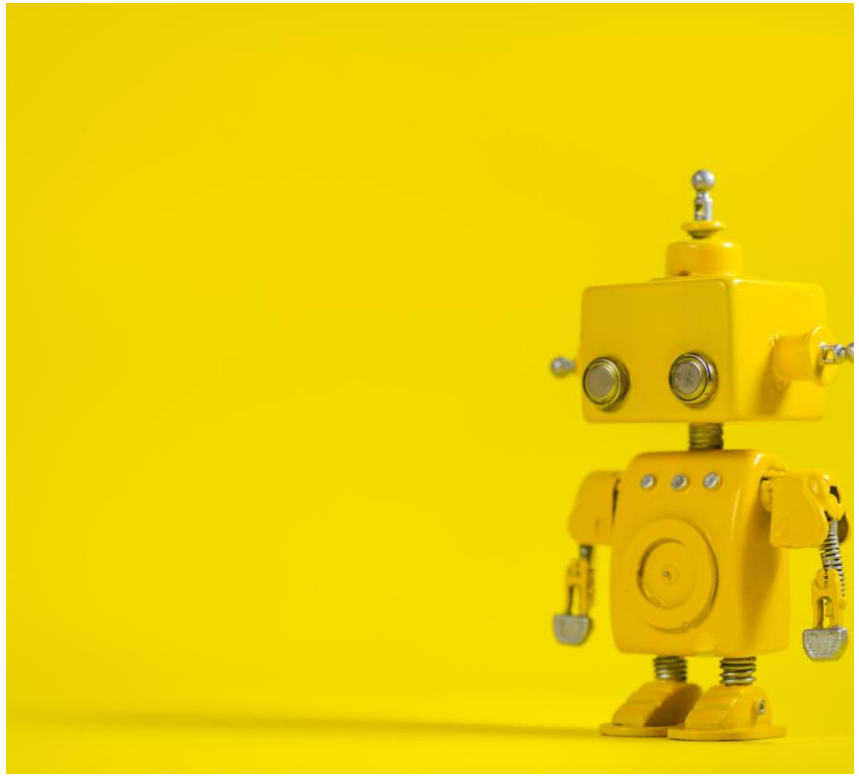


# PROBLEM SOLVING BY SEARCHING

Omkar Basnet

Msc CSIT, CDCSIT, TU Kirtipur

MBA , Lincoln University

HoD , CSIT/BCA

**Texas international College**

# Problem Solving

- Problem solving, particularly in artificial intelligence, may be characterized as a systematic search through a range of possible actions in order to reach some predefined goal or solution.
- Problem-solving methods divide into special purpose and general purpose. A special-purpose method is tailor-made for a particular problem and often exploits very specific features of the situation in which the problem is embedded. In contrast, a general-purpose method is applicable to a wide variety of problems. One general-purpose technique used in AI is means-end analysis—a step-by-step, or incremental, reduction of the difference between the current state and the final goal.

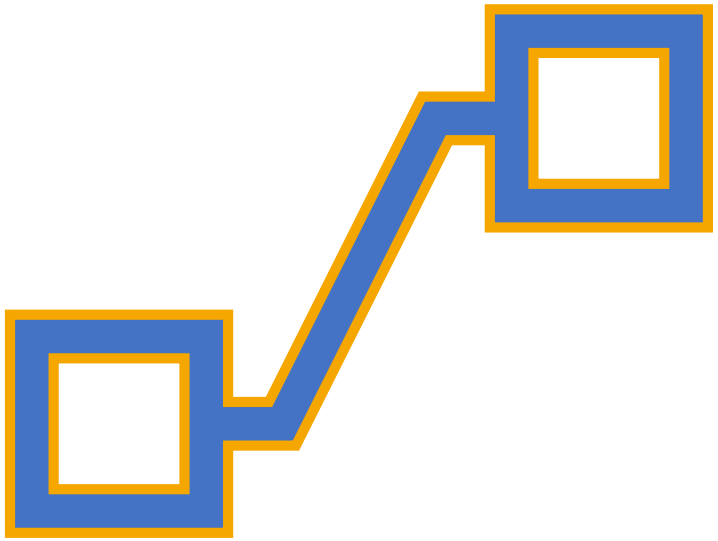
# Four general steps in problem solving

- **Goal formulation**
  - What are the successful world states
- **Problem formulation**
  - What actions and states to consider given the goal
- **Search**
  - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
- **Execute**
  - Give the solution perform the actions.

# Problem formulation

- **A problem is defined by:**
  - An initial state: State from which agent start
  - Successor function: Description of possible actions available to the agent.
  - Goal test: Determine whether the given state is goal state or not
  - Path cost: Sum of cost of each path from initial state to the given state.

  **A solution is a sequence of actions from initial to goal state. Optimal solution has the lowest path cost.**



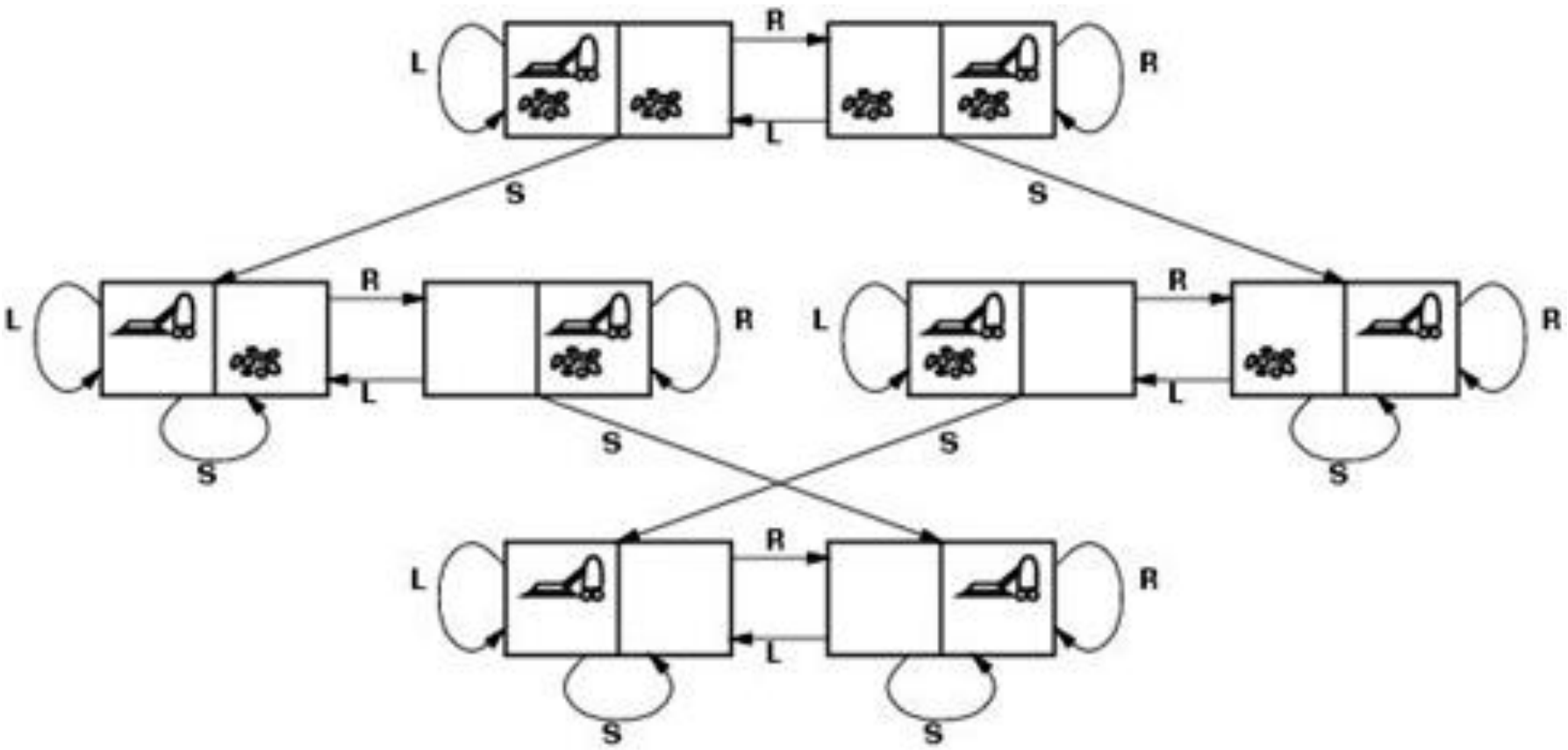
# State Space representation

- The state space is commonly defined as a directed graph in which each node is a state and each arc represents the application of an operator transforming a state to a successor state.
- A **solution** is a path from the initial state to a goal state.

# State Space representation

- **Basically, there are two types of problem approaches:**
  - **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.
  - **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

# State Space representation of Vacuum World Problem:



# State Space representation of Vacuum World Problem:

- **States??** two locations with or without dirt: 2 x $2^2$=8 states.
- **Initial state??** Any state can be initial
- **Actions??** {*Left*, *Right*, *Suck*}
- **Goal test??** Check whether squares are clean.
- **Path cost??** Number of actions to reach goal.

**Reminder :**
step performed by problem solving agent

- **Goal Formulation**
- **Problem Formulation:**
- **Initial State:** It is the starting state or initial step of the agent towards its goal.
- **Actions:** It is the description of the possible actions available to the agent.
- **Transition Model:** It describes what each action does.
- **Goal Test:** It determines if the given state is a goal state.
- **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, **an optimal solution has the lowest path cost among all the solutions.**
- **Initial state, actions**, and **transition model** together define the **state-space** of the problem implicitly.

# 8 Puzzle Problem

- Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blank space. The tile adjacent to the blank space can slide into that space. The objective is to reach a specified goal state similar to the goal state
- **States:** It describes the location of each numbered tiles and the blank tile.
- **Initial State:** We can start from any state as the initial state.
- **Actions:** Here, actions of the blank space is defined, i.e., either **left, right, up or down**
- **Transition Model:** It returns the resulting state as per the given state and actions.
- **Goal test:** It identifies whether we have reached the correct goal-state.
- **Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.

# 8 Puzzle Problem

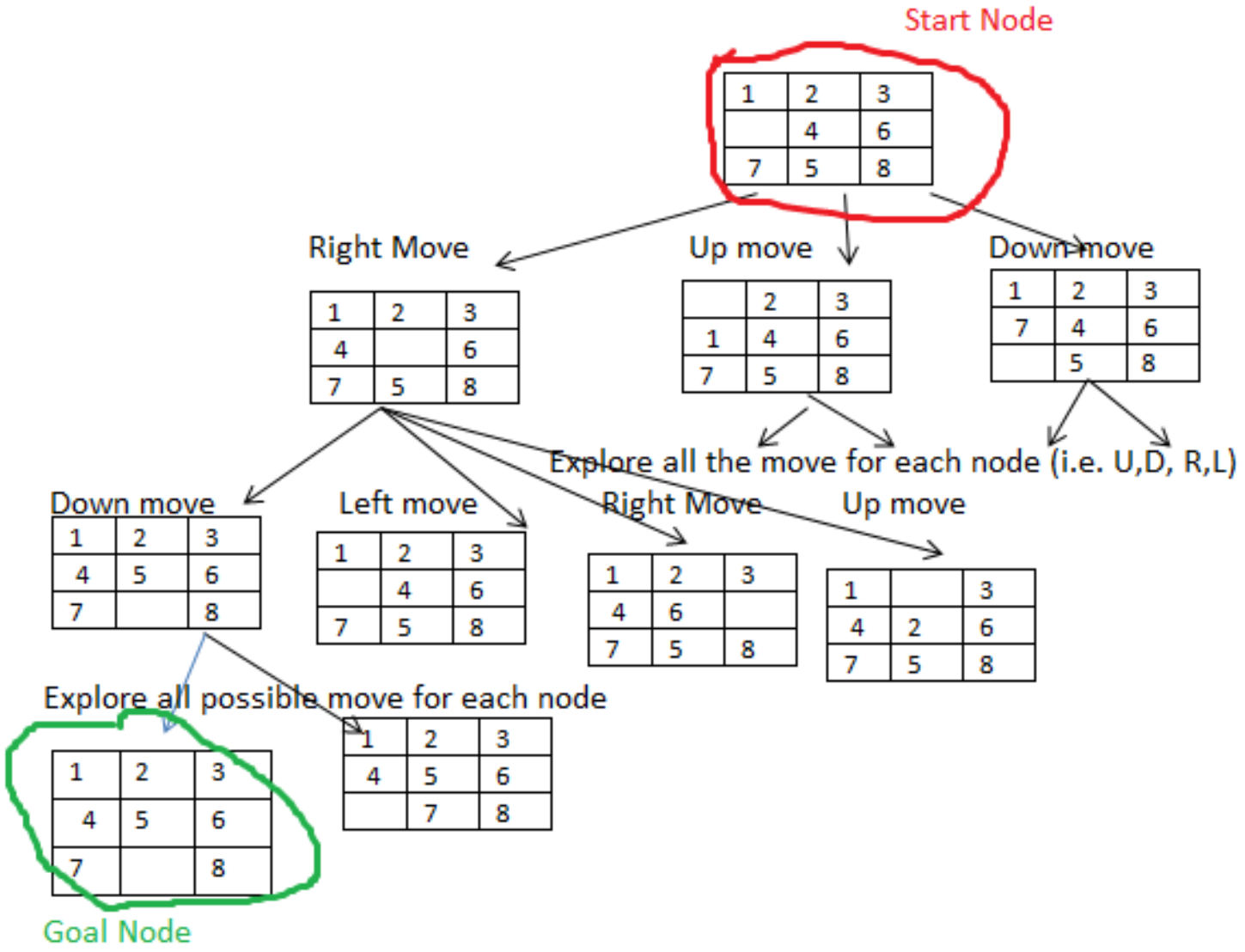
- Here we have initial state and goal state:
- Initial state

| | | |
|---|---|---|
| 1 | 2 | 3 |
| | 4 | 6 |
| 7 | 5 | 8 |

goal state

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

# 8 Puzzle Problem



# Well defined problems

If the given problem is defined properly in five component then it is called well defined problems.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

**Figure 3.1** A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

# Romania problem

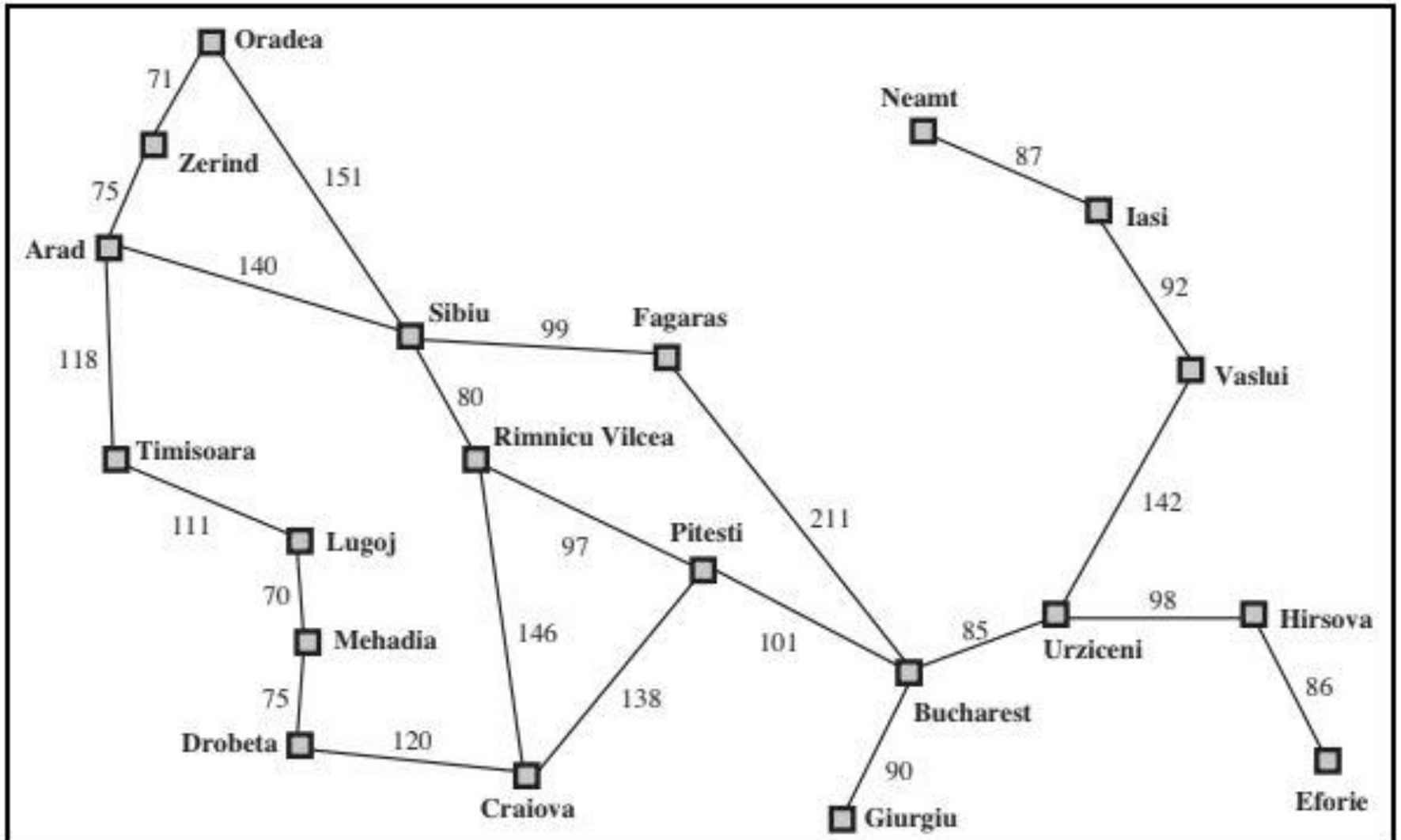
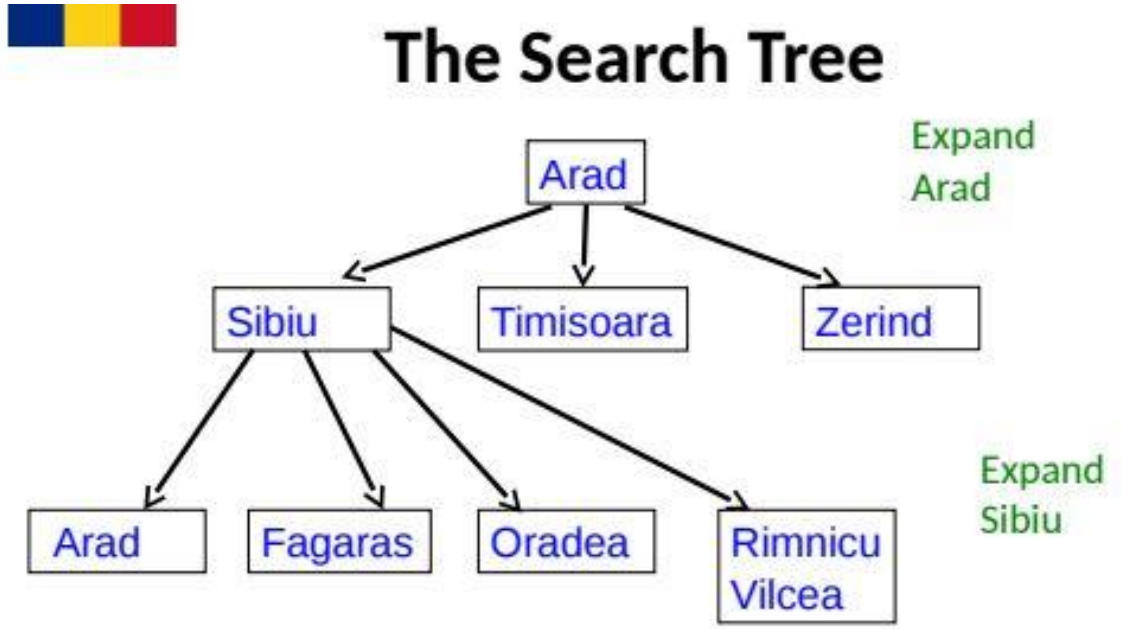



**Figure 3.2** A simplified road map of part of Romania.

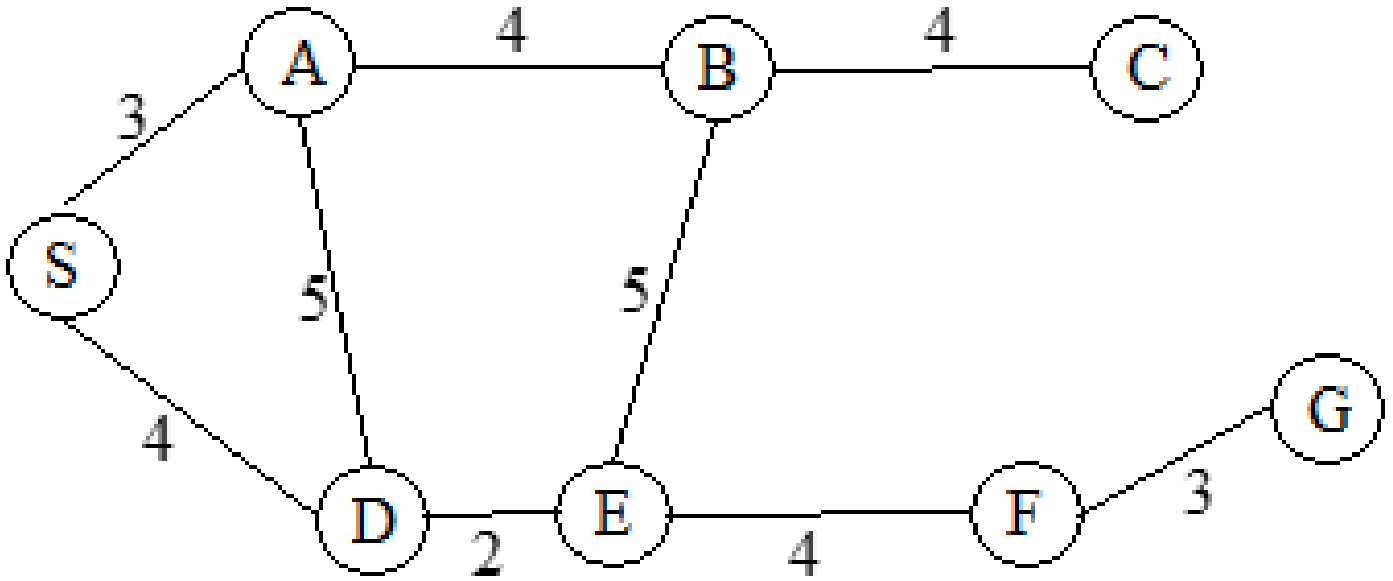
# Romania Problem

# Romania problem

- **Now how can we choose the optimal solution???**
- **We will discuss different search techniques to find out the optimal solution for the given problems.**

# Solving Problems by Searching




- This contains a representation of a map. The nodes represent cities, and the links represent direct road connections between cities. The number associated to a link represents the length of the corresponding road.
- The search problem is to find a path from a city S to a city G

# Solving Problems by Searching

- Search problems are part of a large number of real-world applications:
- VLSI layout , Path planning , Robot navigation etc.
- **There are two broad classes of search methods:**
- - **uninformed (or blind) search methods**
- - **heuristically informed search methods.**
- **In the case of the uninformed search methods**, the order in which potential solution paths are considered is arbitrary, using no domain-specific information to judge where the solution is likely to lie.
- **In the case of the heuristically informed search methods**, one uses domain-dependent (heuristic) information in order to search the space more efficiently.

# Measuring problem Solving Performance

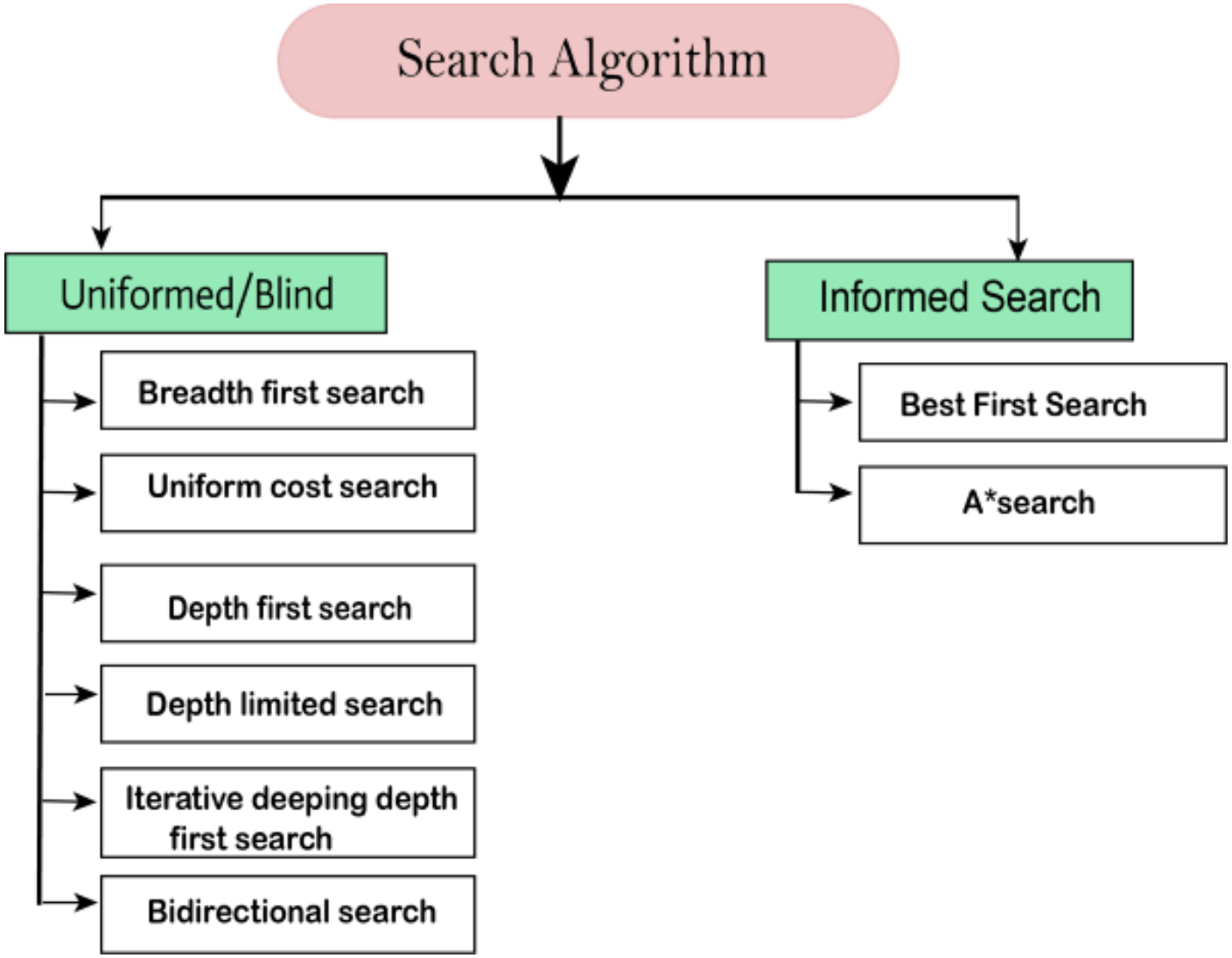
- We will evaluate the performance of a search algorithm in four ways
- **Completeness:** An algorithm is said to be complete if it definitely finds solution to the problem, if exist.
- **Time Complexity:** How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**
- **Space Complexity:** How much space is used by the algorithm? Usually measured in terms of the **maximum number of nodes in memory at a time**
- **Optimality/Admissibility:** If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

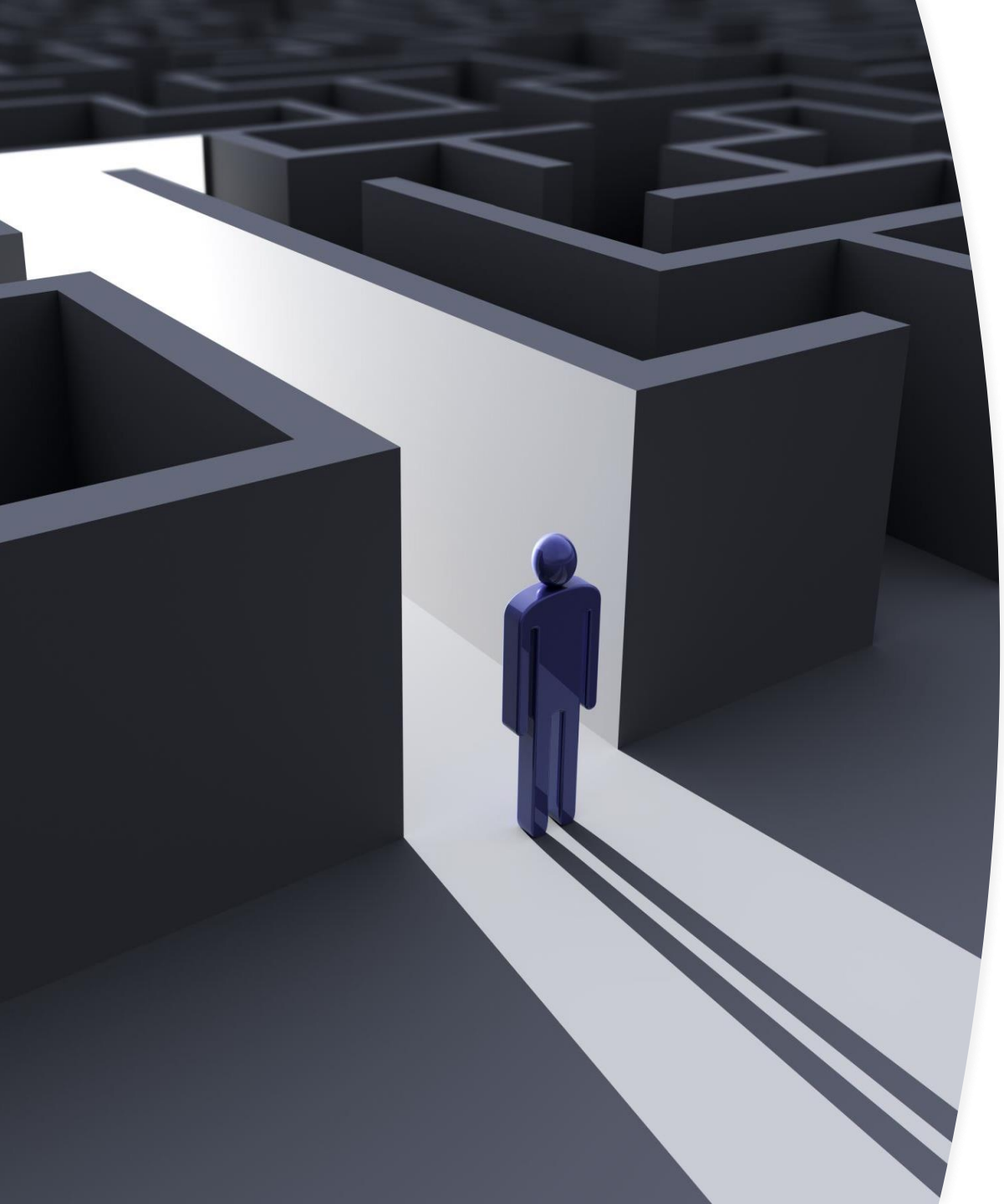
# Measuring problem Solving Performance

- **Time and space complexity are measured in terms of**
- **b -** Maximum branching factor (number of successor of any node) of the search tree
- **m -** depth of the least-cost solution
- **d -** maximum length of any path in the space

# Search Strategies

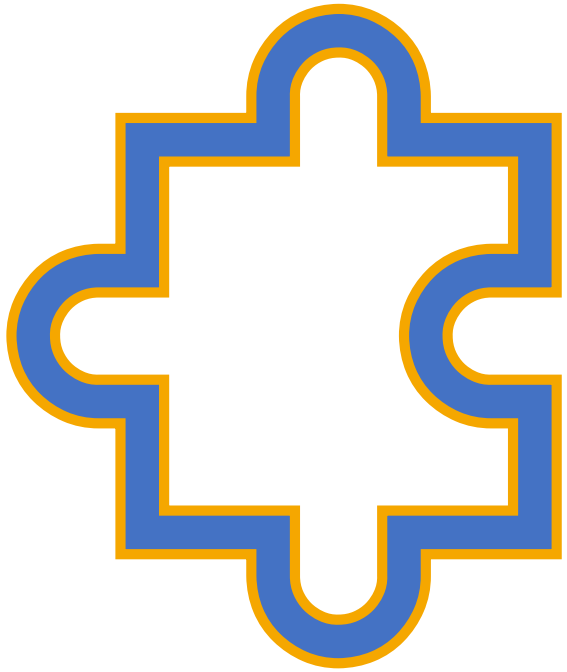
- Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.





# Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.



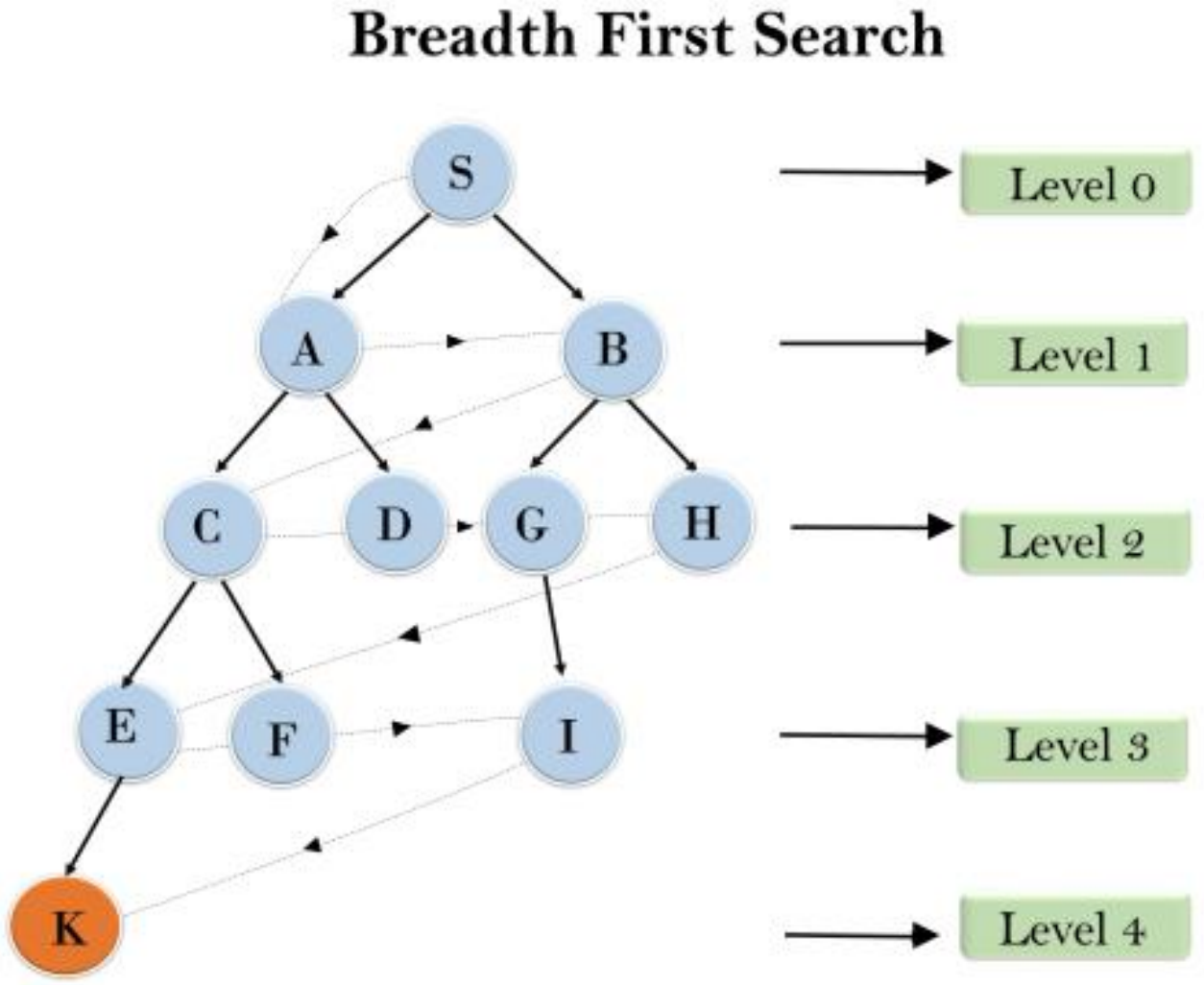
# Breadth-first Search:

- **Advantages:**
- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.
- **Disadvantages:**
- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

# Breadth-first Search:

- In this tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:
- **S---> A--->B---->C--->D---->G--->H**

  **--->E---->F---->I---->K**



# Breadth-first Search:

- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

  $$\mathbf{T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)}$$

- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.
- **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

# Depth-first Search

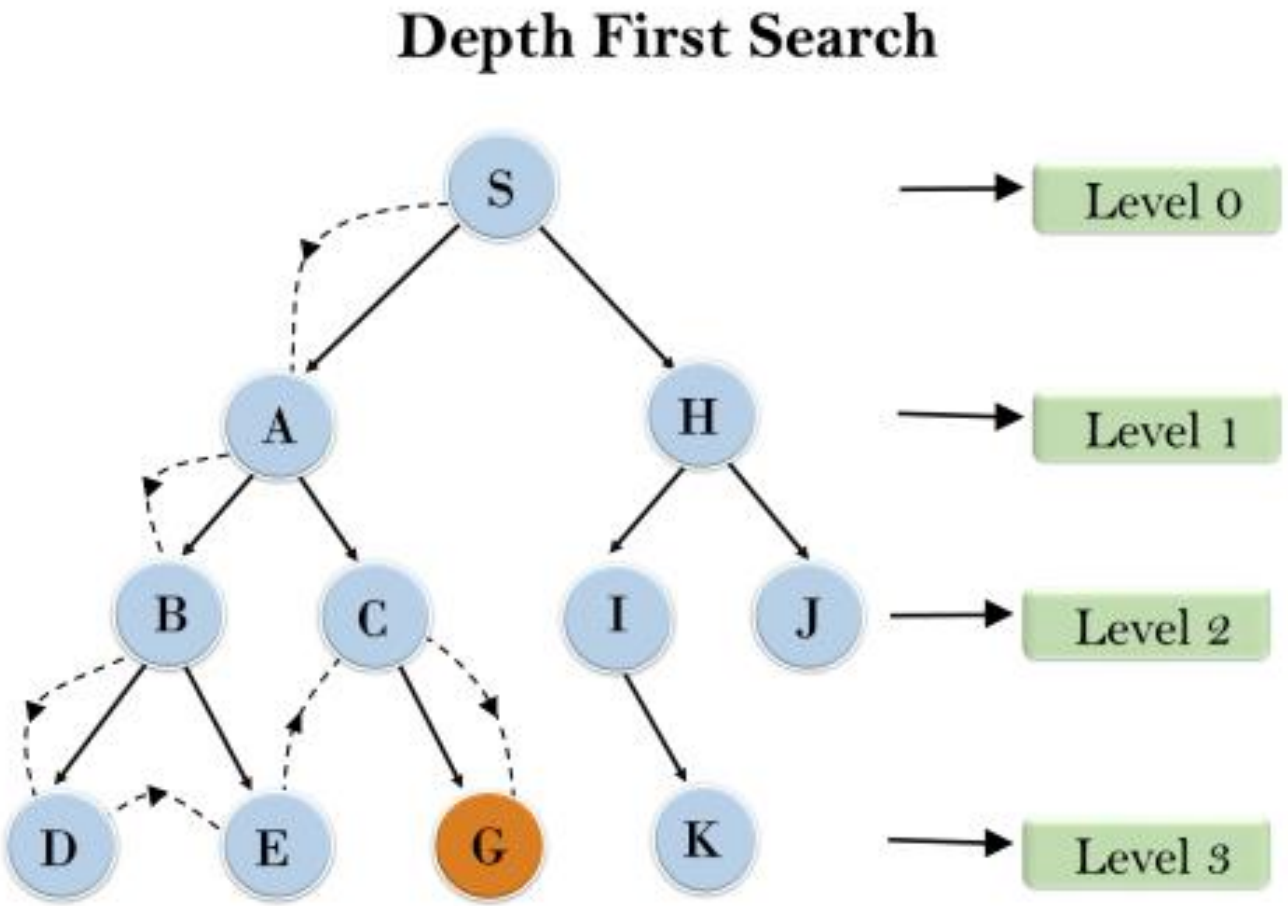
- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

# Depth-first Search

- **Advantage:**
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- **Disadvantage:**
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop

# Depth-first Search

- In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
- **Root node--->Left node ----> right node.**
- It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.
- **S---> A--->B--->D---->E--->C --->G**



# Depth-first Search

- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

  $$\mathbf{T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)}$$

- **Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)**
- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

# Depth-first Search

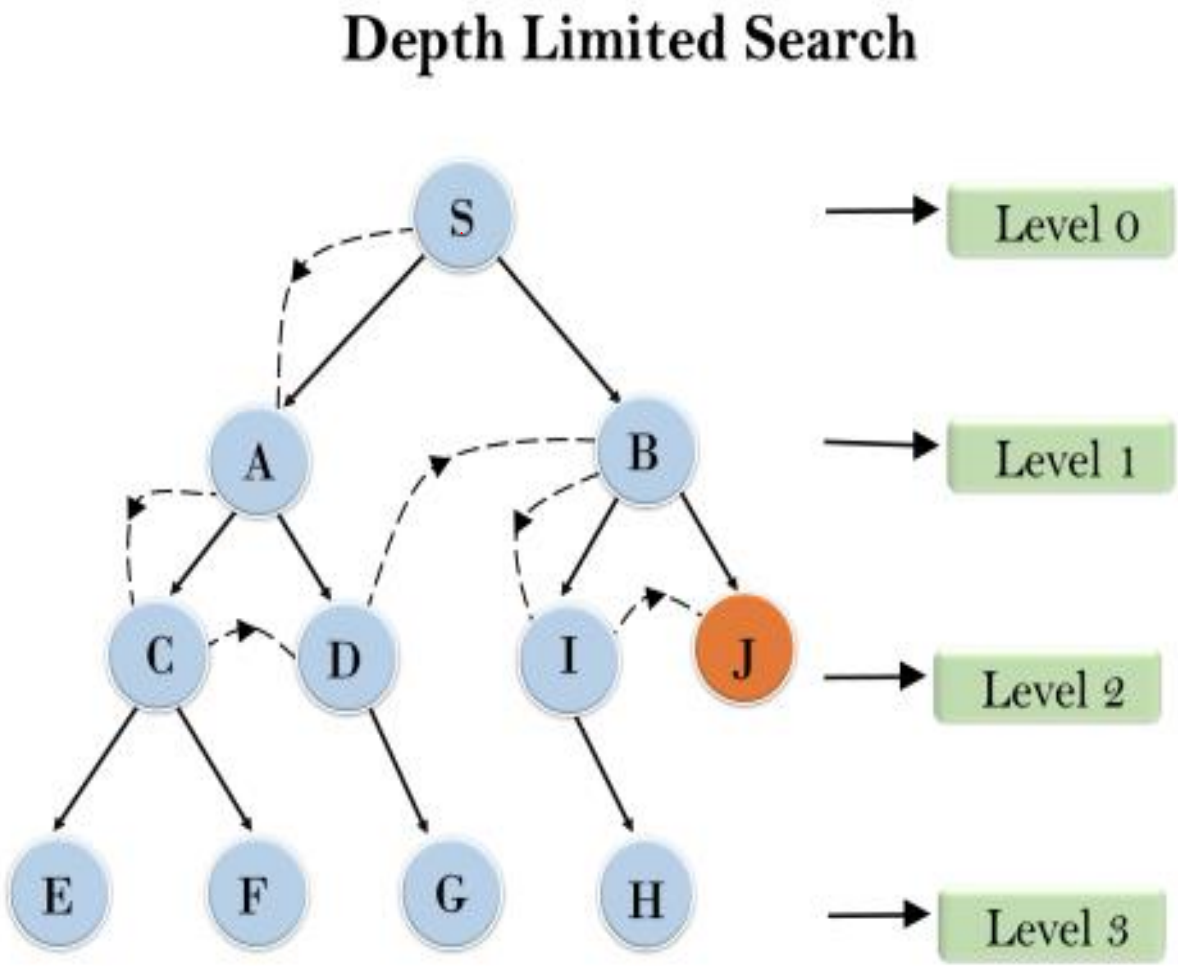
- **Advantage:**
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).
- **Disadvantage:**
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

# Depth Limited Search Algorithm

- The DLS algorithm is one of the uninformed strategies. A depth limited search is close to DFS to some extent. It can find the solution to the demerit of DFS. The nodes at the depth may behave as if no successor exists at the depth. Depth-limited search can be halted in two cases:
- **SFV:** The Standard failure value which tells that there is no solution to the problem.
- **CFV:** The Cut off failure value tells that there is no solution within the given depth.

# Depth Limited Search Algorithm

- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **O($b^{\ell}$)**.
- **Space Complexity:** Space complexity of DLS algorithm is O**(b×ℓ)**.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.
- If ℓ =2 then and element is J then the Search order is
- **S---> A--->C--->D---->B--->I --->J**

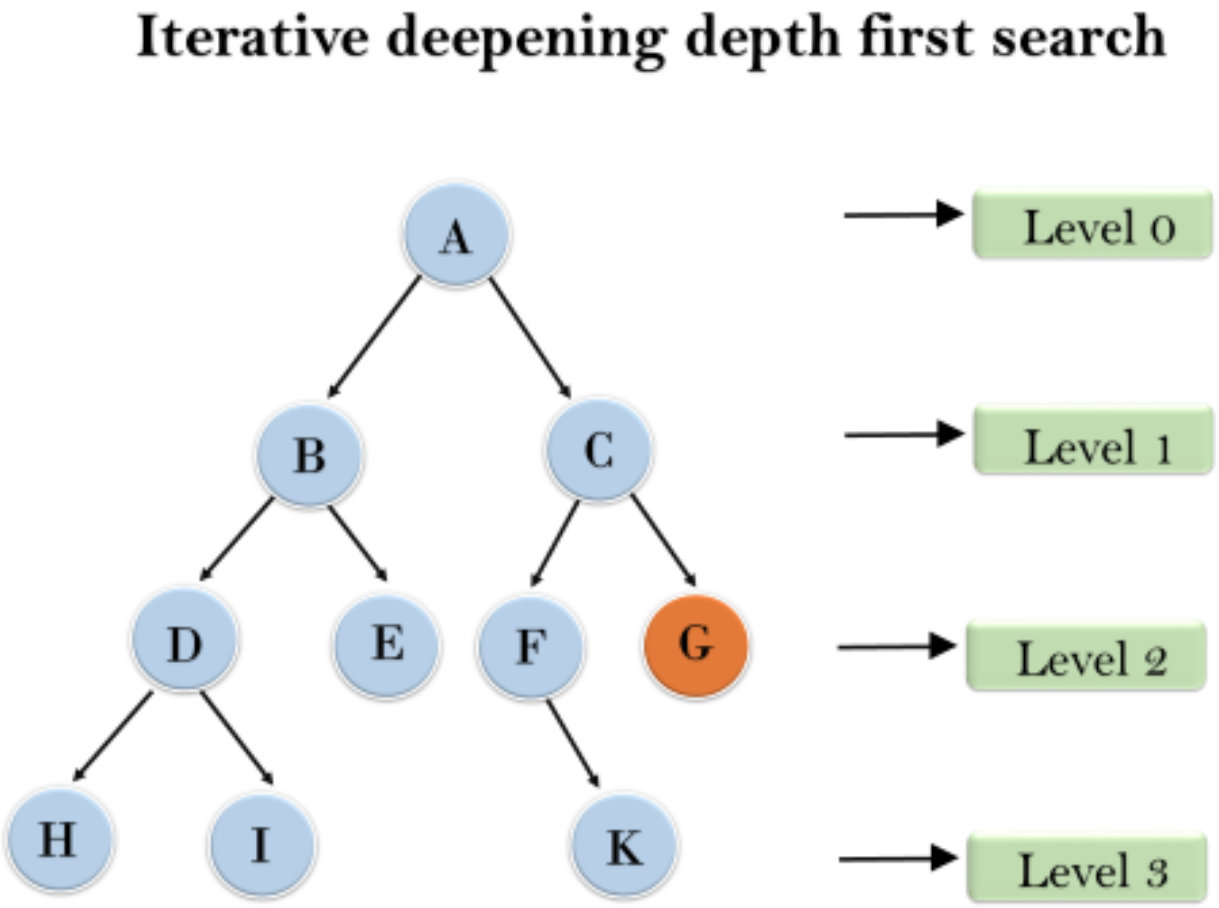


# Iterative deepening depth-first Search(DFS):

- it is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

# Iterative deepening depth-first Search(DFS):

- Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:
- 1st Iteration-----> A
  2'nd Iteration----> A, B, C
  3'rd Iteration------>A, B, D, E, C, F, G
  4'th Iteration------>A, B, D, H, I, E, C, F, K, G
  **In the Third iteration, the algorithm will find the goal node.**

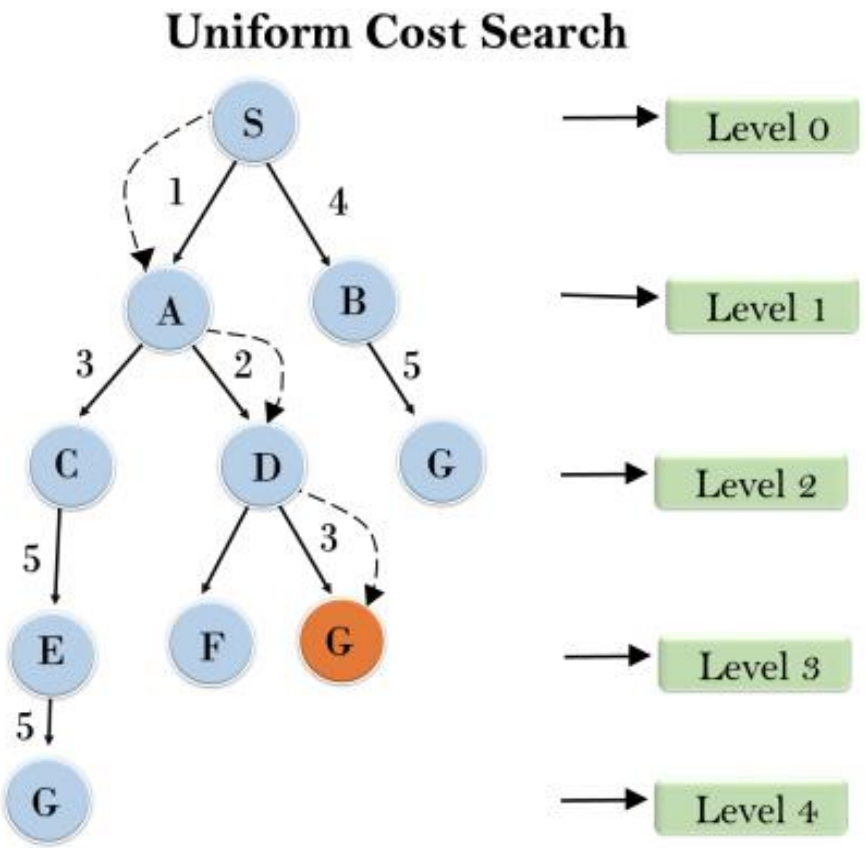


# Iterative deepening depth-first Search(DFS):

- **Completeness:** This algorithm is complete is ifthe branching factor is finite.
- **Time Complexity:** Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.
- **Space Complexity:** The space complexity of IDDFS will be **O(bd)**.
- **Optimal:** it is optimal if path cost is a non- decreasing function of the depth of the node.
- **Advantages:**
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.
- **Disadvantages:** it repeats all the work of the previous phase.

# Uniform-cost Search Algorithm:

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand.
- **Advantages:** it is optimal because at every state the path with the least cost is chosen.
- **Disadvantages:**
- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.




# Uniform-cost Search Algorithm:

- **Completeness:** Uniform-cost search is complete, such as if there is a solution, UCS will find it.
- **Time Complexity:** Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
- Hence, the worst-case time complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})/}$.
- **Space Complexity:** The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.
- **Optimal:** Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

# 3.3 Heuristic search (Informed search)

- Heuristic Search Uses domain-dependent (heuristic) information in order to search the space more efficiently.
- ***Ways of using heuristic information:***
- Deciding which node to expand next, instead of doing the expansion in a strictly breadth-first or depth-first order;
- In the course of expanding a node, deciding which successor or successors to generate, instead of blindly generating all possible successors at one time;
- Deciding that certain nodes should be discarded, or *pruned*, from the search space.

# Heuristic Searches - Why Use?

- It may be too resource intensive (both time and space) to use a blind search
- Even if a blind search will work we may want a more efficient search method
- Informed Search uses domain specific information to improve the search pattern
  - **Define a heuristic function, h(n), that estimates the "goodness" of a node n.**
  - Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
  - **The heuristic function is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal.**

# 3.3.1 Hill climbing

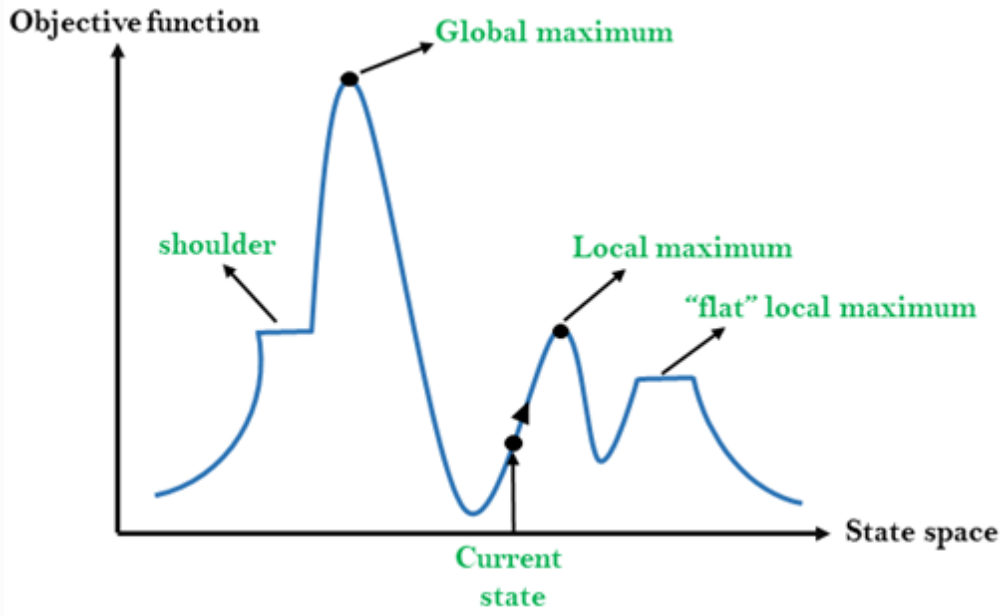
- Hill climbing can be used to solve problems that have many solutions, some of which are better than others.
- **It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates.**
- Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

# Features of Hill Climbing:

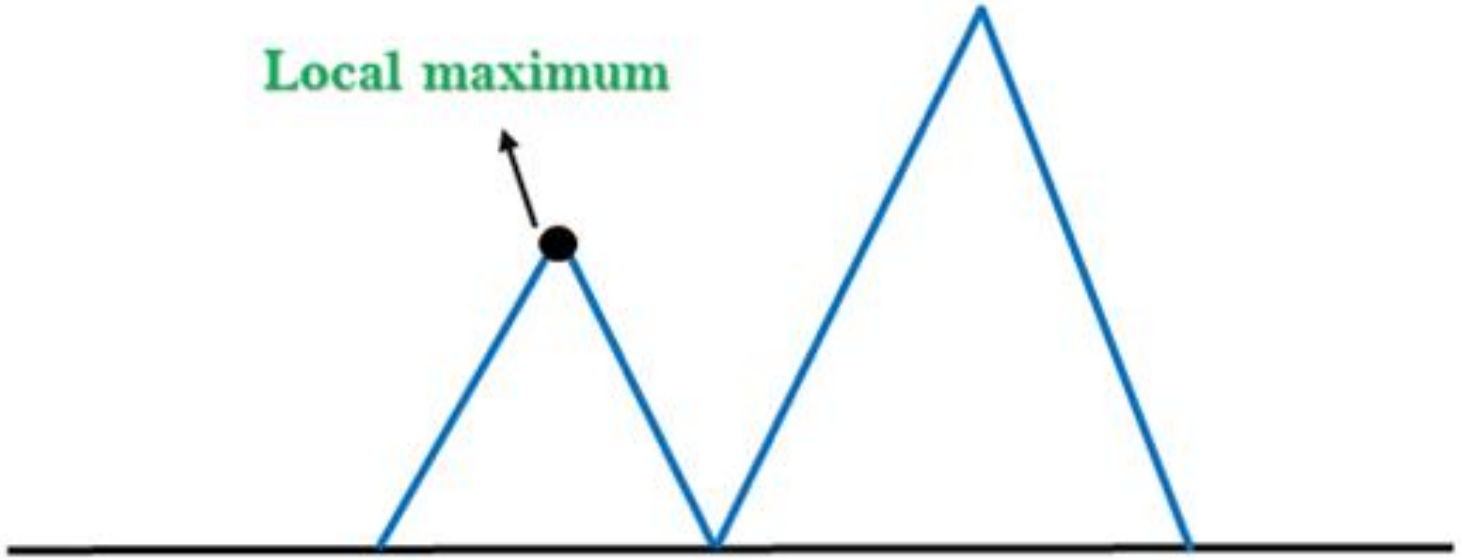
- **Following are some main features of Hill Climbing Algorithm:**
- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

# State-space Diagram for Hill Climbing:

- **Local Maximum:** it is a state which is better than its neighbour states, but there is also another state which is higher than it.
- **Global Maximum:** it is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbour states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.

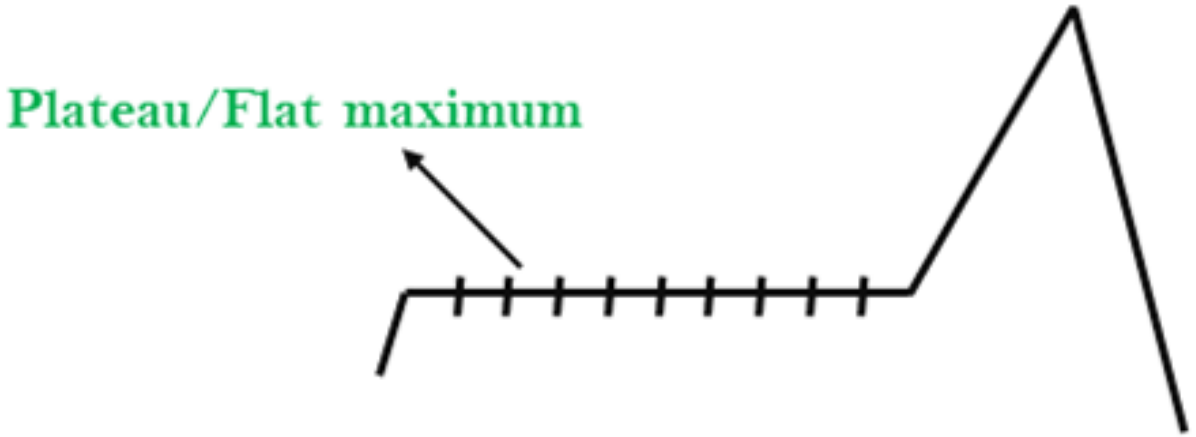


## 3.3.1 The Foothills Problem(Local Maximum:)



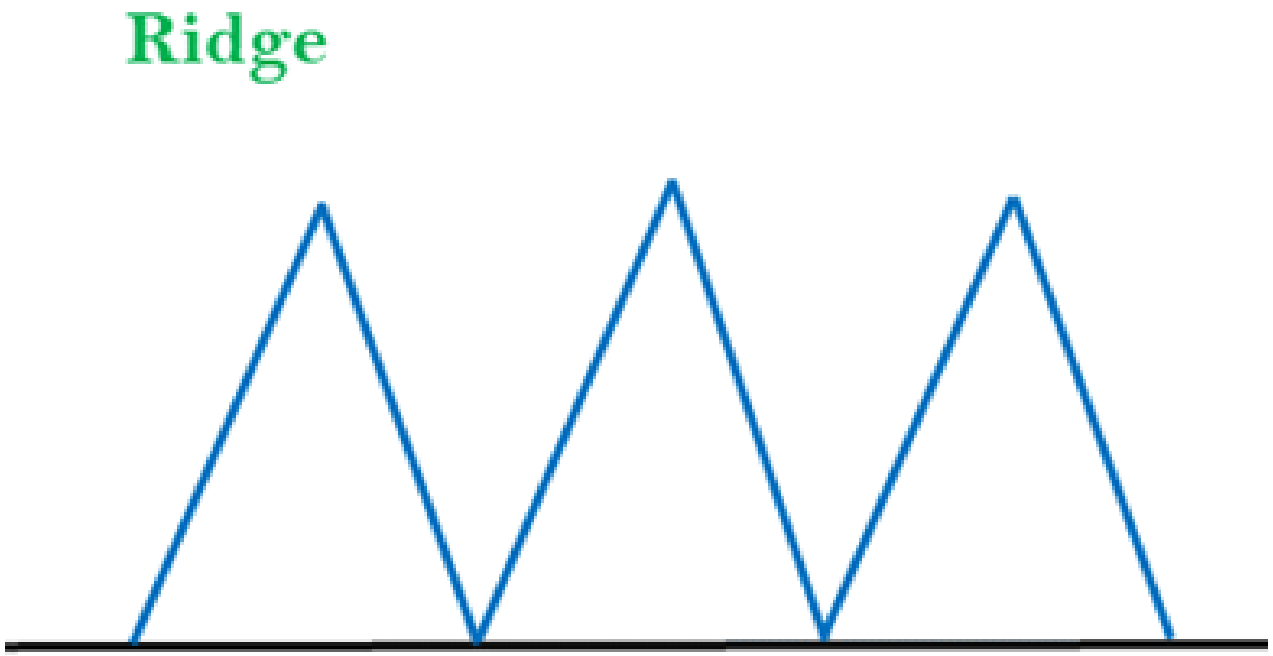

- A local maximum is a peak state in the landscape which is better than each of its neighbouring states, but there is another state also present which is higher than the local maximum.
- **Solution:** Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

# 3.3.2 The Plateau Problem




- **A plateau** is the flat area of the search space in which **all the neighbor states of the current state contains the same value,** because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.
- **Solution:** The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.

# 3.3.3 The Ridge Problem




- **A ridge** is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.
- **Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.

# Pure Heuristic Search:

- Pure heuristic search is the simplest form of heuristic search algorithms.
- It expands nodes based on their heuristic value h(n). It maintains two lists, **OPEN** and **CLOSED** list. In the ***CLOSED list, it places those nodes which have already expanded*** and ***in the OPEN list, it places nodes which have yet not been expanded***.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.
- In the informed search we will discuss two main algorithms which are given below:
- **Best First Search Algorithm(Greedy search)**
- **A* Search Algorithm**

# 3.3.2 Greedy (Best-first) search

- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search.
- With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

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

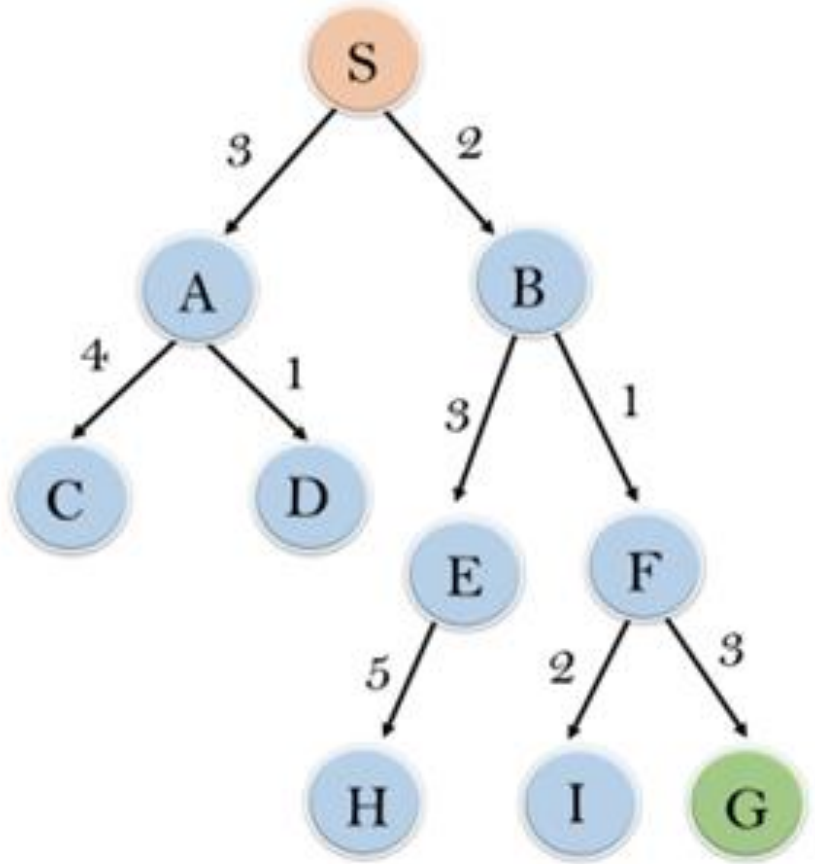
- Were, h(n)= estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

# Best first search algorithm:

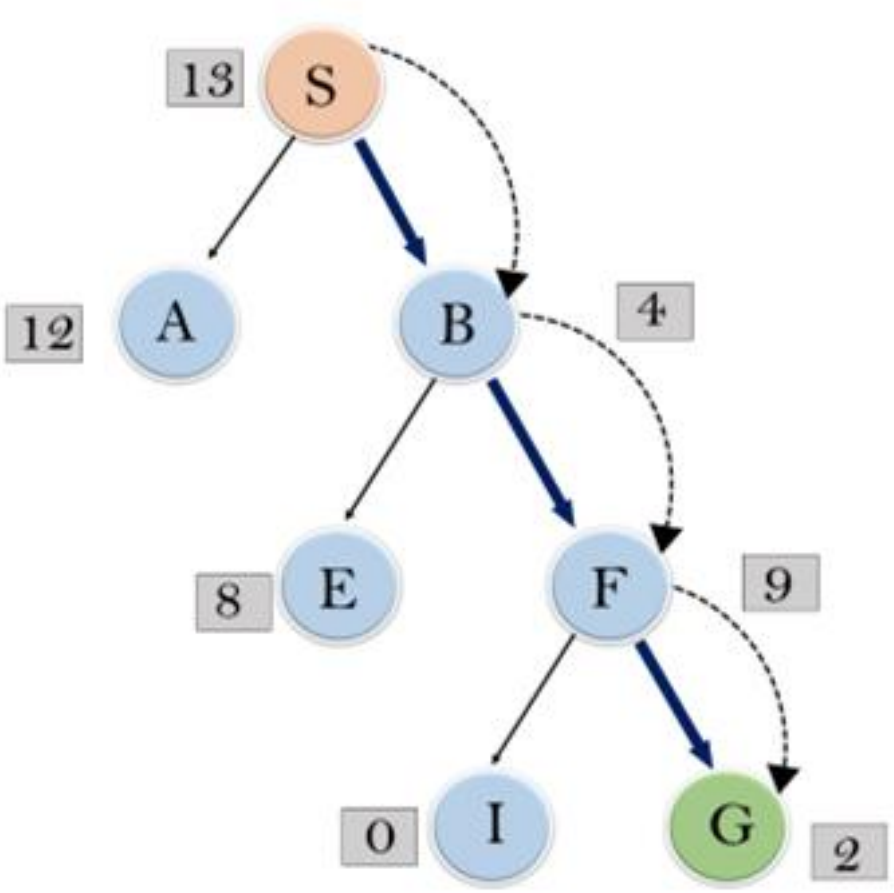
- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- **Step 4:** Expand the node n, and generate the successors of node n.
- **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

# Best first search algorithm:

- Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.




| node | H (n) |
|---|---|
| A | 12 |
| B | 4 |
| C | 7 |
| D | 3 |
| E | 8 |
| F | 2 |
| H | 4 |
| I | 9 |
| S | 13 |
| G | 0 |

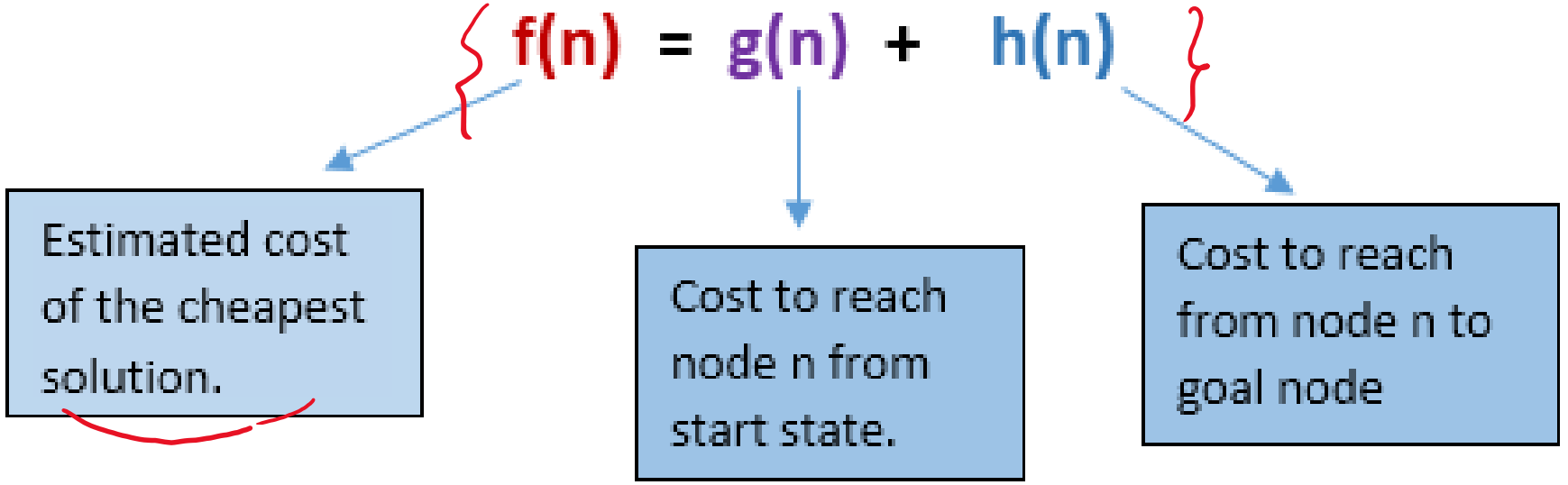



# Best first search algorithm:

- In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.
- **Expand the nodes of S and put in the CLOSED list**
- **Initialization:** Open [A, B], Closed [S]
- **Iteration 1:** Open [A], Closed [S, B]
- **Iteration 2:** Open [E, F, A], Closed [S, B]
  : Open [E, A], Closed [S, B, F]
- **Iteration 3:** Open [I, G, E, A], Closed [S, B, F]
  : Open [I, E, A], Closed [S, B, F, G]
- Hence the final solution path will be: **S----> B----->F----> G**

# Best first search algorithm:

- **Time Complexity:** The worst case time complexity of Greedy best first search is $O(b^m)$.
- **Space Complexity:** The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.
- **Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.
- **Optimal:** Greedy best first search algorithm is not optimal.




## 3.3.3 A* algorithm (search)

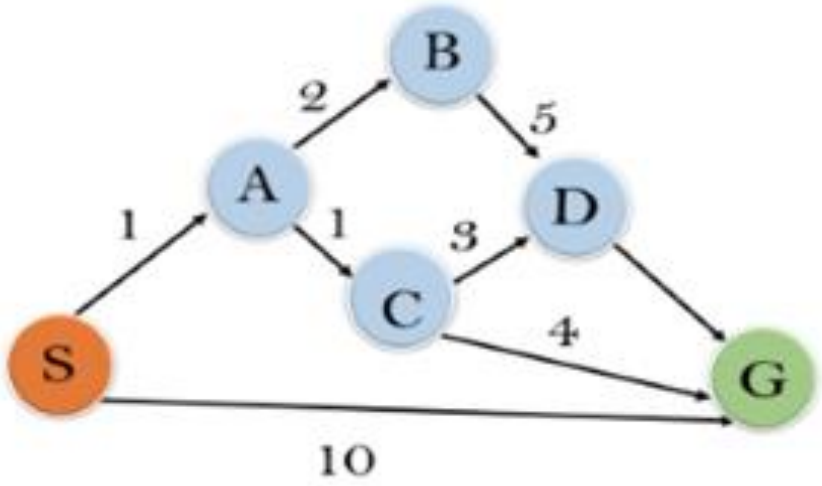
- A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- A* search algorithm finds the shortest path through the search space using the heuristic function. expands less search tree and provides optimal result faster. it is similar to UCS except that it uses f(n)=g(n)+h(n) instead of g(n) called **fitness number**.

# Algorithm of A* search:

- **Step1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
- **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- **Step 6:** Return to **Step 2**.

# A* Search Algorithm:

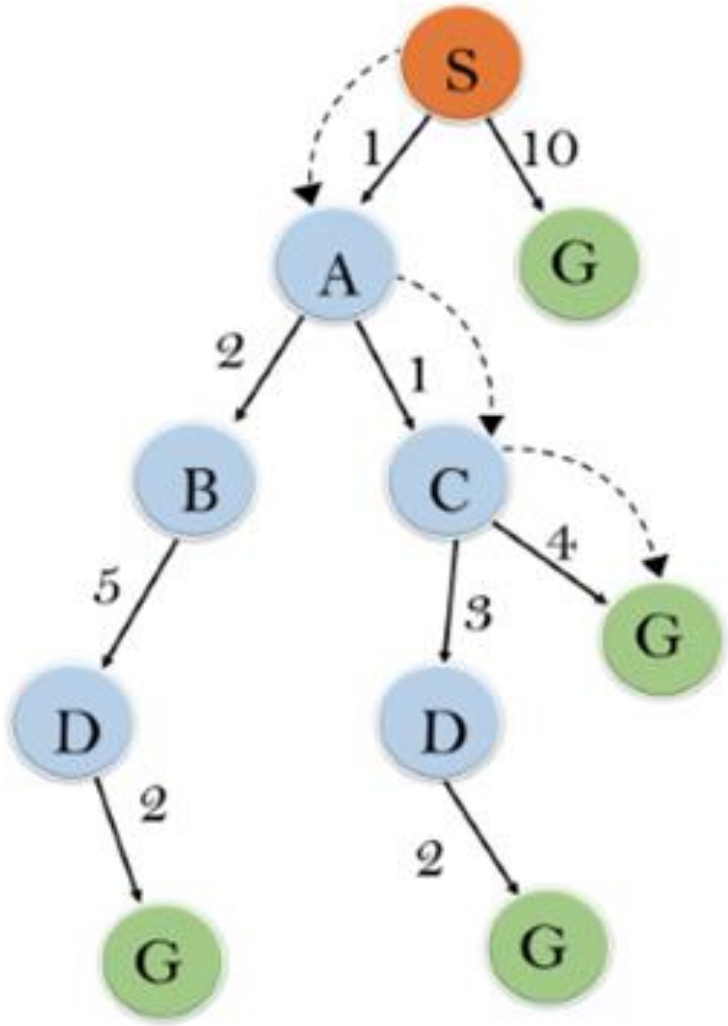
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.




| State | h(n) |
|---|---|
| S | 5 |
| A | 3 |
| B | 4 |
| C | 2 |
| D | 6 |
| G | 0 |

# A* Search Algorithm:

- **Initialization:** {(S, 5)}
- **Iteration1:** {(S--> A, 4), (S-->G, 10)}
- **Iteration2:** {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration3:** {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
- **Iteration 4** will give the final result, as **S--->A-->C--->G** it provides the optimal path with cost 6.



# A* Search Algorithm:

- **Complete:** A* algorithm is complete as long as: Branching factor is finite, Cost at every action is fixed.
- **Optimal:** A* search algorithm is optimal if it follows below two conditions:
- **Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A* graph-search.
- If the heuristic function is admissible, then A* tree search will always find the least cost path.
- **Time Complexity:** The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

# **General Problem Solving (GPS):** Problem solving agents

Slide no : 1 , 2, 3

# 3.4.1 Constraint satisfaction problem

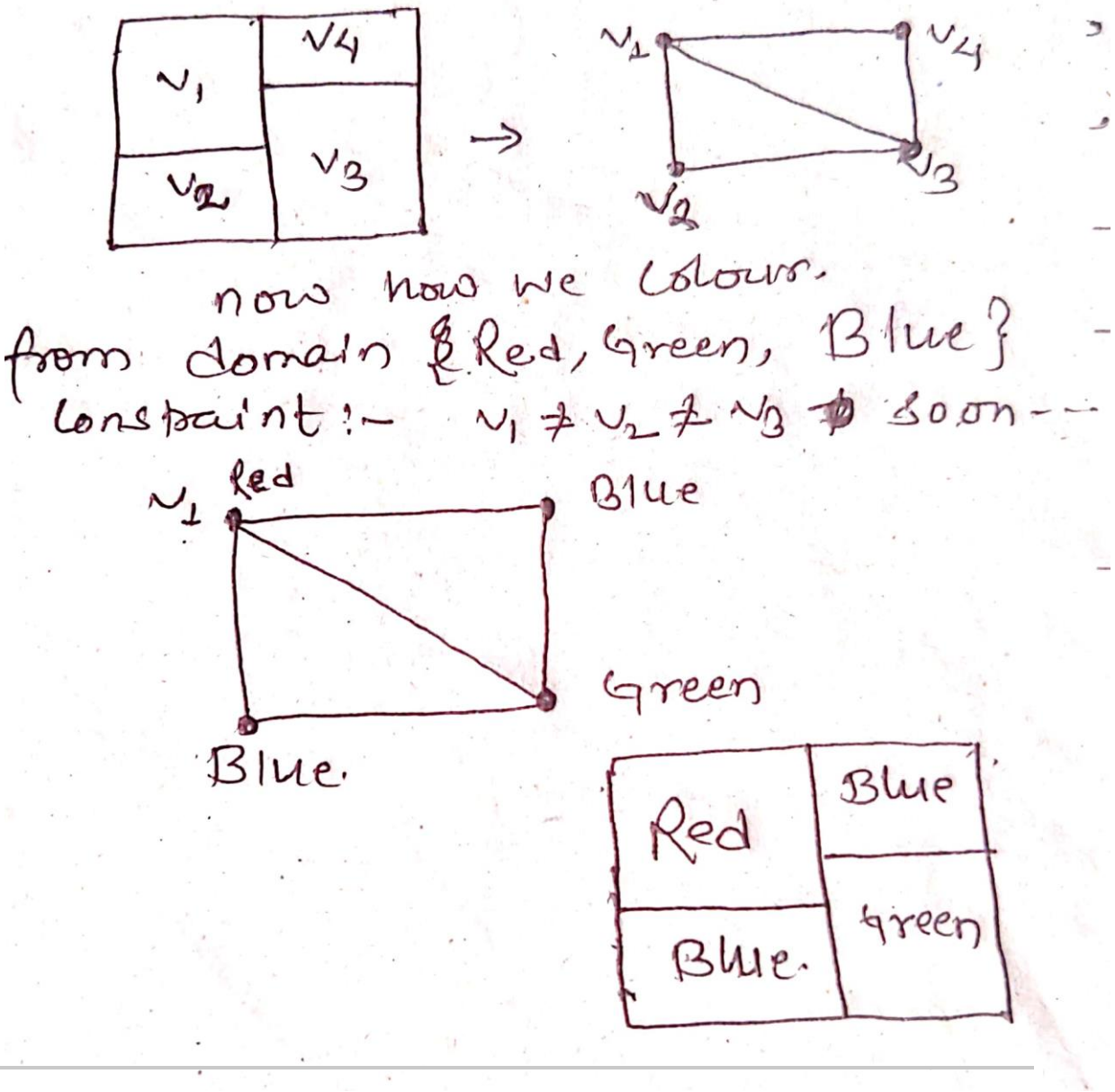
- As originally defined in artificial intelligence, constraints enumerate the possible values a set of variables may take in a given world.
- It is a solution is a way for assigning a value to each variable in such a way that **all constraints are satisfied by these values**.
- CSP is used in variety of problems such as : scheduling ,Satisfactory , Time tabling , supply chain management, graph colouring , machine vision, puzzles etc.

# 3.4.1 Constraint satisfaction problem

- **Formally CSP consist of :**
- A set of variable ,X
- For each variable $X_i$ in X , a domain $D_i$
  - **$D_i$ is a finite set of possible values.**
- A set of constraints restricting touples of values.
  - (if only pairs of values ; it is binary CSP otherwise it is unary)
  - **$X_1 \neq x_7$ (binary) $x_1 \neq 3$ (unary)**
- A solution is ann assignment of value in $D_i$ to each variable $x_i$ such that every constraints satisfied.

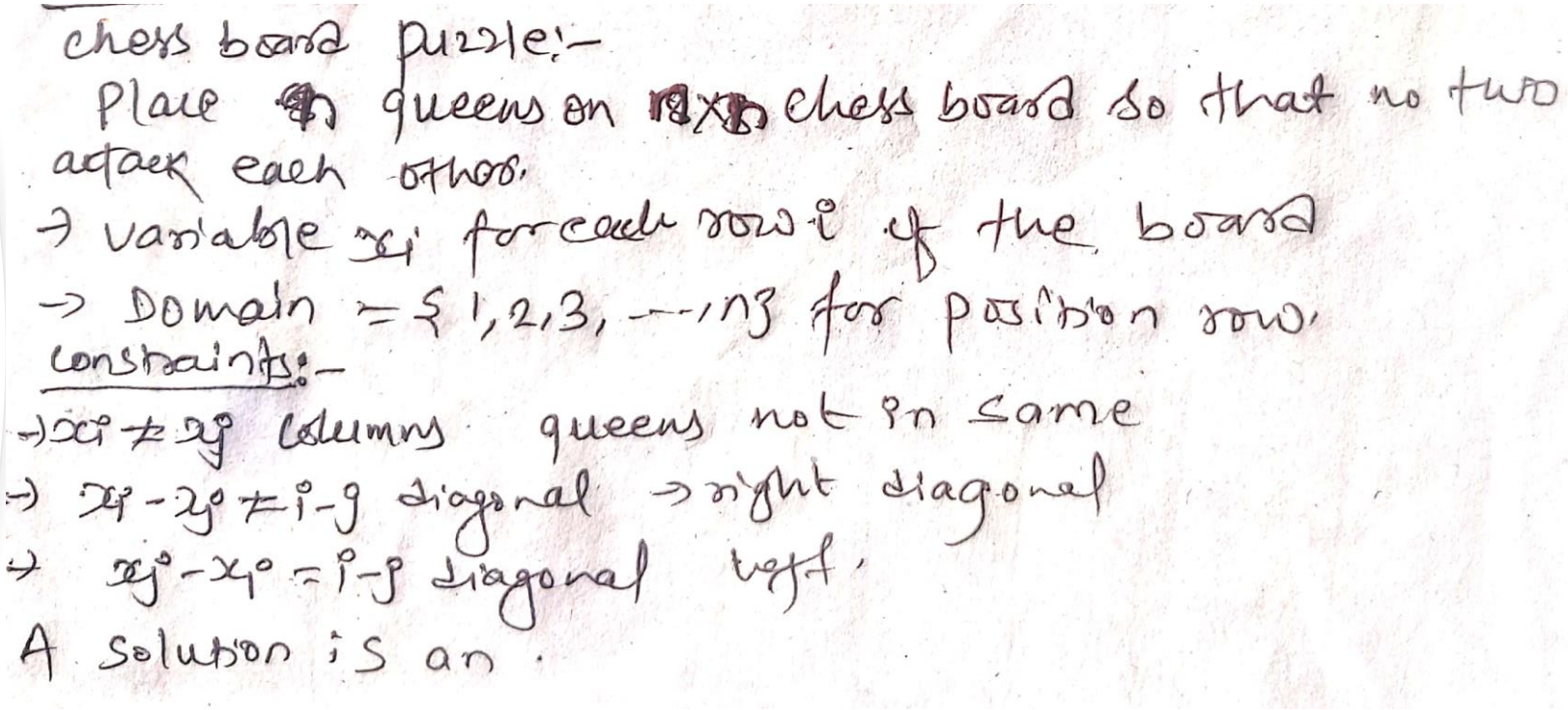
# Colouring as CPS

- We can colour all four region with 3 colours so that no two adjacent region are the same colour.
- **Variable:** four region (node)
- **Domain :** {Red , Green ,Blue}
- **Constraints:** xi ≠ xj
- **Solution :** Colouring the graph

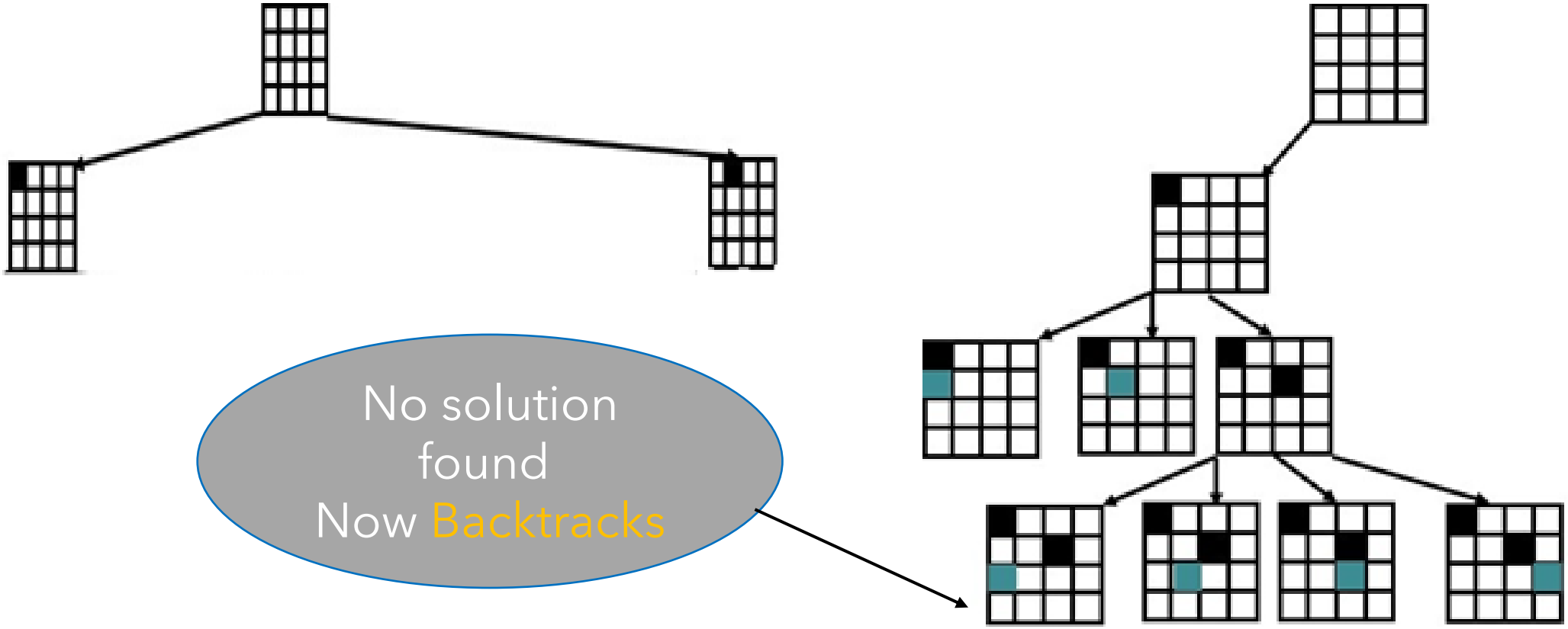


# 3.4.1 Constraint Satisfaction Search

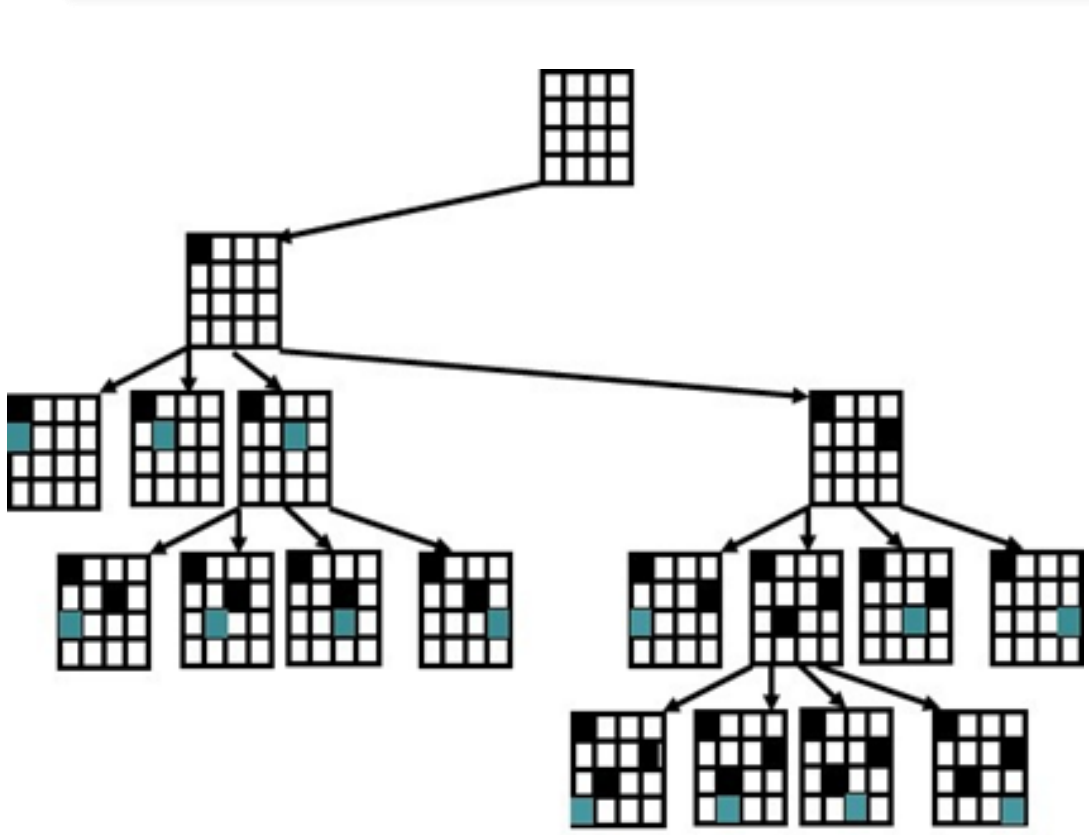
- States are factored into set of variables.
- Search = assigning values to variables.
- Structure of space is encoded with constraints (using backtracking)

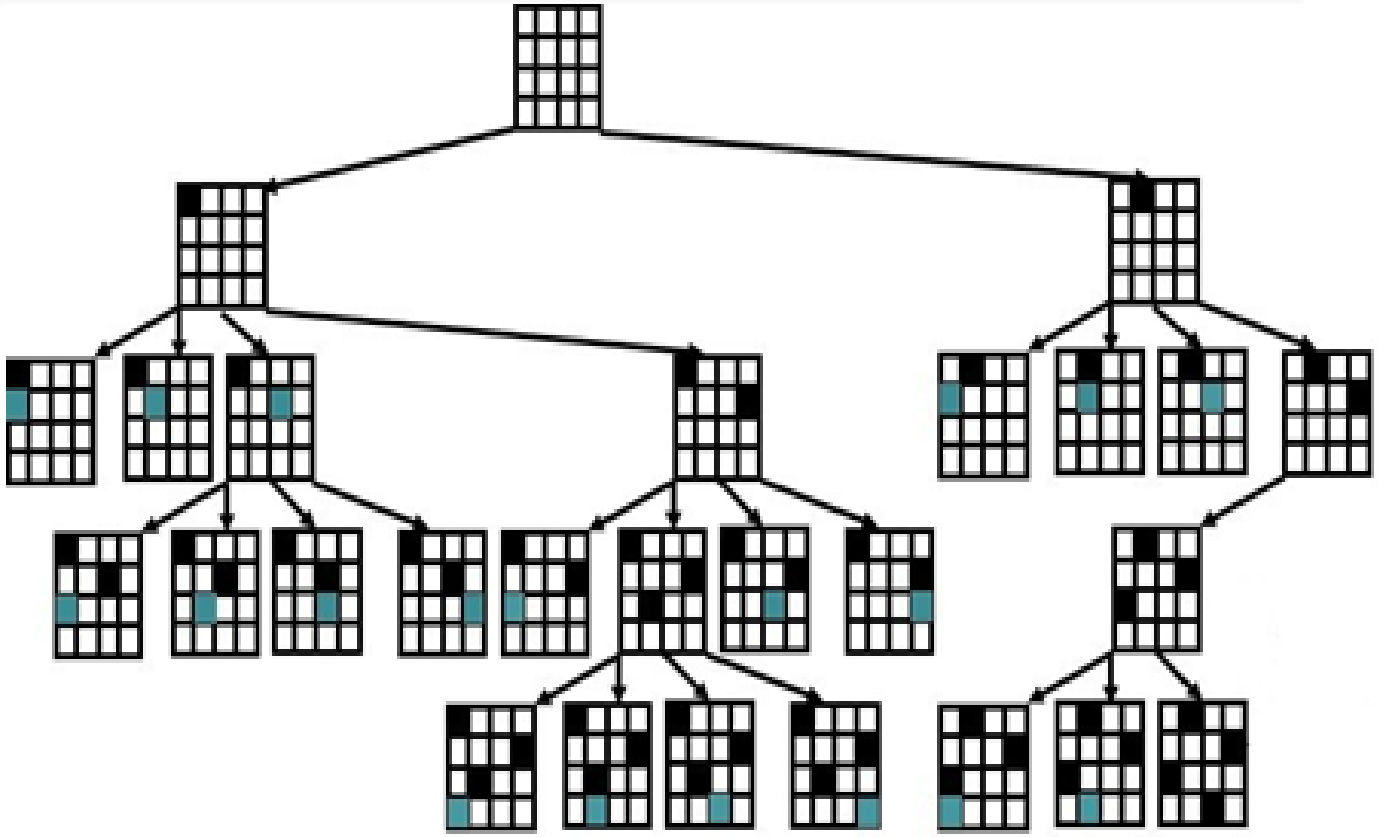


# 4 queens problems (Backtracking)



# 4 queens problems (Backtracking)



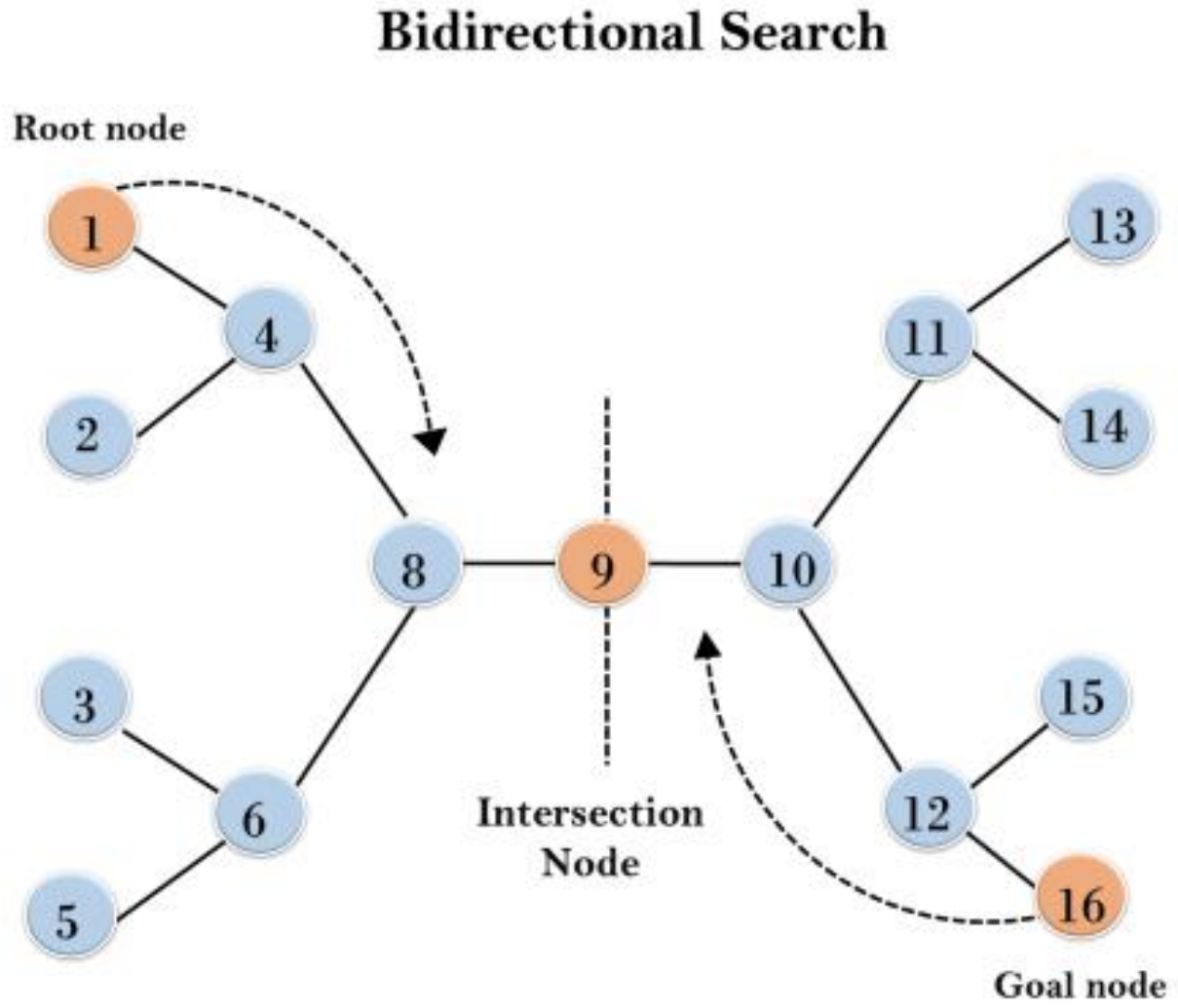


# 3.4.3 The bidirectional search

- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.
- **Advantages:**
- Bidirectional search is fast.
- Bidirectional search requires less memory
- **Disadvantages:**
- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

# The bidirectional search

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.



# The bidirectional search

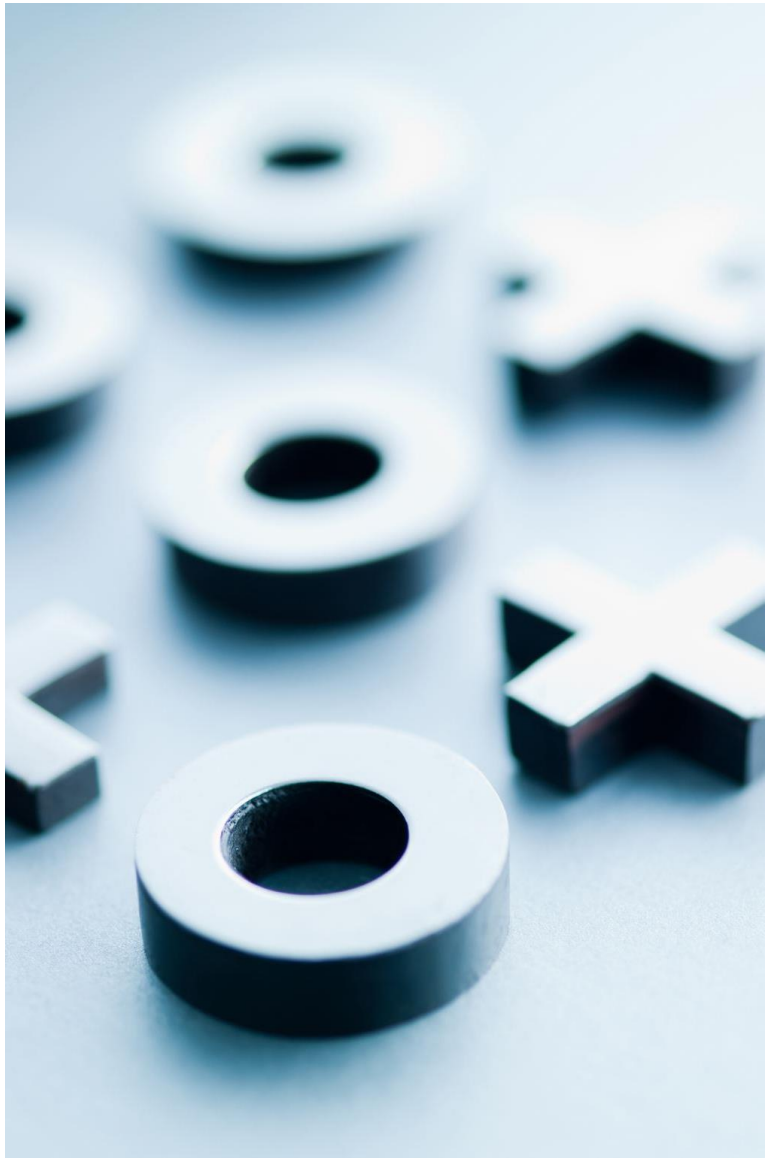
- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is **$O(b^d)$**.
- **Space Complexity:** Space complexity of bidirectional search is **$O(b^d)$**.
- **Optimal:** Bidirectional search is Optimal.

# 3.5 Game playing and AI

- Game Playing is an important domain of AI. Games just require the knowledge of rules, legal moves and the conditions of winning or losing the game.
- Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improve –
- **Generate procedure** so that only good moves are generated.
- **Test procedure** so that the best move can be explored first.

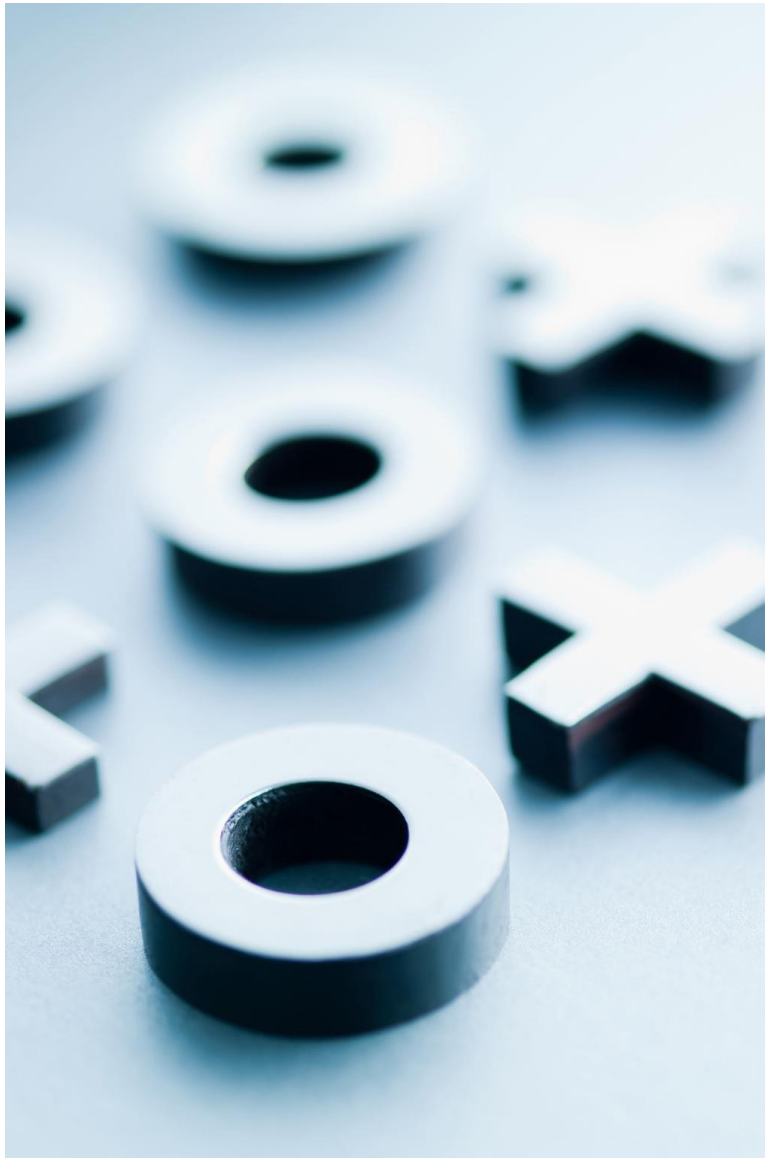
# Game Trees and Minimax Evaluation

- To solve games using AI, we use game tree. The different states of the game are represented by nodes in the game tree.
- In the game tree, the nodes are arranged in levels that correspond to each player's turns in the game so that the “root” node of the tree (usually depicted at the top of the diagram) is the beginning position in the game.
- In tic-tac-toe, this would be the empty grid with no **Xs** or **Os** played yet. Under root, on the second level, there are the possible states that can result from the first player's moves, be it X or O. We call these nodes the “children” of the root node.

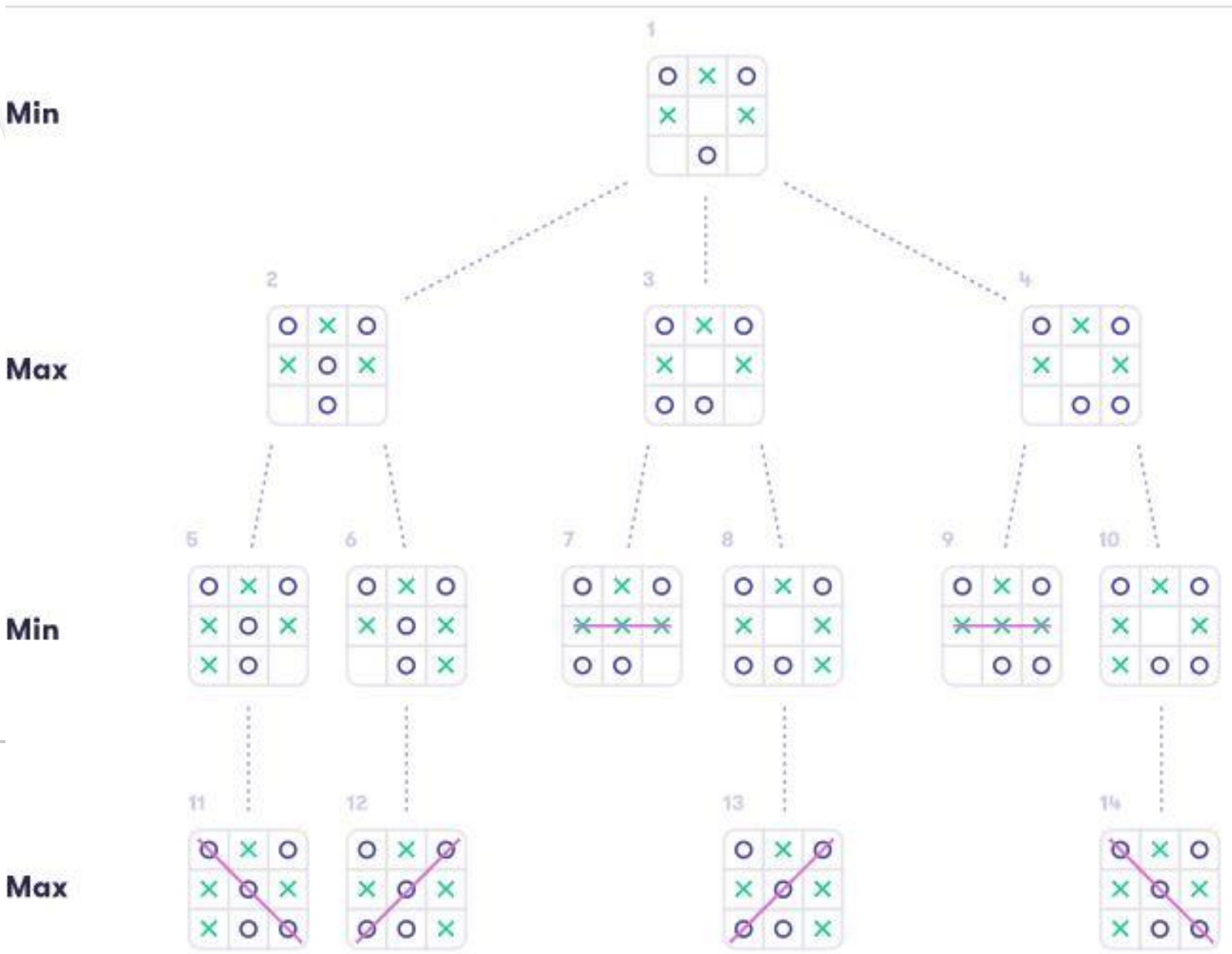


# Minimizing and maximizing value

- In order to be able to create game AI that attempts to win the game, we attach a numerical value to each possible end result.
- To the board positions where X has a line of three so that Max wins, we attach the value +1, and likewise, to the positions where Min wins with three Os in a row we attach the value –1
- For the positions where the board is full and neither player wins, we use the neutral value 0

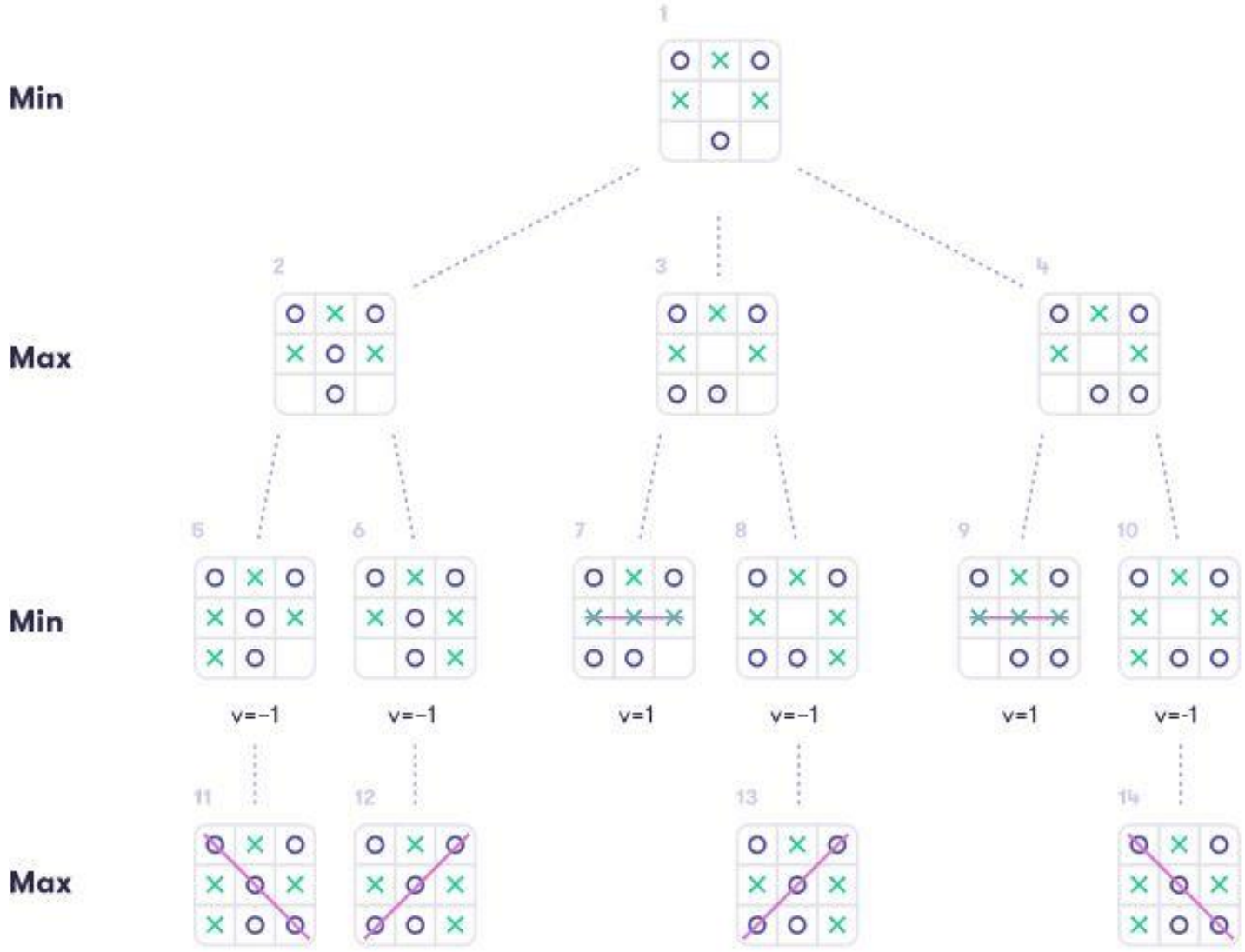


# Game Trees and Minimax Evaluation



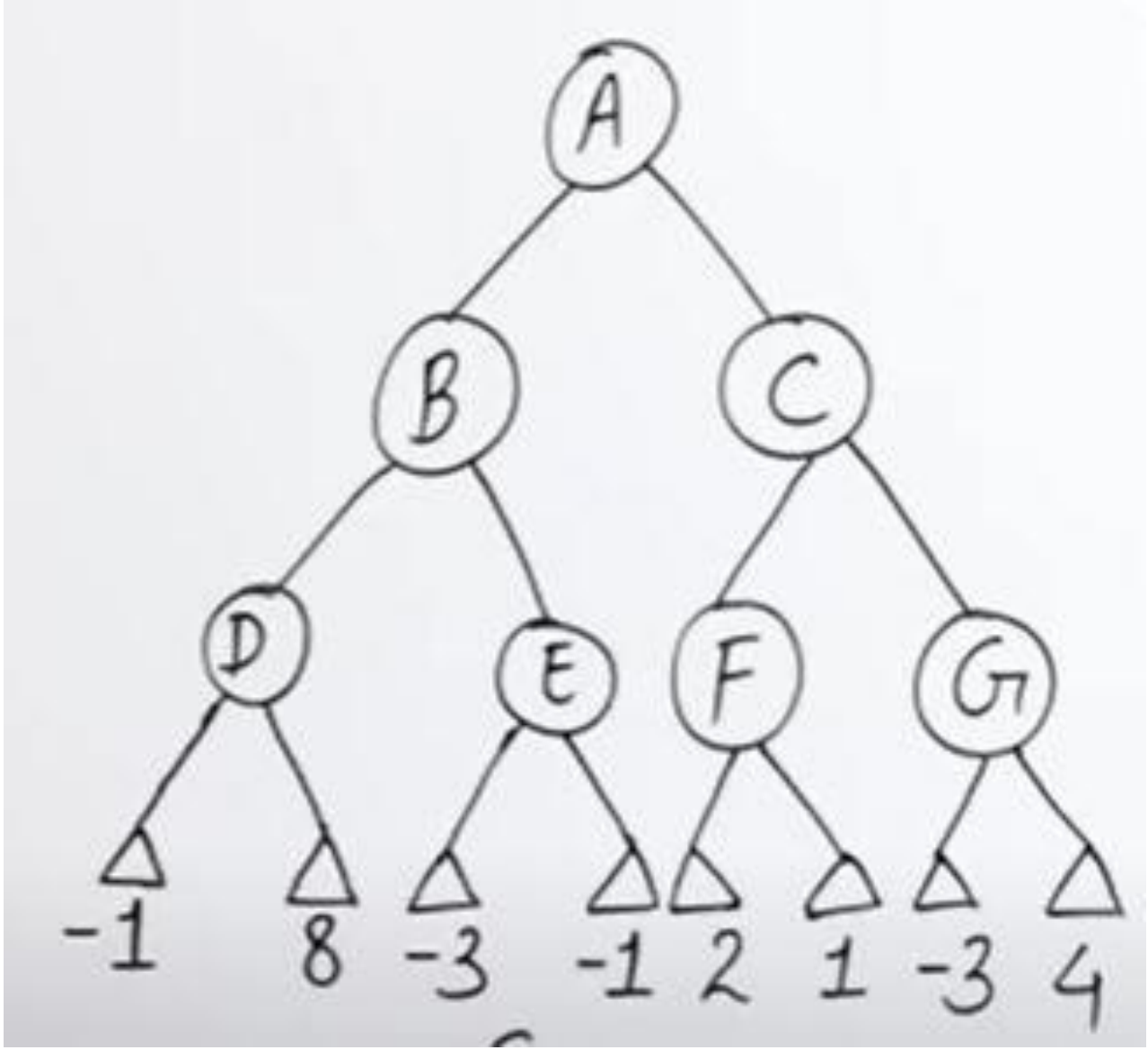
# Game Trees and Minimax Evaluation

In nodes (7) and (9), the game is over, and Max wins with three X's in a row. The value is +1. (5), (6), (8), and (10), the game is also practically over, since Min only needs to place her O in the only remaining cell to win. In other words, we know how the game will end at each node on the second level from the bottom. We can therefore decide that the value of nodes (5), (6), (8), and (10) is also –1.



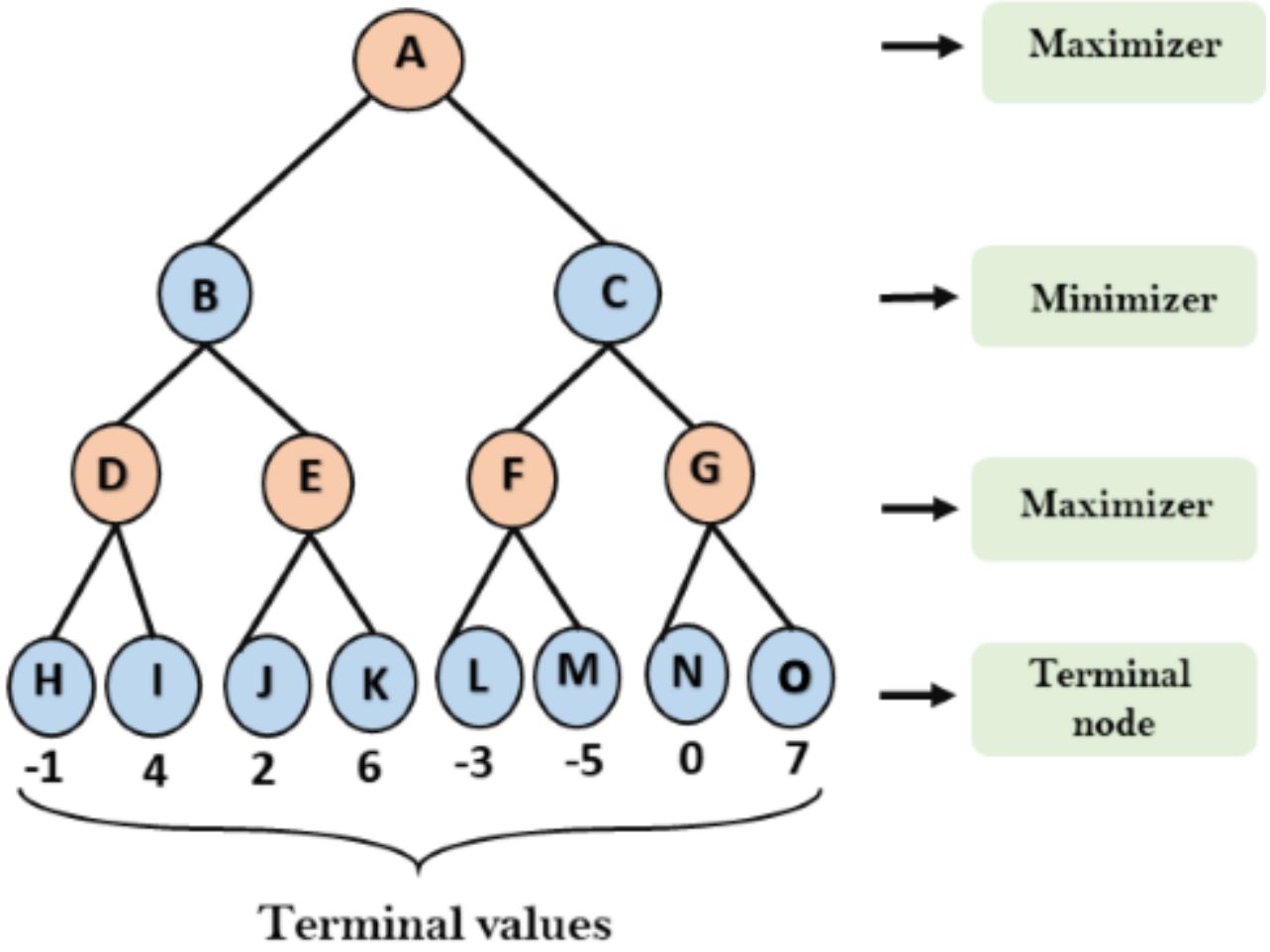
# 3.2.3 Min-max algorithm (search)

- It is backtracking algorithm.
- In min-max serch the best move strategy is used.
- Max will try to maximize its utility(Best Move).
- Min will try to minimize utility(Worst Move).



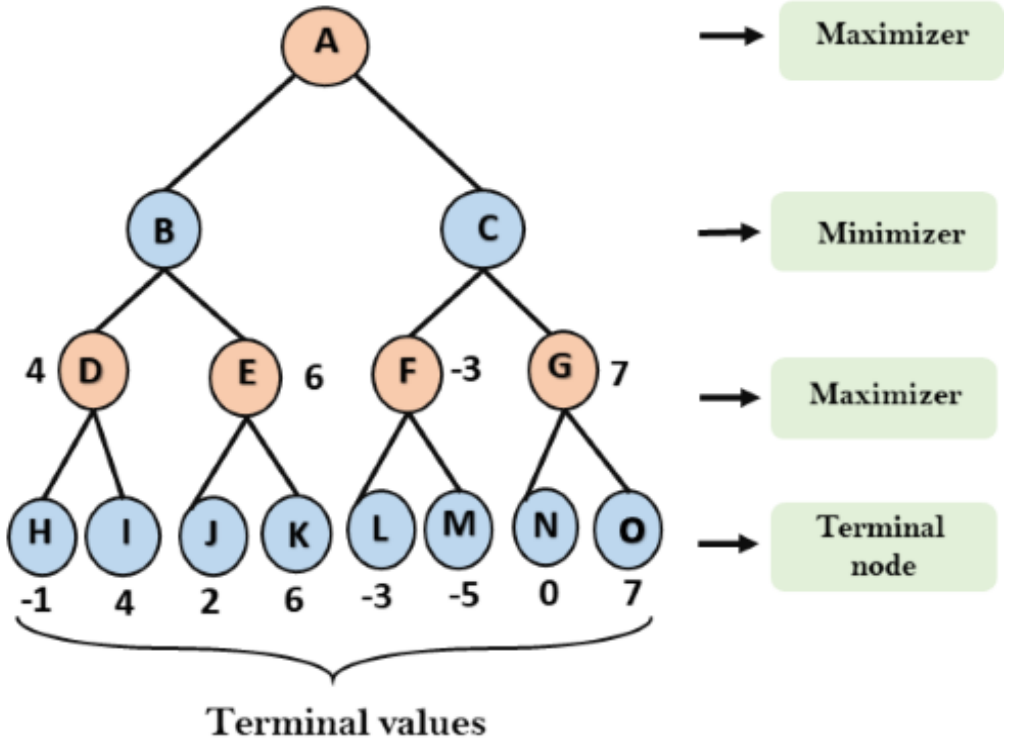
# Example :

- **In the first step,** the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states
- In this diagram, A is the initial state of the tree. Suppose **maximizer** takes first turn which has worst-case initial value =- ∞, and **minimizer** will take next turn which has worst-case initial value = +∞.



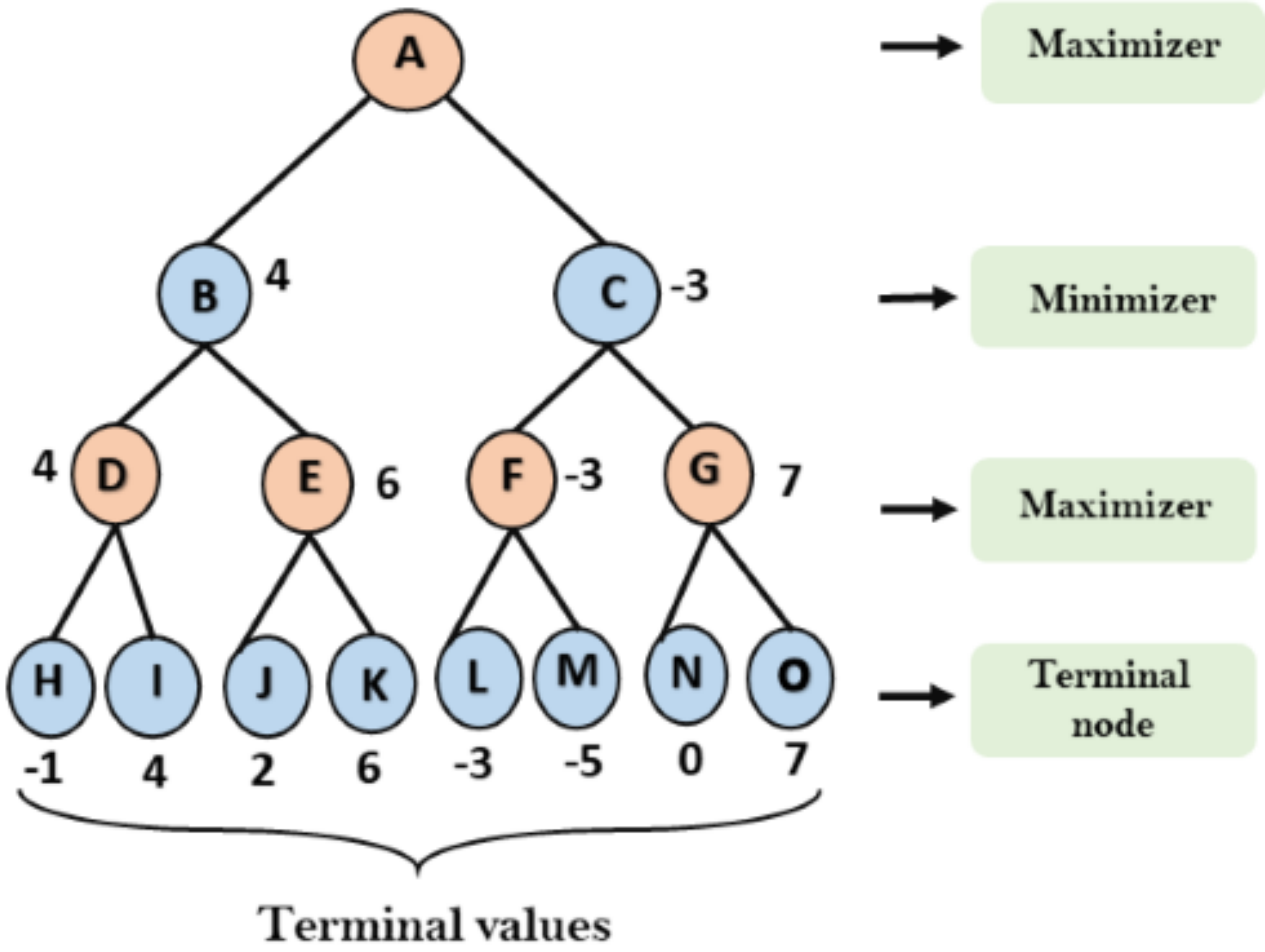
# Example:

- **Step 2:** Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.
- **For node D** max(-1,- -∞) => max(-1,4)= 4
- **For Node E** max(2, -∞) => max(2, 6)= 6
- **For Node F** max(-3, -∞) => max(-3,-5) = -3
- **For node G** max(0, -∞) => max(0, 7) = 7



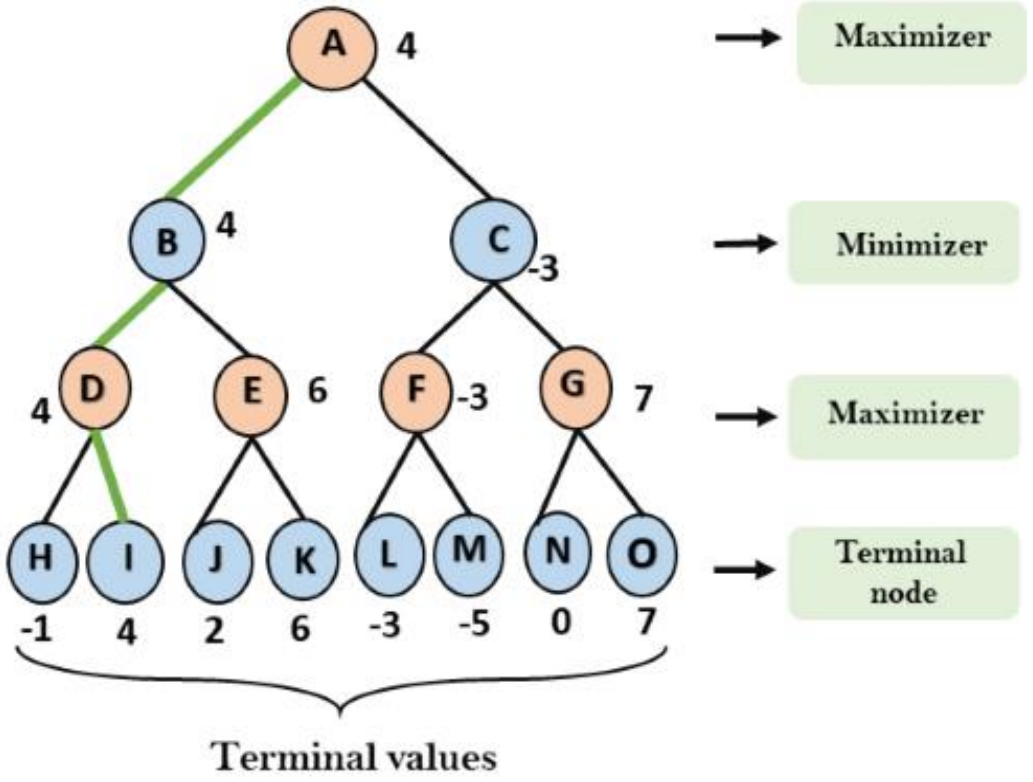
# Example:

- **Step 3:** In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.
- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3

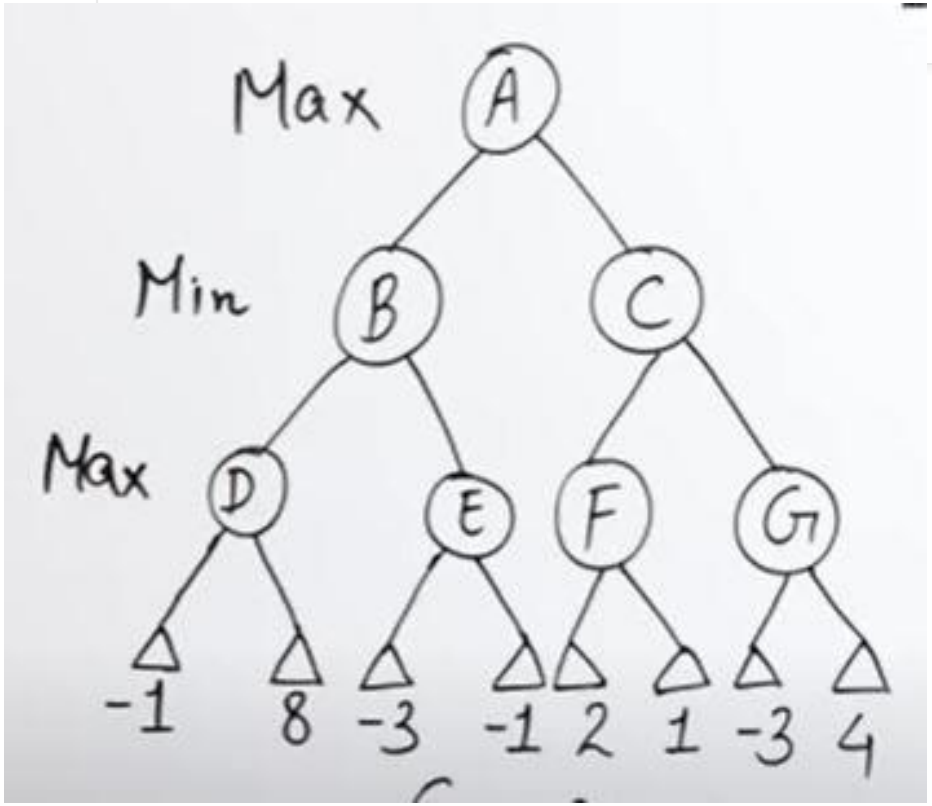


# Example:

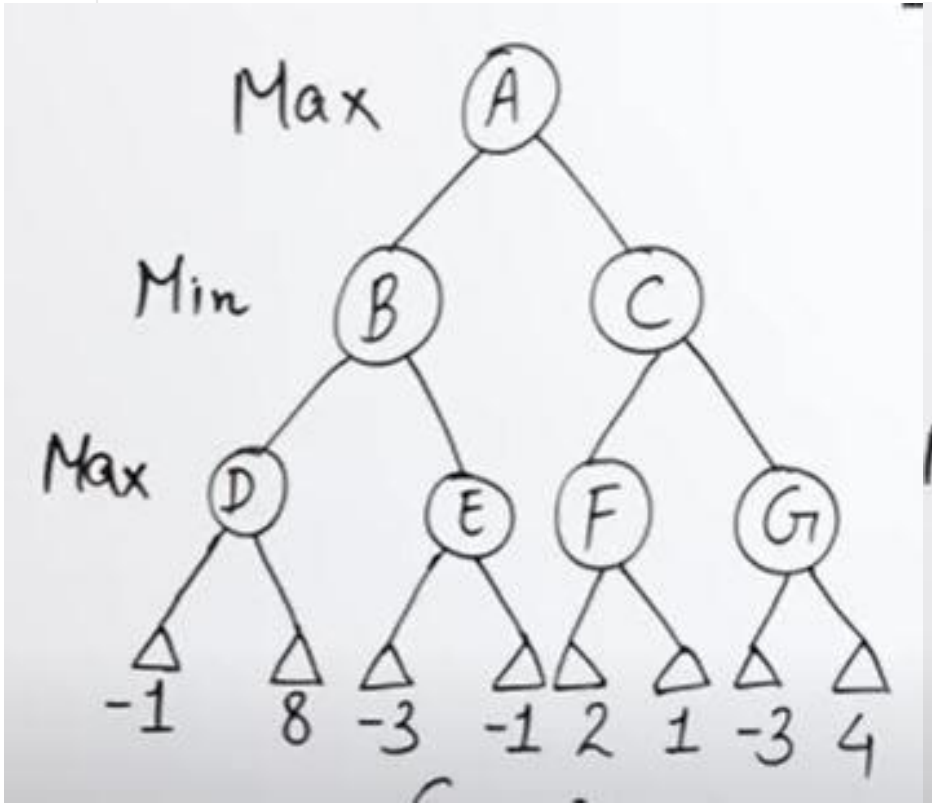
- **Step 3:** Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.
- For node A max(4, -3)= 4

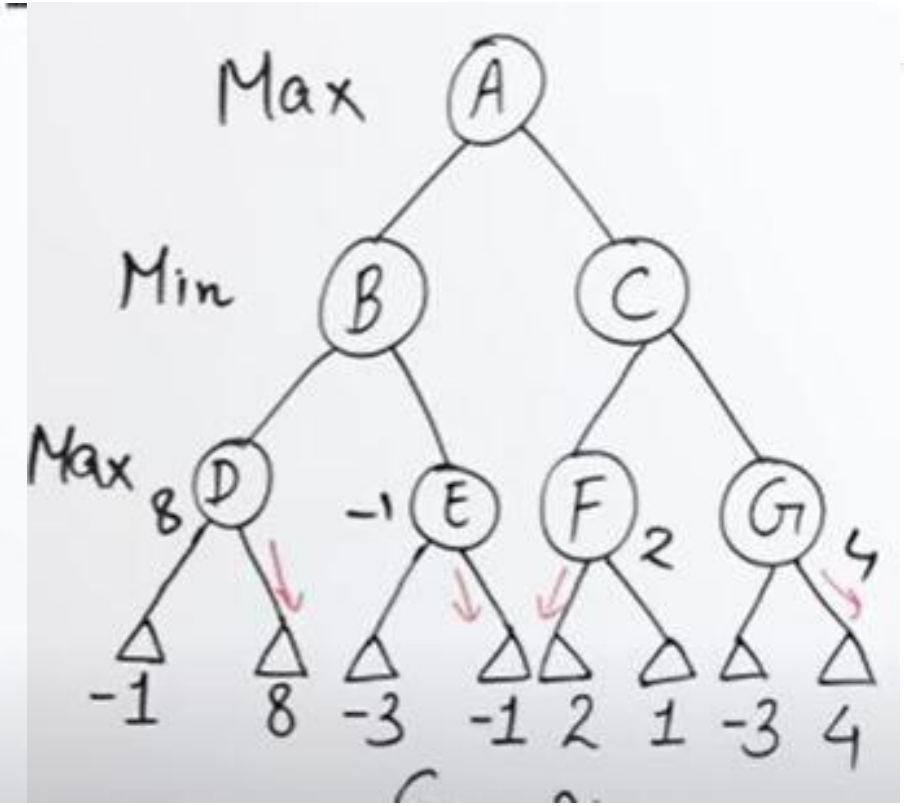


# Try another one:

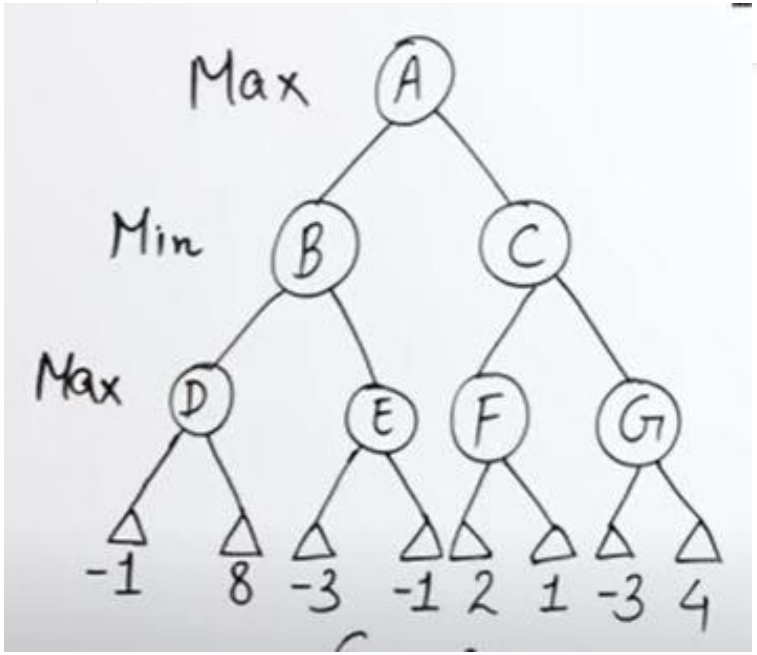


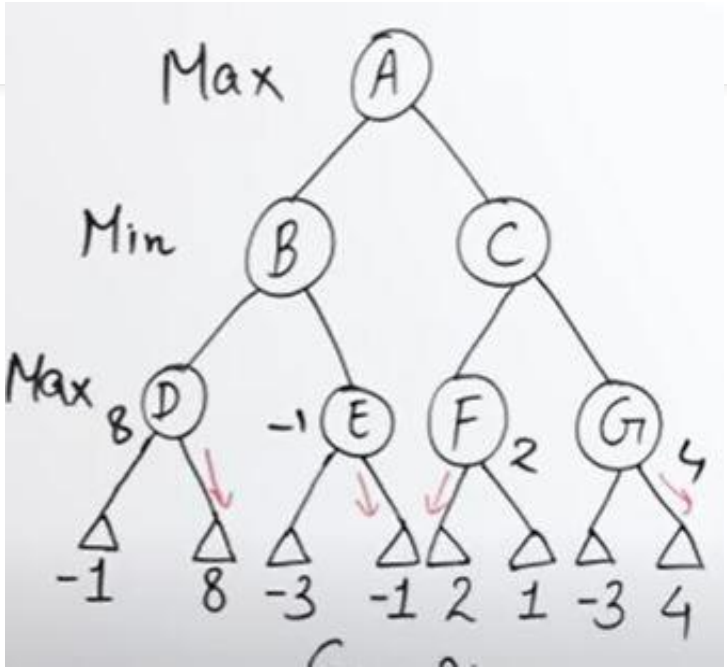
# Try another one:

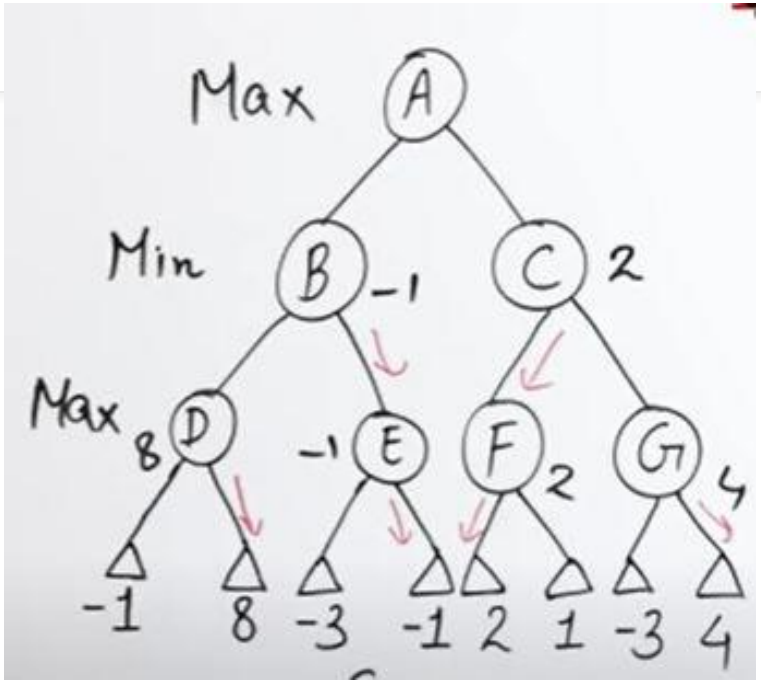


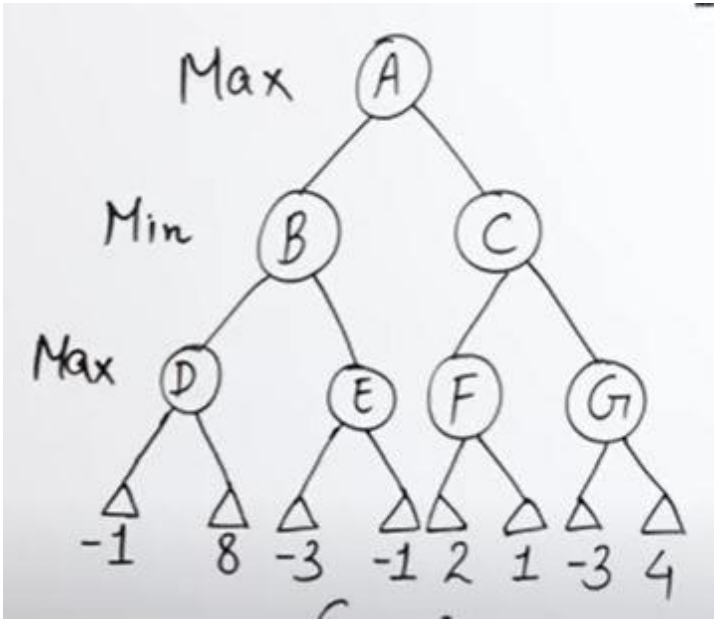
# Try another one:

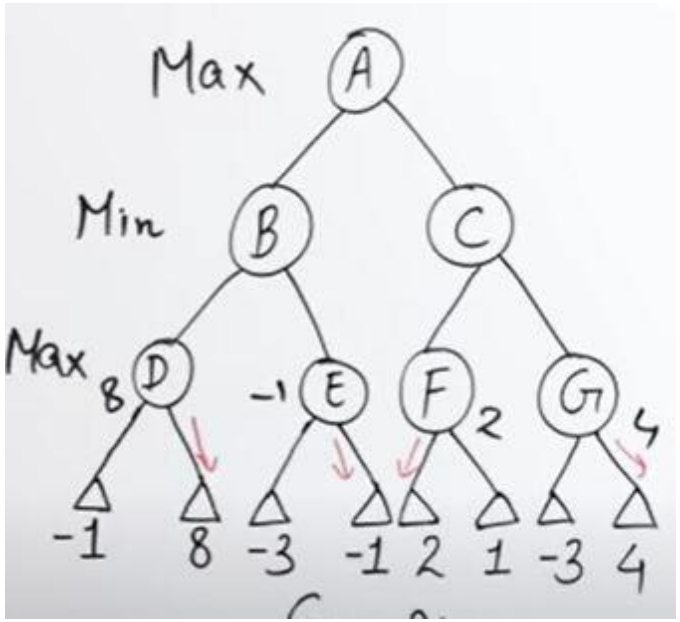


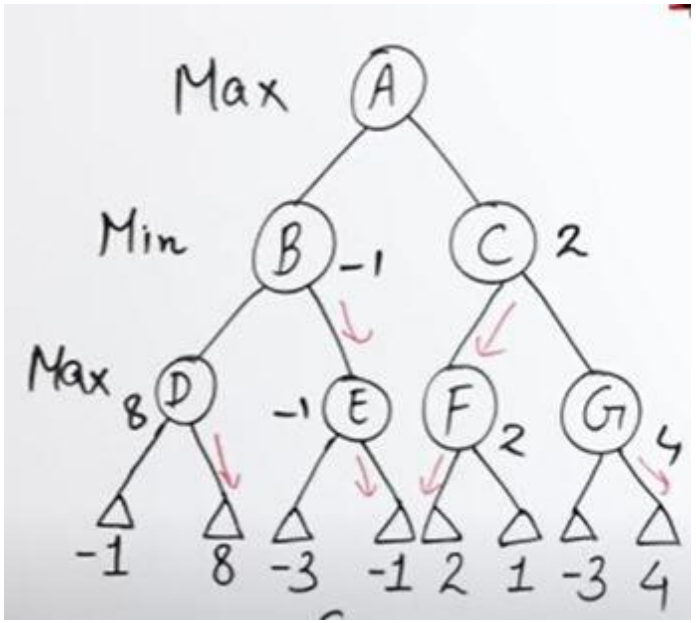



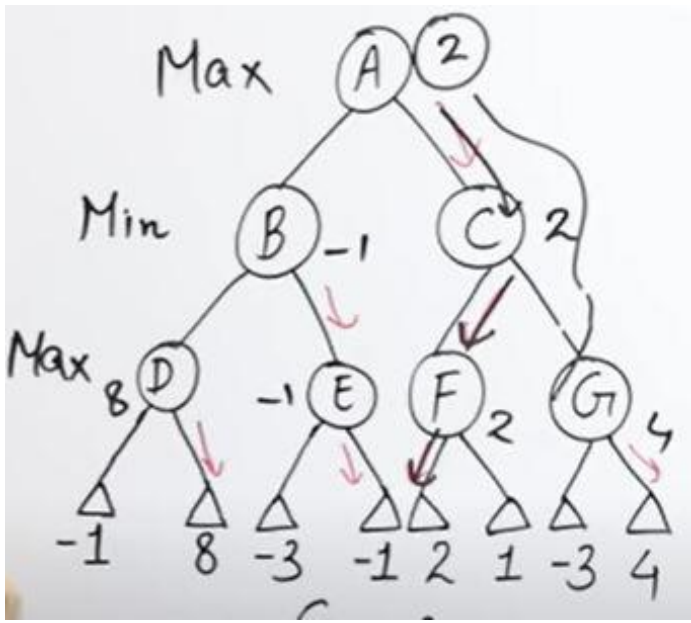



# Try another one:



# Properties of Mini-Max algorithm:

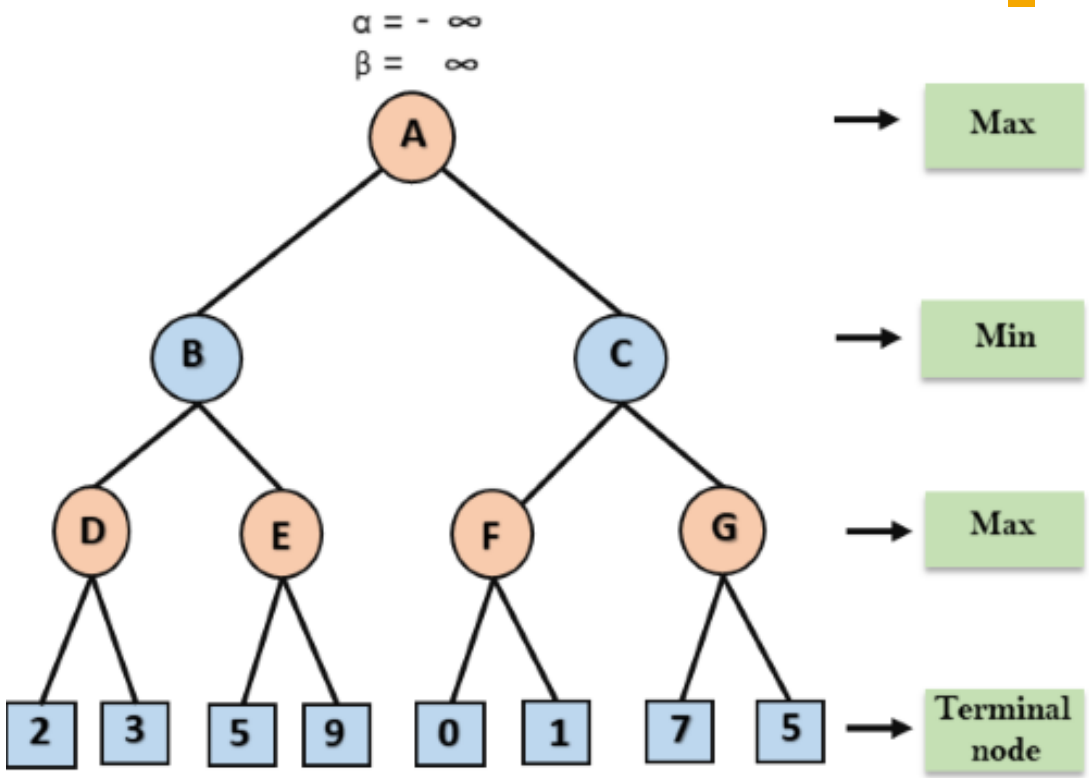
- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

# 3.2.4 Min-max with alpha-beta

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- By using this technique we can minimize the complexity of min max algorithm by taking decision without checking the node which is called **purning**.
- **The two-parameter can be defined as:**
  - **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  - **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.

# Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

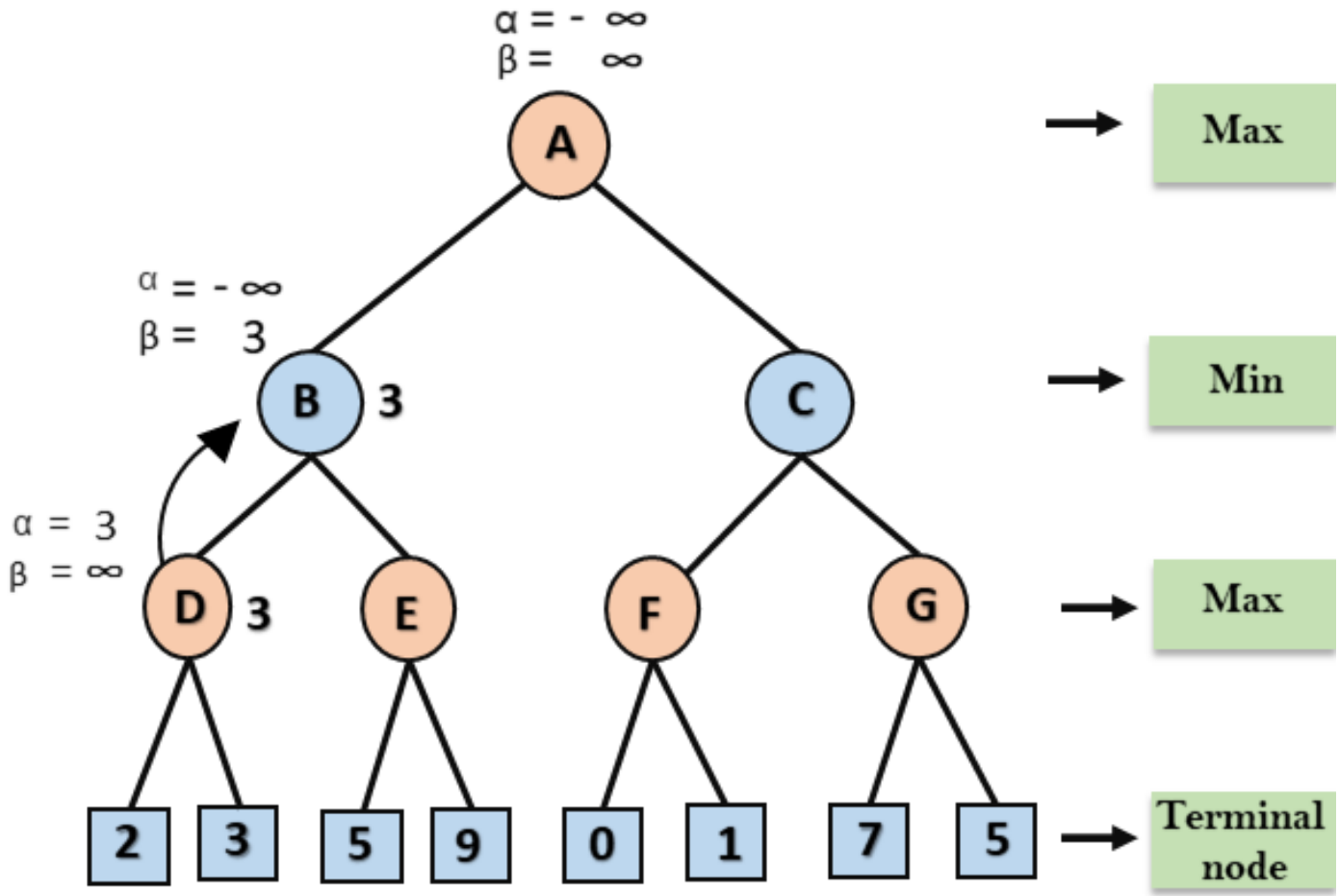



# Working of Alpha-Beta Pruning:

- Let's take an example of two-player search tree to understand the working of Alpha-beta pruning
- **Step 1:** At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.

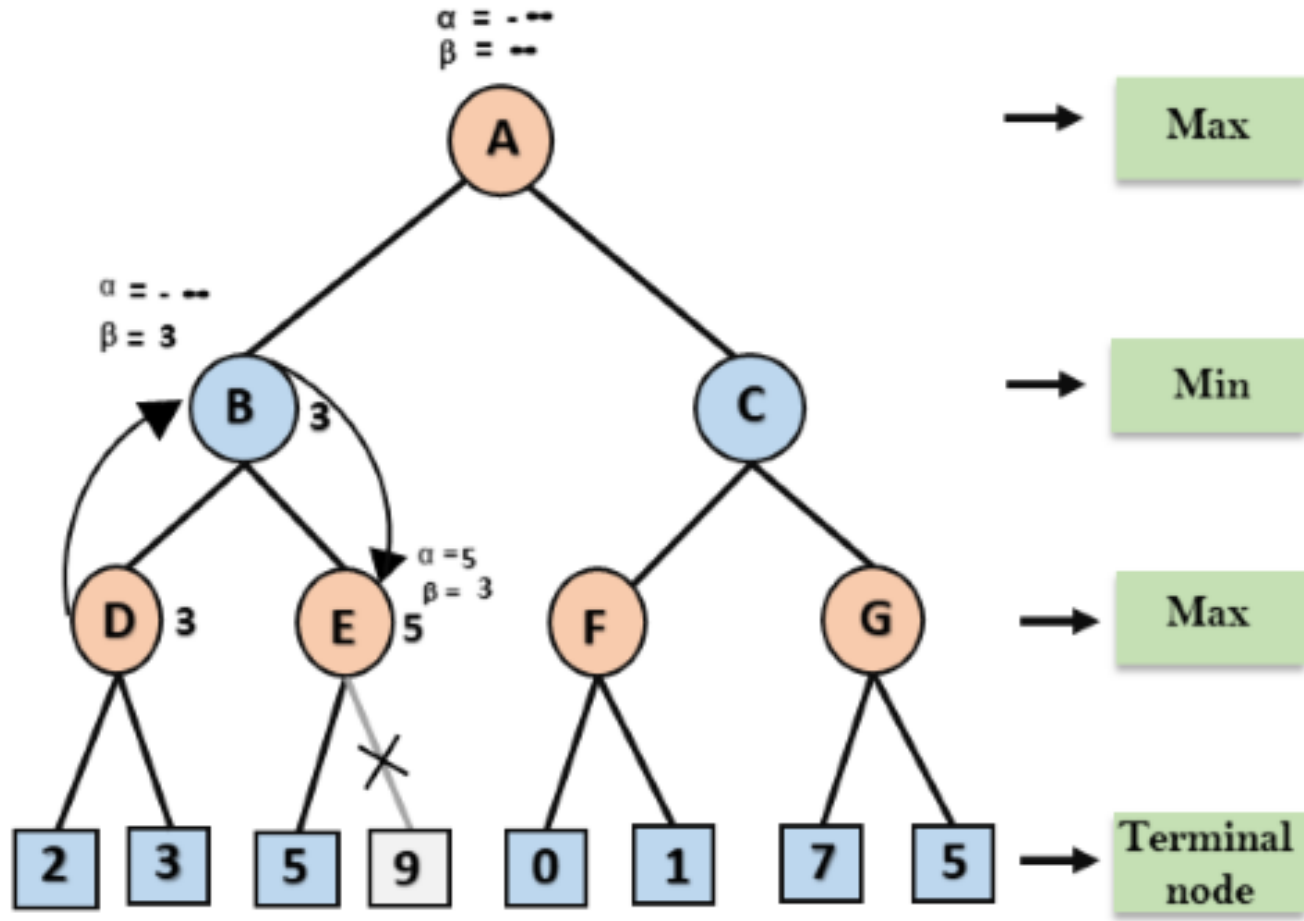
# Working of Alpha-Beta Pruning:

- **Step 2:** At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.
- **Step 3:** Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



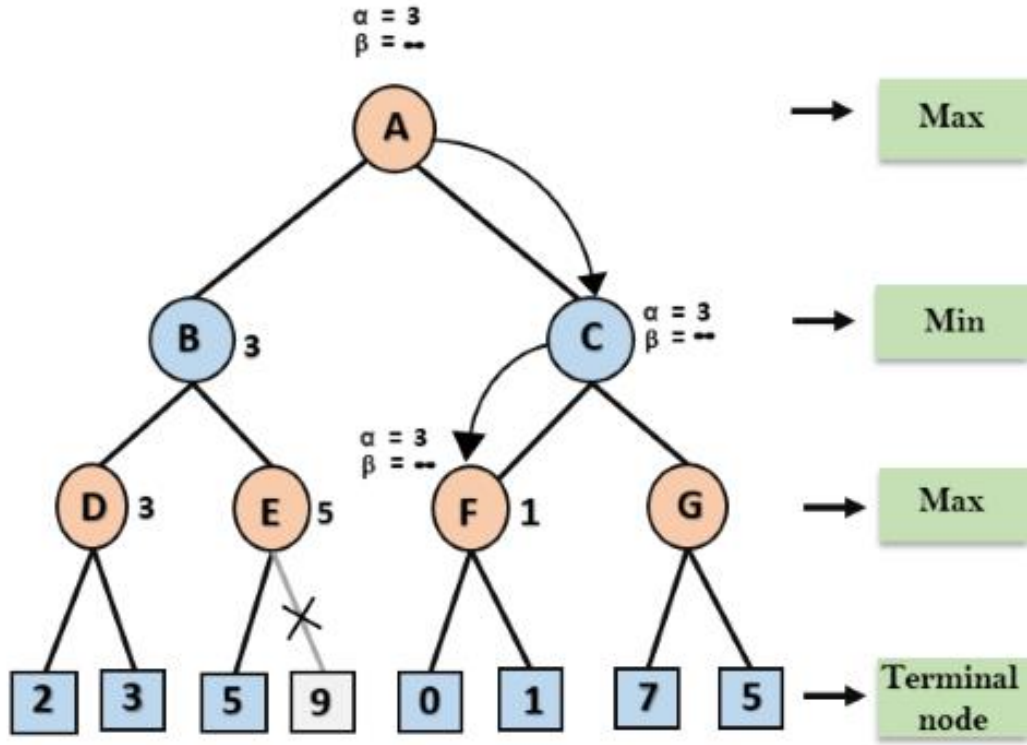
# Working of Alpha-Beta Pruning:

- In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.
- **Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



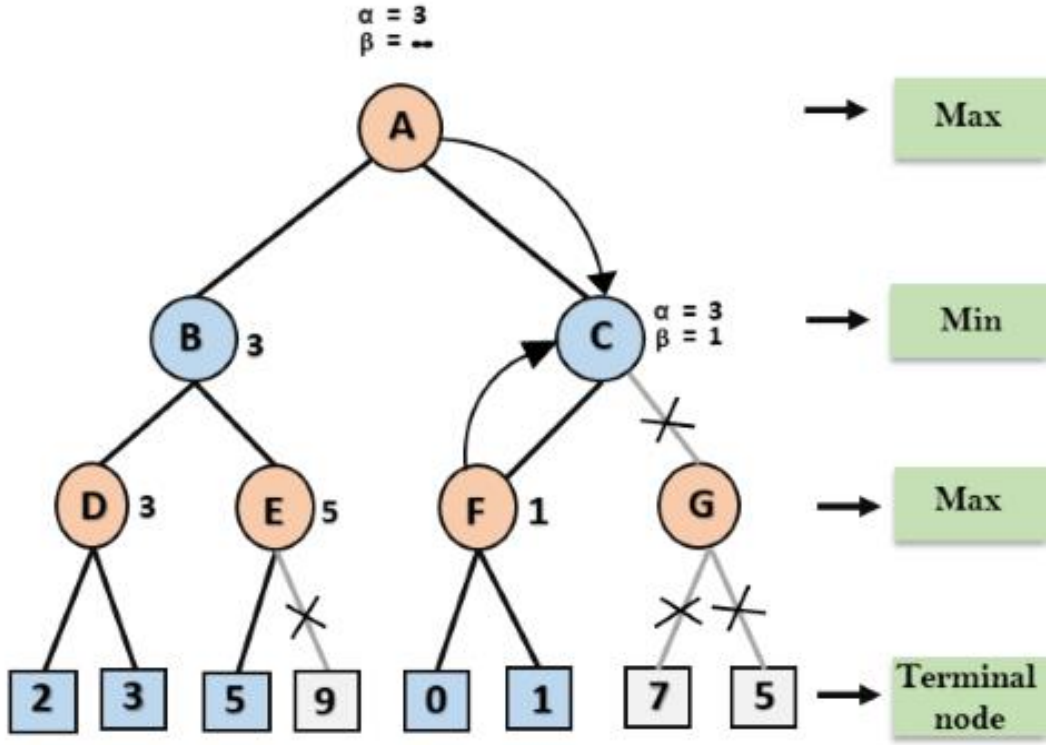
# Working of Alpha-Beta Pruning:

- **Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
- At node C, α=3 and β= +∞, and the same values will be passed on to node F.
- **Step 6:** At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



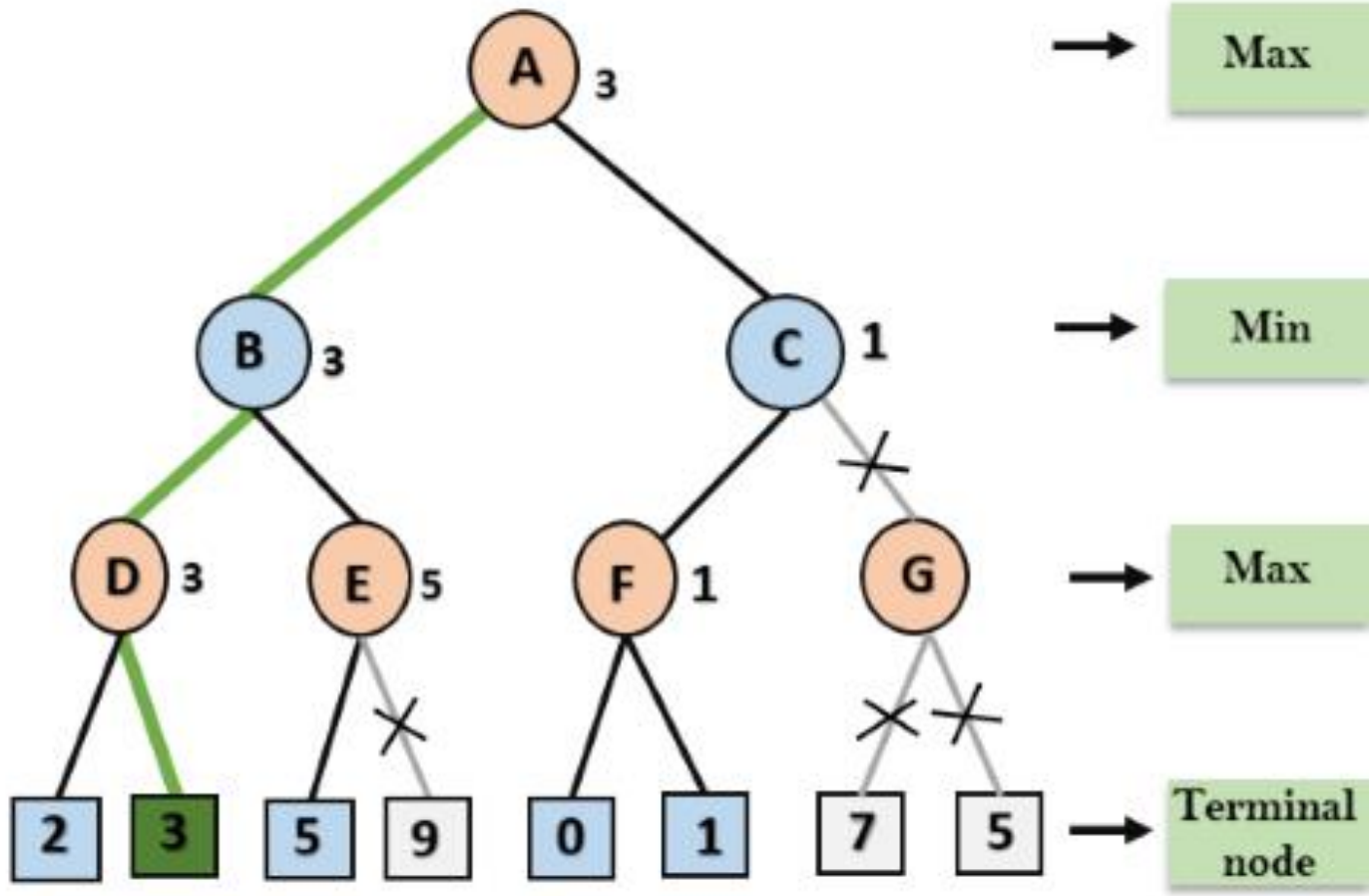
# Working of Alpha-Beta Pruning:

- **Step 7:** Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



# Working of Alpha-Beta Pruning:

**Step 8:** C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



# Thank You