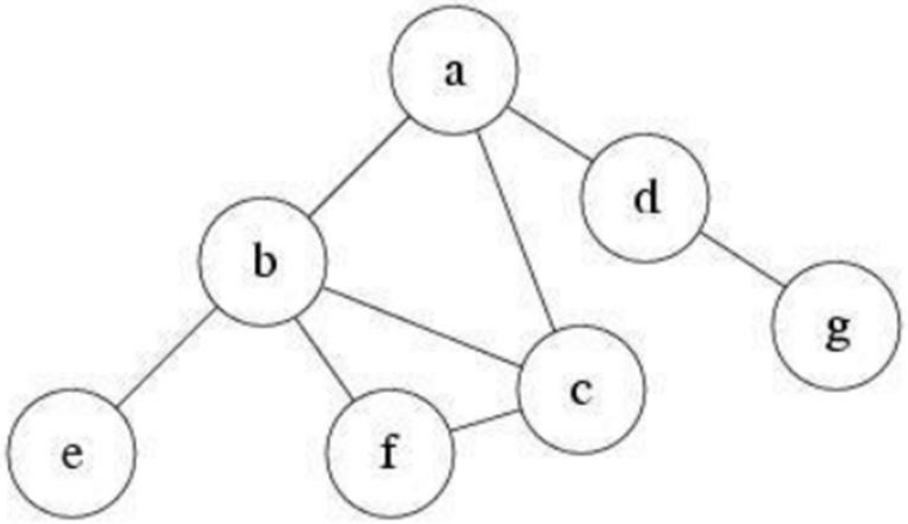
Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

Practice Problem

Below is the graph we will traverse. Same as DFS, we use the stack data structure S2 to record the node we've explored. Suppose the source node is node a. Assume also that the 'solution' node is node g.

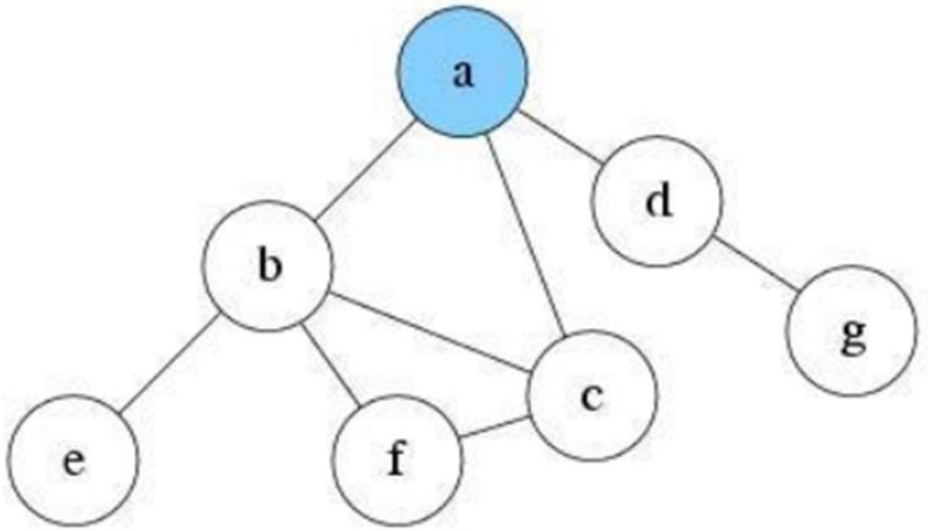


S2:

Practice Problem

At first, the depth limit L = 0. There is only the node a reachable. So push it into S1 and mark as visited.

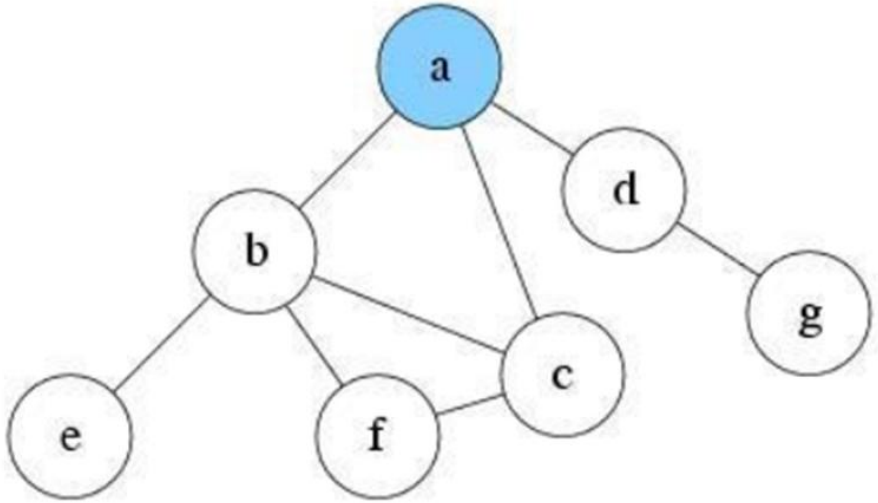
Current level is 0.



S2: a

Practice Problem

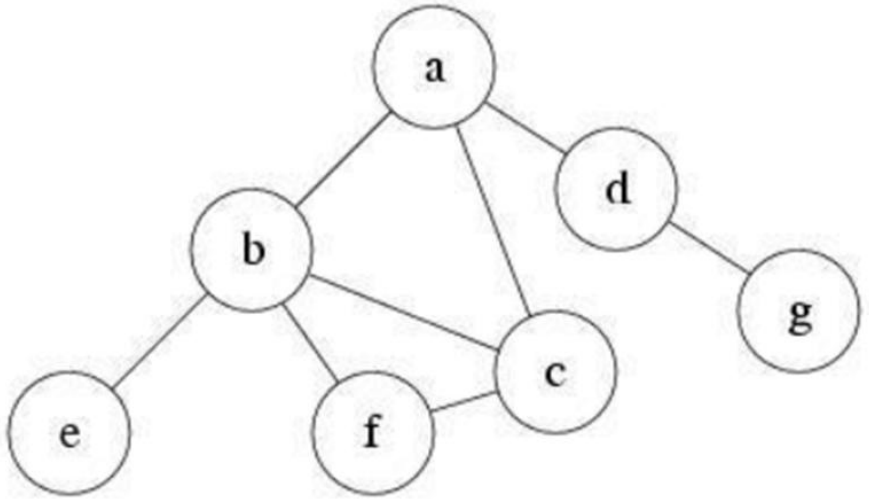
Explore a. Since current level is already the max depth L. No new reachable node will be found. Pop a from S2.



S2:

Practice Problem

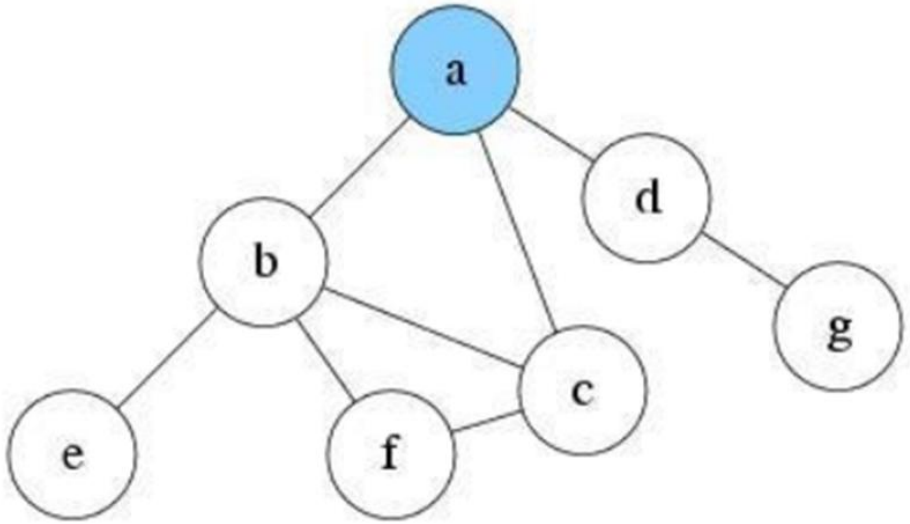
S2 is empty now. Since solution is not found and max depth is not reached, increment depth limit L to 1 and start the search from the beginning.



S2:

Practice Problem

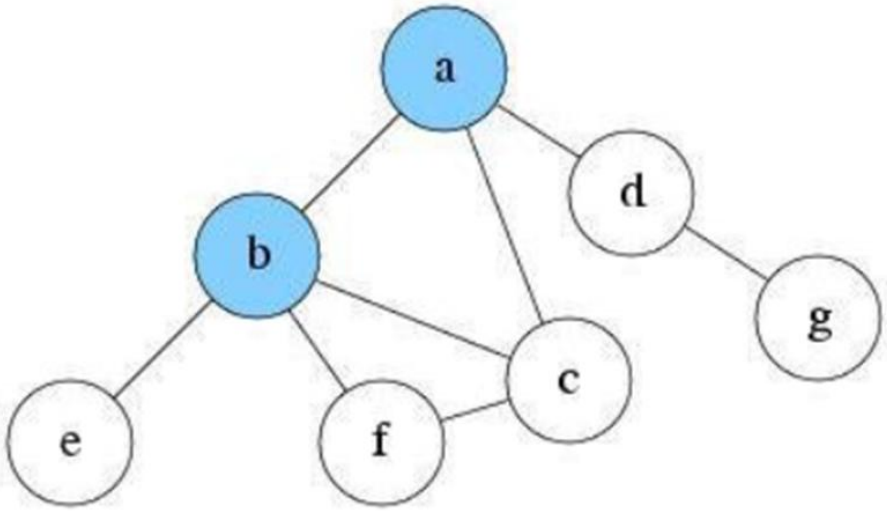
At first, only node a is reachable. So push it into S2 and mark as visited. Current level is 0.



S2: a

Practice Problem

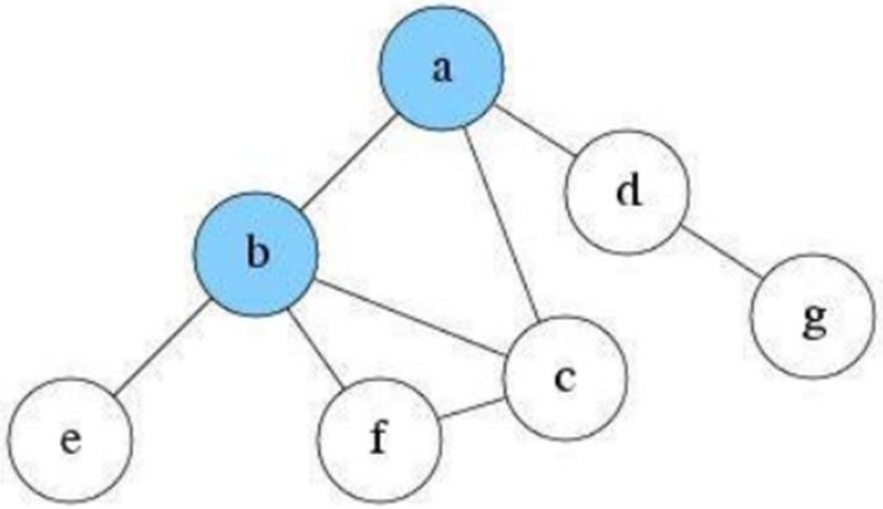
After exploring a, now there are three nodes reachable: b, c and d. Suppose we pick node b to explore first. Push b into S2 and mark it as visited. Current level is 1.



S2: b, a

Practice Problem

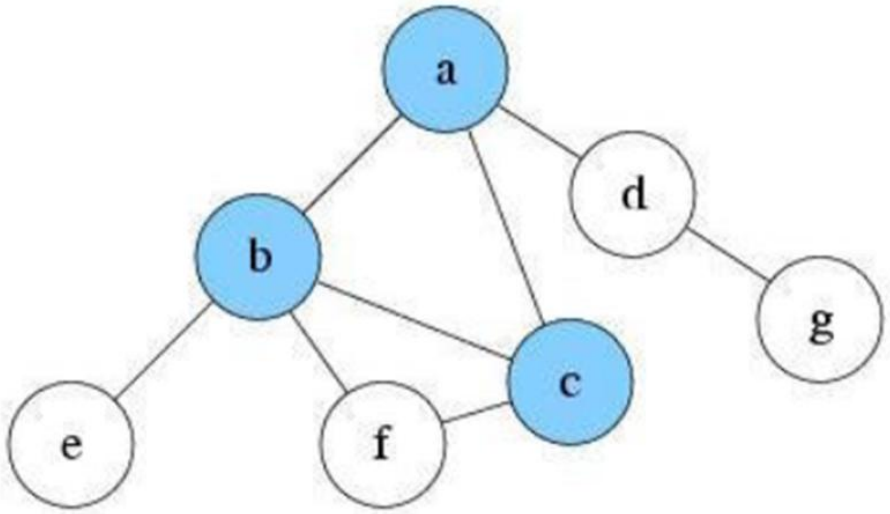
Since current level is already the limited depth L. Node b will be treated as having no successor. So there is nothing reachable. Pop b from S2. Current level is 0.



S2: a

Practice Problem

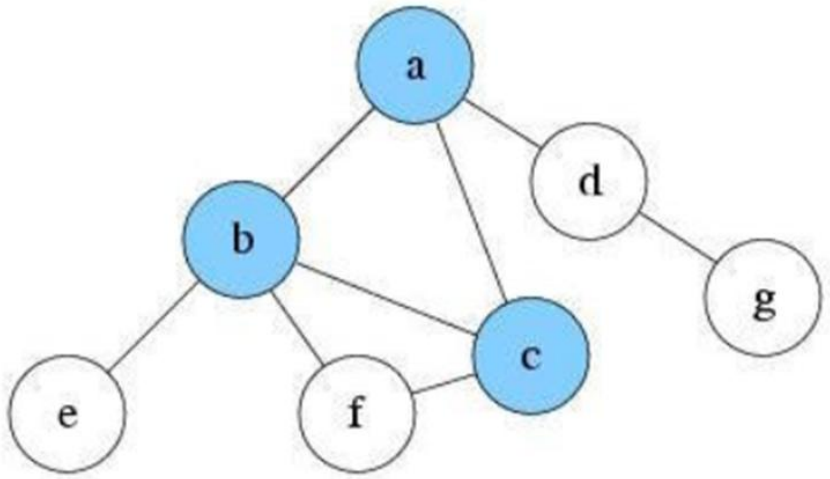
Explore a again. There are two unvisited nodes c and d reachable. Suppose we pick node c to explore first. Push c into S1 and mark it as visited. Current level is 1.



S2: c, a

Practice Problem

Since the current level is already the limited depth L, node c will be treated as having no successor. So there is nothing reachable. Pop c from S2. Current level is 0.

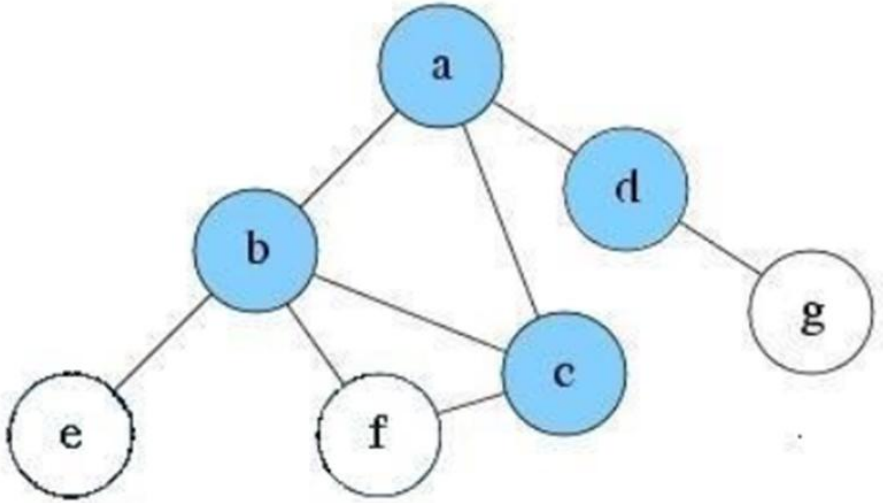


S2: a

Practice Problem

Explore a again. There is only one unvisited node d reachable. Push d into S2 and mark it as visited.

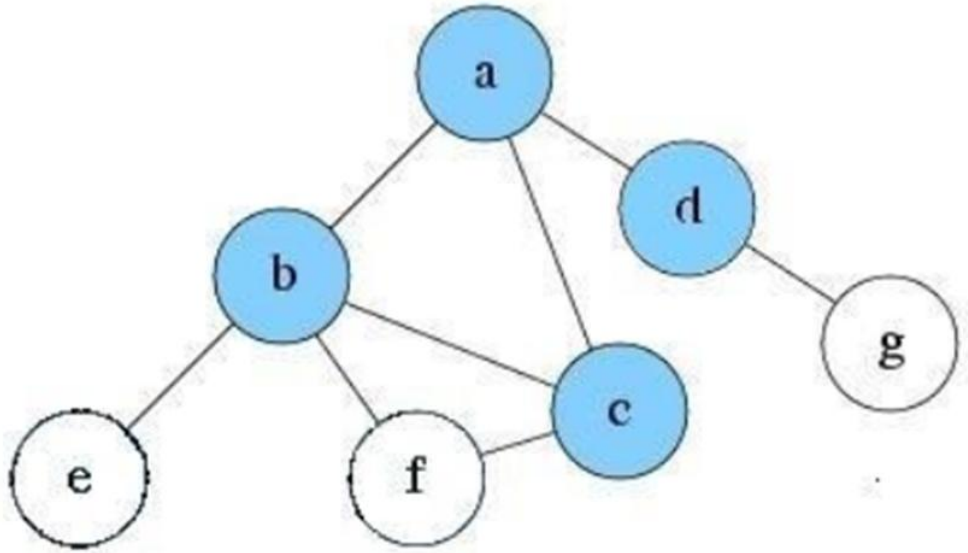
Current level is 1.



S2: d, a

Practice Problem

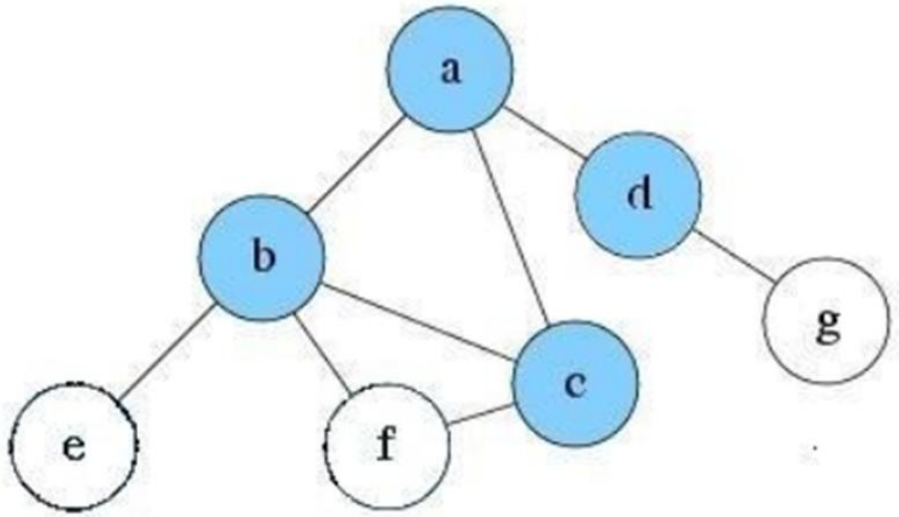
Explore d, and find no new node reachable. Pop d from S2. Current level is 0.



S2: a

Practice Problem

Explore a again. No new reachable node anymore. Pop a from S2.

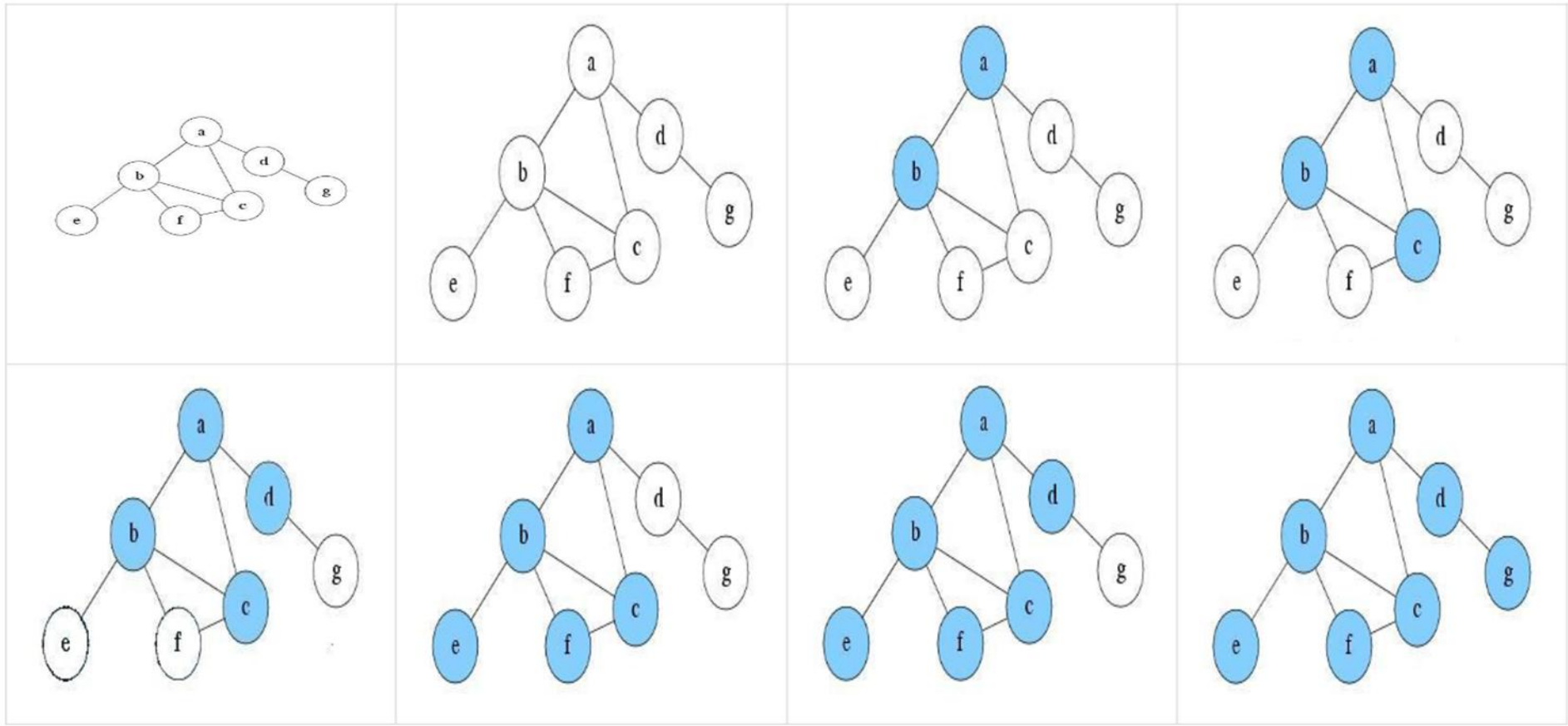


S2:

Practice Problem

S2 is empty now. Since a solution is not found and max depth is not reached, increment depth limit L to 2 and start the search from the beginning.

The progress is very similar to previous two, and is pictured below. The steps described above generalize:



Now we can see that, at depth limit 2, IDS already explored all the nodes reachable from a, and if the solution exists in the graph, then it has been found.

Why DFS-ID is Efficient Despite Repeating Work?

- The number of nodes at deeper levels **grows exponentially**.
- Even though shallower levels are explored multiple times, **most of the time is spent at the deepest level**.

Limitations

- **Exponential worst-case time complexity** (like DFS and BFS).
- Still an **uninformed search** method (does not use heuristics).

Iterative Deepening A* (IDA)

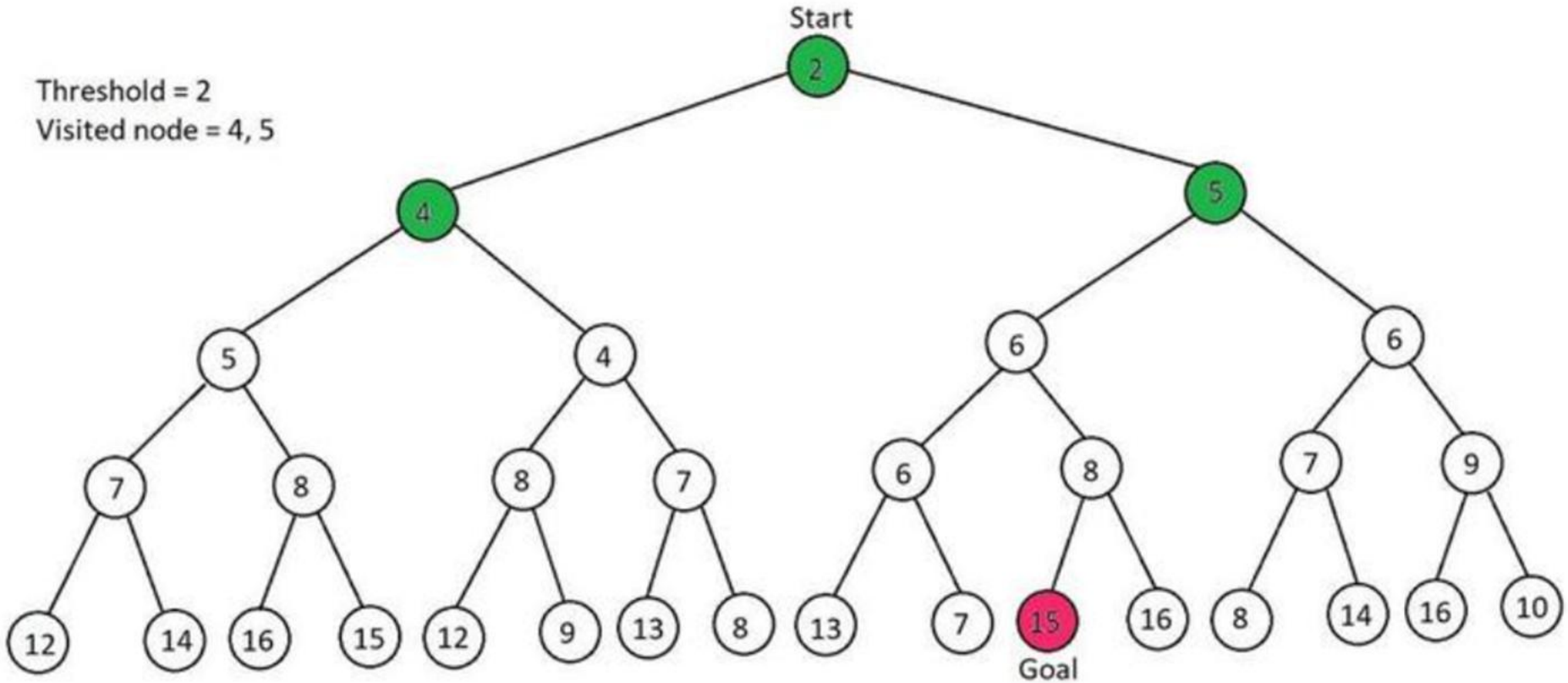
- **Combination of Iterative Deepening (ID) and A***:
  - Uses **depth-first search (DFS)** with iterative deepening.
  - Incorporates the **heuristic evaluation function** from A*.
- **Depth Limit Based on Cost**:
  - Instead of depth, the limit is set on the **total cost (f(n))**.
  - **f(n) = g(n) + h(n)**, where:
    - **g(n)** = cost from start to node nnn.
    - **h(n)** = heuristic estimate to the goal from nnn.
- **Incremental Cost Limit**:
  - Starts with a cost limit set to the heuristic of the start node.
  - Increases the cost limit iteratively to explore deeper paths.

Practice Problem

Example

In the below tree, the **f score** is written inside the nodes means the f score is already computed and the start node is **2** whereas the goal node is **15**. the explored node is colored green color.

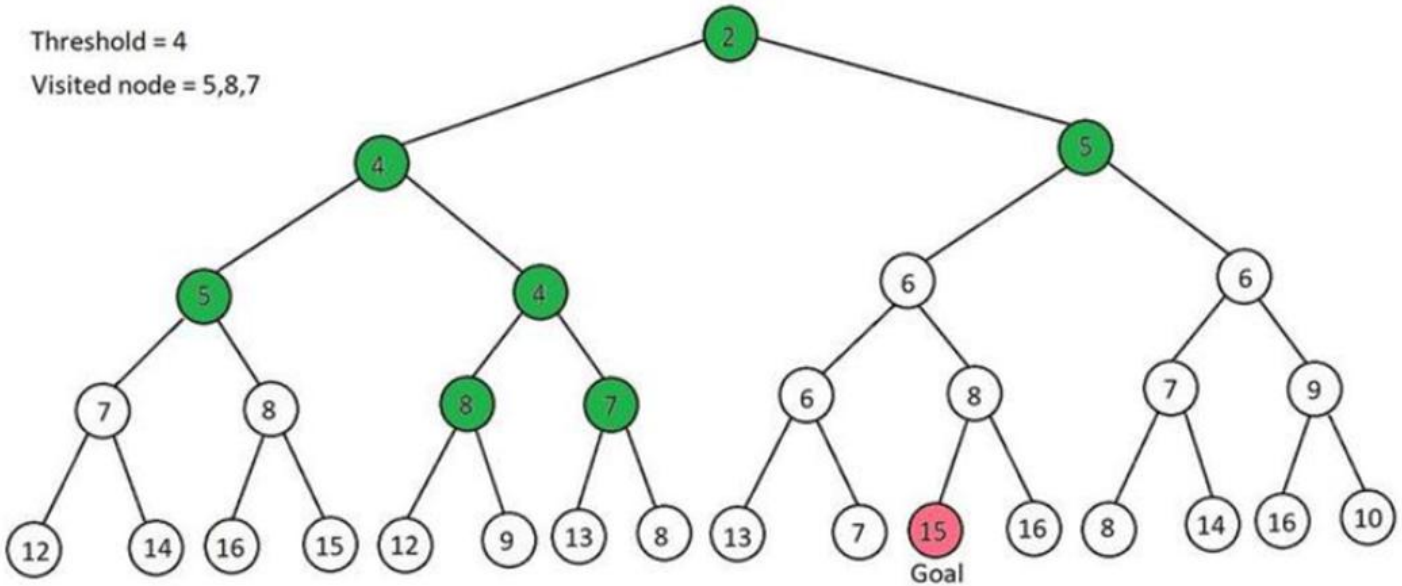
Iteration 1



Iteration 1

- Root node as current node i.e 2
- Threshold = current node value (2=2). So explore its children.
- 4 > Threshold & 5>Threshold. So, this iteration is over and the pruned values are 4, and 5.

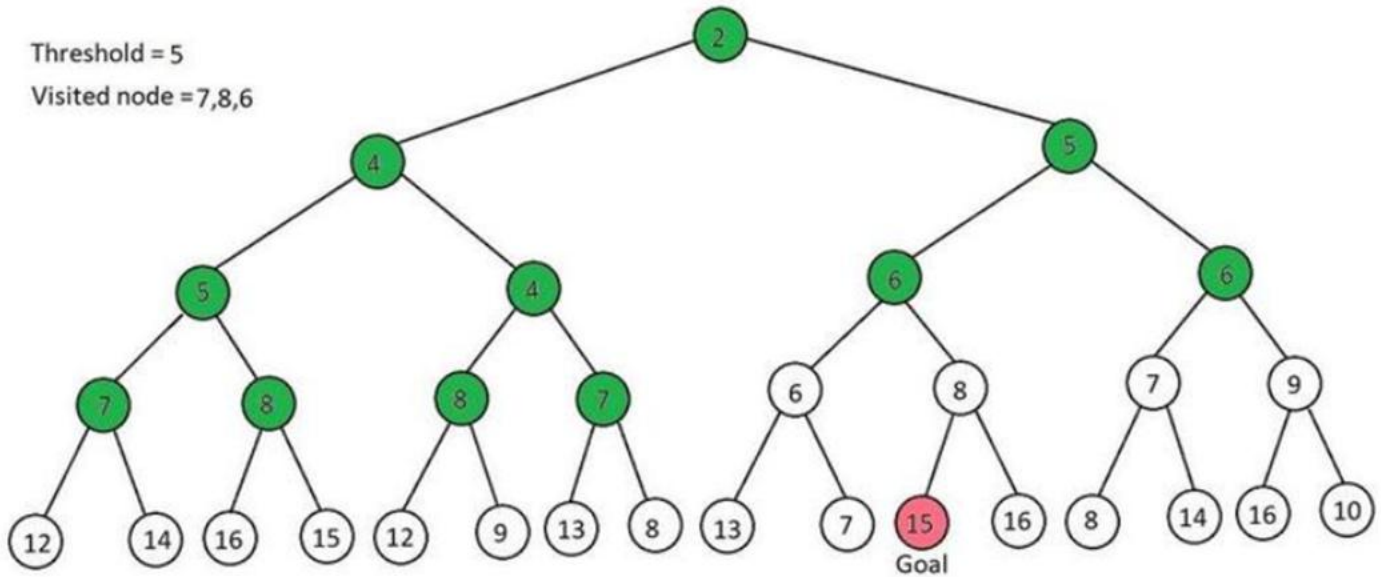
Iteration 2



Iteration 2

- In pruned values, the least is 4, So threshold = 4
- current node = 2 and 2< threshold, So explore its children. i.e two children explore one by one
- So, first children 4, So, set current node = 4 i.e equal to the threshold, so, explored its children also i.e 5, 4 having 5> threshold so, pruned it and explore second child of node 4 i.e 4, so set current node = 4 = threshold, and explore its children i.e 8 & 7 having both 8 & 7 > threshold so, pruned it. At the end of this, our pruned value is 5,8,7
- Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5>threshold, So pruned it.
- So, our pruned value is 5,8,7.

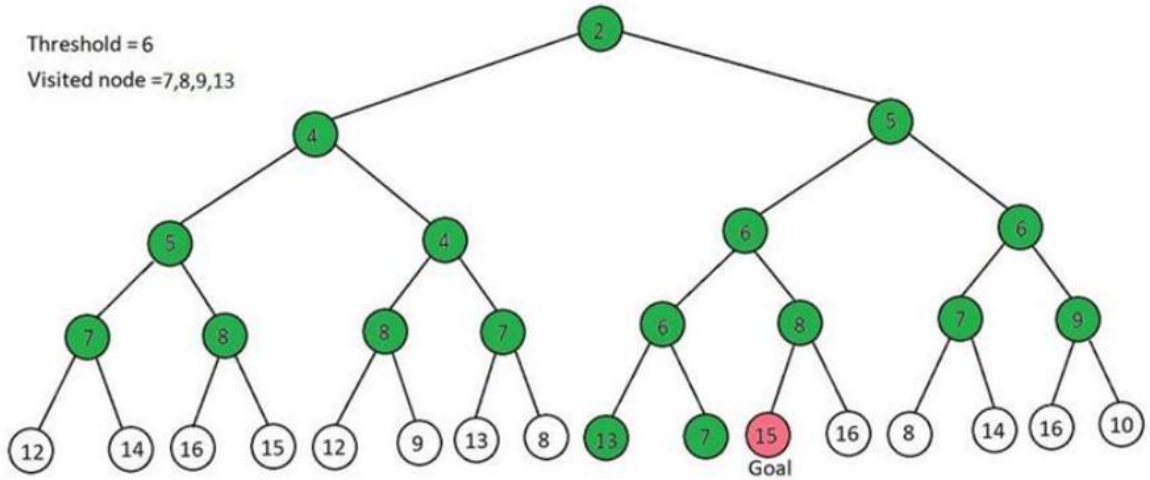
Iteration 3



Iteration 3

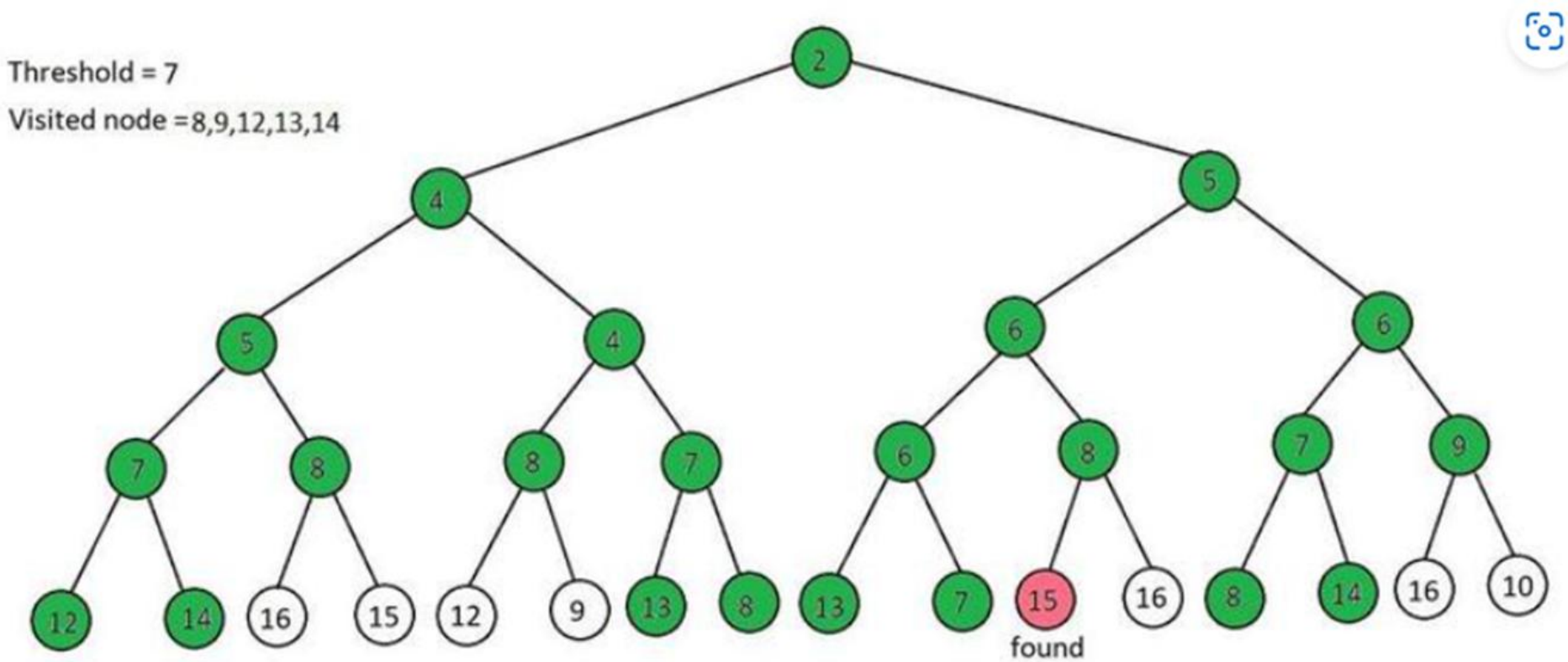
- In pruned values, the least is 5, So threshold = 5
- current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one
- So, first children 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4 having 5= threshold so explore its child also 7&8 > threshold. So, pruned it and explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, both 8 & 7 > threshold so, pruned it. At the end of this, our pruned value is 7 & 8
- Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 = threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 > threshold. So pruned it
- So, our pruned value is 7,8 & 6

Iteration 4



- In pruned values, the least value is 6, So threshold = 6
- current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one
- So, the first child is 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4 having 5< threshold so explore its child also 7&8 > threshold. So, pruned it and explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, both 8 & 7 > threshold so, pruned it. At the end of this, our pruned value is 7 & 8
- Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 = threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 = threshold, So, explore one by one,
- The first 6 has two children i.e 6 & 8, having 6 = threshold. So, explore its child also i.e 13 & 7. here both 13 & 7 > Threshold. So, pruned it. next is 8 > Threshold. pruned it, So, pruned value at this stage is 13,7 & 8.
- Explore the second child of 5 i.e 6 = Threshold. So, explore its child i.e 7 & 9. Both are greater than Threshold. So, pruned it
- So, our pruned values are 13,7,8 & 9.

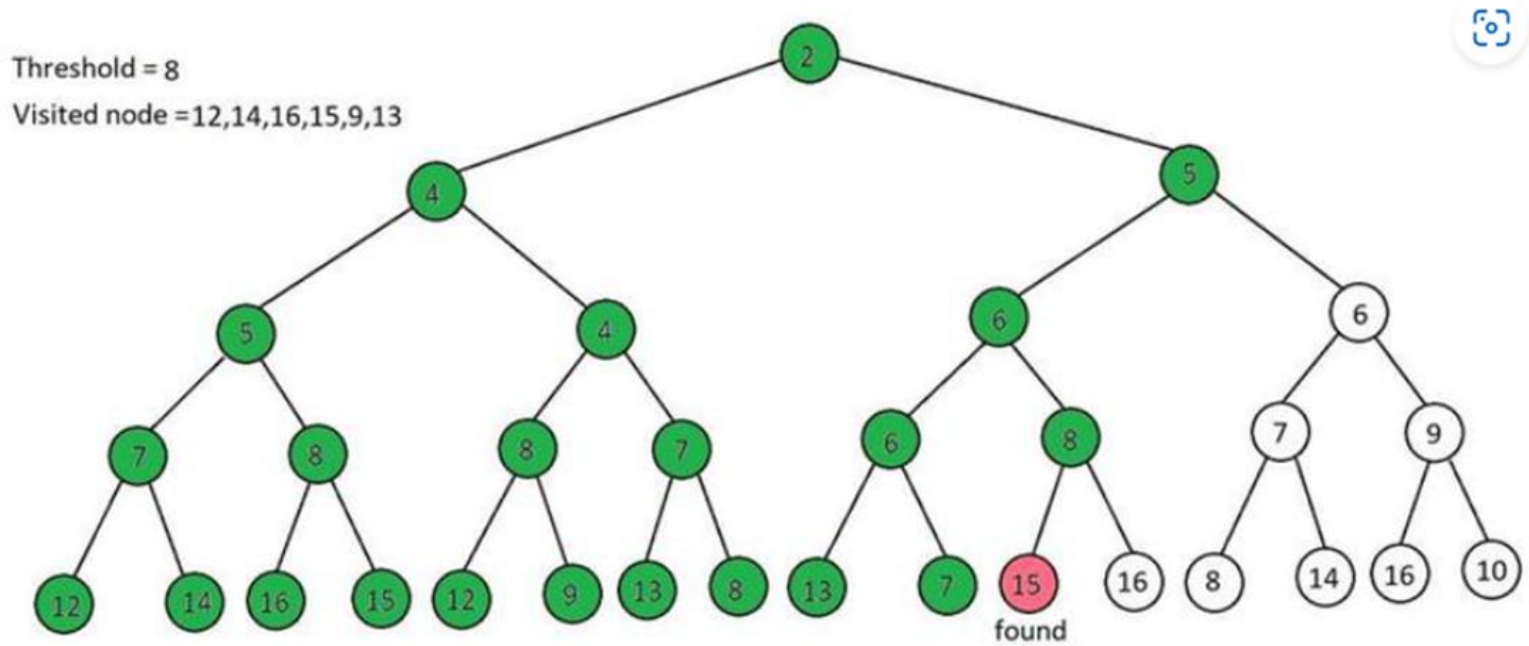
Iteration 5



- In pruned values, the least value is 7, So threshold = 7
- current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one
- So, the first child is 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4

- The first child of 4 is 5 i.e 5< threshold so explore its child also 7&8, Here 7 = threshold. So, explore its children i.e 12 & 14, both > Threshold. So, pruned it. And the second child of 5 is 8 > Threshold, So, pruned it. At this stage, our pruned value is 12, 14 & 7.
- Now explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, 8 > threshold so, pruned it. then go to the second child i.e 7 = Threshold, So explore its children i.e 13 & 8. having both > Threshold. So pruned it. At the end of this, our pruned value is 12,14,8 & 13
- Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 < threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 < threshold, So, explore one by one,
- The first 6 has two children i.e 6 & 8, having 6 < threshold. So, explore its child also i.e 13 & 7. here 13 > Threshold. So, pruned it. And 7= Threshold. And it hasn't any child. So, the shift to the next sub-child of 6 i.e 8 > threshold, So, pruned it. The pruned value at this stage is 12,14,8 &13
- Explore the second child of 5 i.e 6 < Threshold. So, explore its child i.e 7 & 9. Here 7 = Threshold, So, explore its children i.e 8 & 14, Both are greater than Threshold. So, pruned it, Now the sub child of 6 is 9 > Threshold, So, pruned it.
- So, our pruned values are 12,14,8,13 & 9.

Iteration 6



Iteration 6

- In pruned values, the least value is 8, So threshold = 8
- current node = root node = 2 and 2< threshold, So explore its children. i.e two children explore one by one
- So, the first child is 4, So, set current node = 4 < threshold, so, explored its children also i.e 5, 4
- The first child of 4 is 5 i.e 5< threshold so explore its child also 7&8, Here 7 < threshold. So, explore its children i.e 12 & 14, both > Threshold. So, pruned it. And the second child of 5 is 8 = Threshold, So, So, explore its children i.e 16 & 15, both > Threshold. So, pruned it. At this stage, our pruned value is 12, 14, 16 & 15.

- Now explore the second child of node 4 i.e 4, so set current node = 4 < threshold, and explore its children i.e 8 & 7 here, 8 = Threshold, So, So, explore its children i.e 12 & 9, both > Threshold. So, pruned it. then go to the second child i.e 7 < Threshold, So explore its children i.e 13 & 8. having 13 > Threshold. So pruned it. and 8 = Threshold and it hasn't any child. At the end of this, our pruned values are 12, 14, 16, 15, and 13.
- Similarly, Explore the second child of root node 2 i.e 5 as the current node, i.e 5 < threshold, so, explored its children also i.e 6 & 6, i.e both 6 & 6 < threshold, So, explore one by one,
- The first 6 has two children i.e 6 & 8, having 6 < threshold. So, explore its child also i.e 13 & 7. here 13 > Threshold. So, pruned it. And 7<Threshold. And it hasn't any child. So, the shift to the next sub-child of 6 i.e 8 = threshold. So, explored its children also i.e 15 & 16, Here **15 = Goal Node**. So, stop this iteration. Now no need to explore more.
- The goal path is **2–>5–>6–>8–>15**

Small Memory A* (SMA)

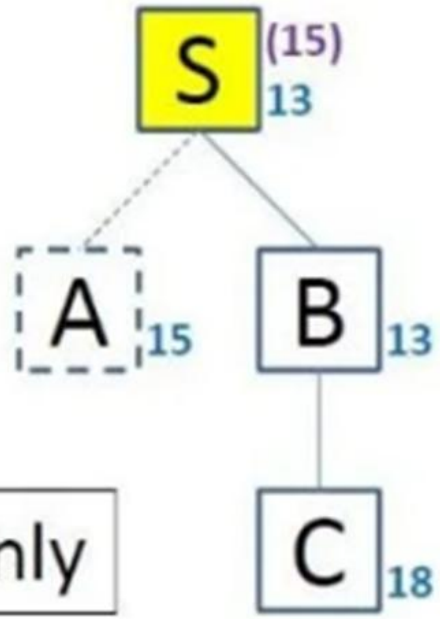
- **SMA*** is an **advanced version of A*** designed to handle **memory constraints** effectively.
- When memory is limited, **SMA* forgets less promising paths** while ensuring optimality.

How SMA* Works?

- Uses a **priority queue (like A*) but with a fixed memory limit**.
- When memory is full, it **removes the least promising nodes** (highest cost).
- Before removing a node, **SMA*** stores a **backup value** (f-cost of the best child) to allow reconstruction later.
- If needed, it **re-expands forgotten nodes** but prioritizes optimal paths.

SMA* Algorithm

- Optimizes A* to work within reduced memory
- ***Key Idea:***
  - **IF** memory **full** for **extra node (C)**
  - **Remove highest f-value leaf (A)**
  - **Remember best-forgotten child** in *each* parent node **(15 in S)**



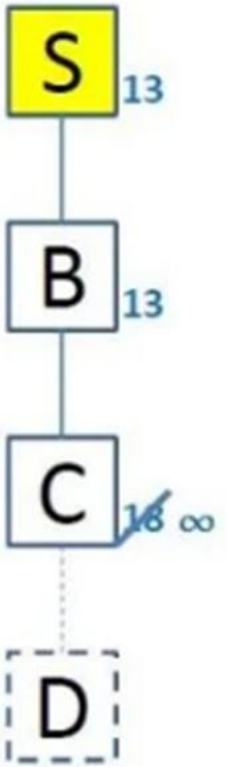
E.g. Memory of 3 nodes only

SMA* Algorithm

- **Generate Children 1 by 1**
  - **Expanding**: add 1 child at the time to QUEUE
  - Avoids **memory overflow**
  - **Allows monitoring** if nodes need deletion

SMA* Algorithm

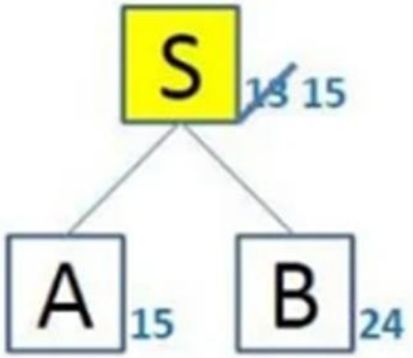
- **Too long paths: Give up**
  - **Extending** path **cannot fit** in **memory**
    - **give up (C)**
  - Set **f-value** node **(C)** to ∞
    - **Remembers:**
      path cannot be found here



E.g. Memory of 3 nodes only

SMA* Algorithm

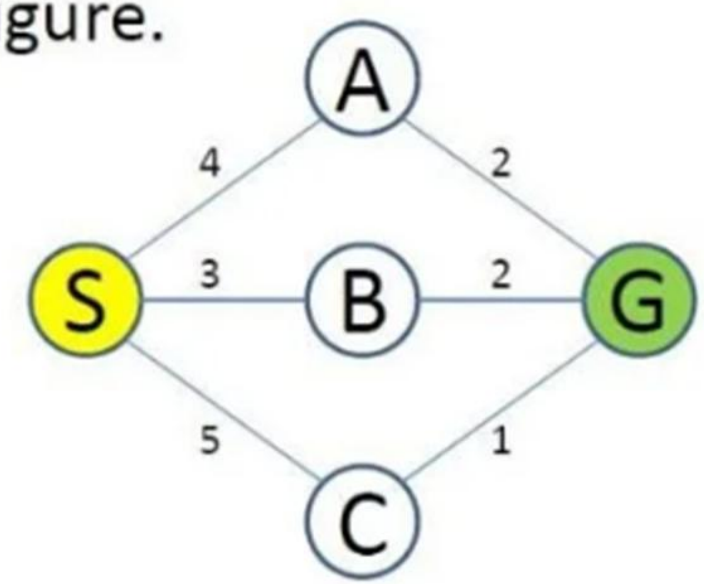
- **Adjust f-values**
  - **IF** all children M_i of node N have been explored
  - **AND** $\forall i$: $f(S...M_i) > f(S...N)$
  - **THEN reset** (through N $\Longrightarrow$ through children)
    - $f(S...N) = \min\{f(S...M_i) \mid M_i \text{ child of N}\}$



Better estimate for f(S)

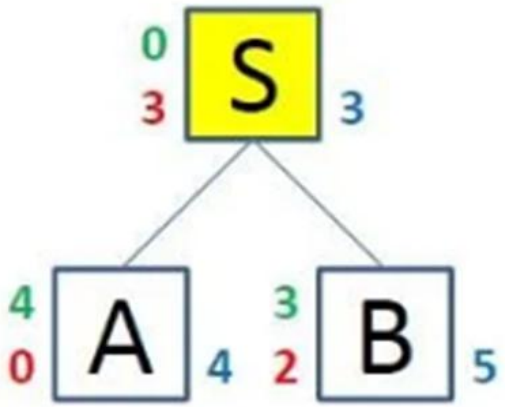
SMA* by Example

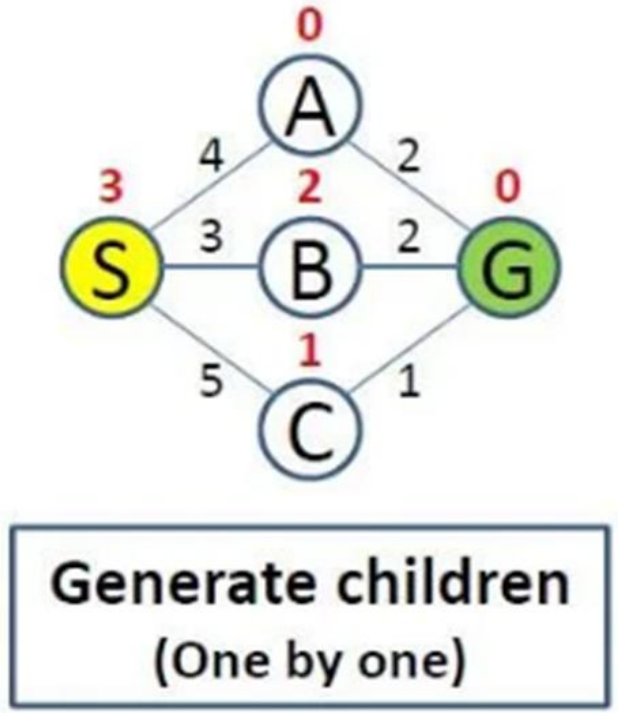
- Perform SMA* (memory: 3 nodes) on the following figure.



	S	A	B	C	G
heuristic	3	0	2	1	0

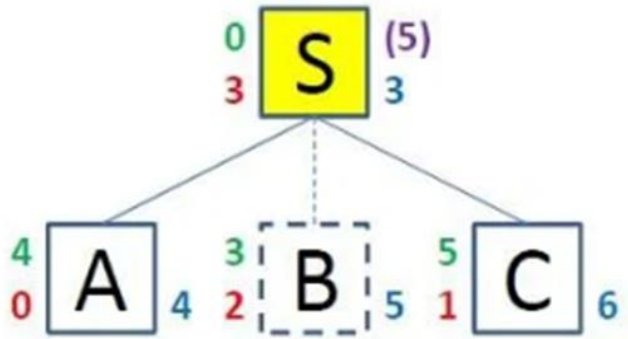
SMA* by Example

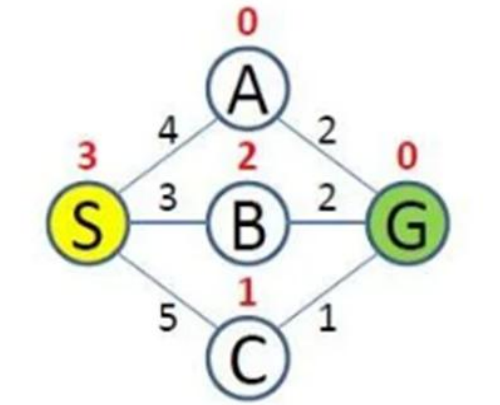




On the right of the node in blue color is the f value, on the left is the g value or the cost to reach that node(in green color), and the h value or the heuristic (in red color).

SMA* by Example



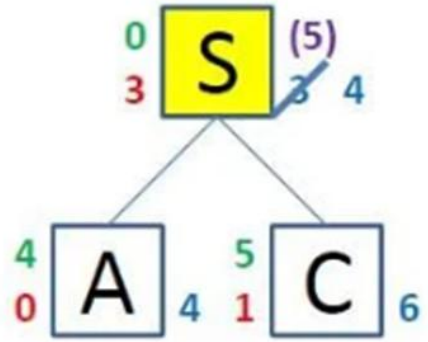


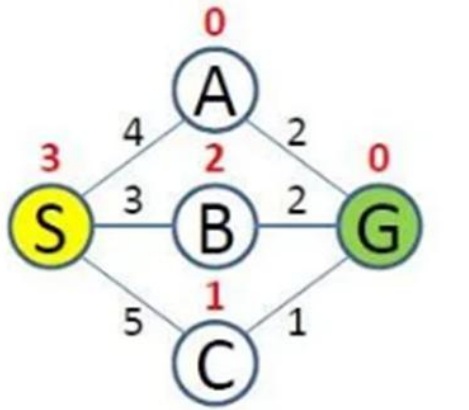
Generate children
(One by one)

Memory full

The already evaluated children with the highest f value get removed, but its value is remembered in its parent(the number in purple color on the right of the “S” node)

SMA* by Example



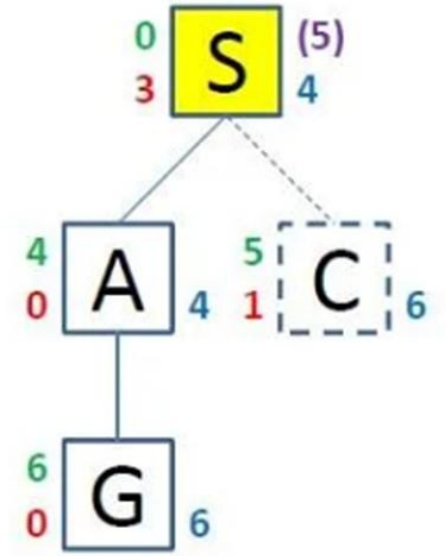


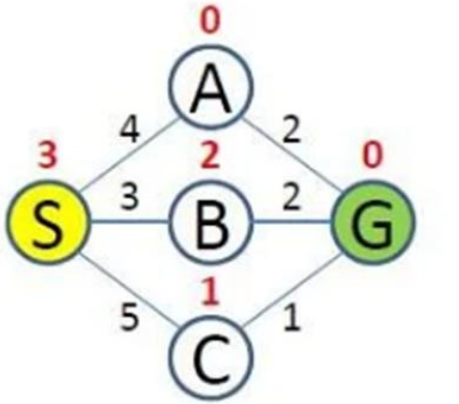
All children are explored

Adjust f-values

if all accessible children nodes are visited or explored we adjust the parent's f value to the value of the children with the lowest f value.

SMA* by Example



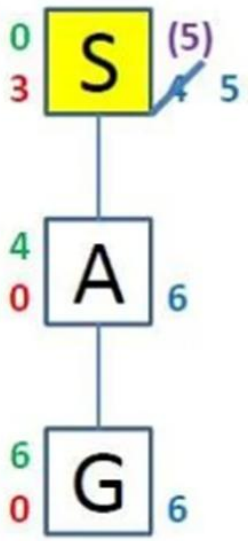


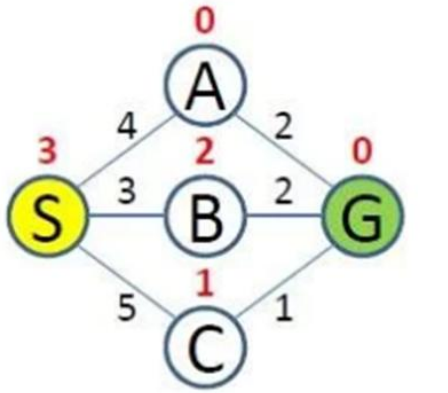
Generate children
(One by one)

Memory full

we remove the “C” node, since it has the highest f value but we don’t remember it, since the “B” node has a lower value. Now we update the f value of the “A” node to 6 because all the children accessible are explored.

SMA* by Example



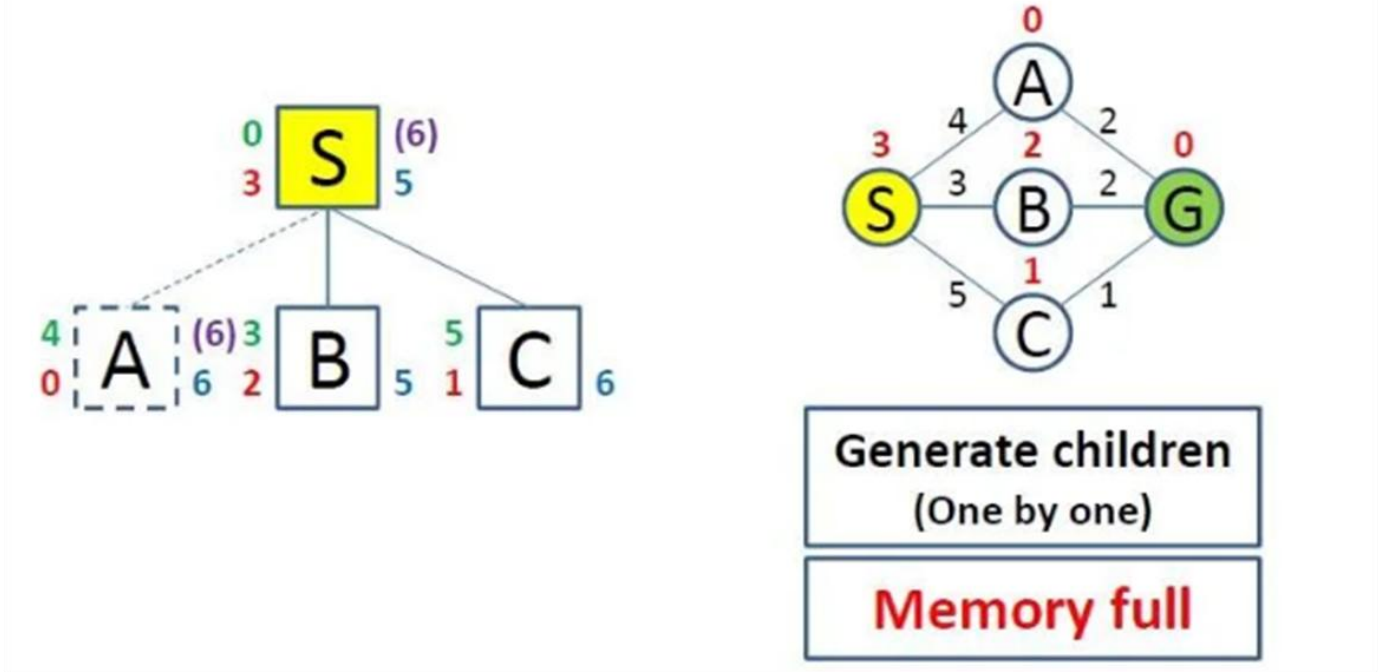


All children are explored (update)

Adjust f-values

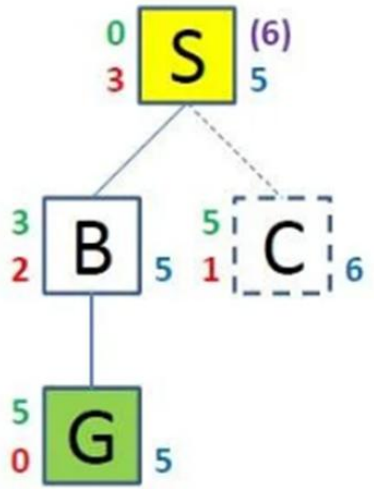
Just because we've reached the goal node, doesn't mean that the algorithms are finished.
As we can see the remembered f value of the "B" node is lower than the f value of the "G" node explored through the "A" node.
We also update the f value of the "S" node to 5 since there is no more child node with value 4

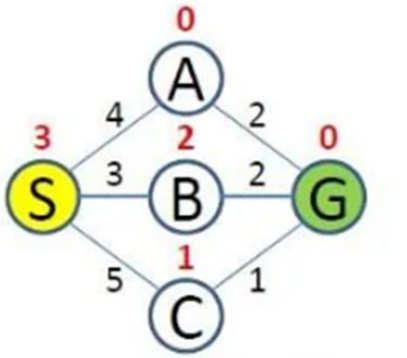
SMA* by Example



we will remove the “A” node since it has already discovered the goal and has the same f value as the “C” node so we will remember this node in the parent “S” node.

SMA* by Example





Generate children
(One by one)

Memory full

We've reached the goal again, this time through the "B" node. This time cost is lower than through the "A" node.
The memory is full, which means we need to remove the "C" node, because it has the highest f value, and it means that if we eventually reach the goal from this node the total cost at best case scenario will be equal to 6, which is worse than we already have.
At this stage, the algorithm terminates, and we have found the shortest path from the "S" node to the "G" node.

Comparison with Other Algorithms

Algorithm	Memory Usage	Optimality	Efficiency
A*	High	Yes	Good
IDA*	Low	Yes	Slightly slower due to re-exploration
SMA*	Bounded	Yes	Best for large problems

While heuristic search strategies focus on reaching a goal, specific heuristic techniques address how to explore the search space effectively.

Heuristic Search Techniques

- **Generate and Test:** This simple technique generates potential solutions and tests each for feasibility.
  - **Limitation:** Inefficient for complex problems.
  - **Example:** Finding a valid configuration for a Sudoku puzzle.
- **Hill Climbing:** A local search algorithm that moves towards a higher-value state based on a heuristic. However, it may:
  - Get stuck at **local maxima.**
  - Struggle with **plateaus** or **ridges.**
  - **Example:** Optimizing network configurations.

Heuristic Search Techniques

- **State Space Search:** Problems are modeled as a graph of states connected by transitions. The goal is to navigate from the initial state to a goal state.
  - **Application:** Pathfinding in mazes or solving puzzles like the 8-puzzle problem.
- **Constraint Satisfaction Problems (CSPs):** These involve finding solutions that satisfy a set of constraints. Techniques include:
  - **Backtracking**
  - **Forward Checking**
  - **Constraint Propagation**
  - **Example:** Scheduling problems, like assigning exam slots for students.

Example of **Generate and Test**

Problem: Find a valid arrangement of 4 queens on a 4x4 chessboard such that no two queens attack each other.

Example of **Generate and Test**

- **Steps**:
- **Generate**: Start by randomly placing 4 queens on the chessboard. For example:
  - Row 1: Queen in column 1.
  - Row 2: Queen in column 2.
  - Row 3: Queen in column 4.
  - Row 4: Queen in column 3.
- **Test**: Check if this arrangement satisfies the constraints:
  - No two queens should be in the same row, column, or diagonal.
  - In this case:
    - Row 1's queen conflicts with Row 2's queen in the same diagonal.
    - Row 3's queen conflicts with Row 4's queen in the same diagonal.
  - **Result**: This arrangement is invalid.

Example of **Generate and Test**

- **Repeat**: Generate a new arrangement and test again:
  - Row 1: Queen in column 1.
  - Row 2: Queen in column 3.
  - Row 3: Queen in column 0.
  - Row 4: Queen in column 2.
- **Test Again**: Check constraints for this new arrangement:
  - No two queens attack each other.
- **Result**: Valid solution found!

This process continues until a valid arrangement is found. While simple to implement, **Generate and Test** can be computationally expensive for large problem spaces like the 8-Queens problem or beyond.

Example of **Hill Climbing** in AI

- **Problem**: You want to find the shortest path to deliver packages in a small city.
- **Scenario**: Imagine you are a delivery person. You have to deliver packages to 5 locations in a city. The **goal** is to minimize the total distance traveled.

Example of **Hill Climbing** in AI

- **Steps in Hill Climbing**:
- **Initial State**: Start with a random route. For example:
  - Route: A → B → C → D → E → A (Total distance = 30 km).
- **Generate Neighbors**: Make small changes to the route to generate neighboring solutions. For example:

  Swap two locations in the route:
  - Neighbor 1: A → C → B → D → E → A (Total distance = 28 km).
  - Neighbor 2: A → B → D → C → E → A (Total distance = 32 km).

Example of **Hill Climbing** in AI

- **Evaluate Neighbors**: Calculate the total distance for each new route and compare:
  - Neighbor 1: 28 km (improvement).
  - Neighbor 2: 32 km (worse).
- **Move to the Best Neighbor**: Choose the neighbor with the shortest distance. Move to Neighbor 1 with a distance of **28 km**.
- **Repeat**: Continue generating and evaluating neighbors by swapping locations until no shorter route is found.

Example of **Hill Climbing** in AI

- **Result**: The algorithm converges at a route with the shortest distance possible for the given changes.
- **Limitations**:
  - **Local Minima**: The algorithm may settle for a route that's shorter than others nearby but not the shortest overall.
  - **Example**: If another route (A → D → C → B → E → A) with 25 km exists but isn't reachable with simple swaps, the algorithm won't find it.
- **Solution**: Techniques like adding random restarts or trying larger changes to the route can help find a better overall solution.

Search and problem-solving techniques are not limited to static problems. They also power dynamic environments like games, where strategies and adversaries must be accounted for.

Introduction to Game Playing

Game-playing algorithms help AI systems make decisions in competitive environments. Games can be categorized as:

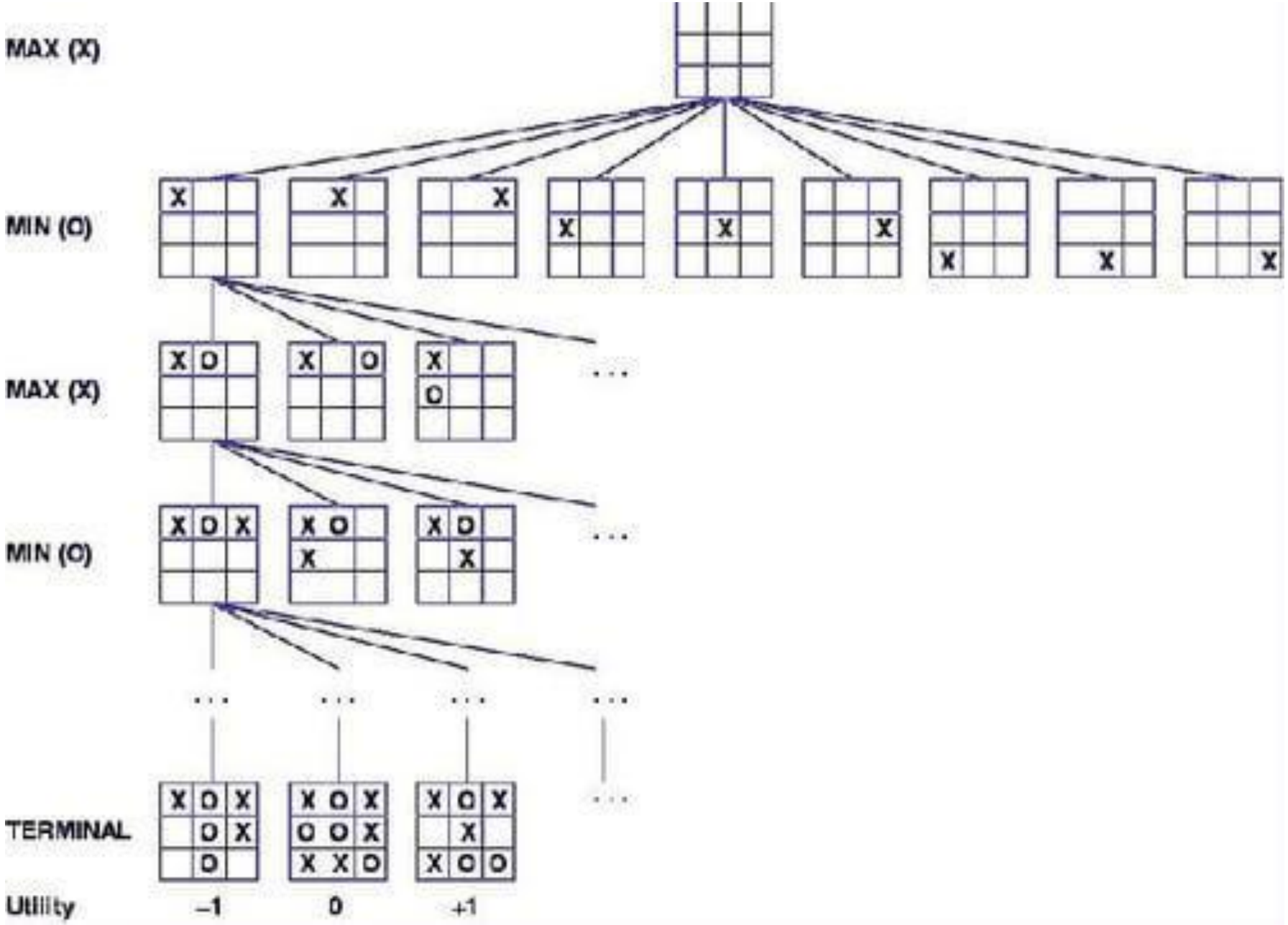
- **Perfect Information Games:** All information is visible (e.g., Chess).
- **Imperfect Information Games:** Some elements are hidden or uncertain (e.g., Poker)

Game Playing Algorithms

- **Minimax Algorithm**
- **Alpha-Beta Pruning**

Minimax Algorithm in Game Search

- Games are important for testing **heuristic search algorithms**.
- Two-player games introduce an **opponent** who actively works against the player.
- **Minimax** is a decision-making algorithm used in adversarial games.

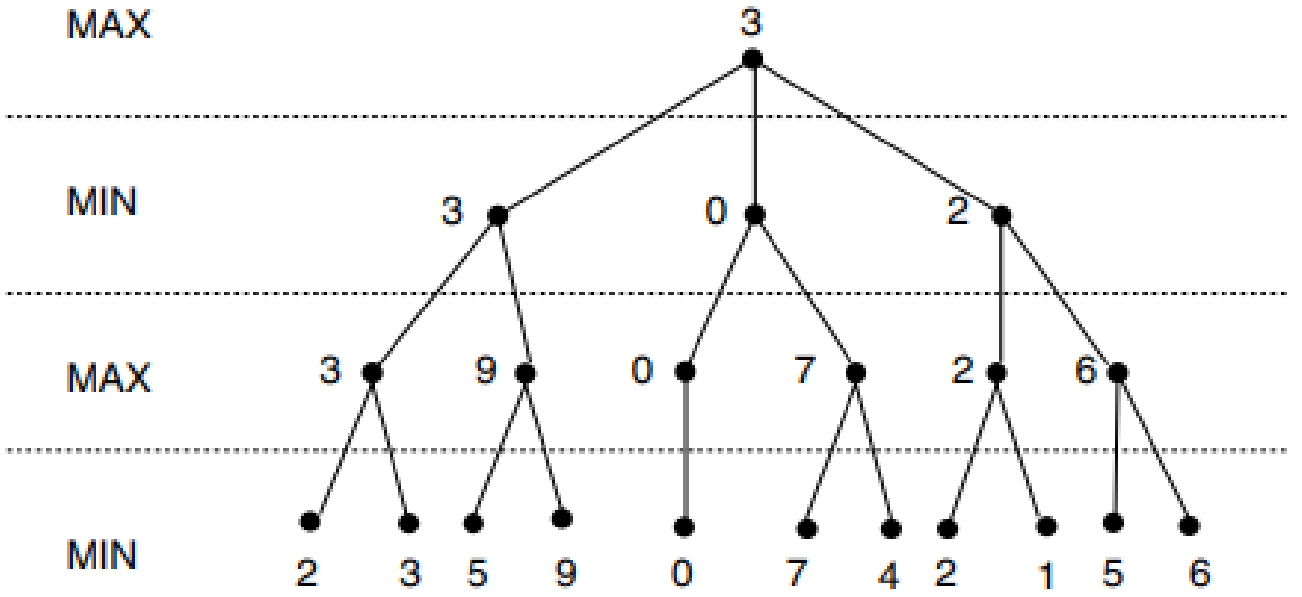

MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
–1
0
+1

Minimax Algorithm Concept

- Two players:
  - **MAX** tries to maximize the score.
  - **MIN** tries to minimize MAX's advantage.
- Game states are **evaluated using heuristics** to determine the best move.
- **Leaf nodes** contain heuristic values (win/loss).
- Internal nodes are assigned values based on **minimax propagation**.

Minimax Example with Heuristic Values

- Example game tree with **heuristic evaluations** at leaf nodes.
- **MIN and MAX levels alternate** in choosing values.
- The final value at the root determines the **best possible move for MAX**.



Minimax to a hypothetical state space. Leaf states show heuristic values; internal states show backed-up values.

Fixed Ply Depth in Minimax

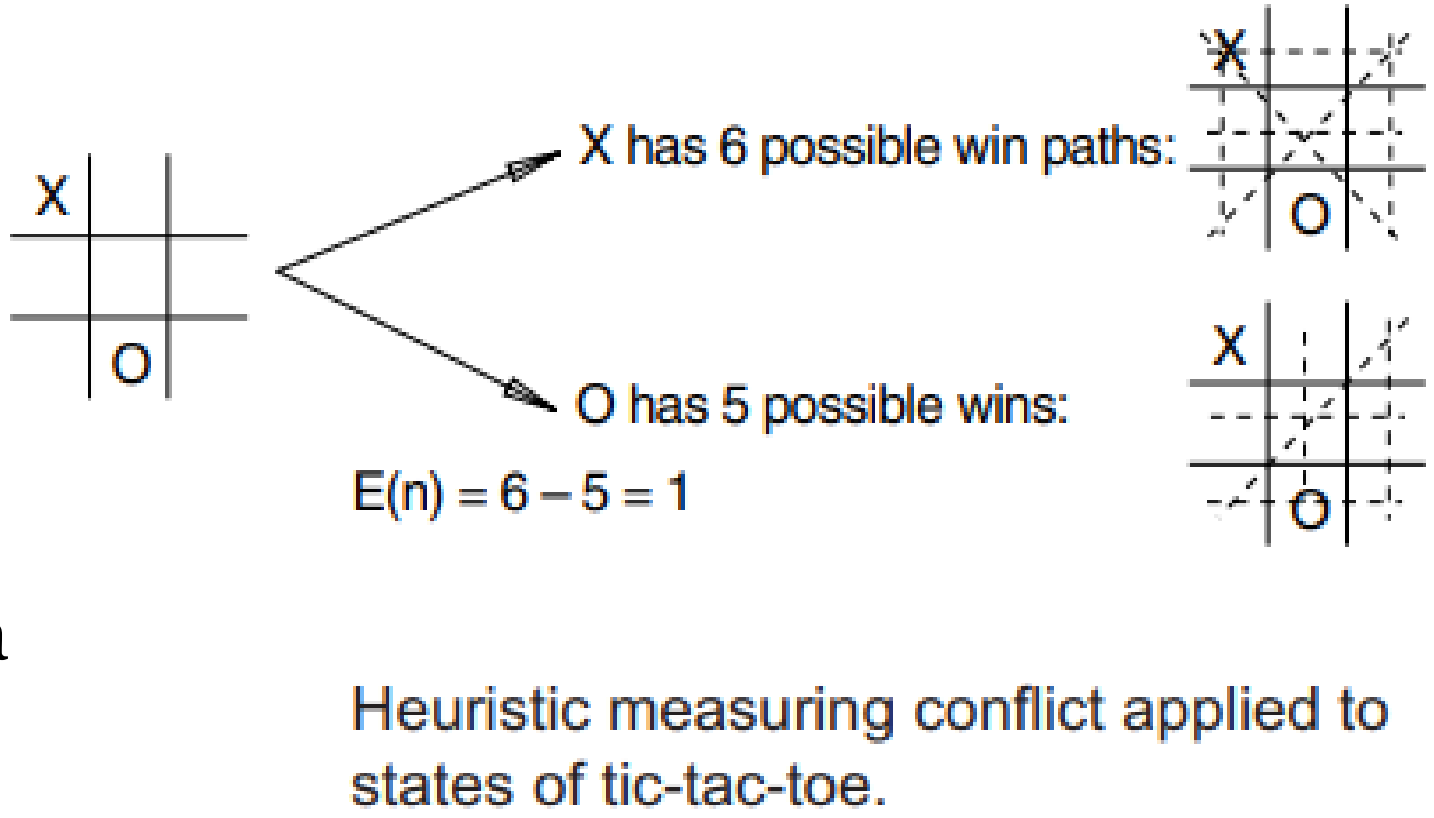
- Searching the entire game tree is often **impractical**.
- A solution: **Fixed-ply depth search**, also called **n-ply lookahead**.
- Evaluates states after **n moves** instead of searching all possible moves.
- **Heuristic evaluation** is applied to non-terminal states.

Heuristic Evaluation in Tic-Tac-Toe

- The heuristic function **E(n) = M(n) - O(n)** is used to evaluate the board state in Tic-Tac-Toe.
- **M(n)** represents the number of possible winning paths for the **player (X)**.
- **O(n)** represents the number of possible winning paths for the **opponent (O)**.
- **E(n) (evaluation value)** determines how favorable a state is for **X**.

Examples from the Tic Tac Toe

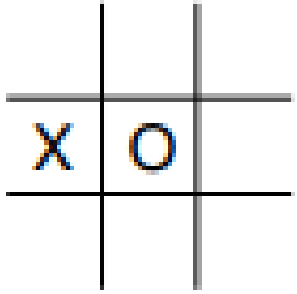
- **First Example: X's Advantage**
- **X has 6 possible winning paths.**
- **O has 5 possible winning paths.**
- **E(n) = 6 - 5 = 1** (X has a slight advantage).



Heuristic measuring conflict applied to states of tic-tac-toe.

Examples from the Tic Tac Toe

- **Second Example: O's Advantage**
- **X has 4 possible winning paths.**
- **O has 6 possible winning paths.**
- **E(n) = 4 - 6 = -2** (O is in a stronger position).



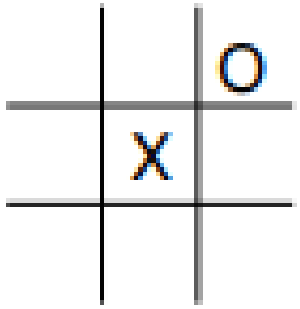
X has 4 possible win paths;
O has 6 possible wins

$E(n) = 4 - 6 = -2$

Heuristic measuring conflict applied to states of tic-tac-toe.

Examples from the Tic Tac Toe

- **Third Example: X's Slight Advantage**
- **X has 5 possible winning paths.**
- **O has 4 possible winning paths.**
- **E(n) = 5 - 4 = 1** (X has a minor advantage).



X has 5 possible win paths;
O has 4 possible wins

E(n) = 5 – 4 = 1

Heuristic measuring conflict applied to states of tic-tac-toe.

Examples from the Tic Tac Toe

- **A positive E(n) favors X**, meaning X has more chances to win.
- **A negative E(n) favors O**, meaning O has more chances to win.
- **E(n) = 0 means a neutral position**, where both players have an equal number of winning paths.
- This heuristic is useful for **minimax decision-making**, helping the AI determine the best move.

Challenges in Minimax

- **Horizon Effect**: A promising move may lead to a bad situation beyond the search depth.
- **Depth Bias**: Deeper searches do not always guarantee better decisions.
- **Computational Complexity**: Minimax is expensive for large state spaces.

Introduction to Alpha-Beta Pruning

- **Optimized version of minimax**, reducing the number of nodes evaluated.
- **Alpha (for MAX) and Beta (for MIN) values** track the best options so far.
- **Pruning occurs** when a branch cannot influence the final decision.

How Alpha-Beta Pruning Works

1. **Alpha (α) - Associated with MAX nodes**
   - **Represents the best value MAX can achieve**.
   - If a MIN node's value is ≤ **α**, we stop searching its subtree.

2. **Beta (β) - Associated with MIN nodes**
   - **Represents the best value MIN can force**.
   - If a MAX node's value is ≥ **β**, we stop searching its subtree.

Advantages of Alpha-Beta Pruning

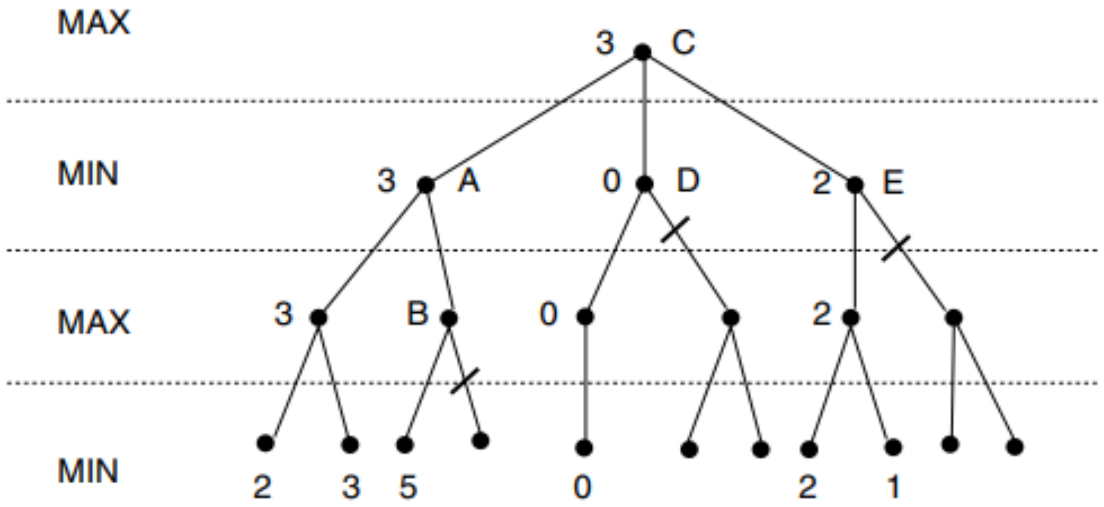
- **Reduces the number of nodes evaluated**, improving efficiency.
- **Can search deeper in the same time compared to minimax**.
- **Best-case scenario: Cuts search space in half**, allowing double the search depth.
- **Worst-case scenario:** Still performs as well as minimax.

Advantages of Alpha-Beta Pruning

- **Minimax ensures optimal play** but is computationally expensive.
- **Alpha-beta pruning optimizes minimax by eliminating unnecessary searches**, significantly improving performance.
- **In Tic-Tac-Toe, Chess, and other strategy games, this technique allows AI to make optimal decisions efficiently.**

Alpha-Beta Pruning Example

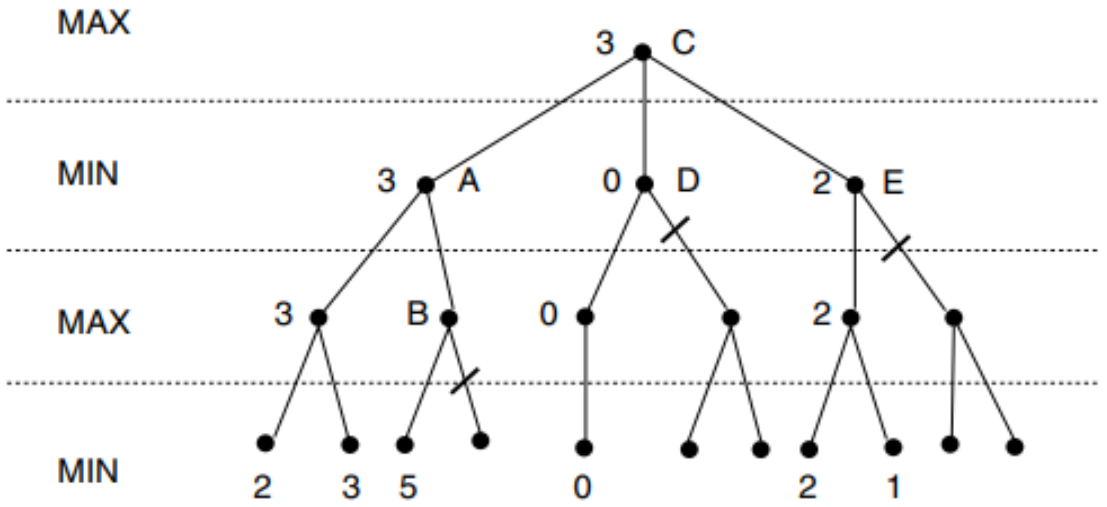
1. **Alpha (α) – Used for MAX nodes**
   - Represents the **best value MAX can achieve** so far.
   - **MAX updates α when it finds a better move**.
   - If **α ≥ β**, pruning occurs, and the subtree is ignored.
2. **Beta (β) – Used for MIN nodes**
   - Represents the **best value MIN can force** on MAX.
   - **MIN updates β when it finds a worse move**.
   - If **β ≤ α**, pruning occurs, and the subtree is ignored.



A has β = 3 (A will be no larger than 3)
B is β pruned, since 5 > 3
C has α = 3 (C will be no smaller than 3)
D is α pruned, since 0 < 3
E is α pruned, since 2 < 3
C is 3

Alpha-Beta Pruning Example

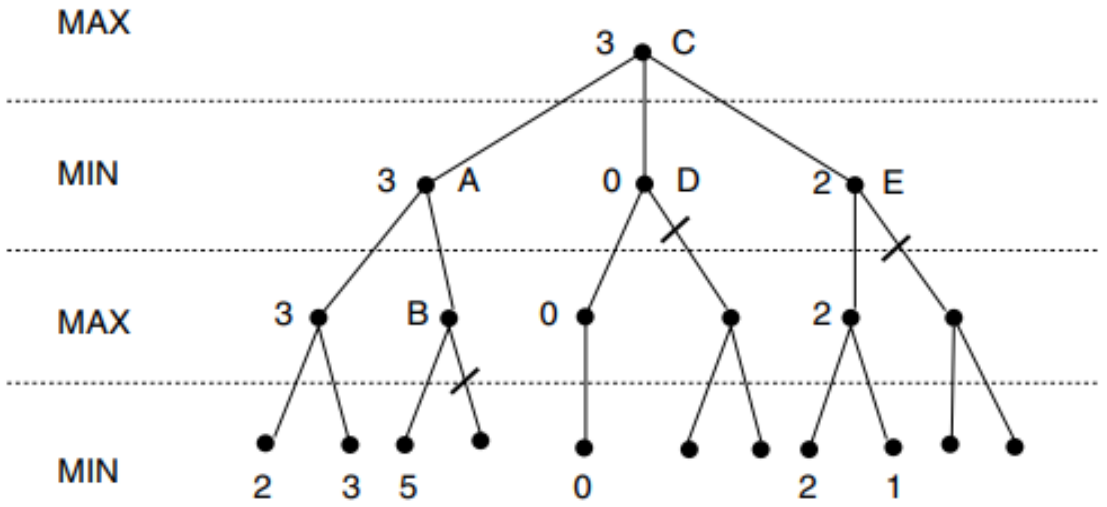
- **MAX at the root (C) starts evaluating its children (A, D, and E).**
  - The **goal is to find the highest possible value**.
- **Evaluating Node A (MIN level):**
  - **A evaluates B and finds a value of 3**.
  - **A has β = 3** (since MIN selects the smallest value).
  - **B's second child has a value of 5**, which is greater than β (3), so B is **pruned**.
  - A is assigned **β = 3**.



A has β = 3 (A will be no larger than 3)
B is β pruned, since 5 > 3
C has α = 3 (C will be no smaller than 3)
D is α pruned, since 0 < 3
E is α pruned, since 2 < 3
C is 3

Alpha-Beta Pruning Example

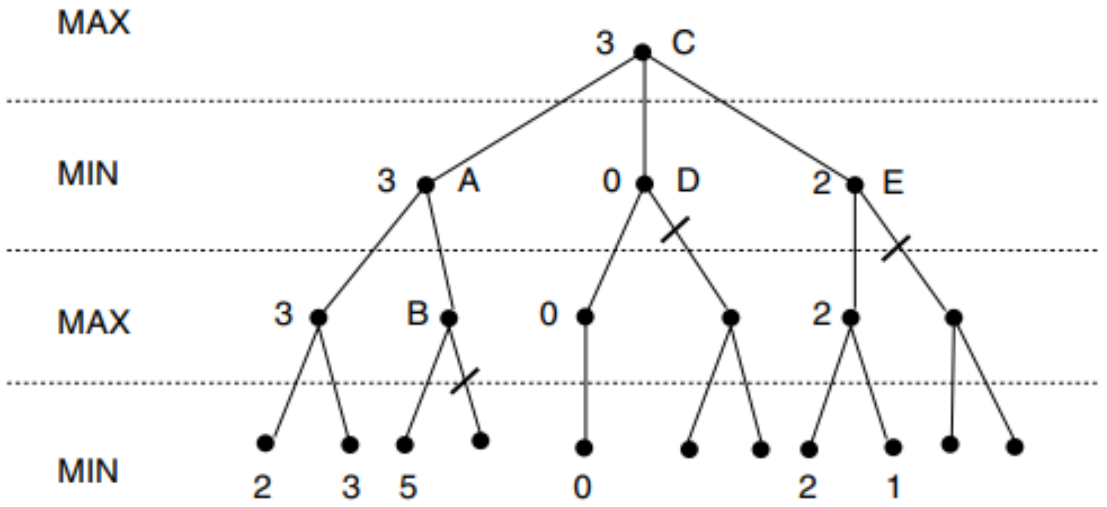
- **Evaluating Node D (MIN level):**
  - **D's first child has a value of 0.**
  - **Since 0 < 3 (current α from A), node D is pruned.**
  - **C will not consider D's children further.**



A has $\beta = 3$ (A will be no larger than 3)
B is β pruned, since 5 > 3
C has $\alpha = 3$ (C will be no smaller than 3)
D is α pruned, since 0 < 3
E is α pruned, since 2 < 3
C is 3

Alpha-Beta Pruning Example

- **Evaluating Node E (MIN level):**
  - **First child has value 2.**
  - **Since 2 < 3 (α from A), node E is pruned.**
  - **No need to explore further.**
- **Final Value for Root Node (C):**
  - Since node A **provided α = 3**, and all other nodes were pruned, **C takes value 3.**



A has β = 3 (A will be no larger than 3)
B is β pruned, since 5 > 3
C has α = 3 (C will be no smaller than 3)
D is α pruned, since 0 < 3
E is α pruned, since 2 < 3
C is 3

Why Does This Pruning Happen?

- **For MAX nodes**, once we find a move that guarantees a higher value than a previous MIN choice, **we don’t need to check other branches**.
- **For MIN nodes**, once we find a move that guarantees a lower value than a previous MAX choice, **we can stop evaluating that branch**.

Applications of Game Playing

Game playing algorithms have diverse applications, including:

- **Training AI Systems**: Games like Chess and Go are used to train AI for strategic reasoning.
- **Simulation and Decision-Making**: Game strategies simulate real-world scenarios, such as economic forecasting or military planning.
- **Entertainment**: Used in video games for creating intelligent opponents or NPCs.
- **Research**: Understanding human cognition and decision-making.

Perfect and Imperfect Decision Games

- **Perfect Decision Games:** These games have no randomness or hidden information, allowing deterministic strategies.
  - **Example:** Connect Four, Checkers.
- **Imperfect Decision Games:** These involve uncertainty, such as hidden cards or dice rolls, requiring probabilistic reasoning.
  - **Example:** Poker, Monopoly.