Artificial Intelligence

By

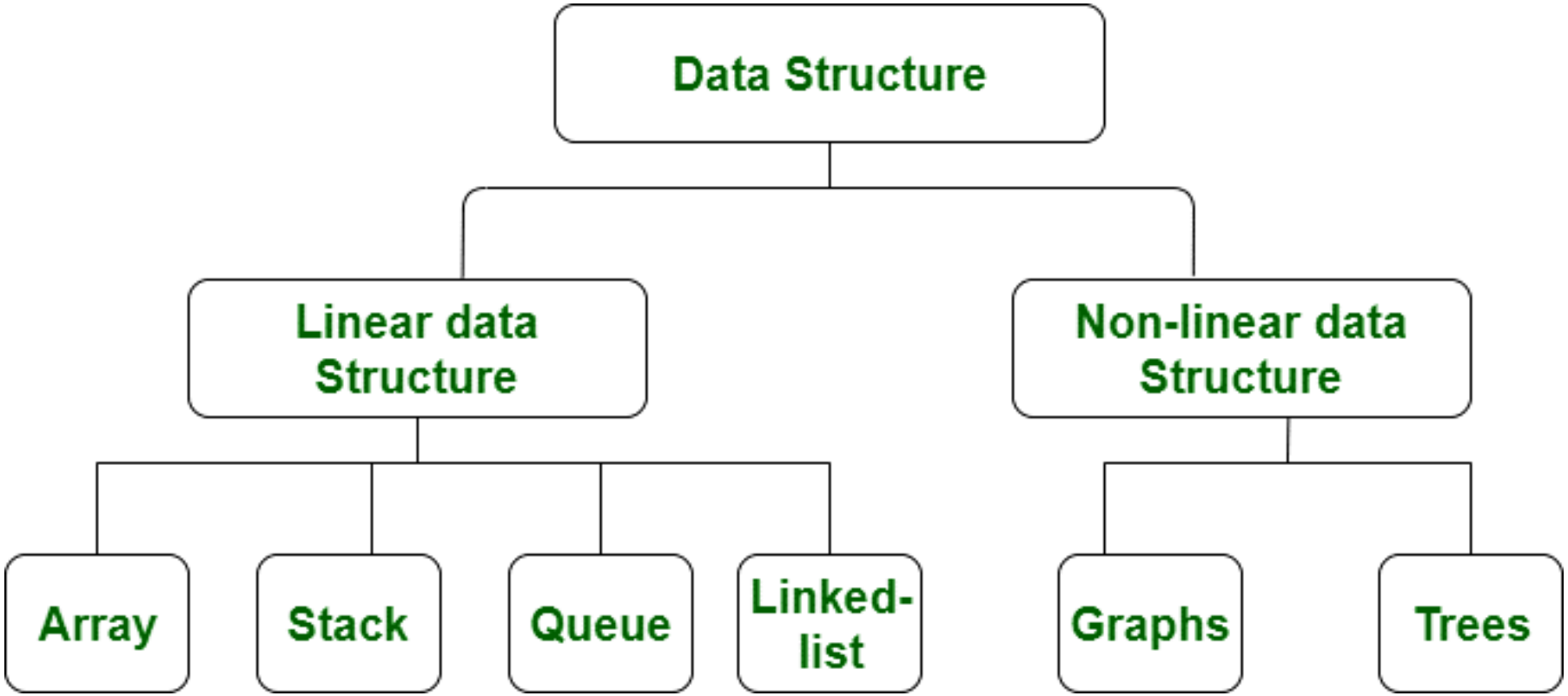
Dr. Manoj Kumar

University School of Automation and Robotics

GGSIP University, East Campus, Delhi, India

Data structure

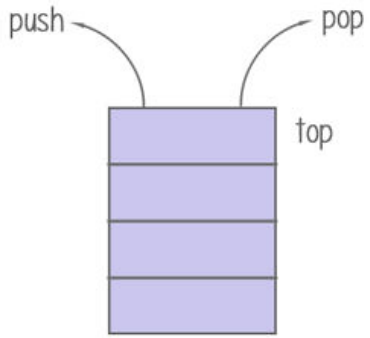
- A tree is a data structure that arranges the data similar to a tree.



Linear Data Structure

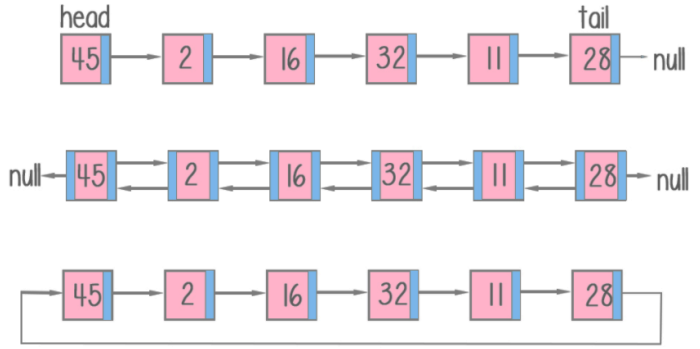
Array: An array is an index-based data structure, which means every element is referred by an index. An array holds same data type elements.

Stack: a stack is LIFO data structure in which only the top element can be accessed. The data is added by push and removed by pop on top.

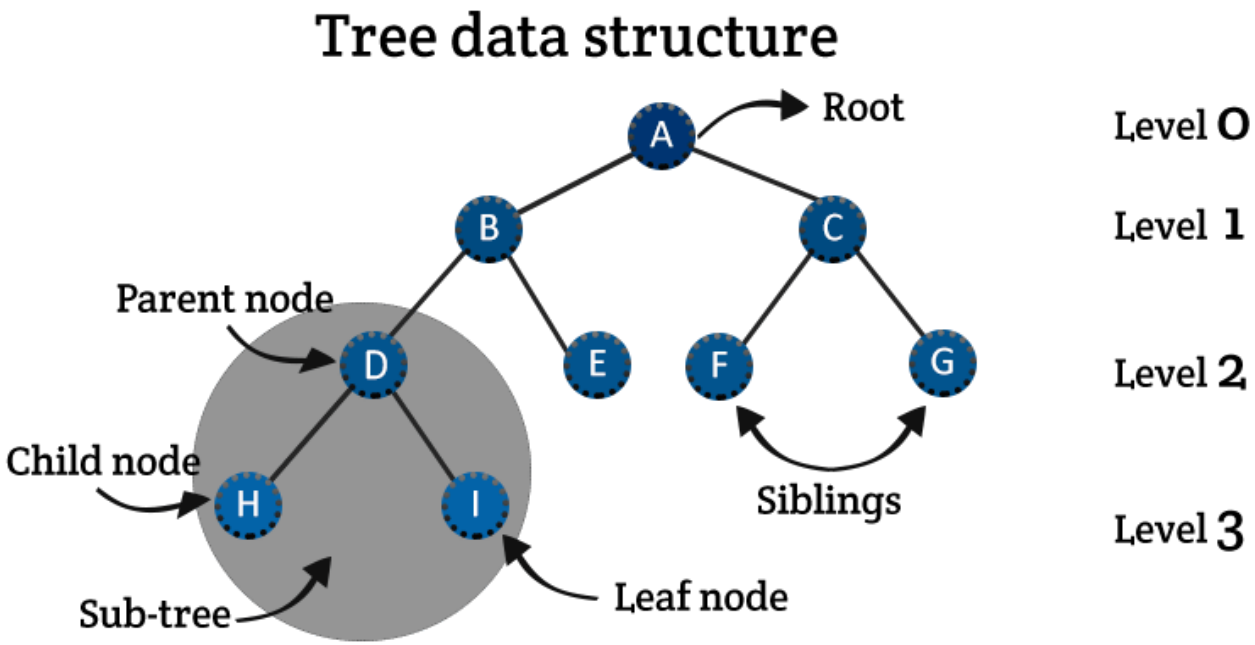


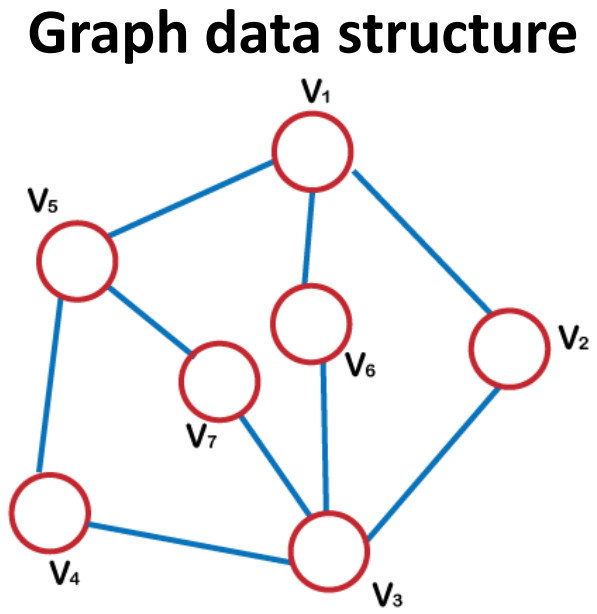
Queue: A queue is FIFO data structure. In this structure, new elements are inserted at one end and existing elements are removed from the other end.

Linked list: A linked list is a sequence of nodes in which each node is connected to the node following it. This forms a chain-link of data storage. It consists of data elements and a reference to the next record.



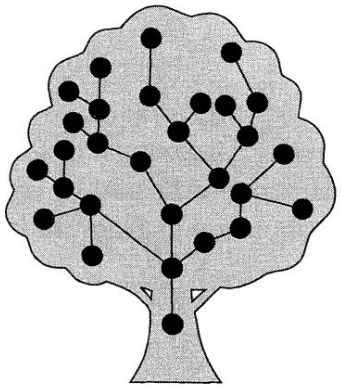
Nonlinear Data Structure





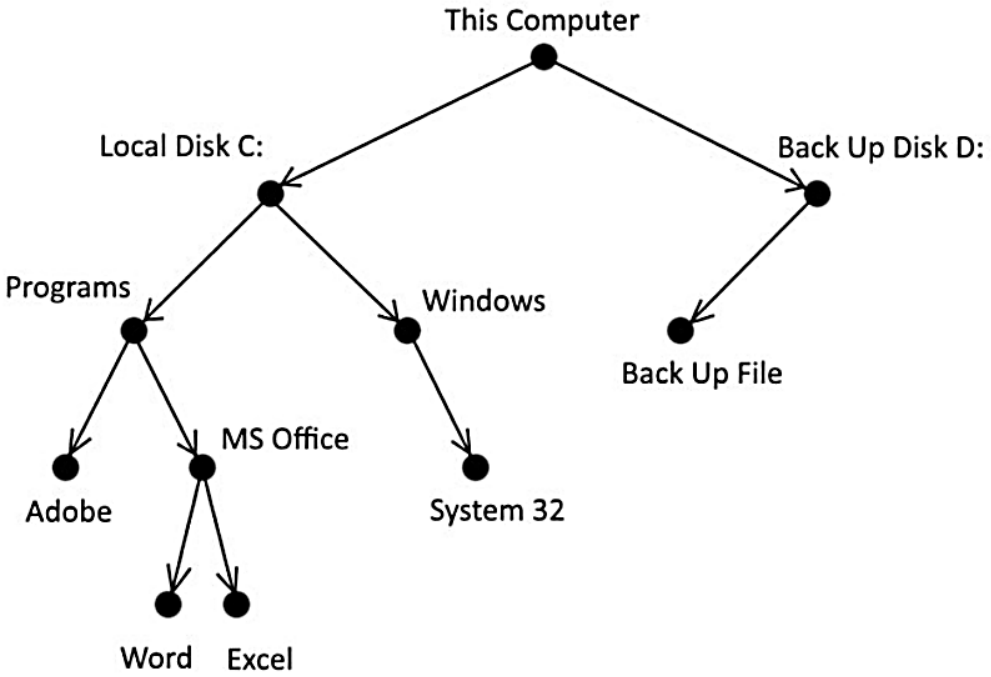
What is tree

- **A tree is a data structure that arranges the data similar to a tree.**

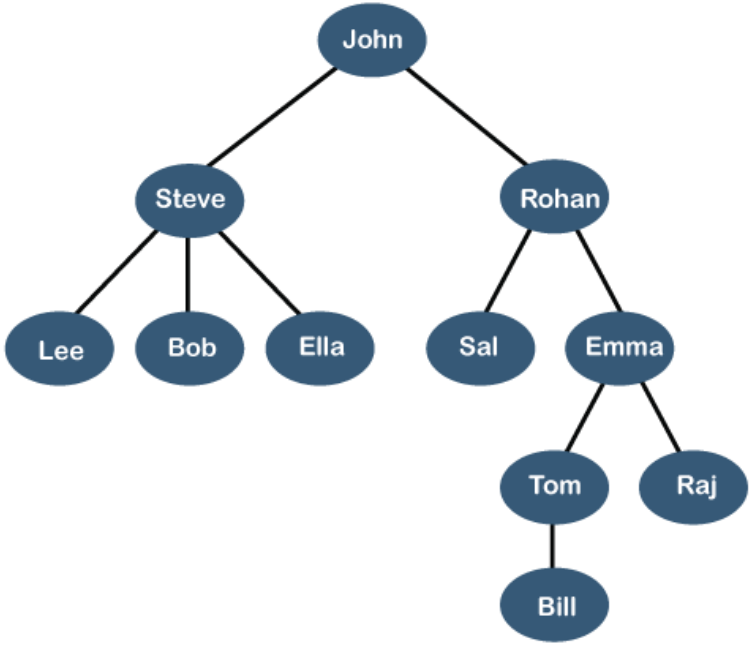


- **They don’t have any cyclic relations and there is only one path to a particular node.**
- **A data structure is a particular way of organizing data in a computer so that it can be used effectively.**

A computer's file system is an example of a tree.



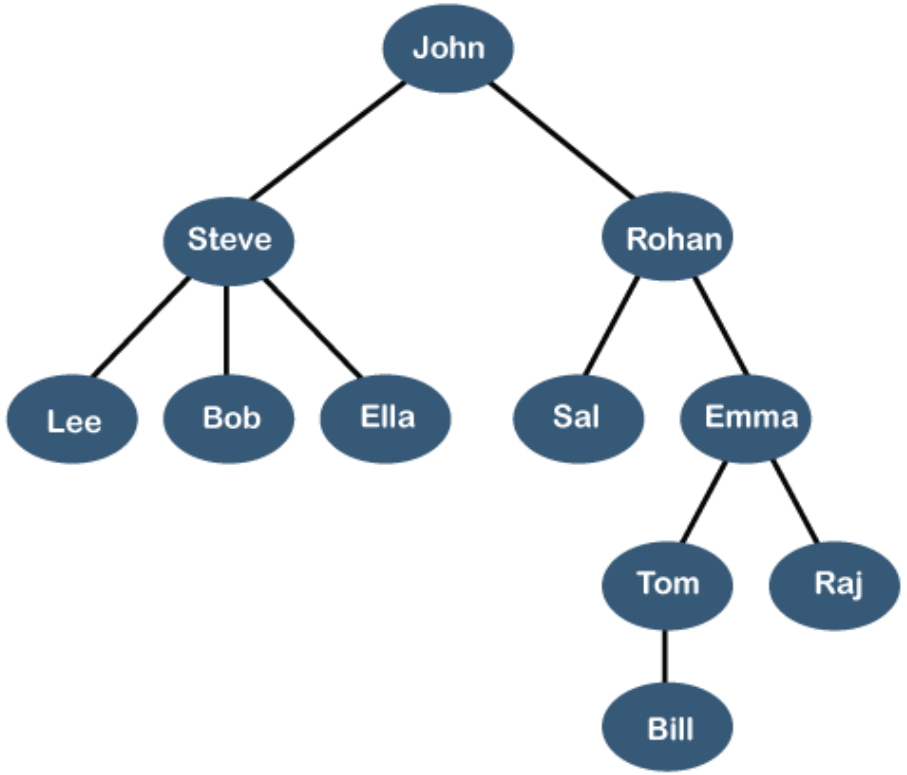
The employees and their positions in a company



What is tree

- **A tree is a data structure that arranges the data similar to a tree.**
  - **They don't have any cyclic relations and there is only one path to a particular node.**

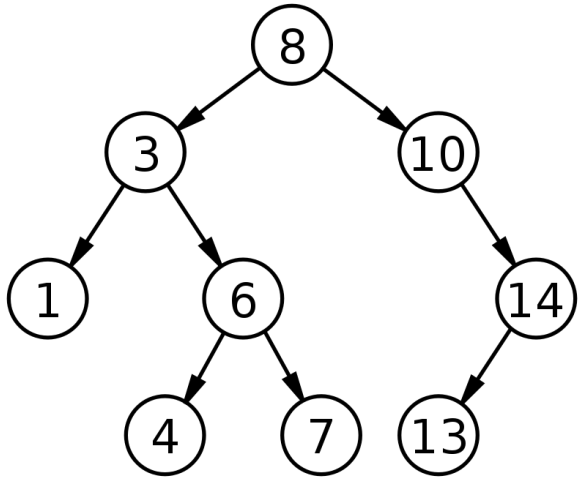
The employees and their positions in a company

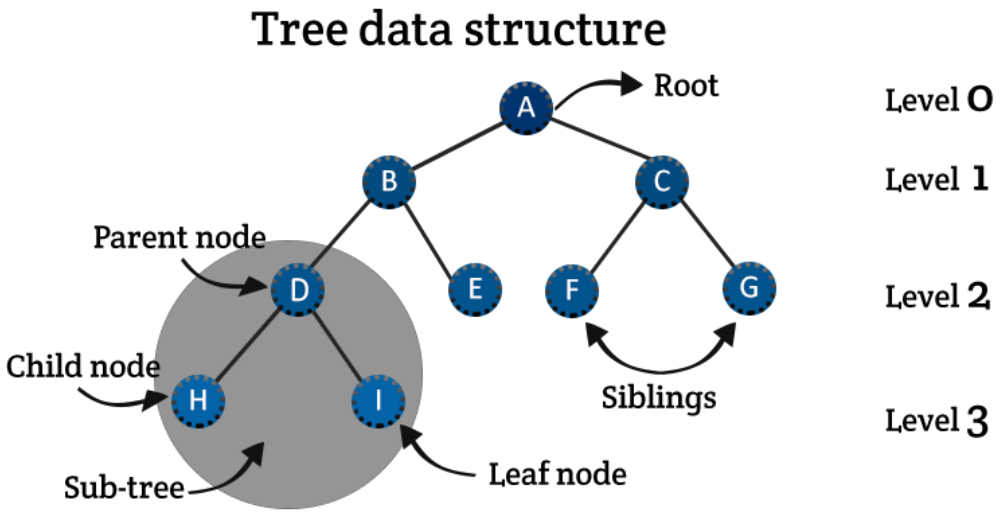


In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***.

What is tree

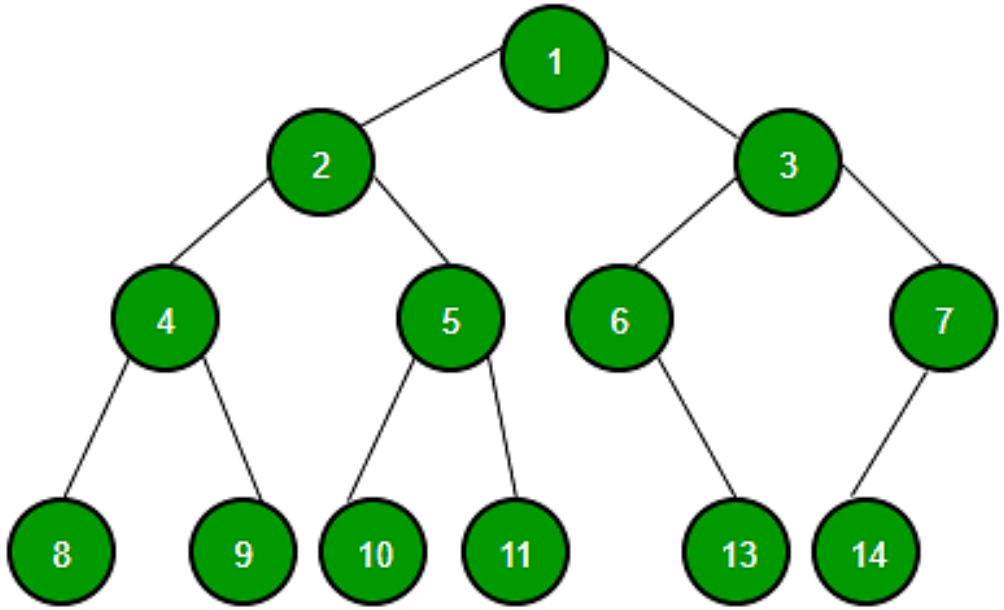
- **Root node** is the topmost data item in the tree. Element 8 is the root node in the above image.
- **Edge** helps to connect nodes. For example, in the above tree, edges connect 8 and 3, 8 and 10.
- **Parent node** is a node other than the root node that connects upwards by an edge. For instance, 3 is the parent node of 1 and 6. Similarly, 6 is the parent node of 4 and 7.
- **Child node** is a node that connects downwards by an edge. For example, 4 and 7 are child nodes of 6.
- **Leaf node** is a node that does not have any child nodes. 1, 4,7,13 are leaf nodes in the above tree.
- **Subtree** is a descendant of a node. For example, the section to the left of the root node (8) that begins from 3 is a subtree. Similarly, the section to the right of the root node that begins from 10 is a subtree.
- **Level** represents the generation of nodes. As an example, the root node belongs to level 0. 3 and 10 belongs to level 1 and so on.

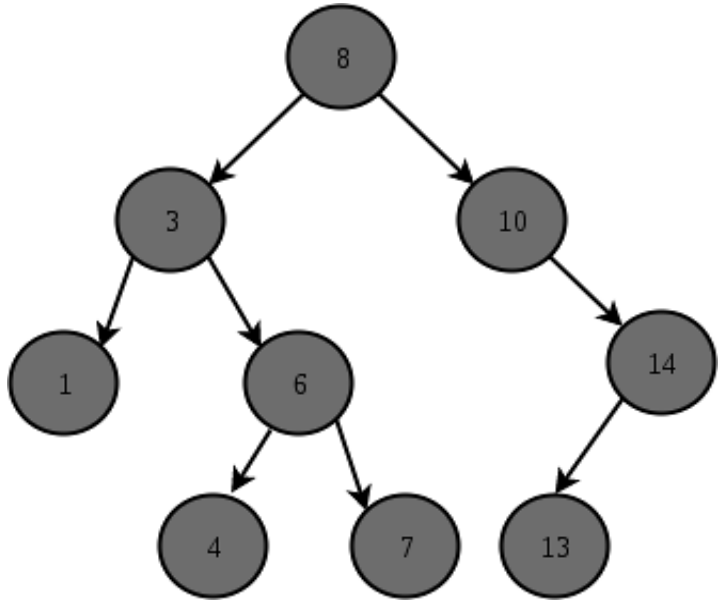




Binary Tree

- There are two major tree types as **binary tree** and **binary search tree**.



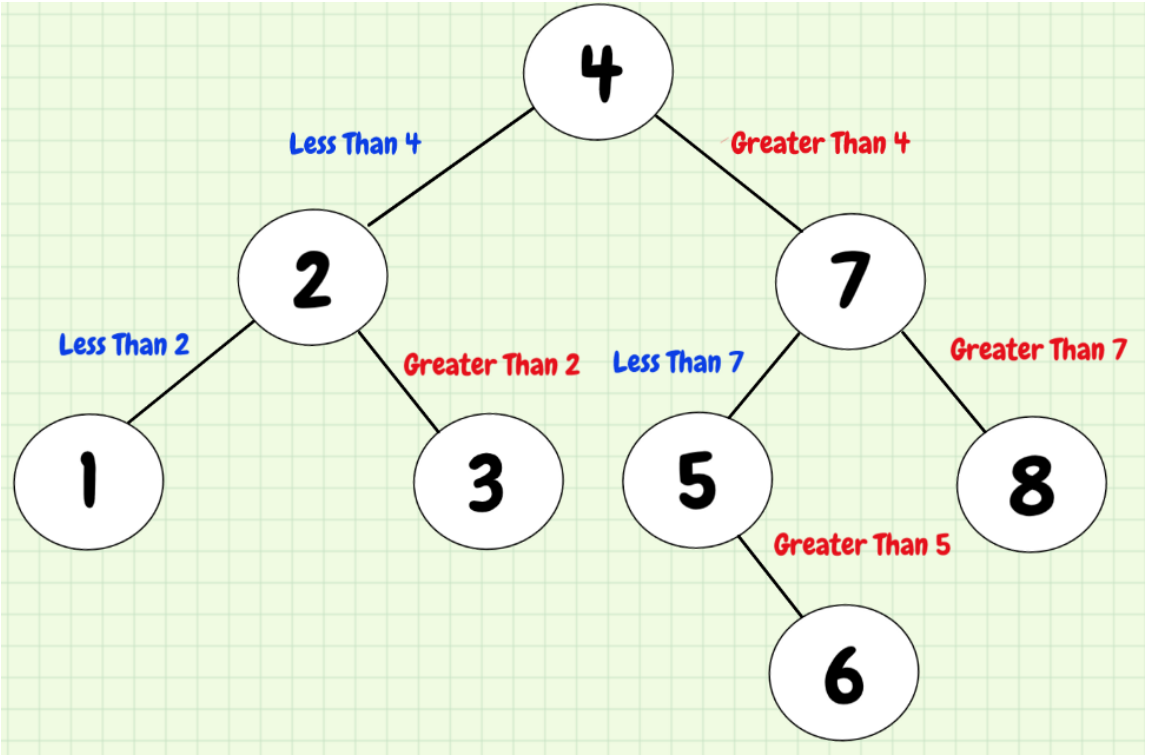


A tree whose elements have **at most (or maximum) 2 children** is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

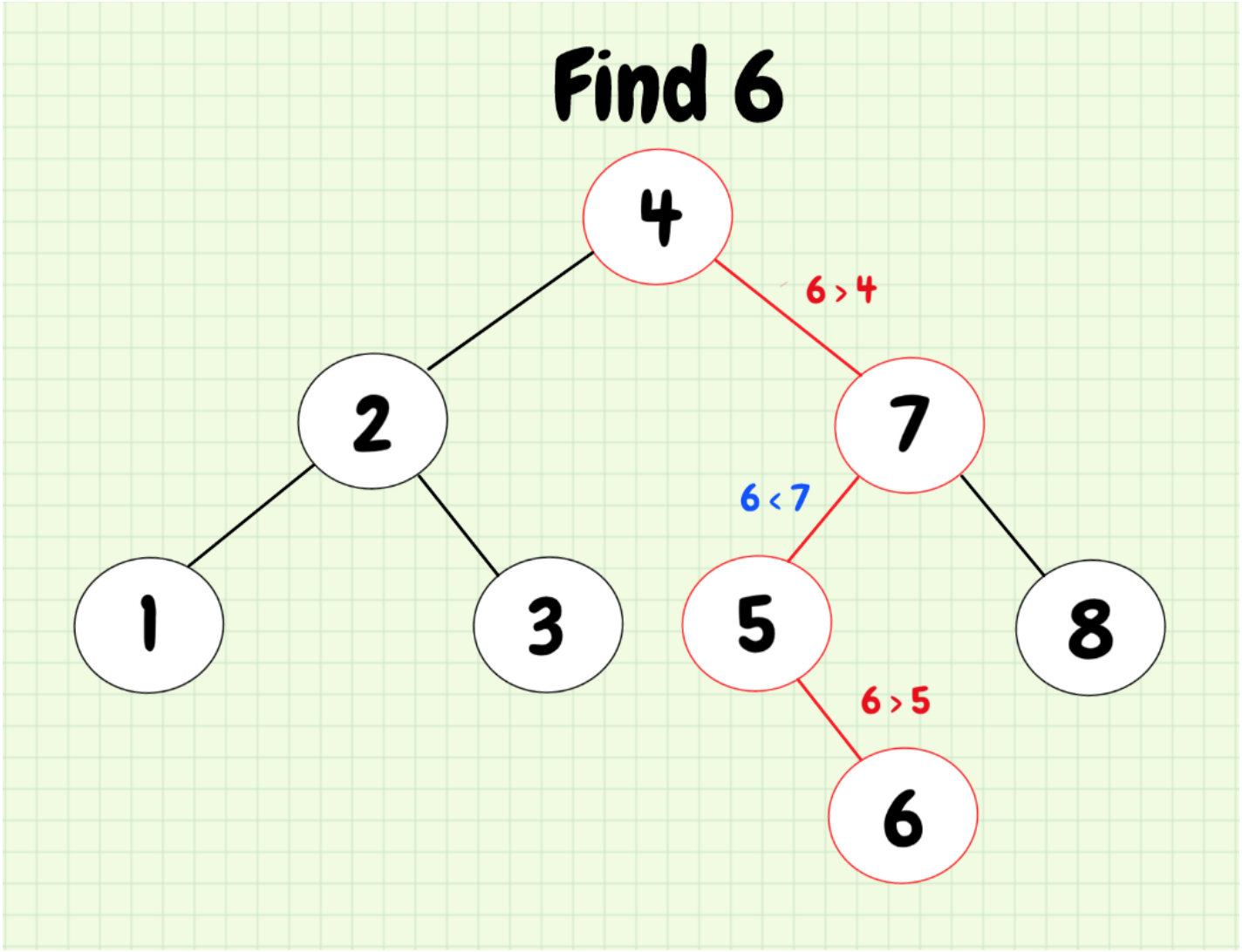
Binary Search Tree

- **Binary Search Tree is a node-based binary tree data structure which has the following properties:**

  - **The left subtree of a node contains only nodes with keys lesser than the node's key.**
  - **The right subtree of a node contains only nodes with keys greater than the node's key.**
  - **The left and right subtree each must also be a binary search tree.**
  - **There must be no duplicate nodes.**



Example

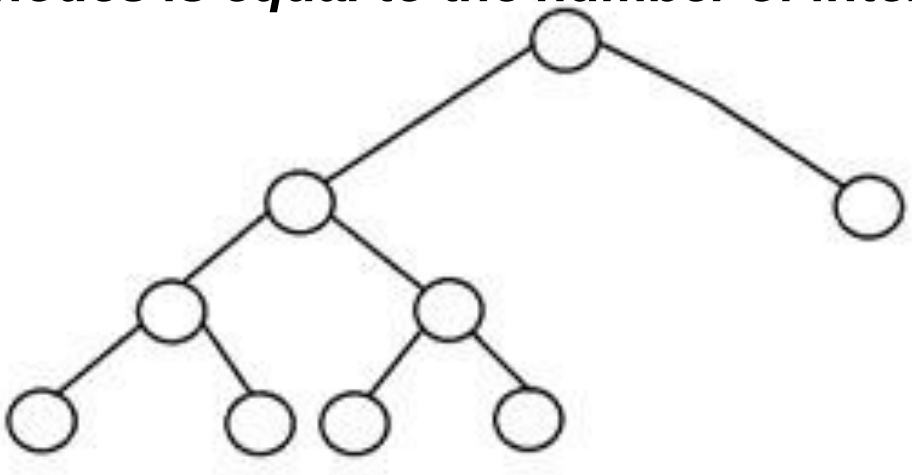


Other types of trees

1. Full Binary Tree
2. Complete Binary Tree
3. Perfect Binary Tree
4. Balanced Binary Tree
5. Degenerate Binary Tree

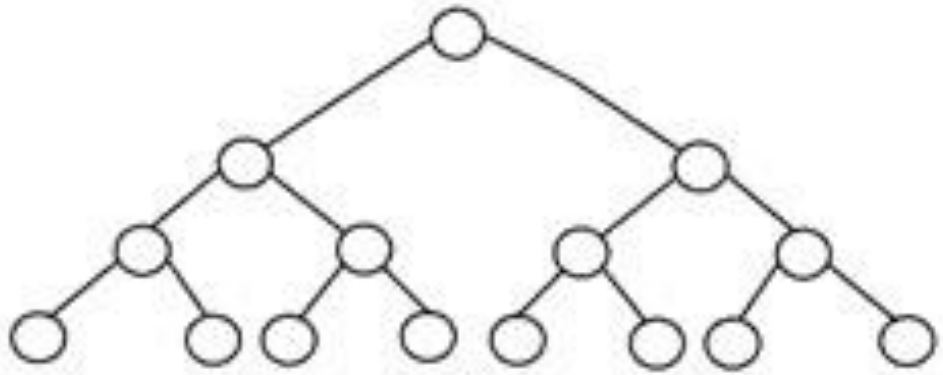
1. Full Binary Tree

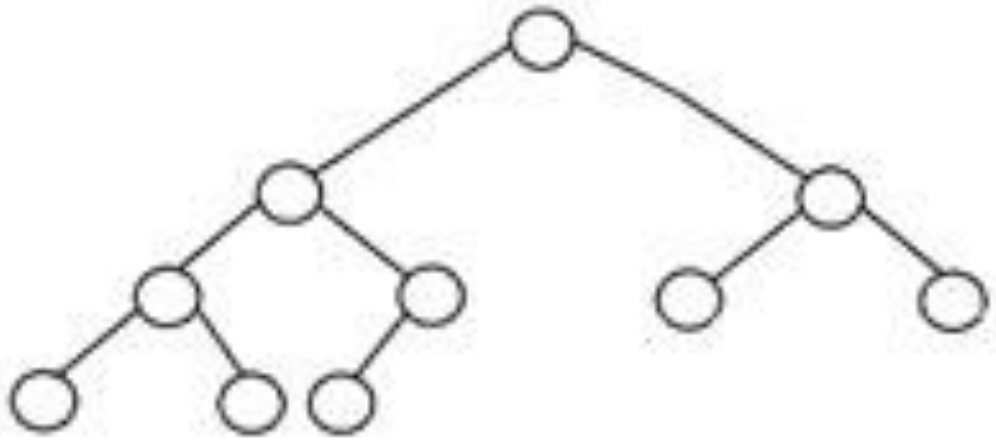
- **It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.**
- **A full binary tree is a unique binary tree where every node except the external node has two children. When it holds a single child, such a binary tree will not be a full binary tree.**
- ***Here, the quantity of leaf nodes is equal to the number of internal nodes plus one.***



2. Complete Binary Tree

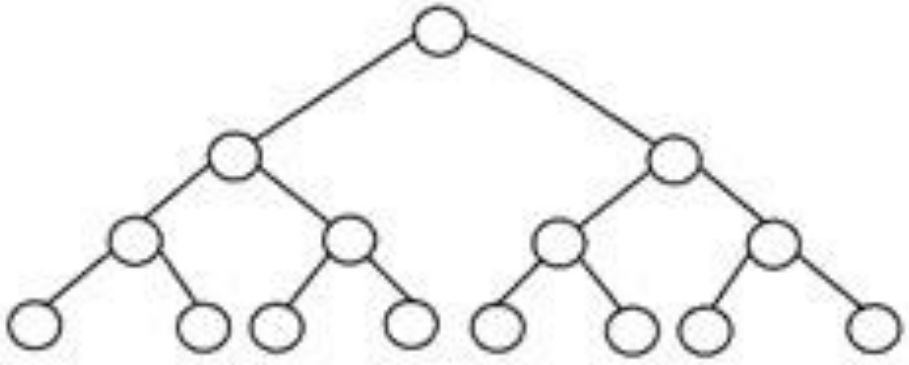
- **A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side. Here is the structure of a complete binary tree:**





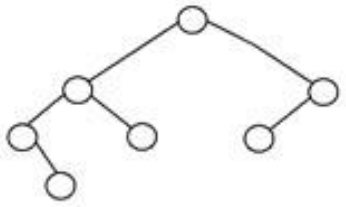
3. Perfect Binary Tree

- **A binary tree is said to be ‘perfect’ if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree.**
- **A perfect binary tree having height ‘h’ has $2^h - 1$ node. Here is the structure of a perfect binary tree:**

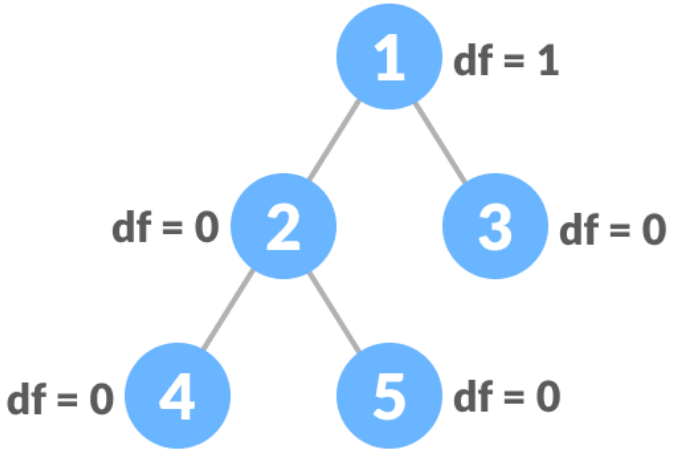


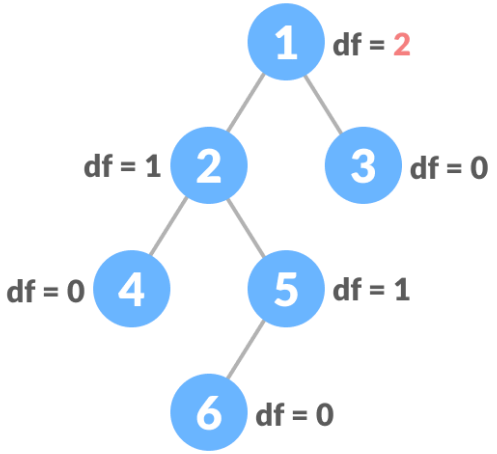
4. Balanced Binary Tree

- **In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree. Here is an example of a balanced binary tree:**



Height of lft sub tree-
height of right subtree

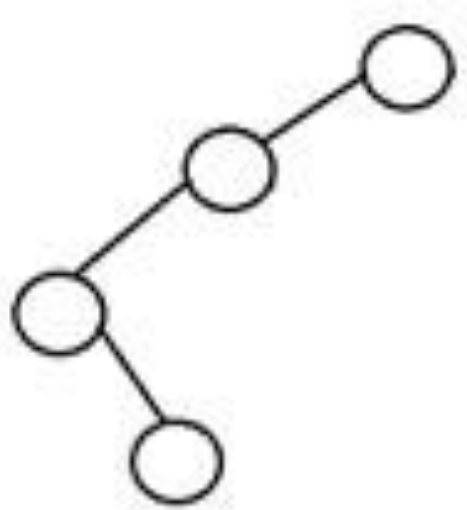




df = |height of left child - height of right child|

5. Degenerate Binary Tree

- **A binary tree is said to be a degenerate binary tree (pathological) if every internal node has only a single child. Such trees are similar to a linked list performance-wise. Here is an example of a degenerate binary tree:**



Artificial Intelligence

By

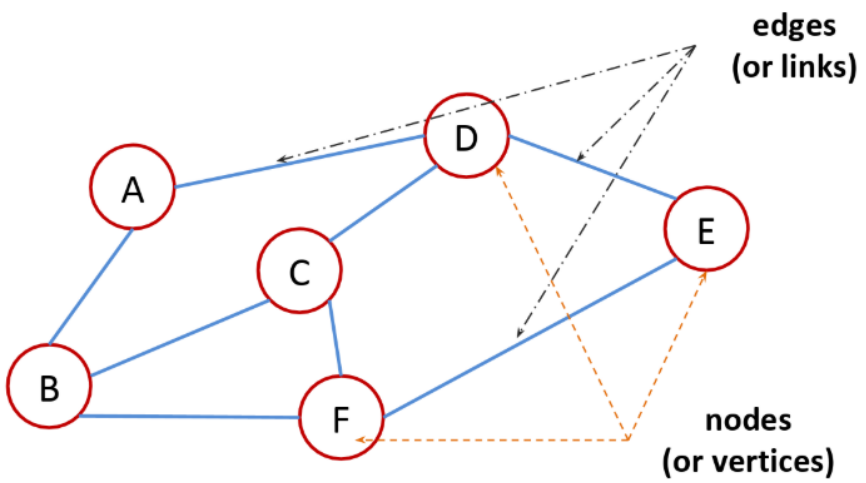
Dr. Manoj Kumar

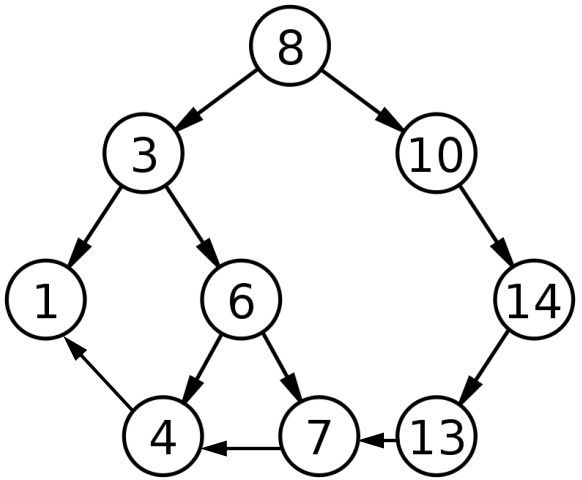
University School of Automation and Robotics

GGSIP University, East Campus, Delhi, India

What is Graph

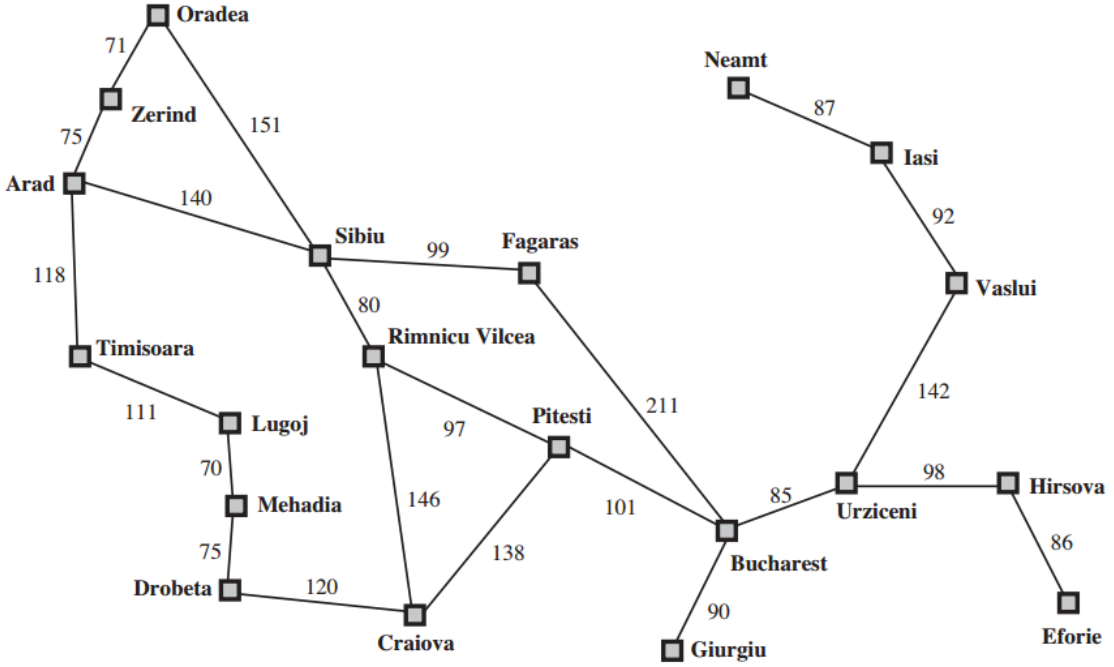
- **Trees don’t have any cyclic relations and there is only one path to a particular node.**
  - **A graph is like a tree data structure is a collection of objects or entities known as nodes that are connected to each other through a set of edges.**
  - **A tree follows some rule that determines the relationship between the nodes,**
  - **whereas graph does not follow any rule that defines the relationship among the nodes.**
  - **A graph contains a set of edges and nodes, and edges can connect the nodes in any possible way.**
  - **Mathematically, it can be defined as an ordered pair of a set of vertices, and a set of nodes where vertices are represented by 'V' and edges are represented by 'E'.** **G= (V , E)**

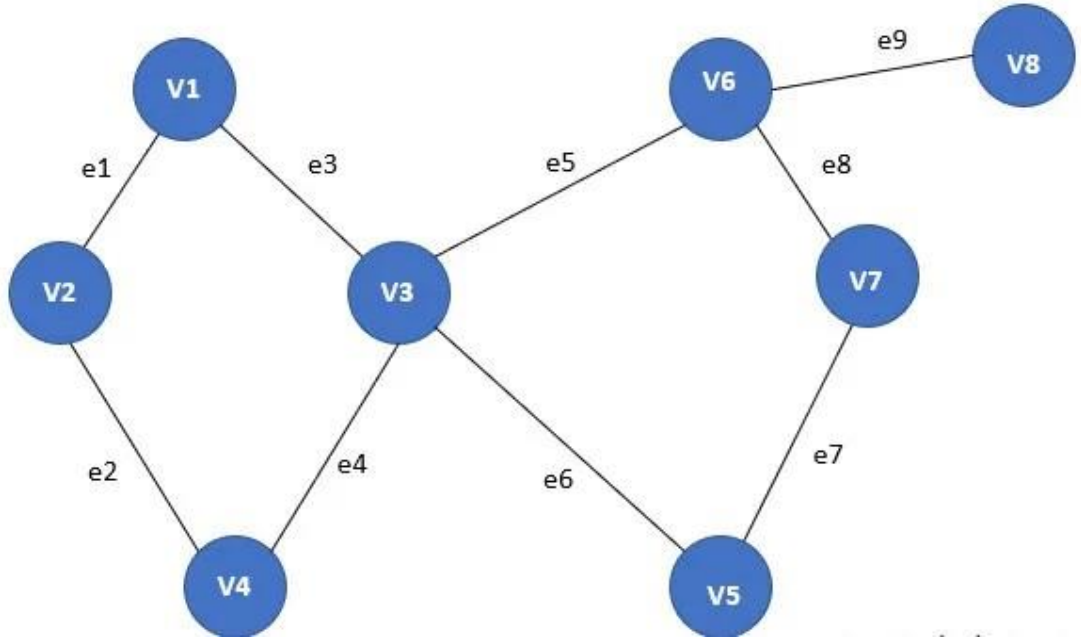




What is Graph

- In Graph, each node has a different name or index to uniquely identify each node in the graph.
- The graph shown below has eight vertices named as v1, v2, v3, v4, v5, v6, v7, and v8.
- There is no first node, a second node, a third node and so on.
- There is no ordering of the nodes.
- An edge can be represented by the two endpoints in the graph.
- We can write the name of the two endpoints as a pair, that represents the edge in a graph.

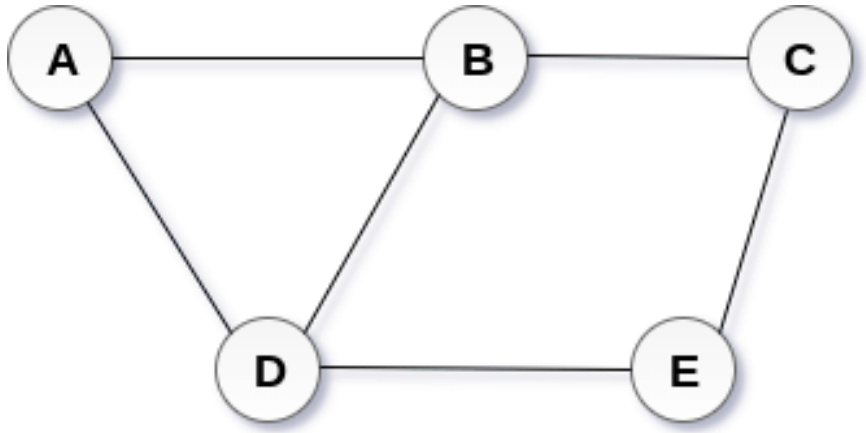




What is a Graph

- A graph can be defined as group of vertices and edges that are used to connect these vertices.
- A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

- A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
- A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Undirected Graph

Types of Graph

There are two types of edges:

•**Undirected edge:** The undirected edge is two-way means that there is no ***origin*** and ***destination***. For example, there are two vertices U and V, then undirected would represent two paths, i.e., from U to V as well as from V to U. An undirected edge can be represented as an unordered pair because the edge is ***bi-directional***.

•**Directed edge:** The directed edge represents one endpoint as an origin and another point as a destination. The directed edge is one-way. For example, there are two vertices U and V; then directed edge would represent the link or path from U to V, but no path exists from V to U. If we want to create a path from V to U, then we need to have one more directed edge from V to U.

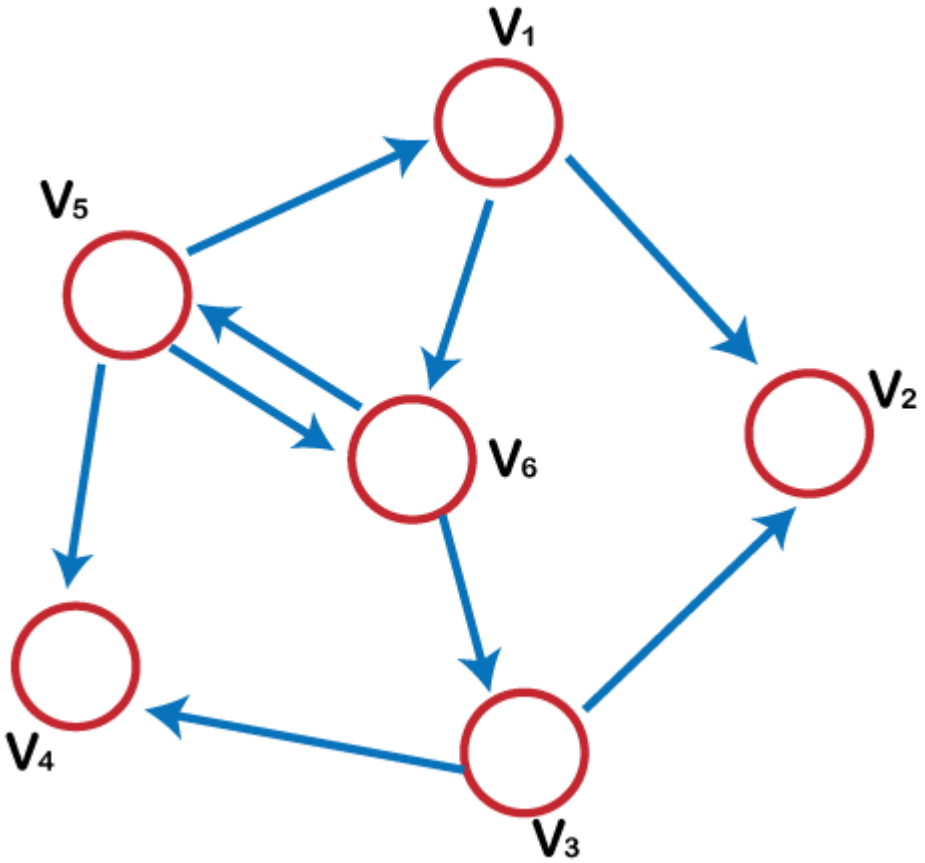
The directed edge can be represented as an ordered pair in which the first element is the origin, whereas the second element is the destination.

The tree data structure contains only directed edges, wher
the graph can have both types of edges, i.e., ***directed as w***
undirected. But, we consider the graph in which all the edg
are either directed edges or undirected edges.

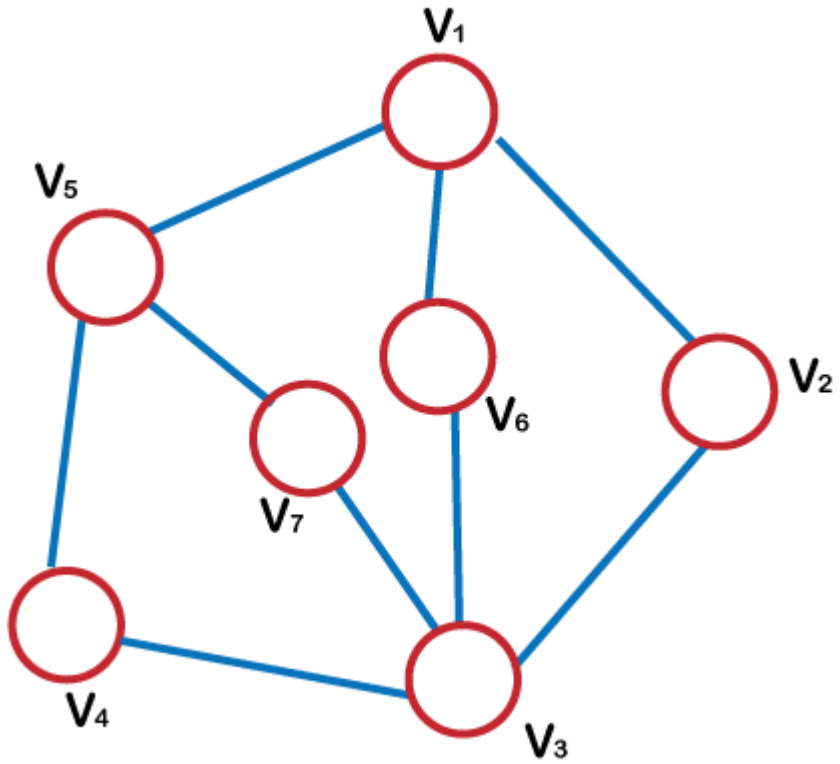
Types of Graph

- **There are two major graph types as directed and binary search tree.**

- **Directed graph:** The graph with the directed edges known as a ***directed graph***.



- **Undirected graph:** The graph with the undirected edges known as a ***undirected graph***. The directed graph is a graph in which all the edges are uni-directional, whereas the undirected graph is a graph in which all the edges are bi-directional.



Graph Terminology

- **Path:** A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
- **Closed Path:** A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.
- **Simple Path:** If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.
- **Cycle:** A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
- **Connected Graph:** A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- **Complete Graph:** A complete graph is the one in which every node is connected with all other nodes.

Example

- **Weighted Graph:** In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.
- **Digraph:** A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- **Loop:** An edge that is associated with the similar end points can be called as Loop.
- **Adjacent Nodes**: If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- **Degree of the Node:** A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graphs Examples

State Space Representation

- In the *state space representation* of a problem, the *nodes* of a graph corresponds to partial problem solution *states*, the *arcs* corresponds to *steps* in a problem-solving process.
- *State space search* characterize problem solving as the process of finding *a solution path* from the start state to a goal.

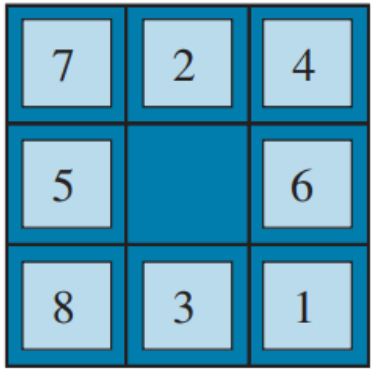
- Precise
- Analyse

State Space Representation

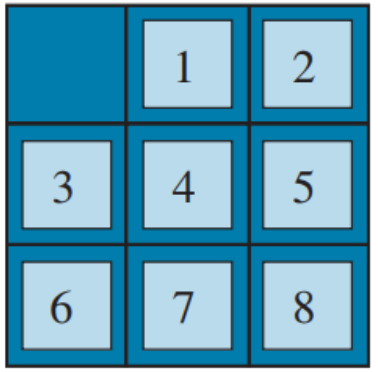
➢ **Set of states in which a can be result**

- **S: {S, A, Action(s), Result(s,a), Cost(s,a)**
- **S: Set of all possible states**
- **A: Set of all action possible**
- **Action (s): Which action is possible/valid for current state**
- **Result (s,a): state reached by performing action 'a' on state 's'**
- **s: initial state**

Eight-tile puzzle



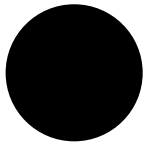
Start State



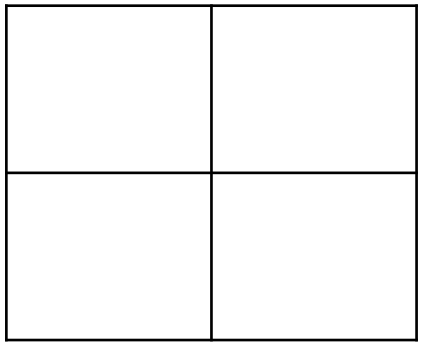
Goal State

Action possible: Up, Down, Left, Right — **Legal moves for a state**

State Space Representation

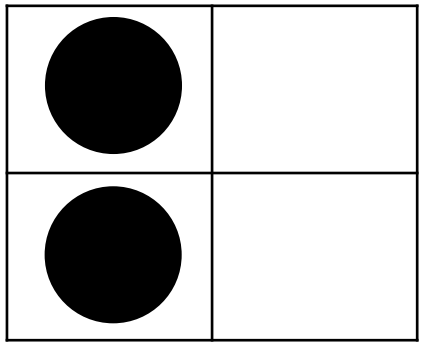


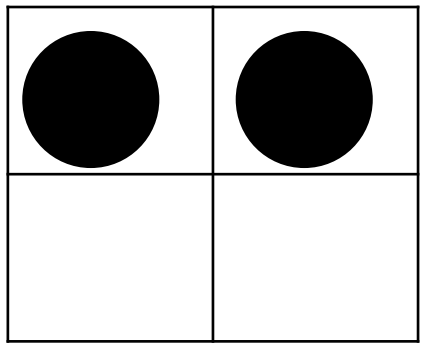
Problem: we have to place two balls in these 4 boxes

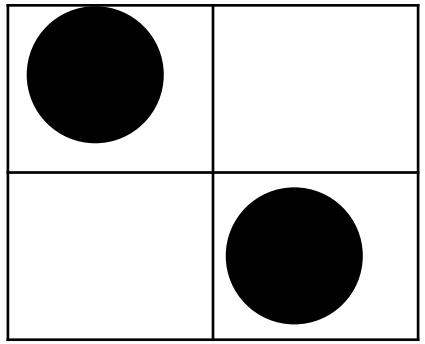


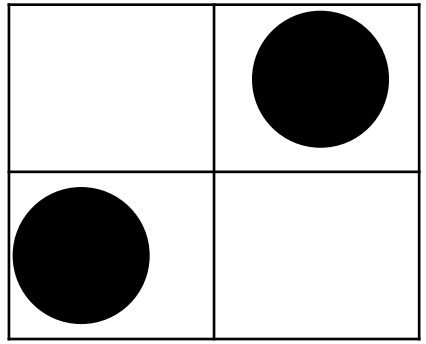
Rule: They should not lie in the same row or column

Balls can be placed diagonally









State Space Representation

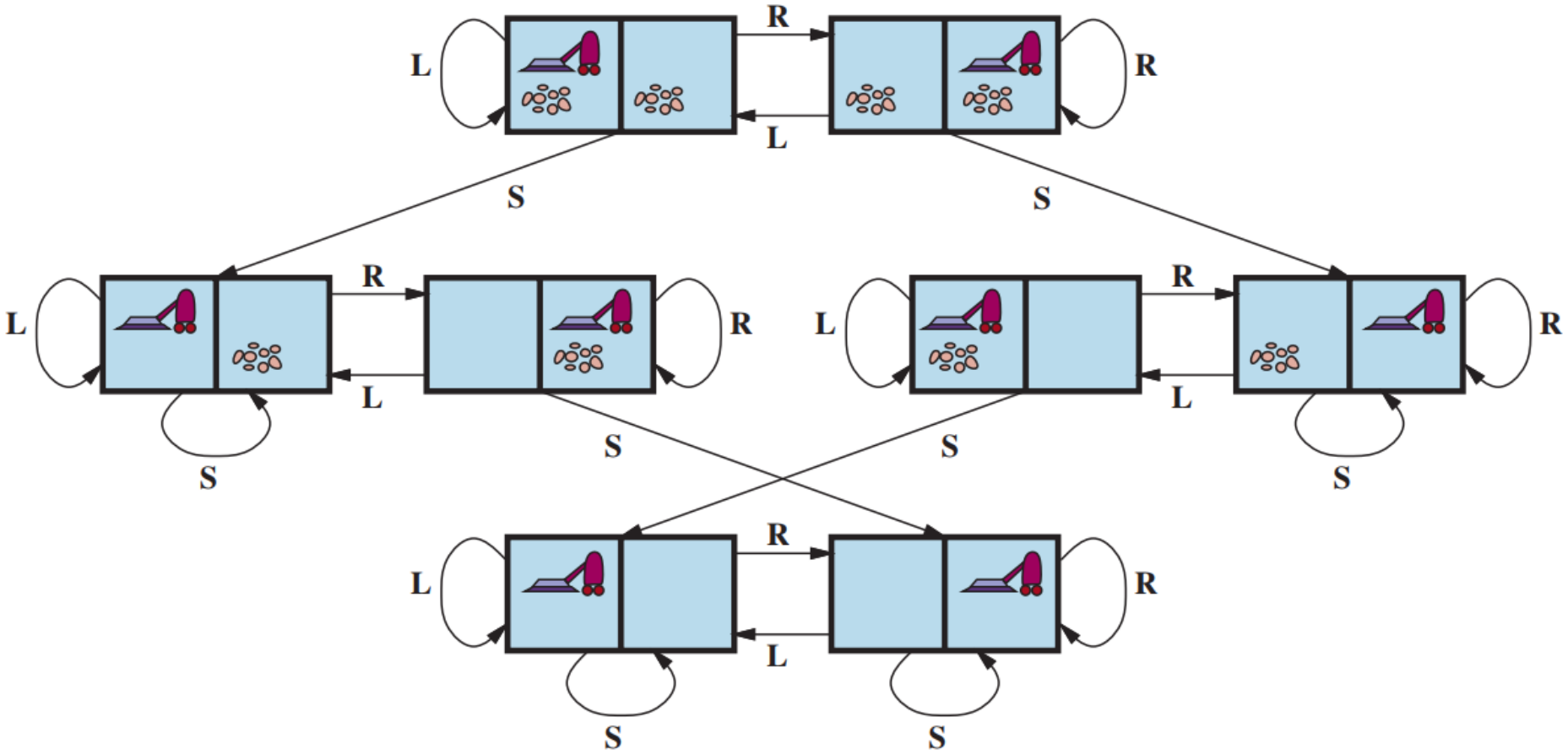


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Search Tree

- In computer science, a **search tree** is a tree data structure used for locating specific keys from within a set. In order for **a tree to function as a search tree**, the key for each node must be greater than any keys in subtrees on the left, and less than any keys in subtrees on the right.
- The advantage of search trees is their efficient search time given the tree is reasonably balanced, which is to say the leaves at either end are of comparable depths. Various search-tree data structures exist, several of which also allow efficient insertion and deletion of elements, which operations then have to maintain tree balance.

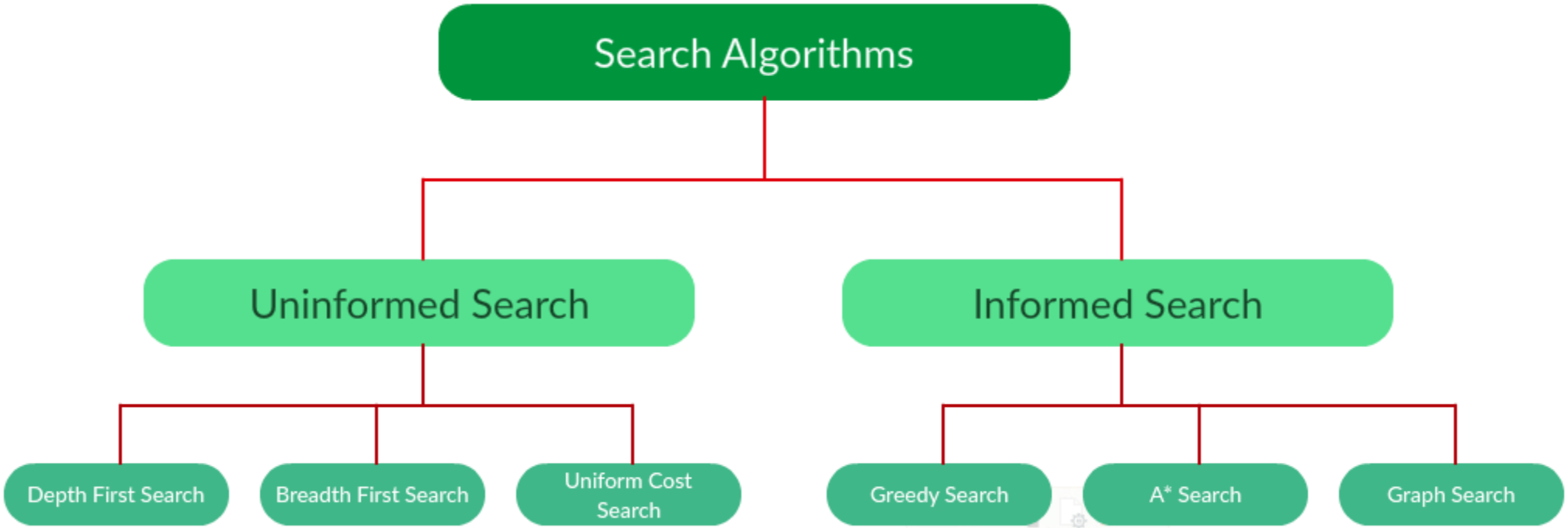
Search Graph

- A graph search is **an algorithm scheme that visits vertices or vertices and edges in a graph**, in an order based on the connectivity of the graph. The most general searches visit both edges and vertices.

- In computer science, graph traversal (also known as graph search) refers **to the process of visiting (checking and/or updating) each vertex in a graph**. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

- We start at the source node and keep searching until we find the target node.

Search Graph



Artificial Intelligence

By

Dr. Manoj Kumar

University School of Automation and Robotics

GGSIP University, East Campus, Delhi, India

Search Algorithms

- **Search is following up one option now and putting the others aside for later on when the first choice does not lead to a solution**
- **We continue choosing, testing and expanding until either a solution is found or there are no more states to be expanded**
- **The choice of which state is to be expanded is determined by the search method**
- **This is basically why we need various search methods in Al**
- **Very broadly, there are two classes of state space search:**

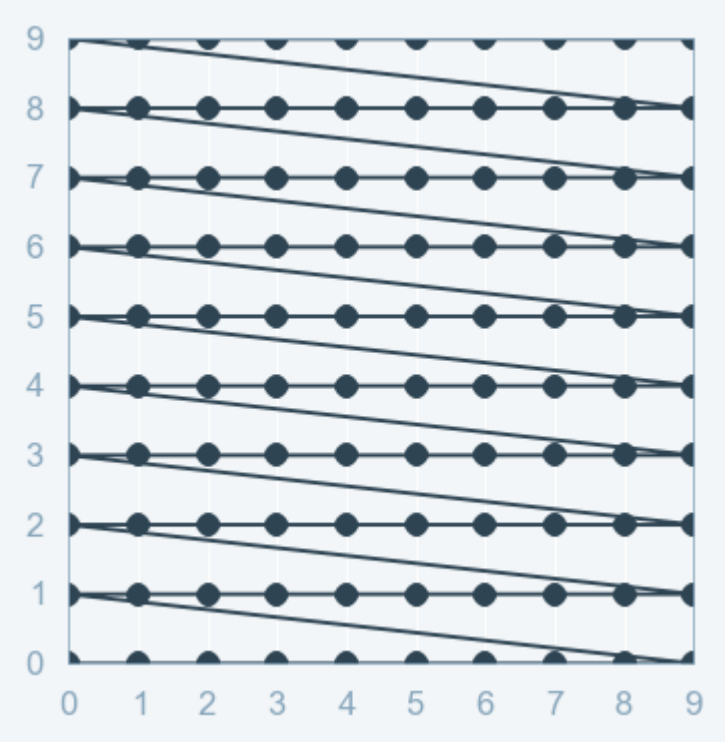
  (1) Uninformed search or Blind search

  (2) Informed search or Heuristic search

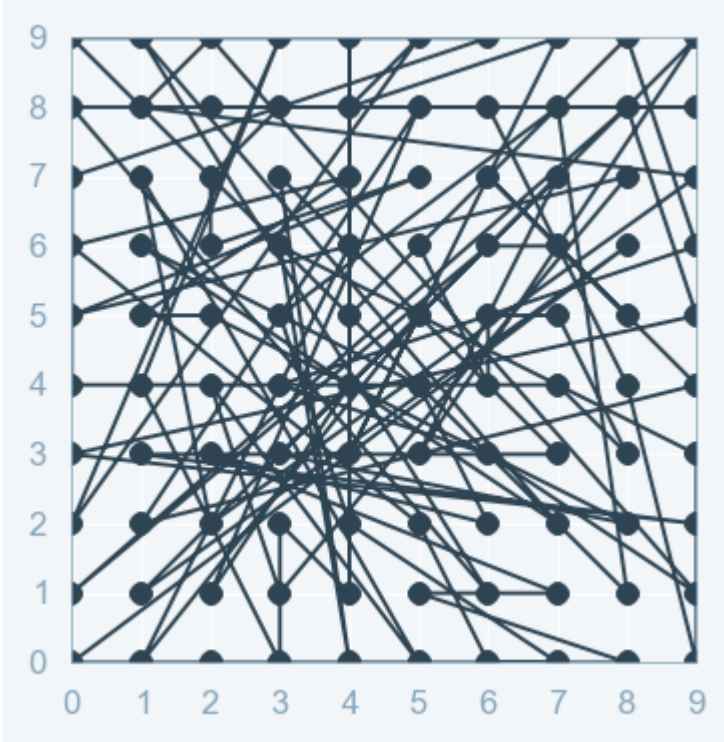
Note: that all search strategies are distinguished by the order in which nodes are expanded.

Search Algorithms

- In Grid Search, we try every combination of a preset list of values of the hyper-parameters and evaluate the model for each combination. The pattern followed here is similar to the grid, where all the values are placed in the form of a matrix.



- A random search algorithm refers to an algorithm that uses some kind of randomness or probability.



Search Algorithms

- A **parameter** is something that is learnt during machine learning process

- In machine learning, a **hyperparameter** is a parameter whose value is used to control the learning process (manually specified).

- *Hyperparameters* are model-specific properties that are ‘fixed’ even before the model is trained or tested on the data. For Example: In the case of a random forest, hyper parameters include the number of decision trees in the forest , for a neural network, there is the learning rate, the number of hidden layers, the number of units in each layer, and several other parameters.

- **Hyperparameter Tuning** is nothing but searching for the right set of hyperparameter to achieve high precision and accuracy.

Search Algorithms

When you say, learning to drive, you have to go through learning sessions with an instructor teaching you.
Here, you are getting trained with the instructor's help. The instructor helps you through the lessons. S/he helps you practice driving until you are confident and capable enough to drive on the road alone.
Once trained and you are able to drive, you would not need the instructor to train you. In this scenario, the instructor is the hyper-parameter and the student the parameter.
As seen earlier, parameters are the variables that the system estimates the value of during the training.

PARAMETER	HYPERPARAMETER
Estimated during the training with historical data.	Values are set beforehand.
It is a part of the model.	External to the model.
The estimated value is saved with the trained model.	Not a part of the trained model and hence the values are not saved.
Dependent on the dataset that the system is trained with.	Independent of the dataset.

Random Search

Algorithm Steps

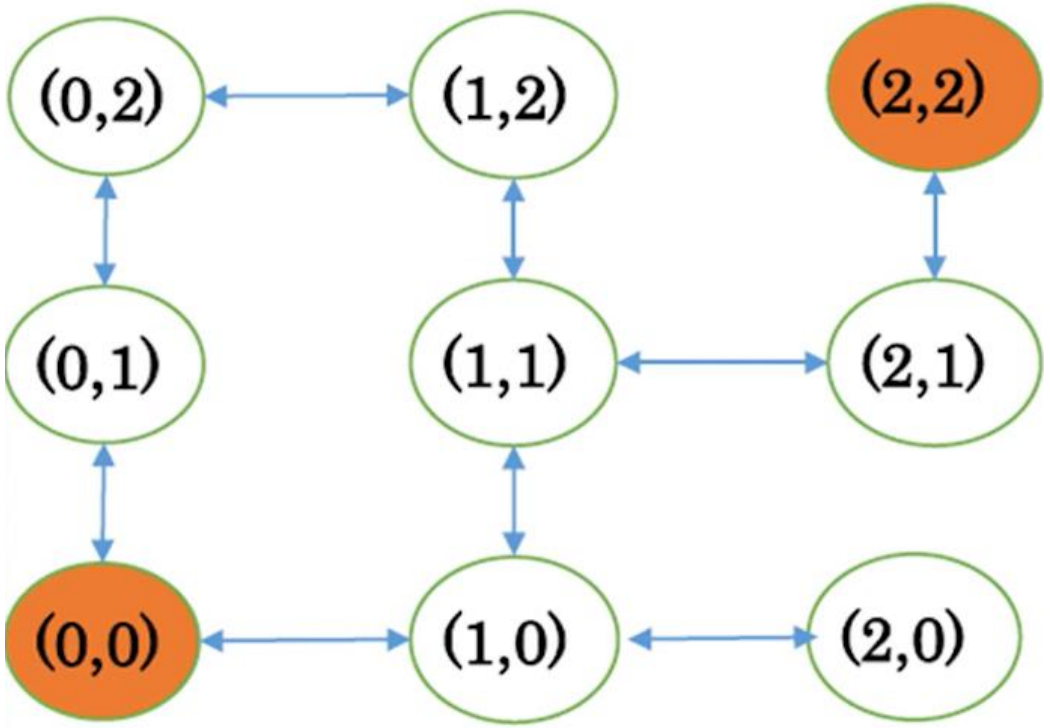
Step 1: Current node x=initial node;

Step 2: If x=target node, stop with success;

Step 3: Expand x, and get a set S of child nodes;

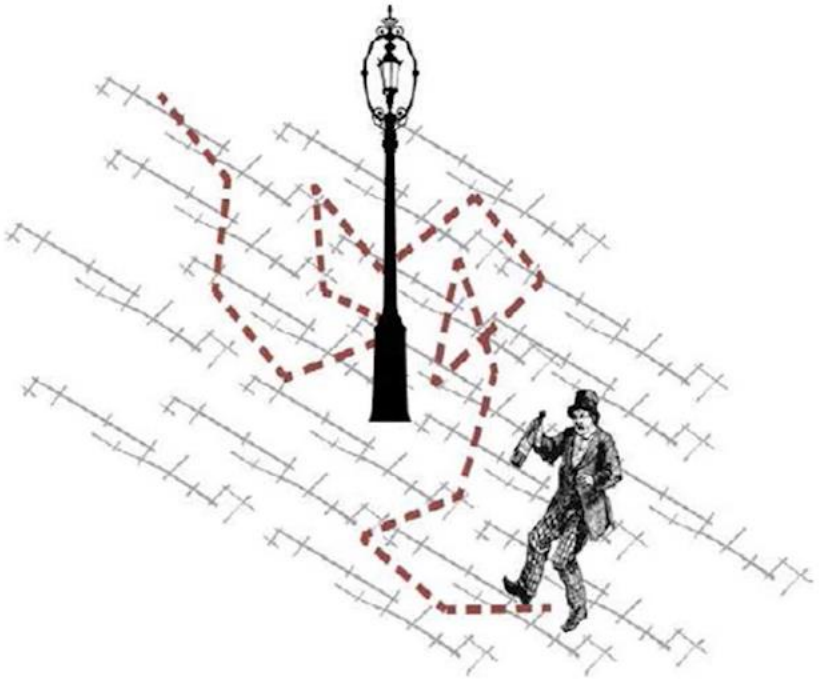
Step 4: Select a node x' from S at random;

Step 5: x=x', and return to Step 2.



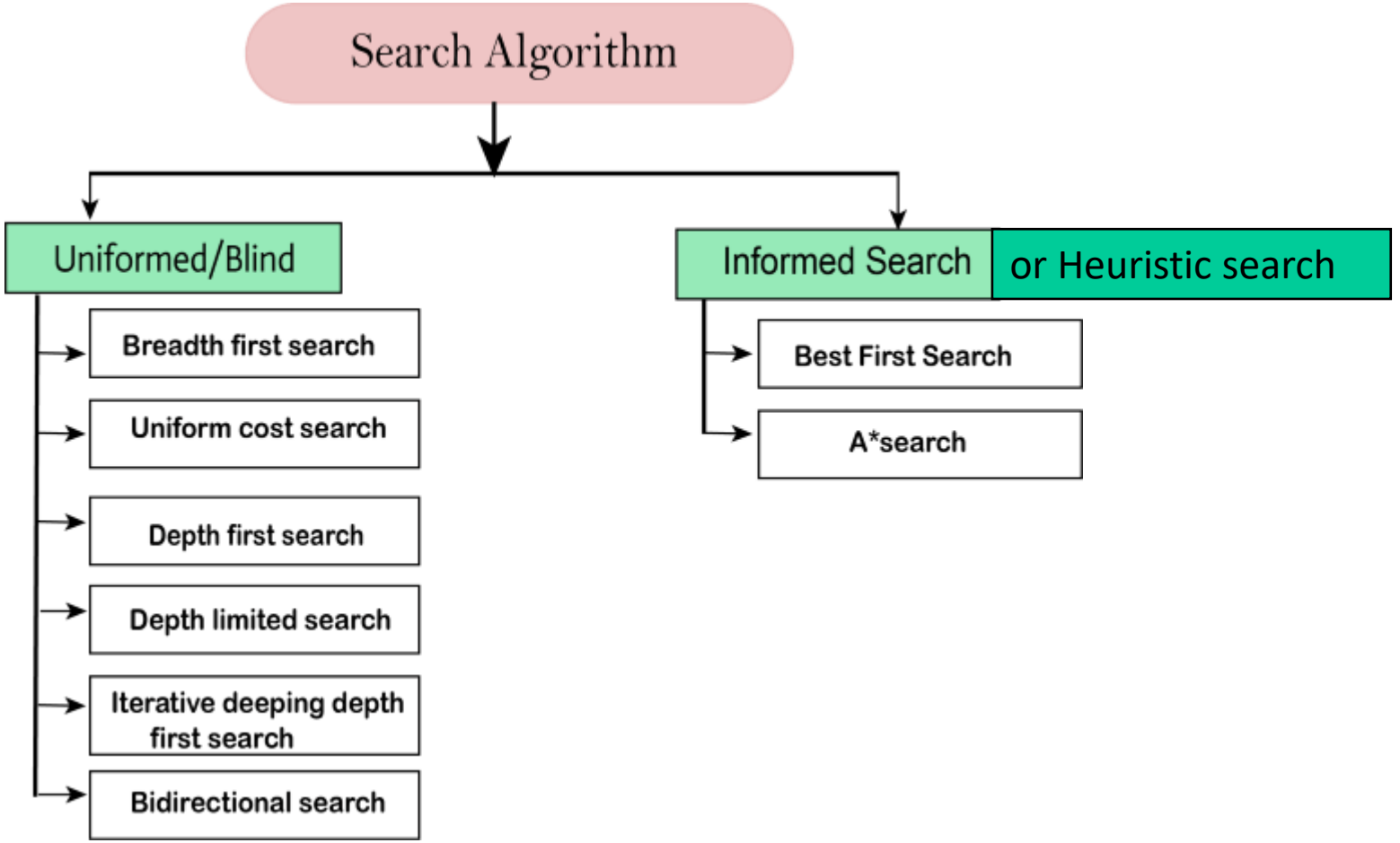
Problem with Random Search

- At each step, the next node is determined at random.
- We cannot guarantee to reach the target node; or even if we can reach the target, the path so obtained is very redundant (extremely long).



Types of Search Algorithms

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



Properties of Search Algorithms

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

- **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Uninformed/Blind search

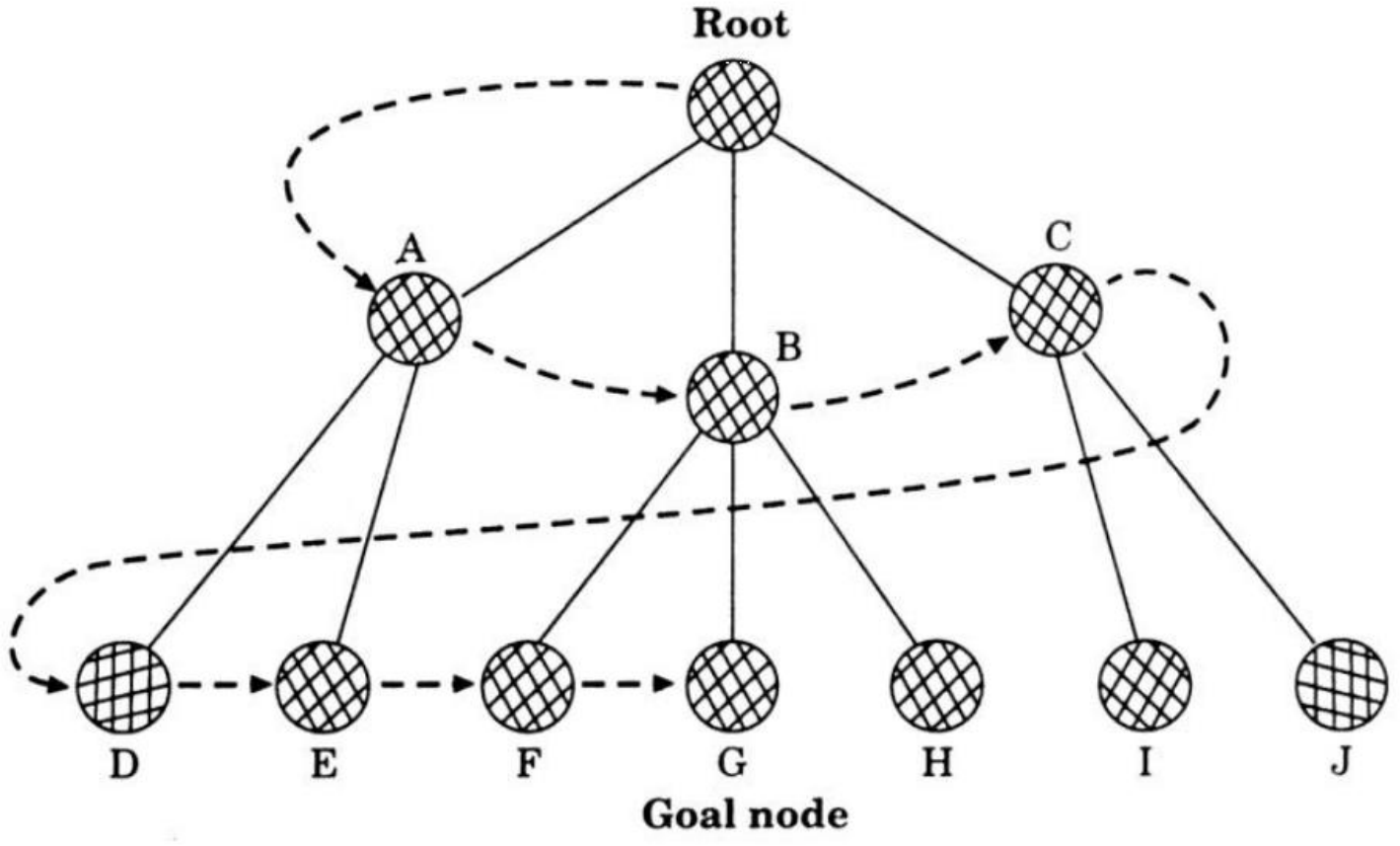
- **The uninformed search does not contain any domain knowledge such as closeness, the location of the goal.**
- **It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.**
- **Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search.**
- **It examines each node of the tree until it achieves the goal node.**

It can be divided into five main types:

- **Breadth-first search**
- Uniform cost search
- **Depth-first search**
- Iterative deepening depth-first search
- Bidirectional Search

Breadth-first search

- Breadth-first search is the simplest form of uniformed search. In this type of search the root node is expanded first, then all the successors of the root node are expanded next. then their successors and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. Note from the figure that we are using the convention that the alternatives are tried in the left-to-right order.



Breadth-first search Algorithm

1. Put the start node on a list, called OPEN of unexplored nodes. If the start node is goal node, a solution has been found.
2. If OPEN is empty, no solution exist.
3. Remove the first node, n form OPEN and place it in a list called CLOSED, of expanded nodes.
4. Expand node n. If it has no successor go to (2).
5. Place all successors of node n at the end of OPEN.
6. If any of the successors of node n is a goal state, solution has been found.

Advantages of Breadth-first Search

(1) Breadth-first search will not get trapped exploring a blind alley.

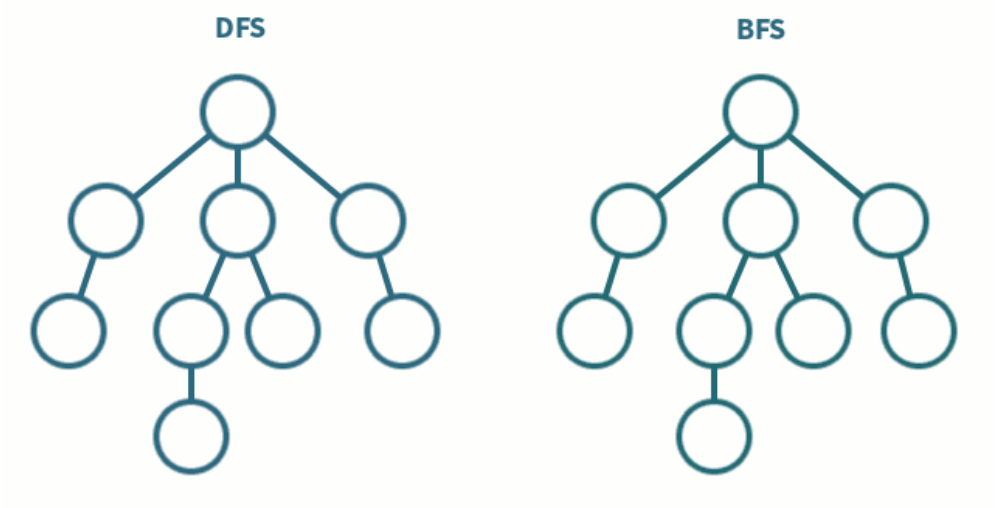
(2) If any solution exists, this method is guaranteed to find it.

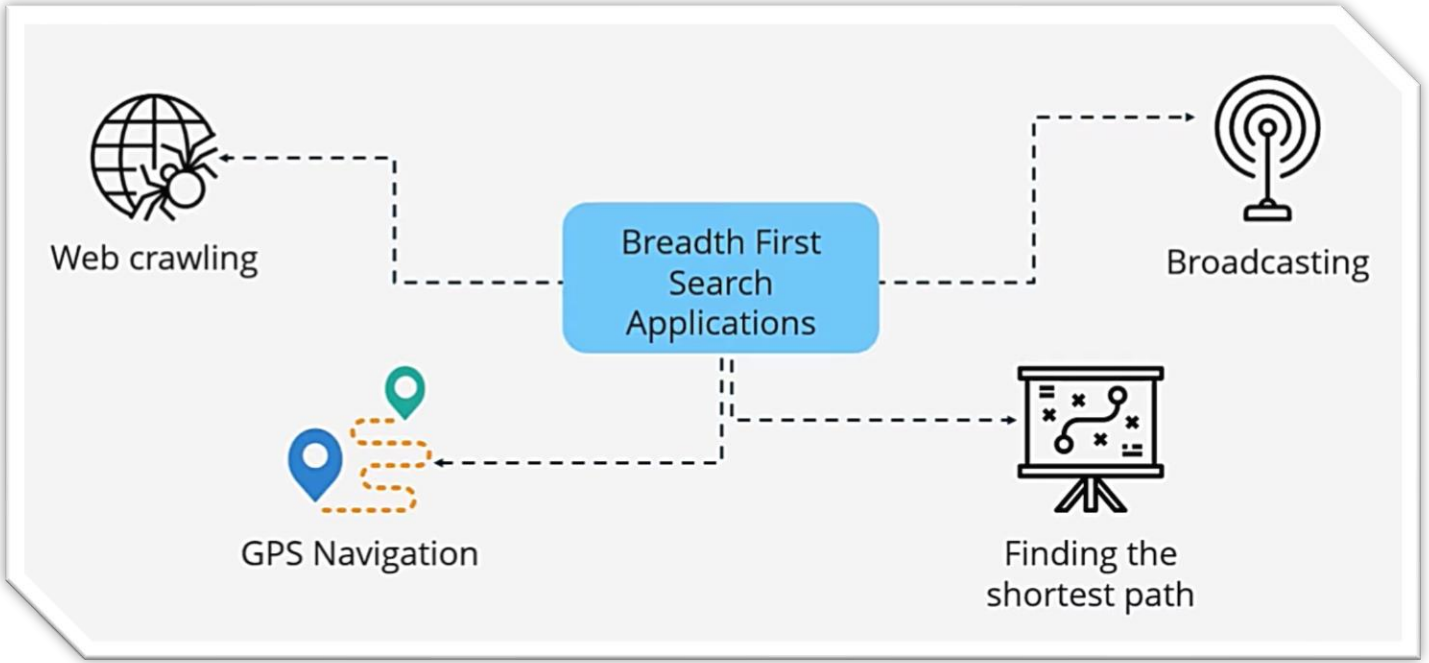
Disadvantages of Breadth-first Search

(1) The amount of time needed to generate all the nodes is considerable.

(2) The searching process remembers all unwanted nodes which is of no practical use for the search.

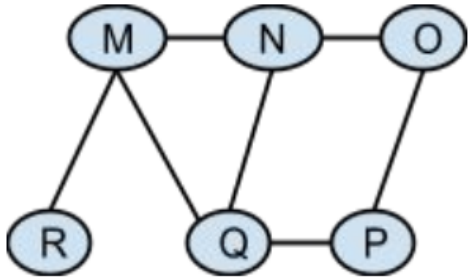
Applications of BFS Algorithm





Breadth-first search

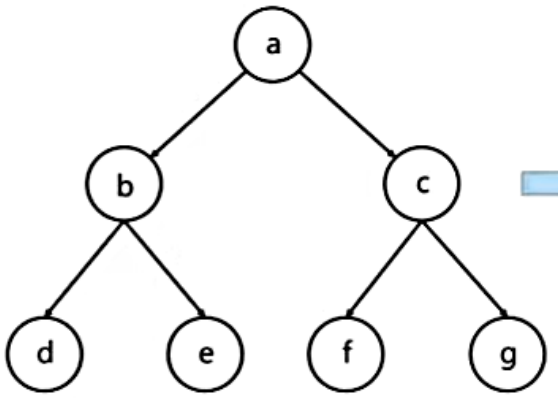
11. Consider the following graph.

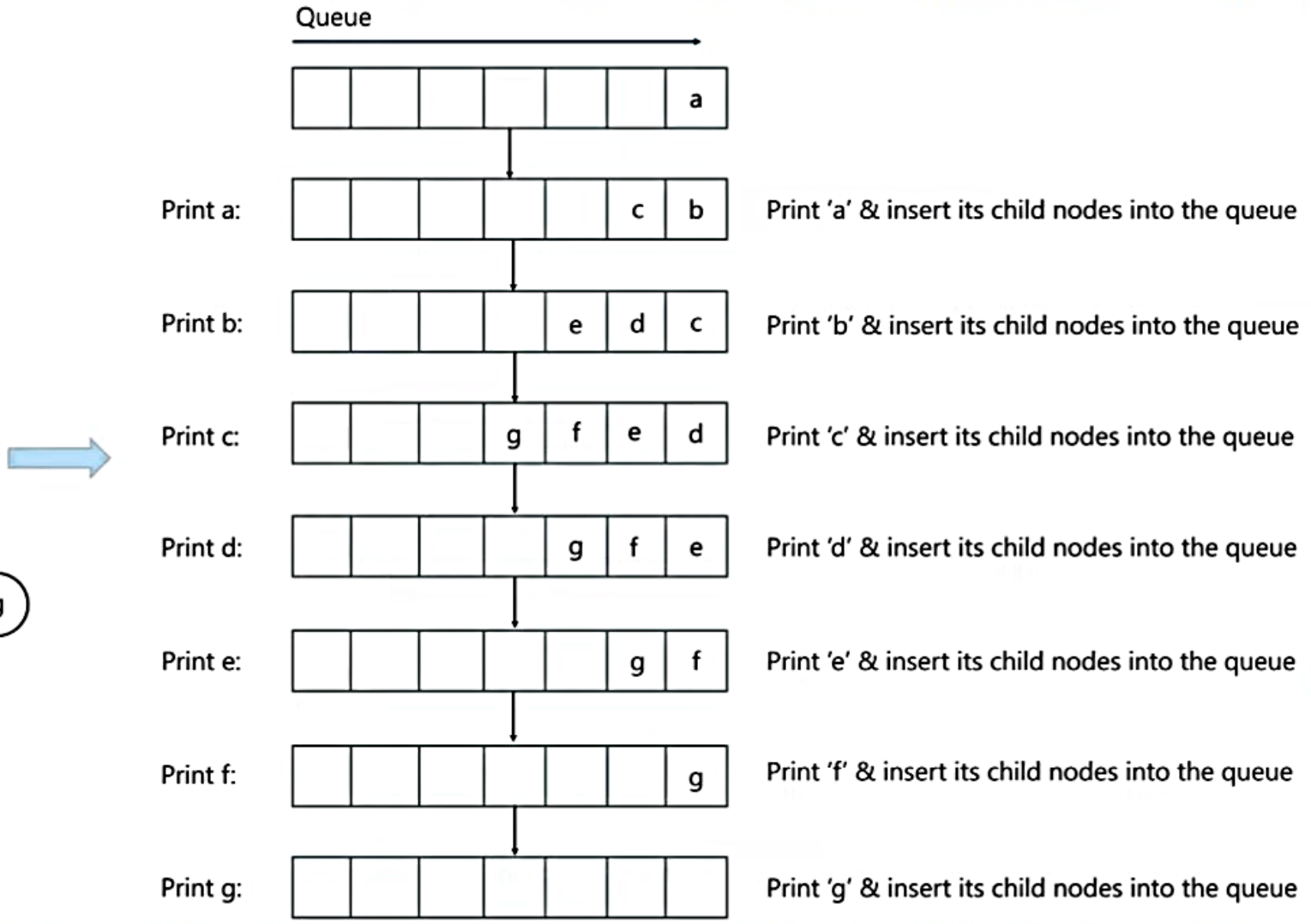


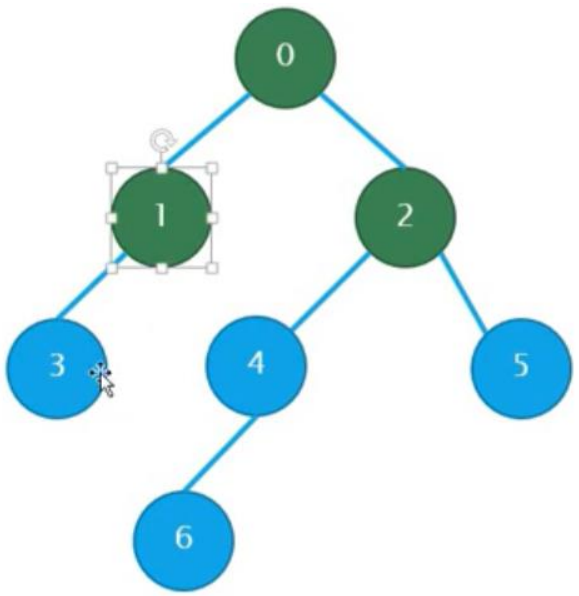
If we run breadth first search on this graph starting at any vertex, which one of the following is a possible order for visiting the nodes ?

A. MNOPQR
B. NQMPOR
C. QMNPRO
D. QMNPOR

Breadth-first search

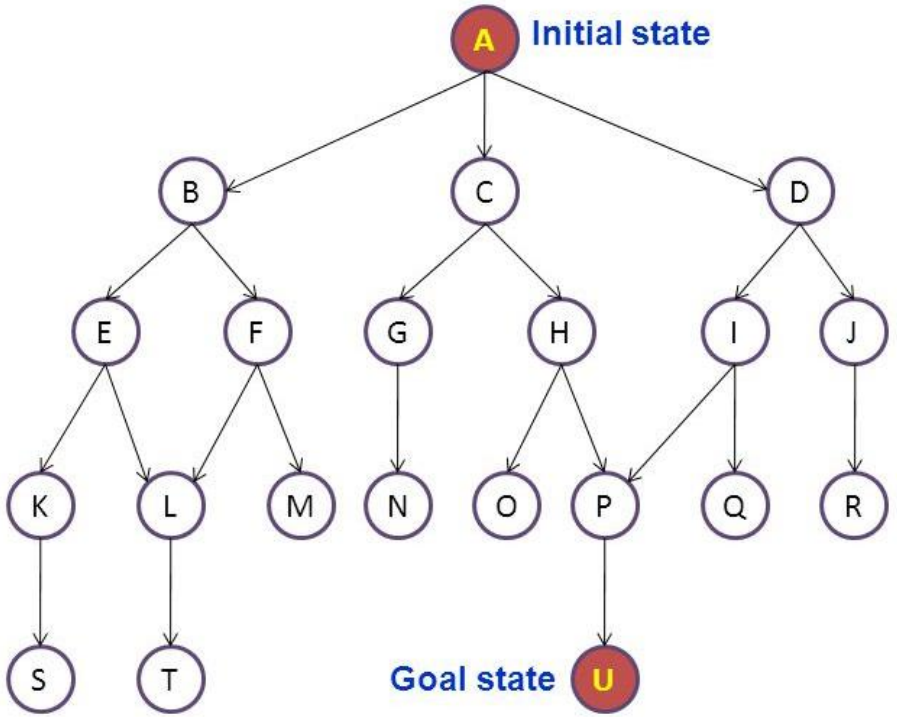






Breadth First Search examines the nodes in the following order:

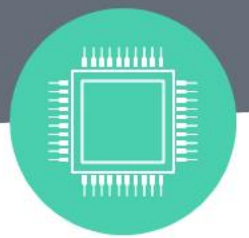
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U



OPEN	CLOSE
A	
BCD	A
CDEF	BA
DEFGH	CBA
EFGHIJ	DCBA
FGHIJKL	EDCBA
GHIJKLM	FEDCBA
HIJKLMN	GFEDCBA
IJKLMNOP	HGFEDCBA
JKLMNOPQ	IHGFEDCBA
KLMNOPQR	JIHGFEDCBA
LMNOPQRS	KJIHGFEDCBA
MNOPQRST	LKJIHGFEDCBA
NOPQRST	MLKJIHGFEDCBA
OPQRST	NMLKJIHGFEDCBA
PQRST	ONMLKJIHGFEDCBA
QRSTU	PONMLKJIHGFEDCBA
RSTU	QPONMLKJIHGFEDCBA
STU	RQPONMLKJIHGFEDCBA
TU	SRQPONMLKJIHGFEDCBA
U	TSRQPONMLKJIHGFEDCBA
U	hold

Artificial Intelligence

By
Dr. Manoj Kumar



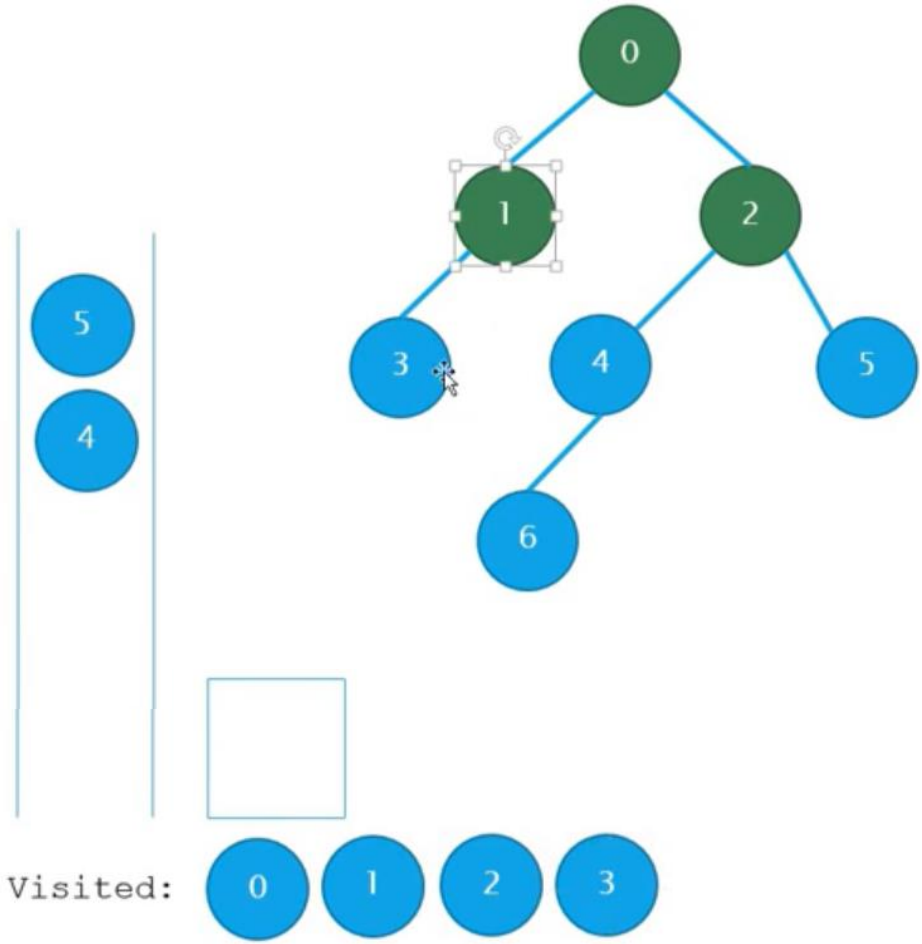
University School of Automation and Robotics
GGSIP University, East Campus, Delhi, India

Open and Close list search

Open list: Contains those nodes that have already completely expanded

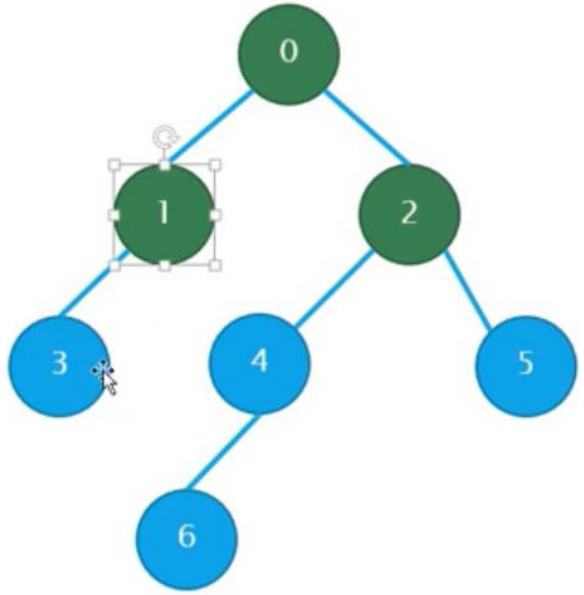
Closed list: Contains those nodes that have been generated but not yet expanded

- Initially, just the root node is included on the Open list, and the Close list is empty.
- At each cycle of the algorithm, an Open node of lowest cost is expanded, moved to Closed, and its children are inserted back to Open.
- The Open list is maintained as a priority queue
- The algorithm terminates when a goal node is chosen for expansion, or there are no more nodes remaining in Open list.



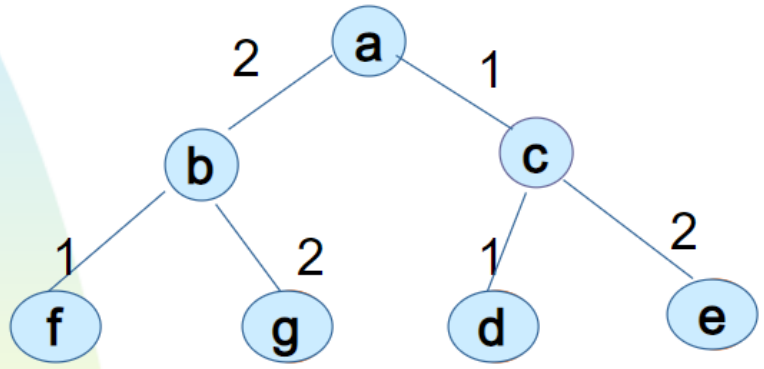
Open and Close list search

- Let g(n) be the sum of the edge's costs from root to node n.
- Initially the root node is placed in Open with a cost of zero.
- At each step, the next node n to be expanded is an Open node whose cost g(n) is lowest among all Open nodes.



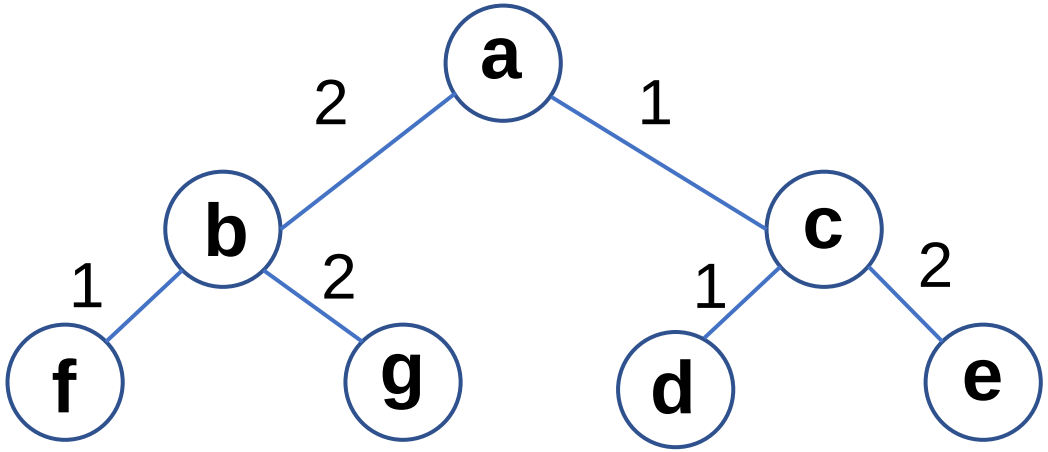
Open and Close list search

- **Assume an example tree with different edge costs, represented by numbers next to the edges.**

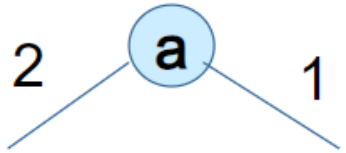


Notations for this example:

- generated node
- expanded node

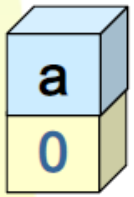


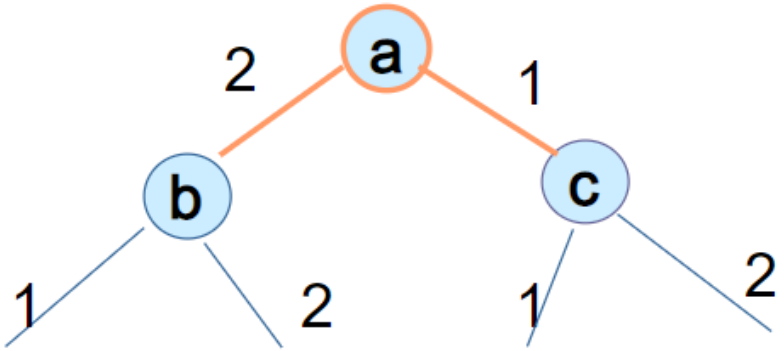
Open and Close list search



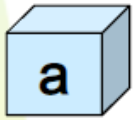
Closed list:

Open list:

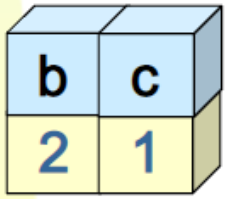




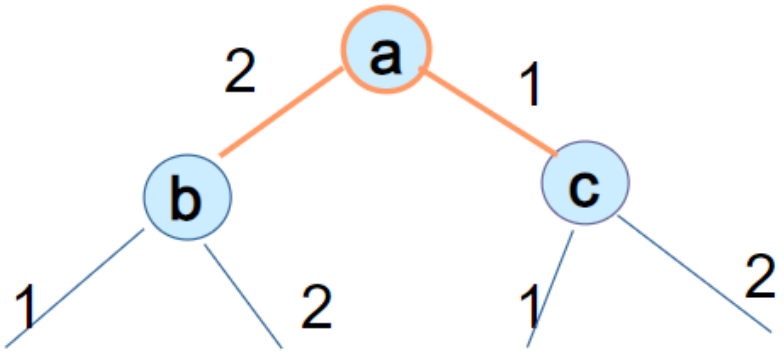
Closed list:



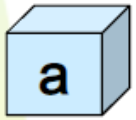
Open list:



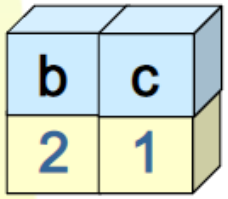
Open and Close list search

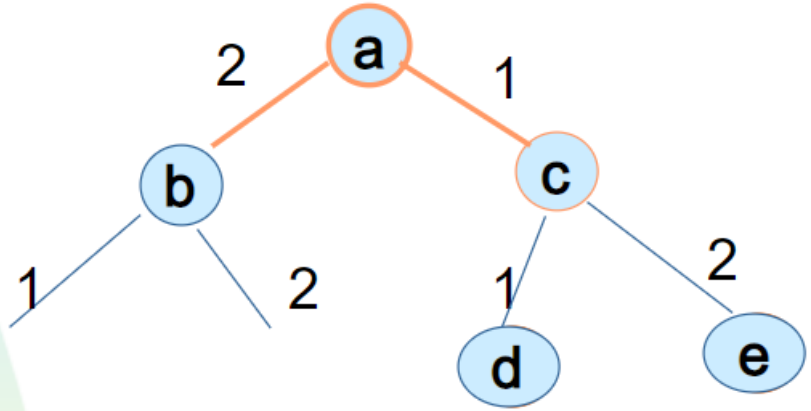


Closed list:

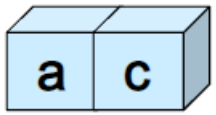


Open list:

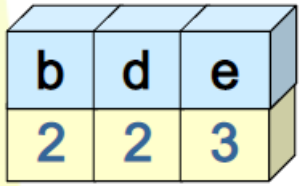




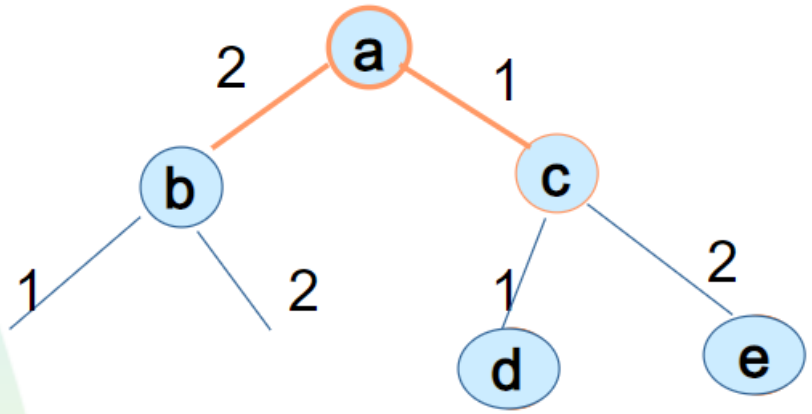
Closed list:



Open list:



Open and Close list search

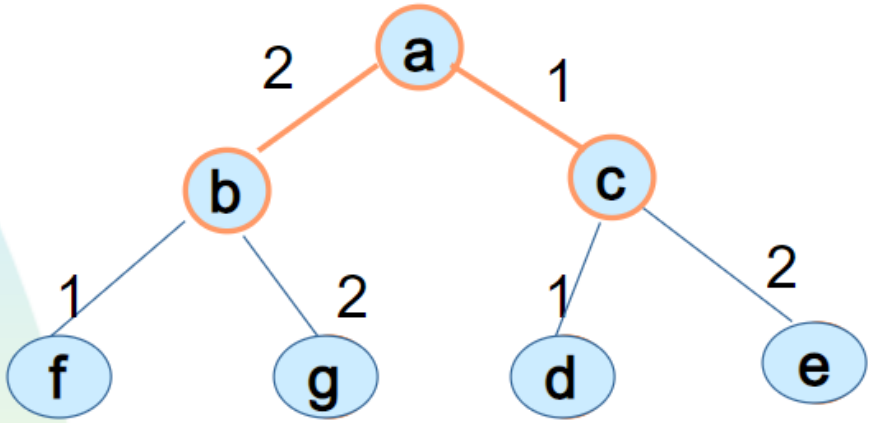


Closed list:

a	c

Open list:

b	d	e
2	2	3



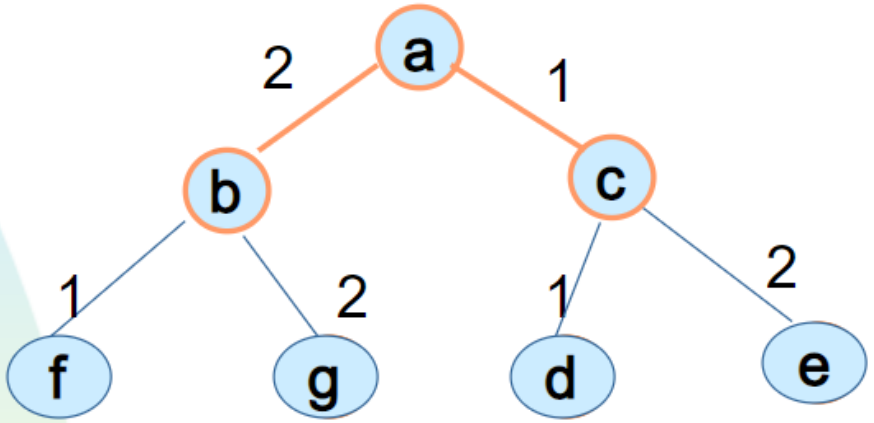
Closed list:

a	c	b

Open list:

d	e	f	g
2	3	3	4

Open and Close list search

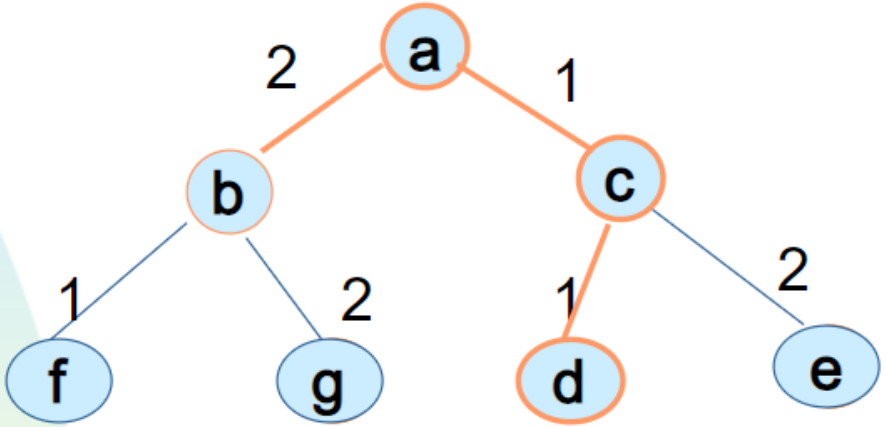


Closed list:

a	c	b

Open list:

d	e	f	g
2	3	3	4



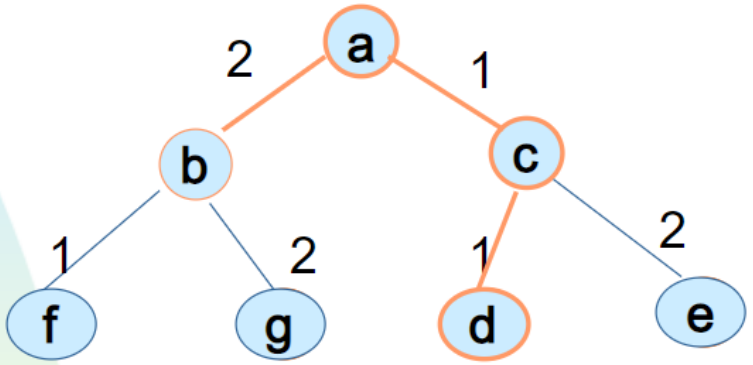
Closed list:

a	c	b	d

Open list:

e	f	g
3	3	4

Open and Close list search

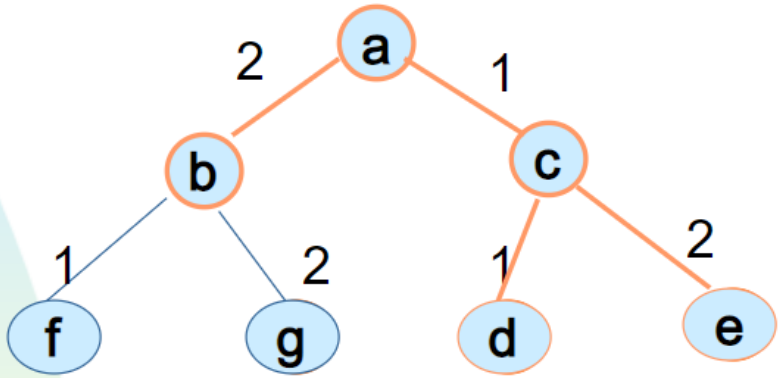


Closed list:

a	c	b	d

Open list:

e	f	g
3	3	4



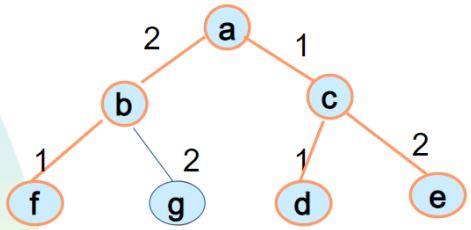
Closed list:

a	c	b	d	e

Open list:

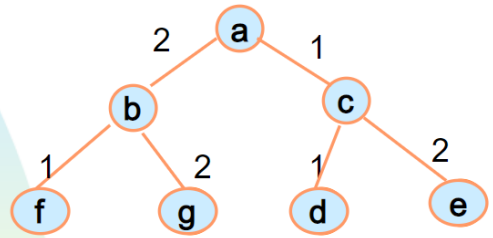
f	g
3	4

Open and Close list search



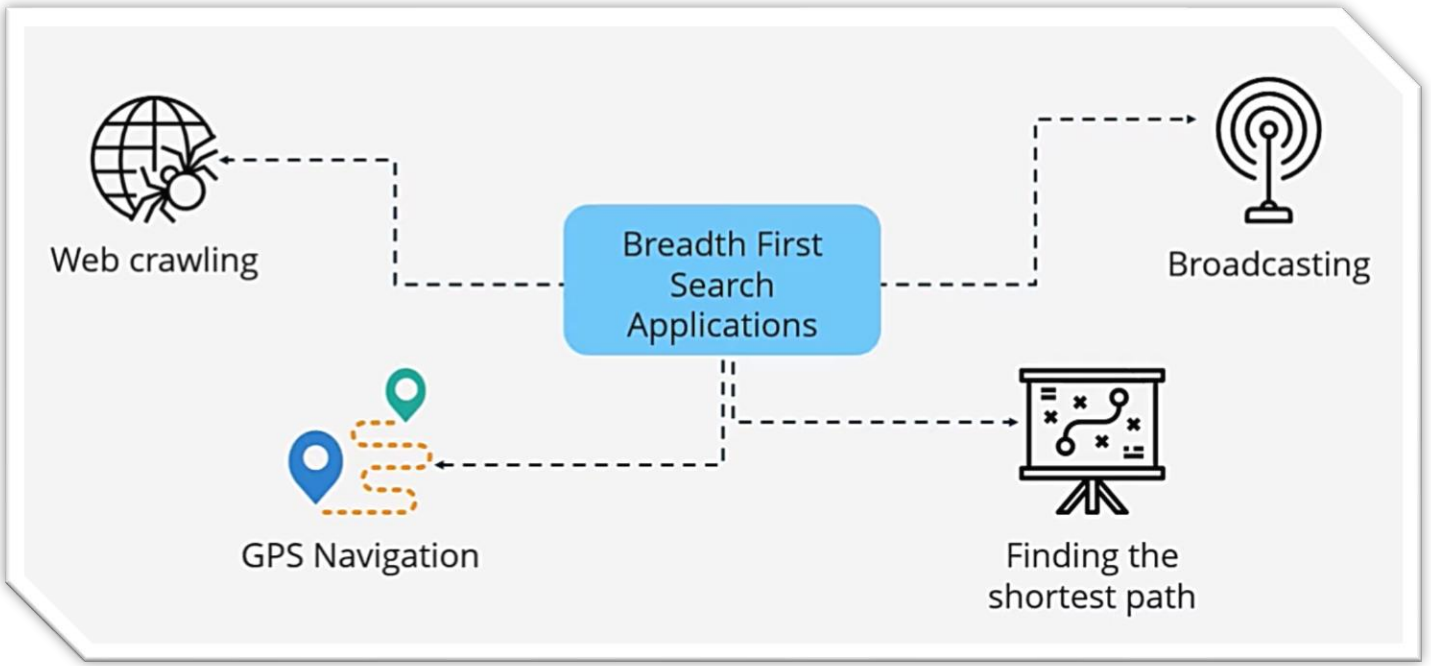
Closed list: a c b d e f

Open list: g 4



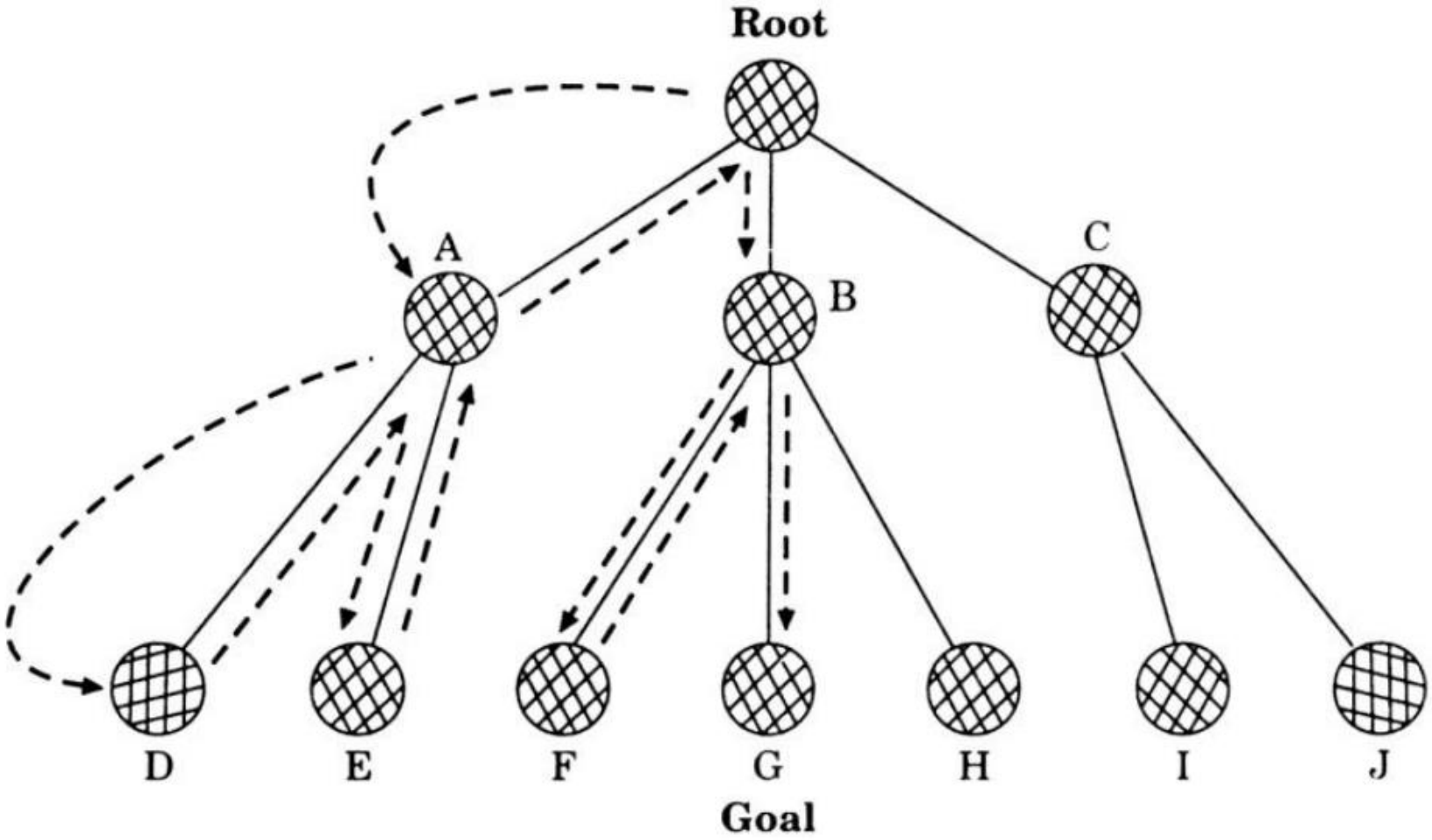
Closed list: a c b d e f g

Open list:



Depth-first search

- **Depth-first search is characterized by the expansion of the most recently generated, or deepest node, first. The progress of the search can be understood from figure. The most elementary form of a depth first search:**



Depth-First Search Algorithm

```
begin
        if n is a goal node then
                begin
                        solution : = n;
                        exit;
                end;
        for each successor n i of n do
        If n i is not an ancestor of n then
                        UDFS (n i);
                return;
end;
```

(1) It has a modest memory requirement. It only needs to store the path from the root to the leaf node as well as the unexpanded nodes.

(2) Depth-first search is neither complete nor optimal. If depth-first search goes down an infinite branch, it will not terminate unless a goal state is found. Even if a solution is found, there may be a better solution at a higher level in the tree.

Depth-First Search Algorithm

Advantages of Depth-First Search

(1) Depth-first search requires less memory since only the nodes on the current path are stored.

(2) It is by chance (or we make take care in ordering the alternative successor states) that we may find a solution without examining much of the search space.

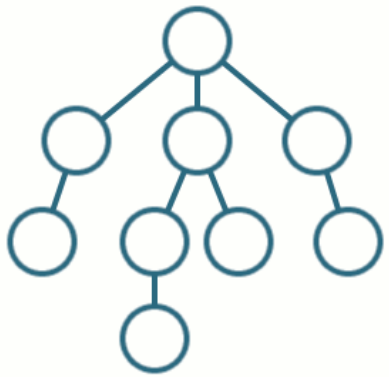
Disadvantages of Depth-First Search

This type of search can go on and on, deeper and deeper into the search space and we can get lost (blind alley).

Applications of BFS Algorithm

Breadth-first	Depth-first
1. No blind alley exists.	1. Blind alley exists.
2. If a solution exists it will be found.	2. Even if solution exists, it explores one branch only and it may declare failure. Chance of success becomes still less when loop exists.
3. May find many solutions. If many solutions exist, minimal solution can be found.	3. It stops after one solution is found. Minimal solution may not be found as it explores only one branch.
4. Requires more memory because all the off-springs of the tree must be explored on level n before a solution on level $(n + 1)$ is to be examined.	4. Requires less memory because only one branch is scanned. It stops after solution is found.

DFS



BFS

Depth-First Search Algorithm

The depth first search is not considered to be best-first-search, because it does not maintain Open and Closed lists, in order to run in linear space.

Depth-First Search Algorithm

A strategy is defined by picking the order of node expansion

Strategies are evaluated along the following dimensions:

completeness – does it always find a solution if one exists?

optimality – does it always find a least-cost solution?

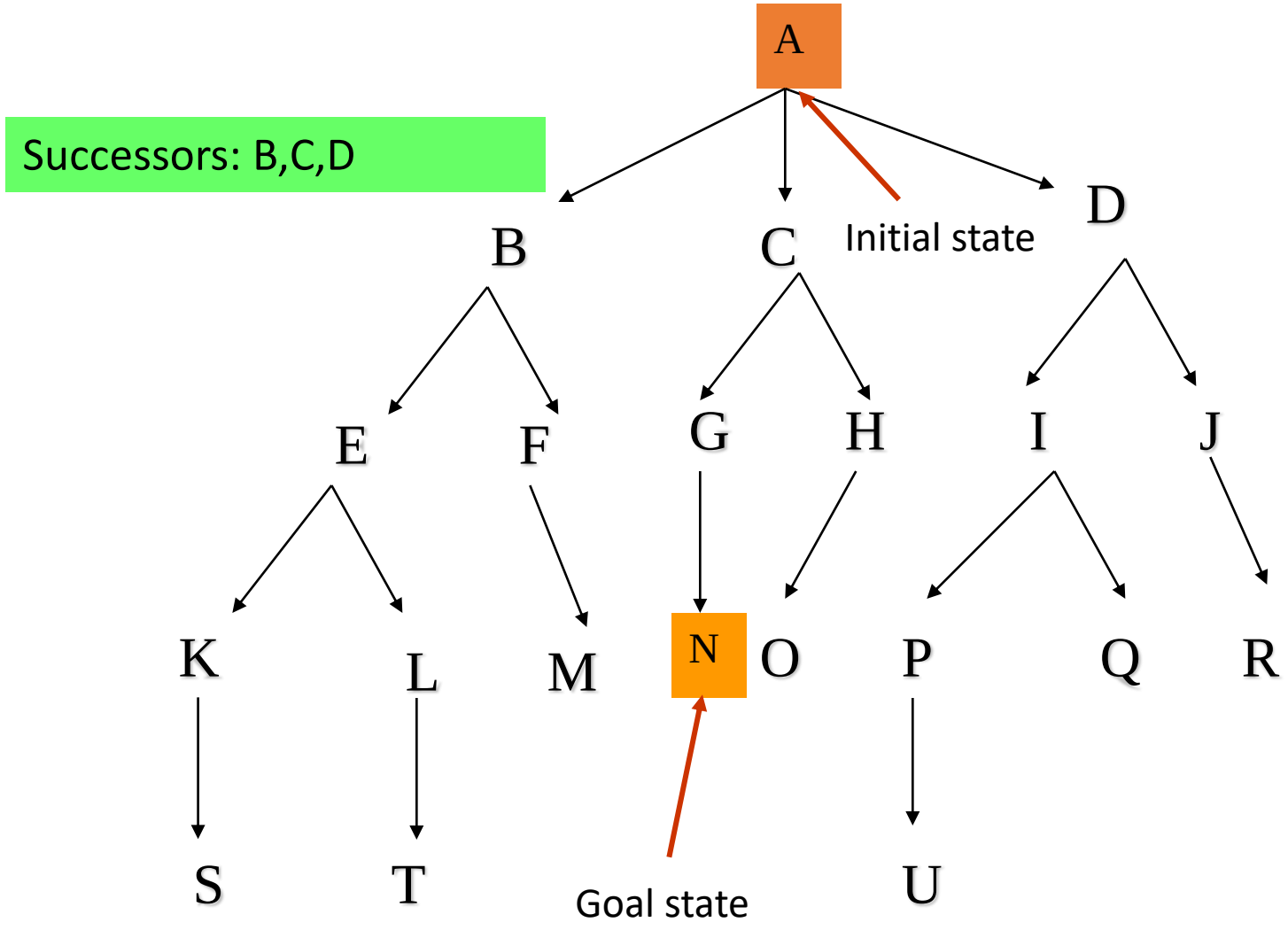
time complexity – number of nodes generated/expanded

space complexity – maximum number of nodes in memory

Time and space complexity are measured in terms of

- b – maximum branching factor of the search tree
- d – depth of the least-cost solution
- m – maximum depth of the state space (may be infinite)

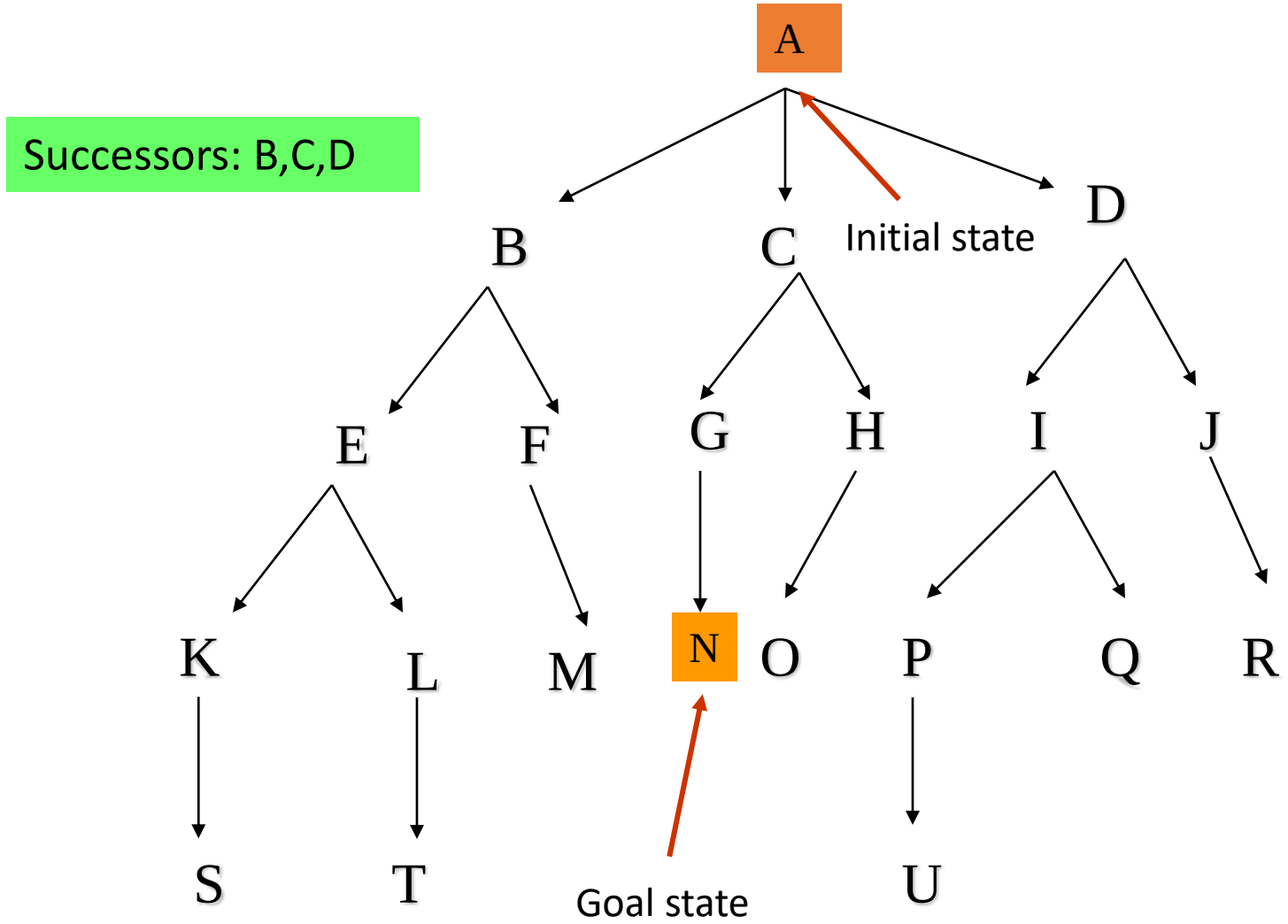
DFS Example



Fringe: A (LIFO)

Visited:

DFS Example

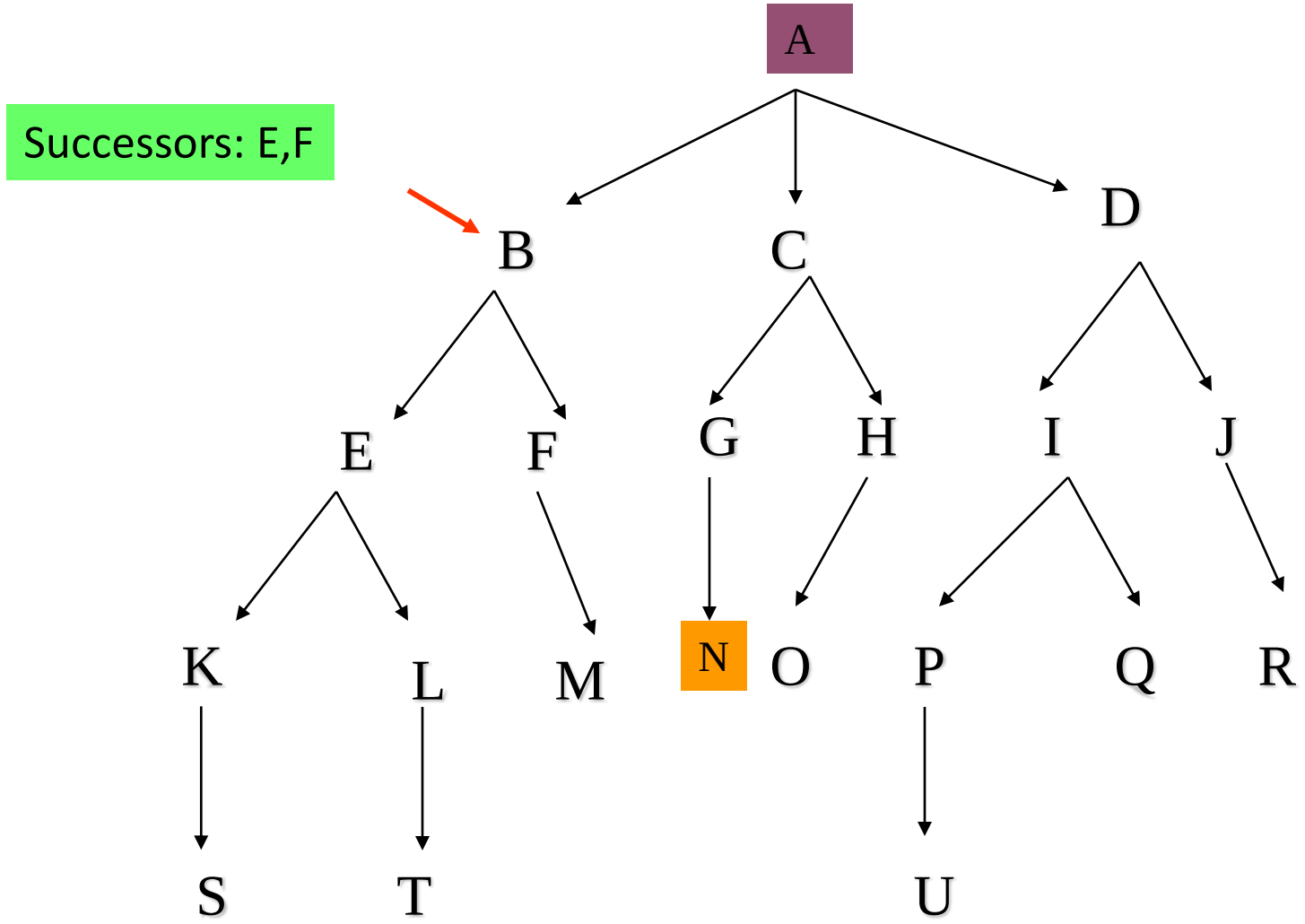


Successors: B,C,D

Fringe: A (LIFO)

Visited:

DFS Example

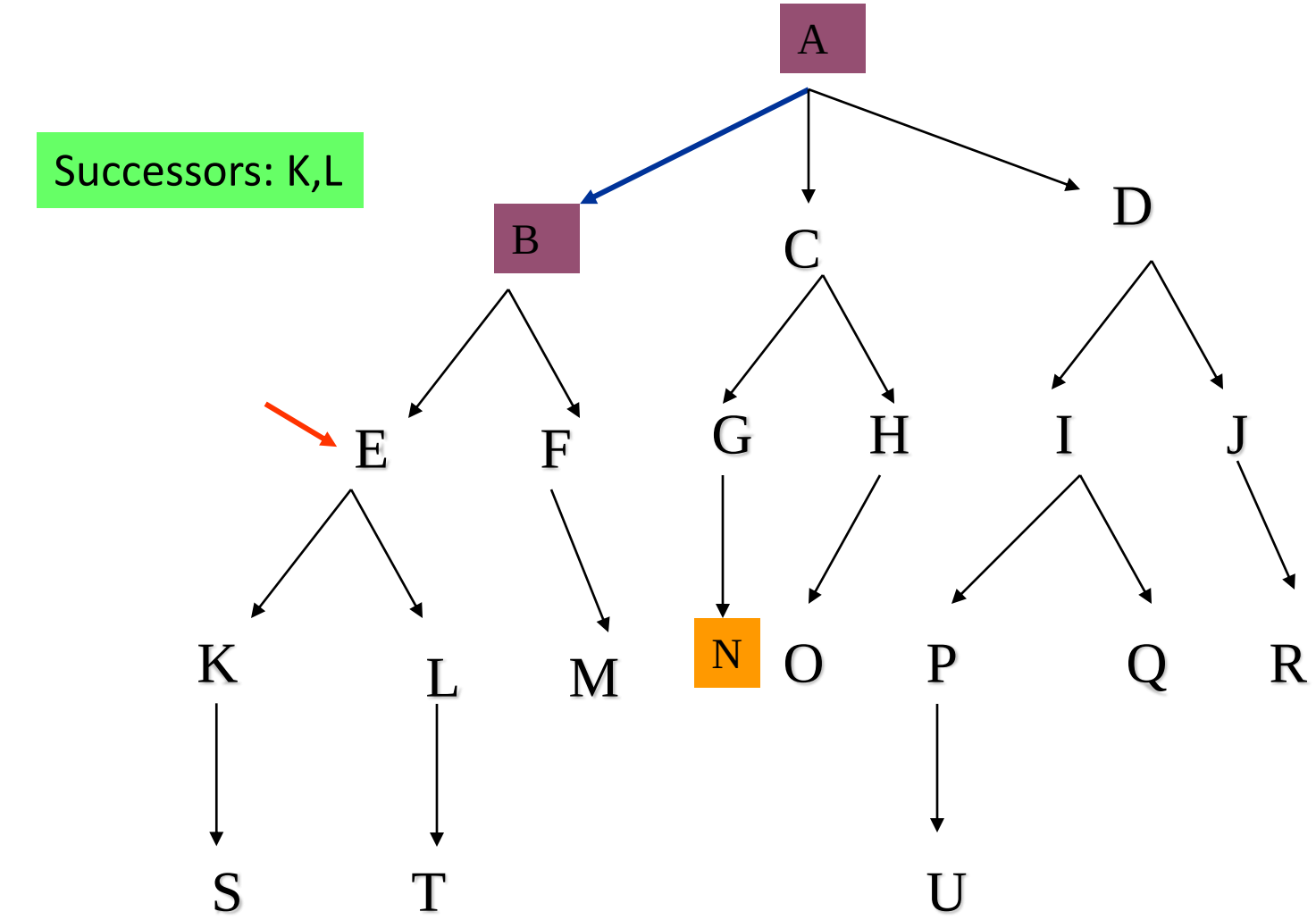


Successors: E,F

Fringe: B,C,D (LIFO)

Visited: A

DFS Example

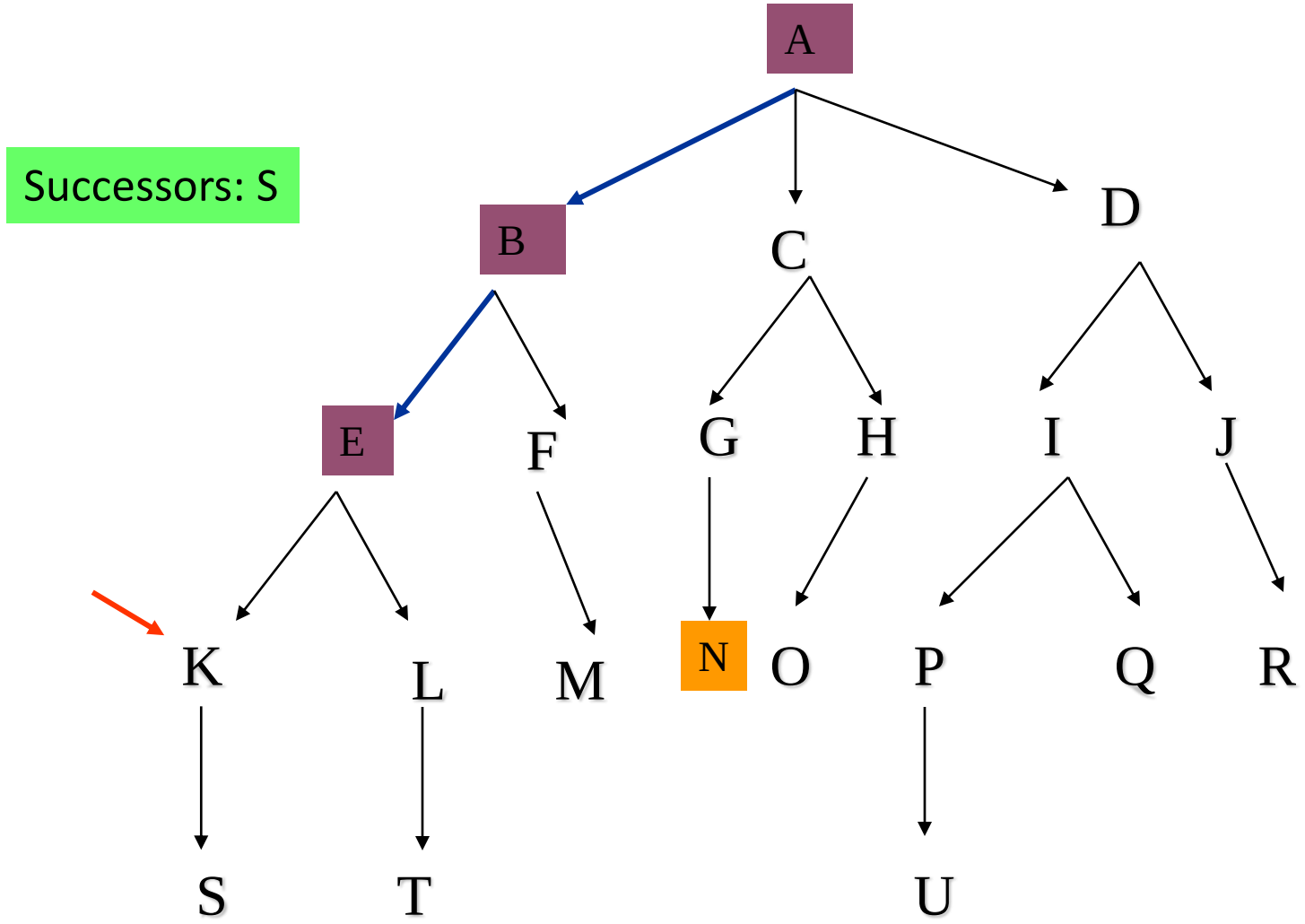


Successors: K,L

Fringe: E,F,C,D (LIFO)

Visited: A, B

DFS Example

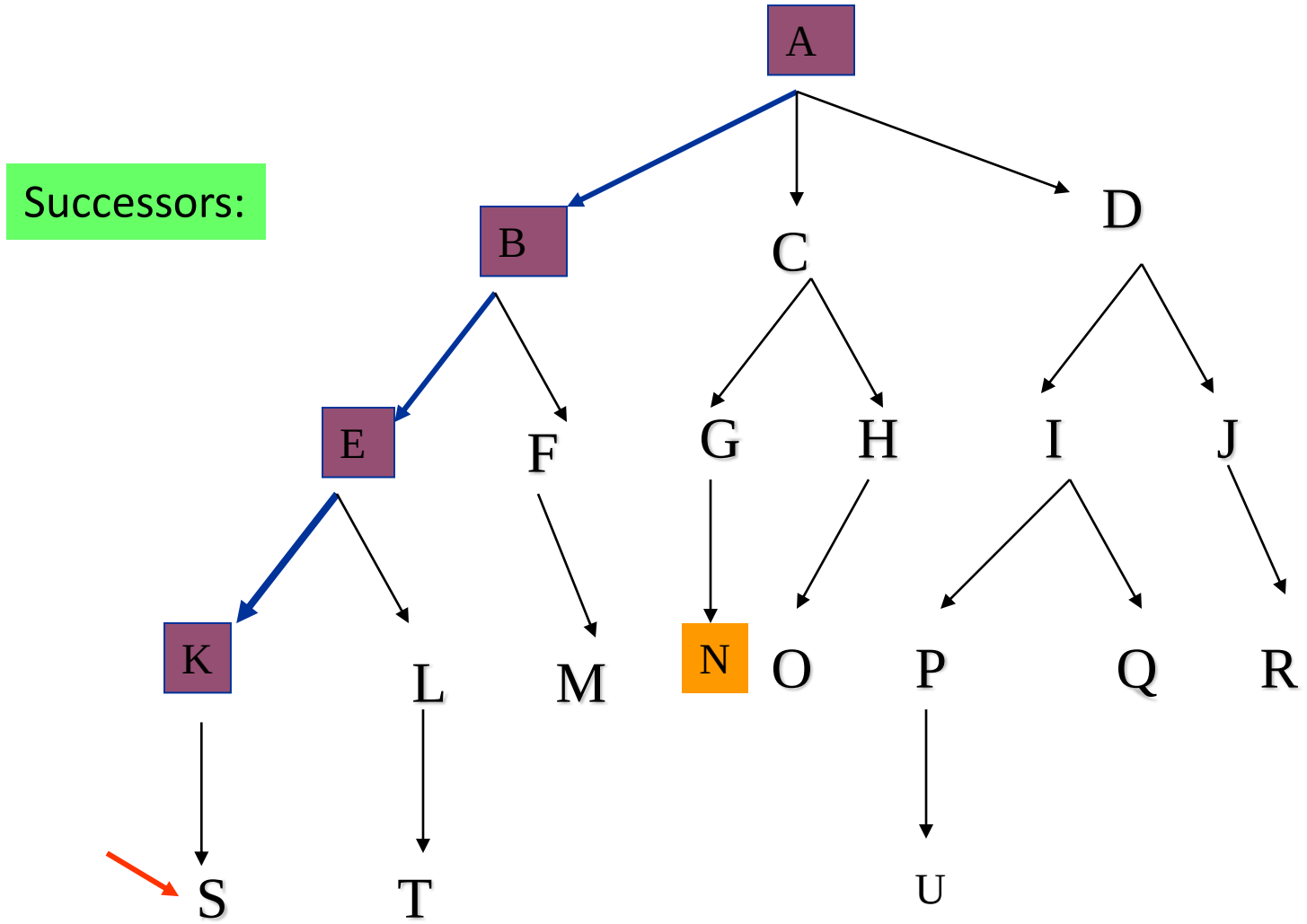


Successors: S

Fringe: K,L,F,C,D (LIFO)

Visited: A, B, E

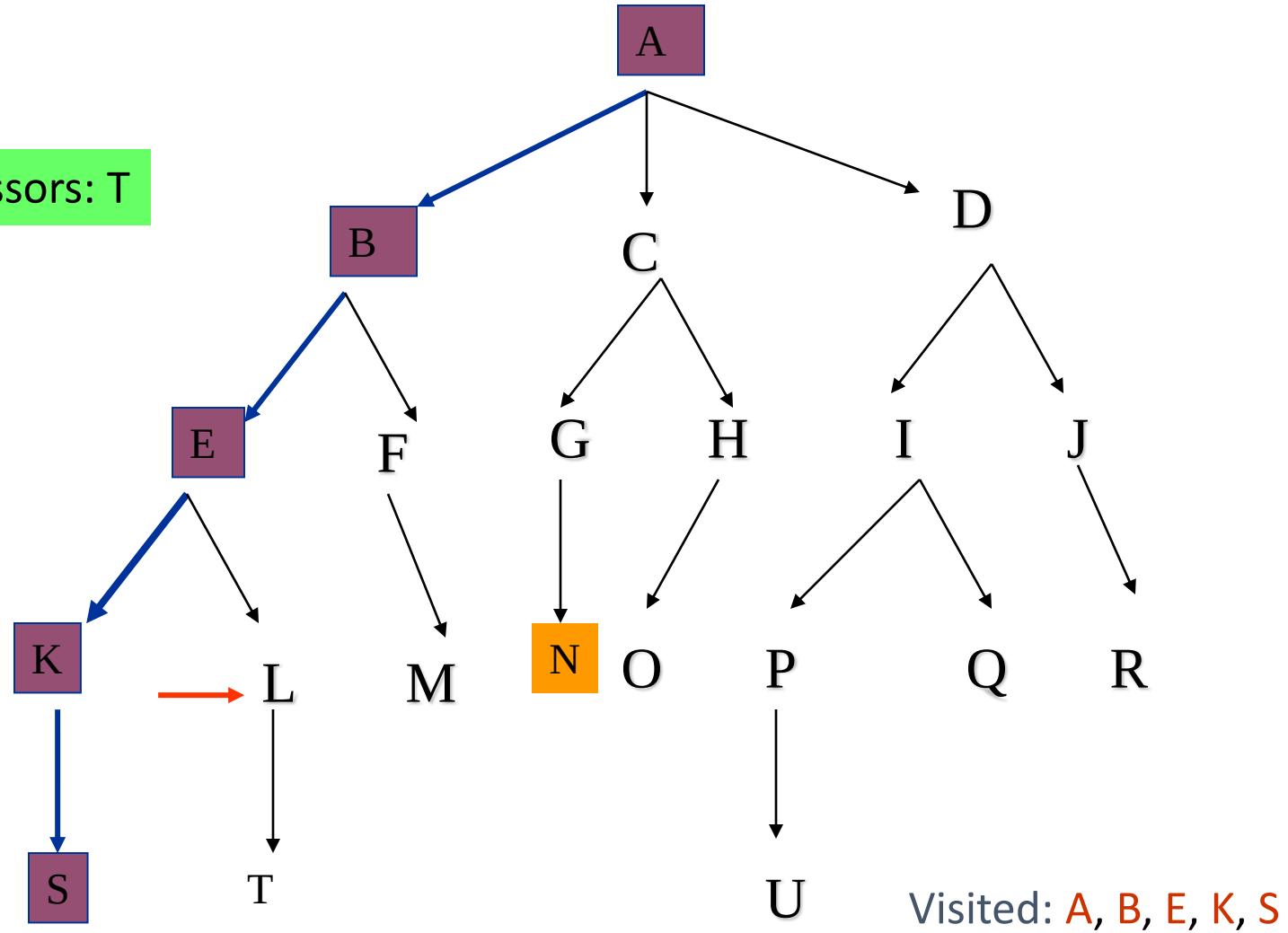
DFS Example



Fringe: S,L,F,C,D (LIFO)

Visited: A, B, E, K

DFS Example



Successors: T

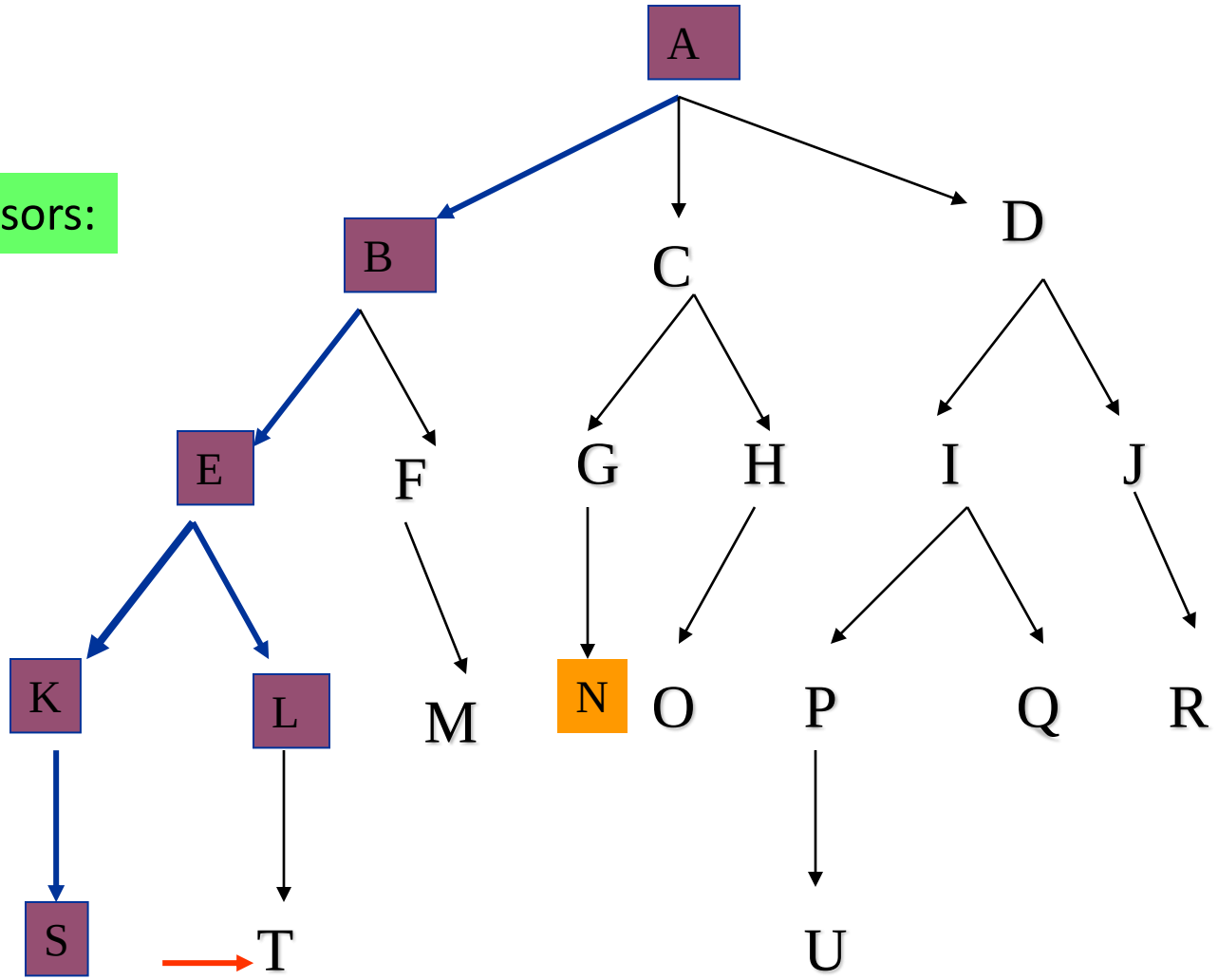
Fringe: L,F,C,D (LIFO)

Visited: A, B, E, K, S

Backtracking

DFS Example

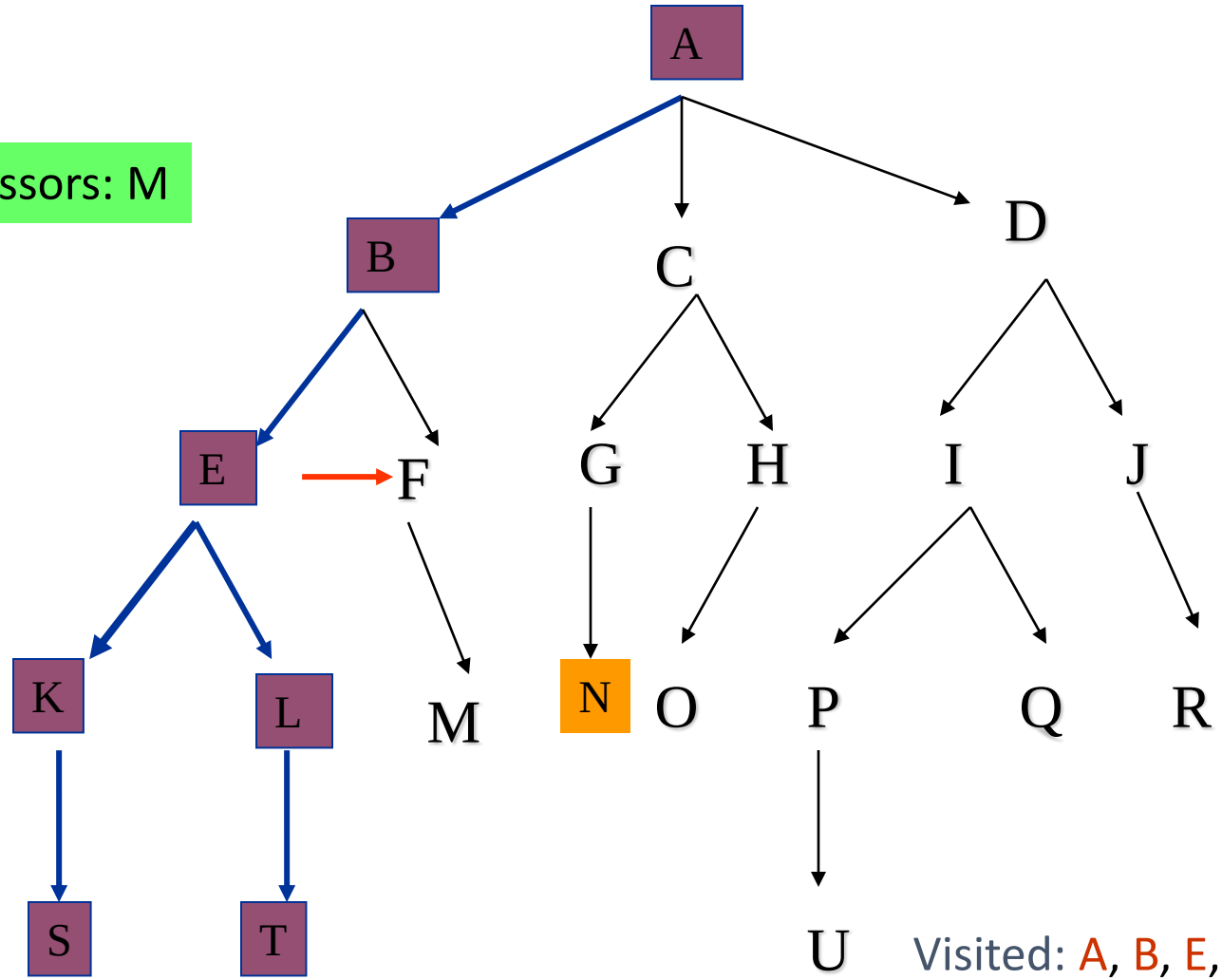
Successors:



Fringe: T,F,C,D (LIFO)

Visited: A, B, E, K, S, L

DFS Example



Successors: M

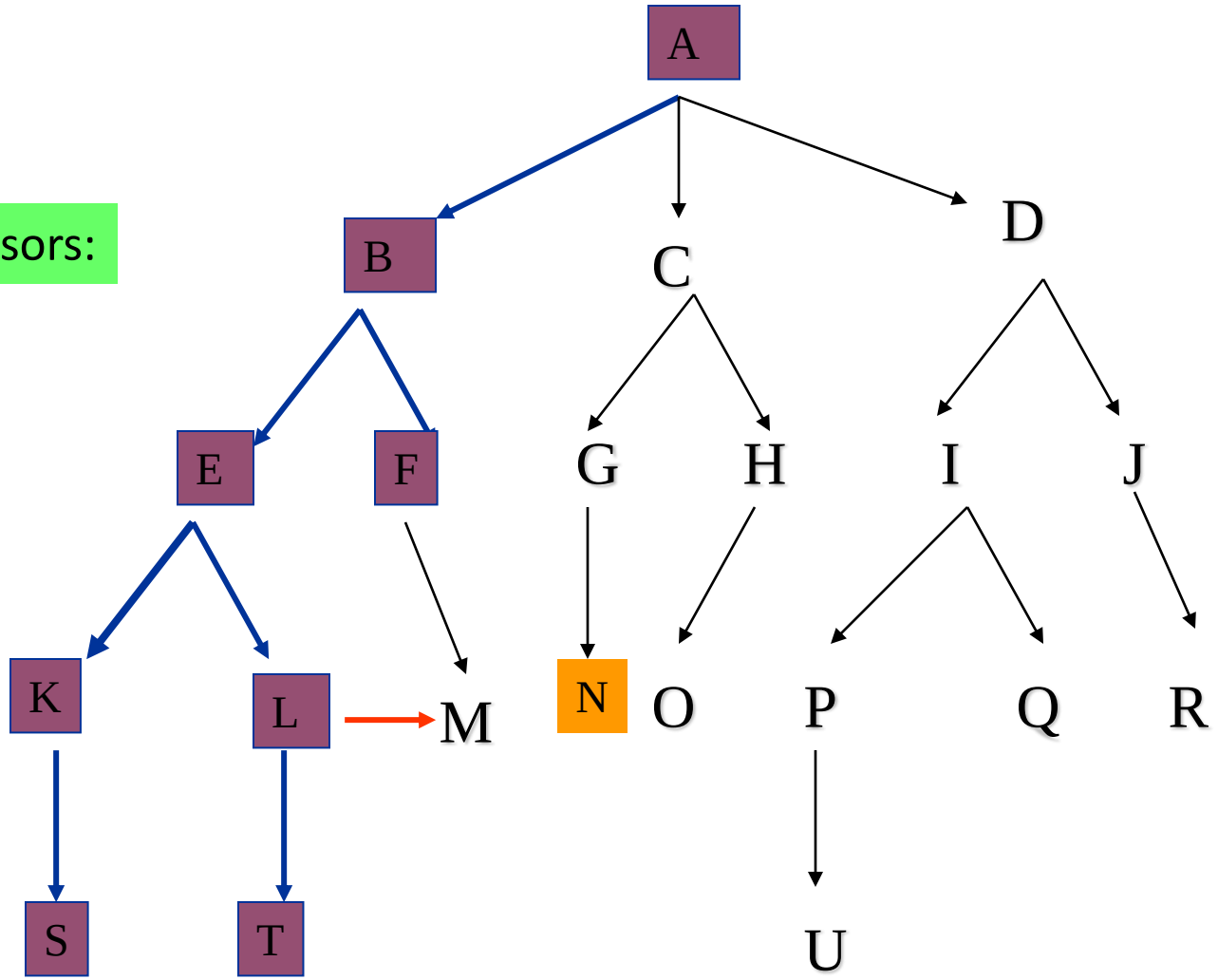
Visited: A, B, E, K, S, L, T

Fringe: F,C,D (LIFO)

Backtracking

DFS Example

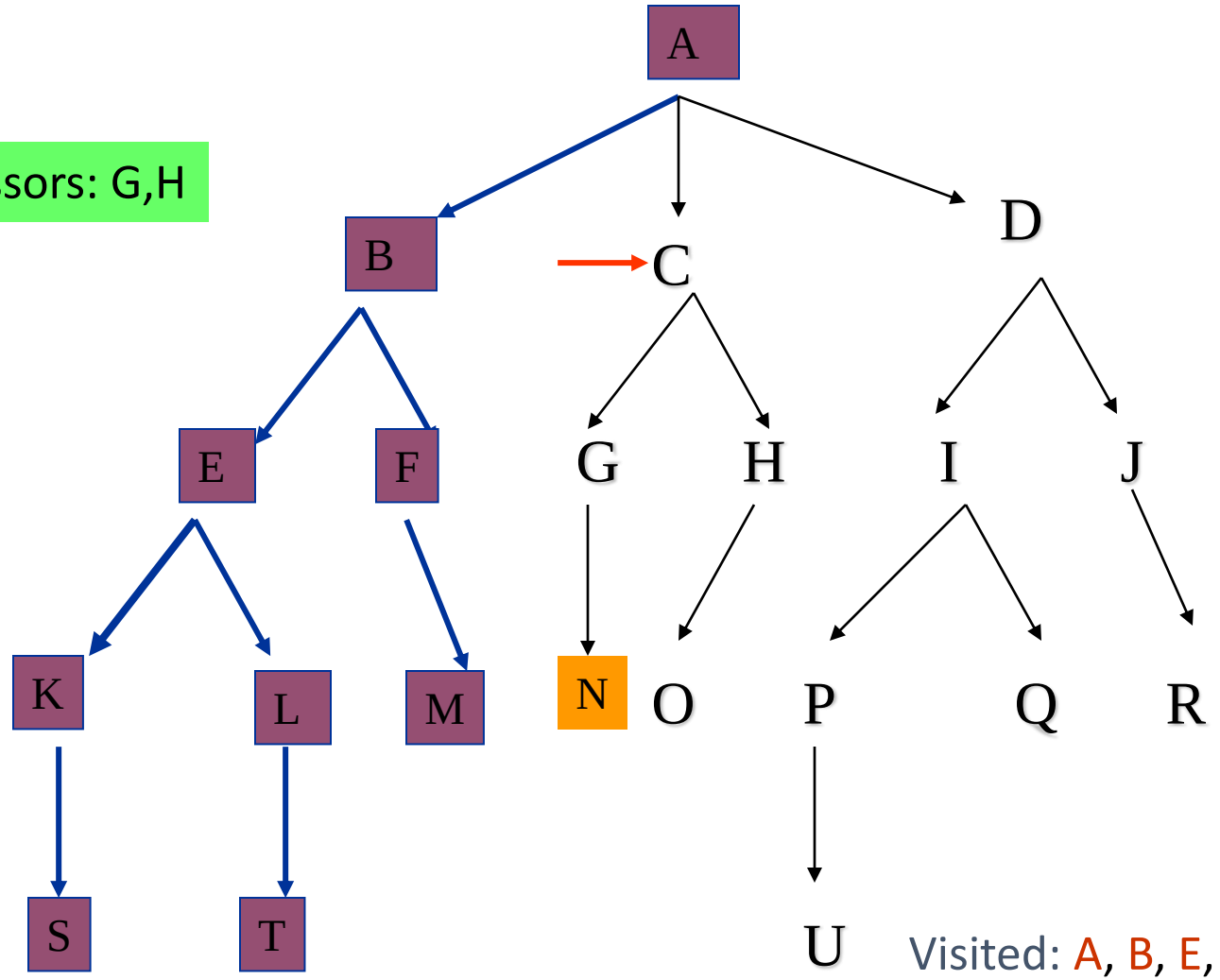
Successors:



Fringe: M,C,D (LIFO)

Visited: A, B, E, K, S, L, T, F

DFS Example



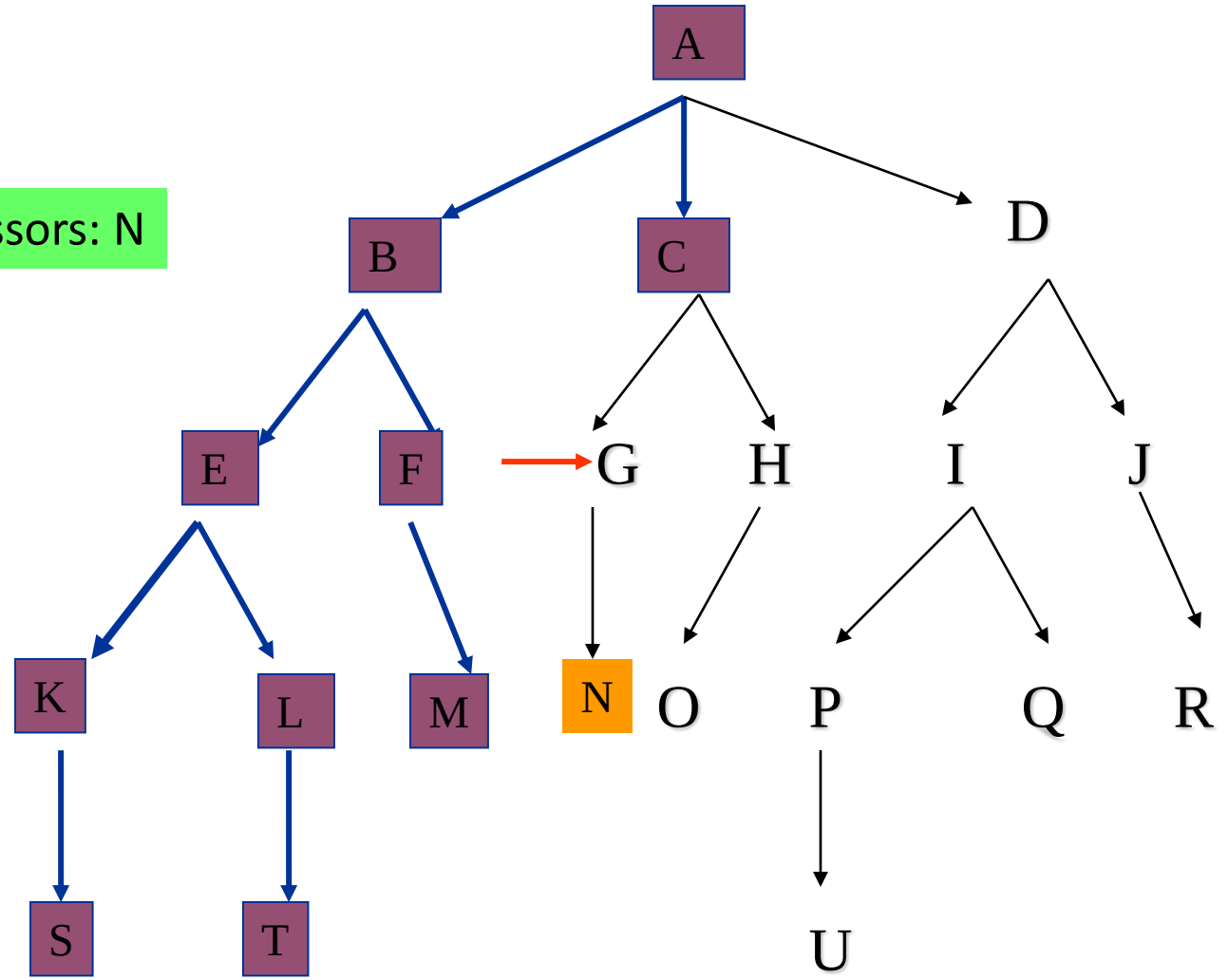
Successors: G,H

Fringe: C,D (LIFO)

Visited: A, B, E, K, S, L, T, F, M

Backtracking

DFS Example

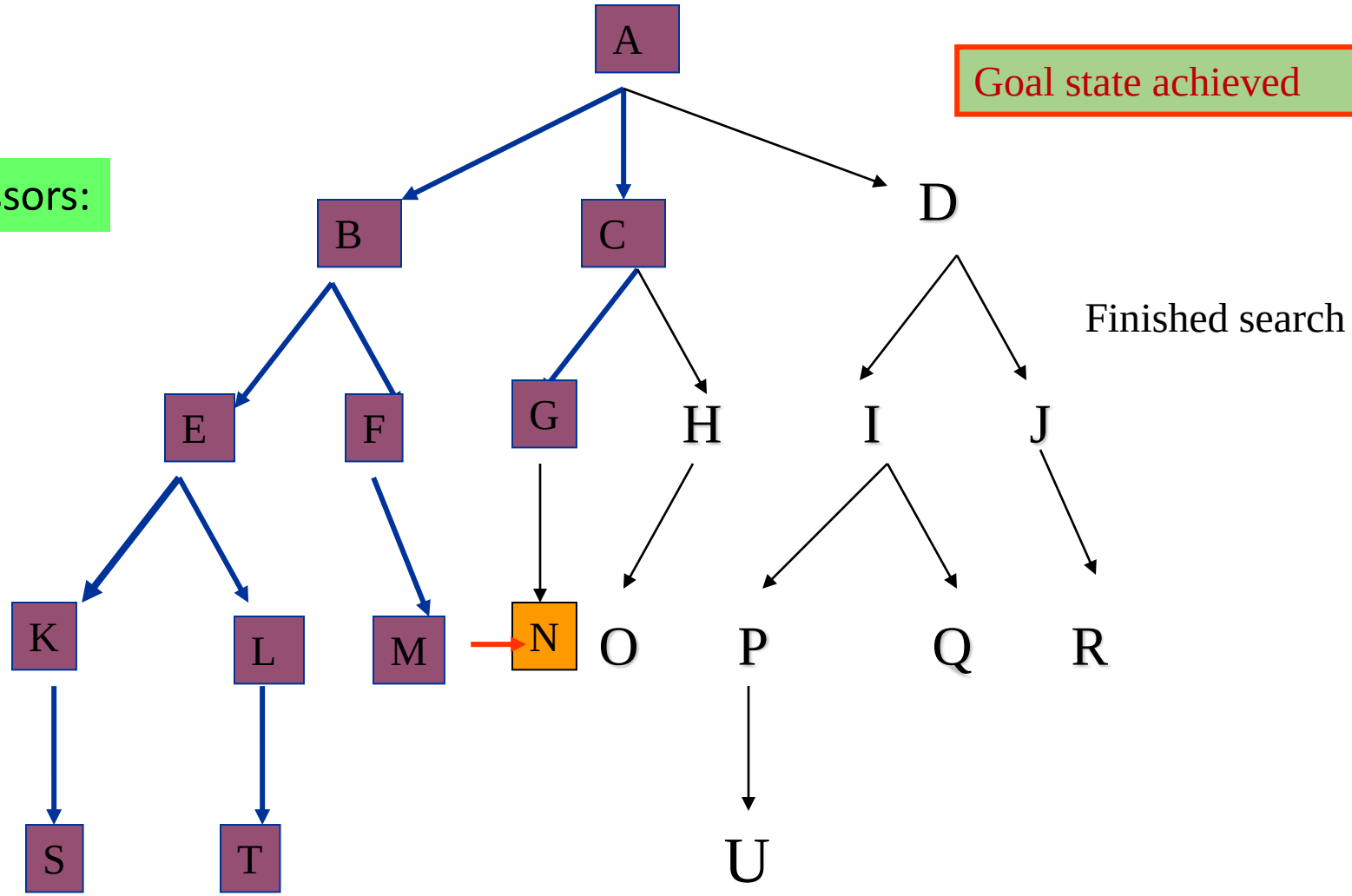


Successors: N

Fringe: G,H,D (LIFO)

Visited: A, B, E, K, S, L, T, F, M, C

DFS Example



Goal state achieved

Successors:

Finished search

Fringe: N,H,D (LIFO)

Visited: A, B, E, K, S, L, T, F, M, C, G

Depth-limited search

- **To keep depth-first search from wandering down an infinite path, we can use depth-limited search, a version of depth-first search in which we supply a depth limit, ℓ, and treat all nodes at depth ℓ as if they had no successors**

Is a depth-first search with a predetermined depth limit l

Implementation:

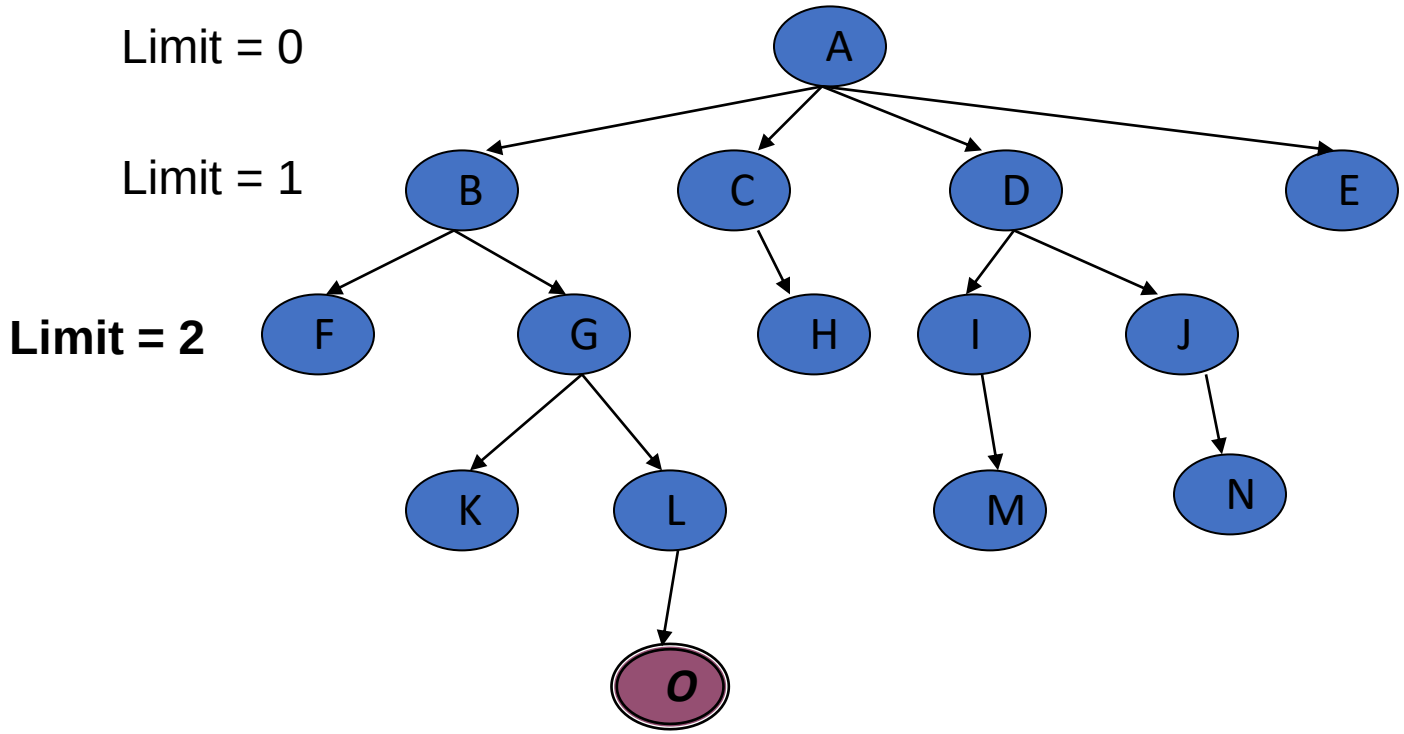
Nodes at depth l are treated as if they have no successors.

Complete: if cutoff chosen appropriately then it is guaranteed to find a solution.

Optimal: it does not guarantee to find the least-cost solution (If choosing $l > d$)

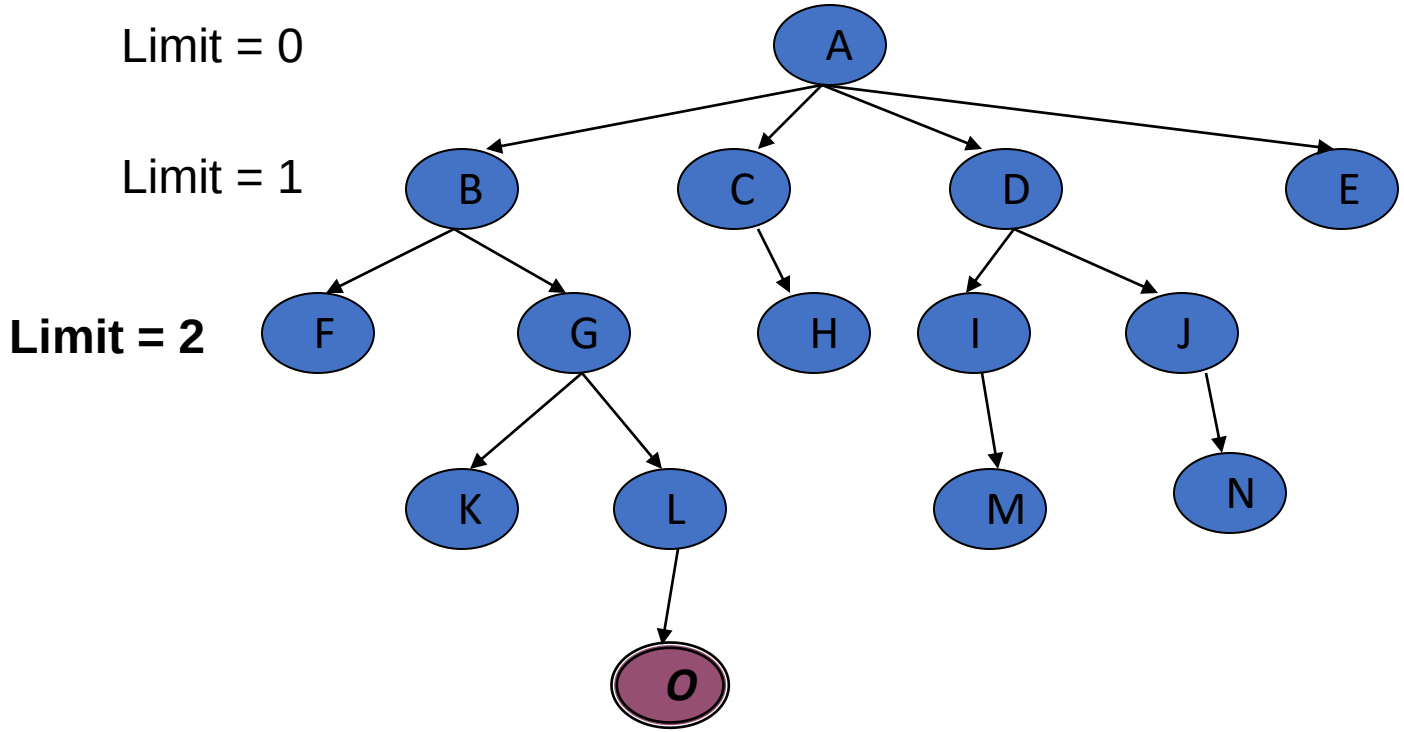
Depth-limited search

Given the following state space (tree search), give the sequence of visited nodes when using DLS (Limit = 2):



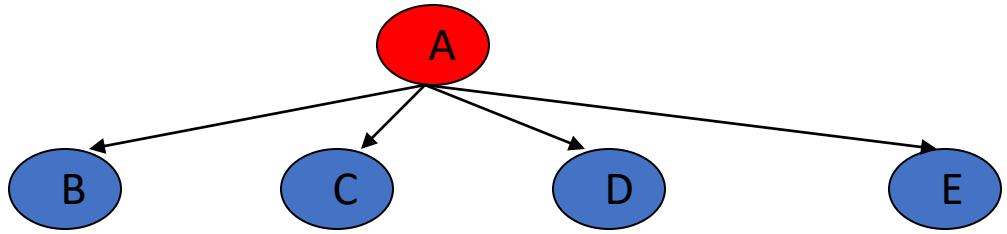
Depth-Limited Search (DLS)

Given the following state space (tree search), give the sequence of visited nodes when using DLS (Limit = 2):



Depth-Limited Search (DLS)

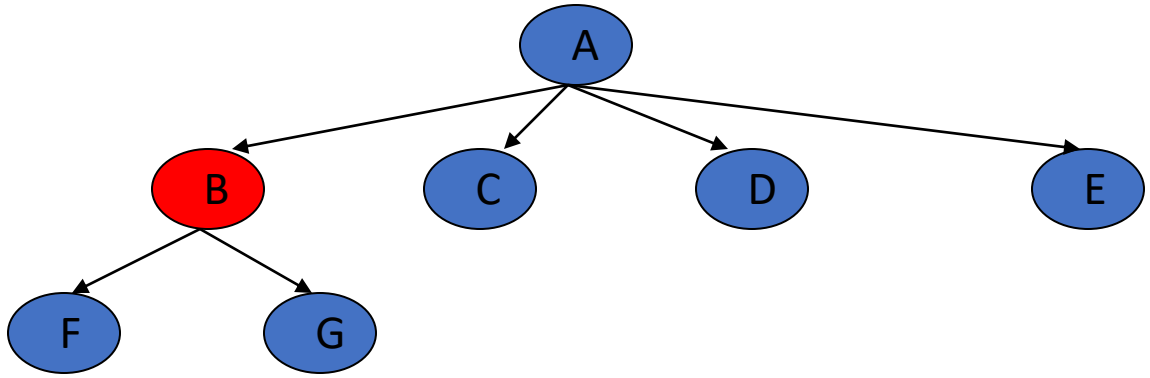
- A,



Limit = 2

Depth-Limited Search (DLS)

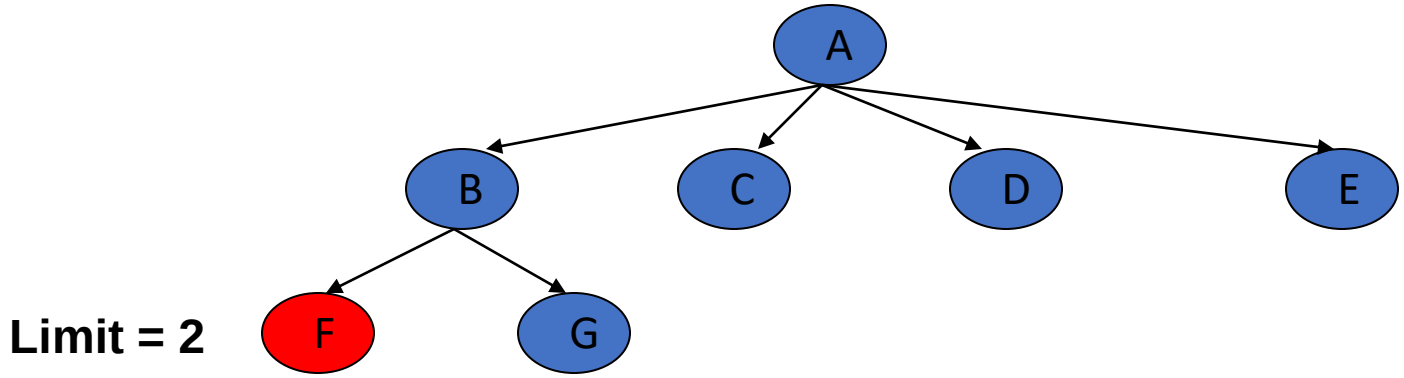
- A,B,



Limit = 2

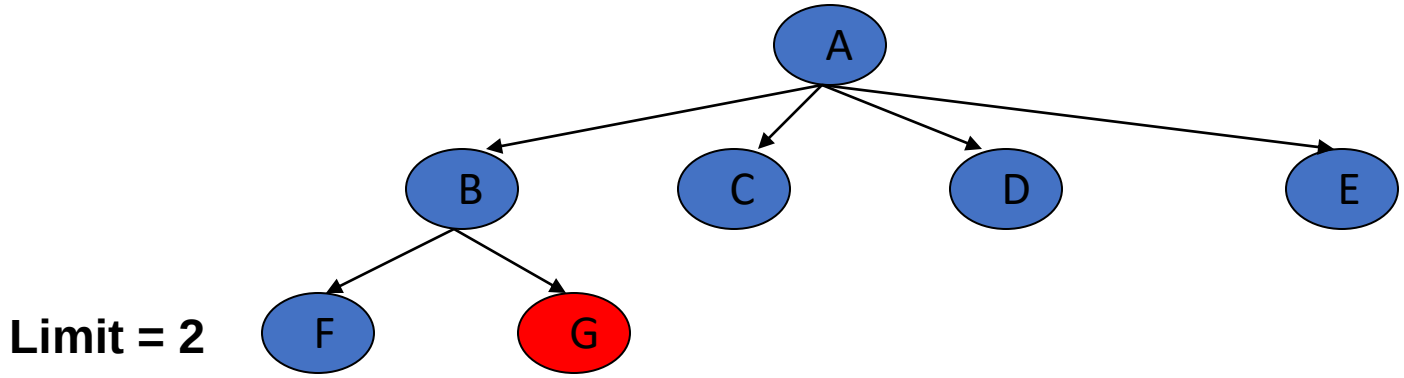
Depth-Limited Search (DLS)

- A,B,F,



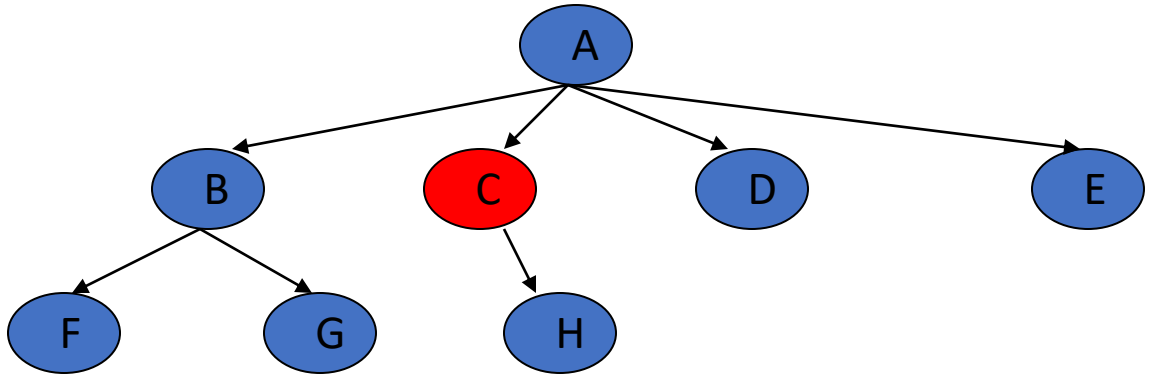
Depth-Limited Search (DLS)

- A,B,F,
- G,



Depth-Limited Search (DLS)

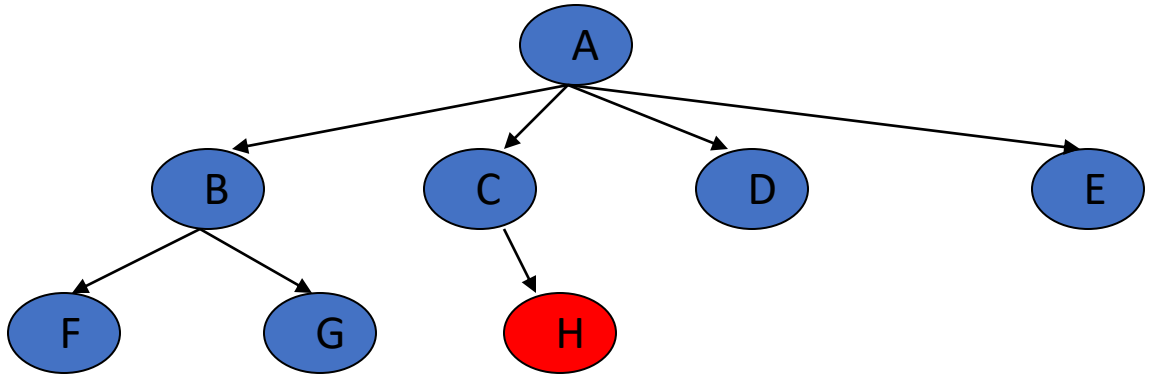
- A,B,F,
- G,
- C,



Limit = 2

Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,

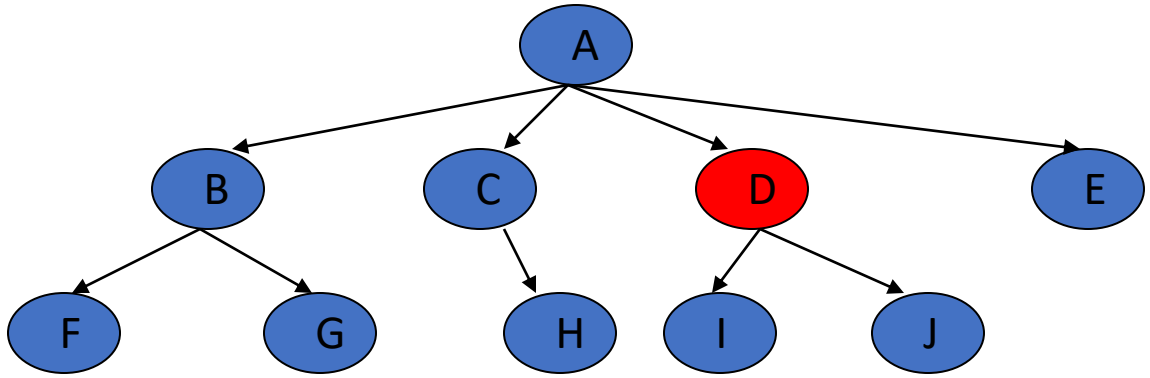


Limit = 2

Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,
- D,

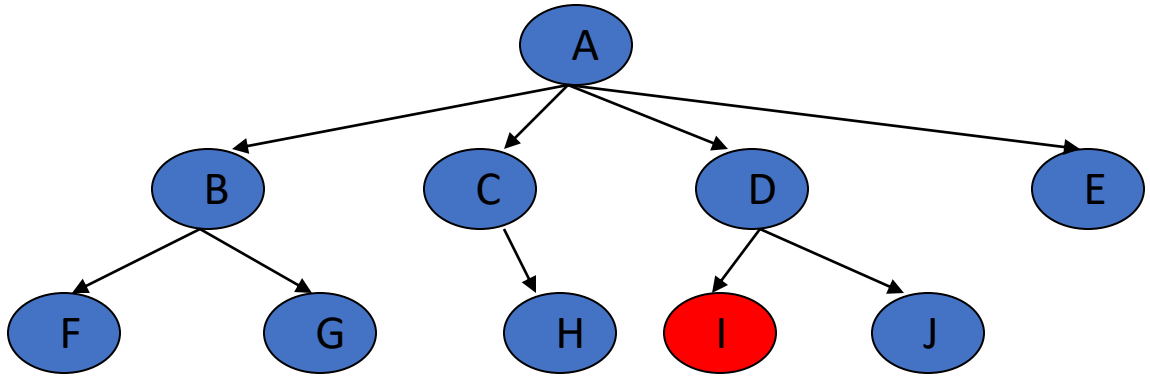
Limit = 2



Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,
- D,I

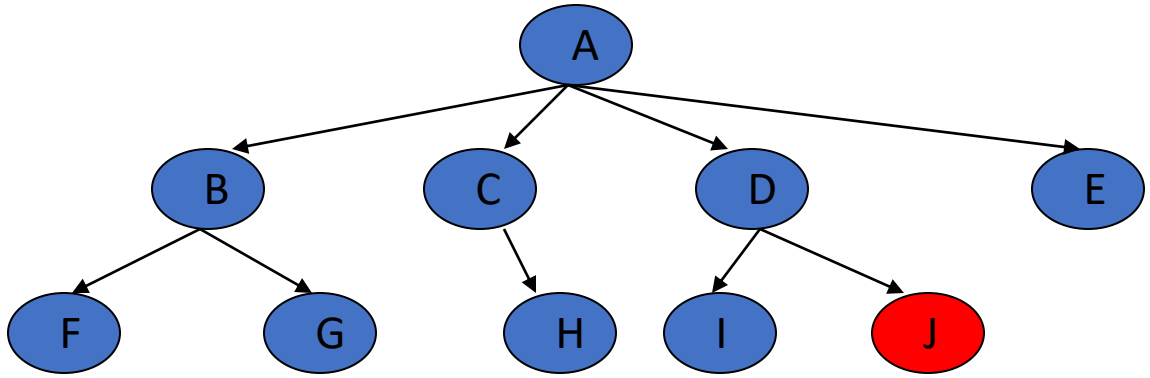
Limit = 2



Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,
- D,I
- J,

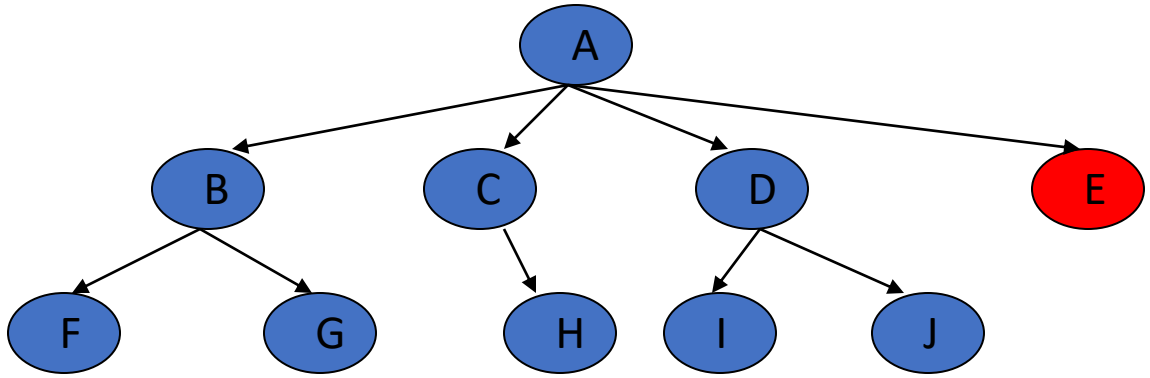
Limit = 2



Depth-Limited Search (DLS)

- A,B,F,
- G,
- C,H,
- D,I
- J,
- E

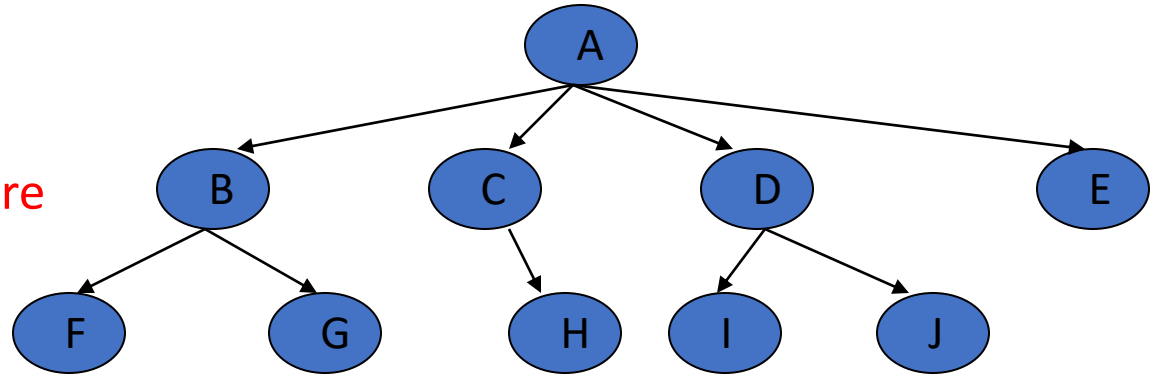
Limit = 2



Depth-Limited Search (DLS)

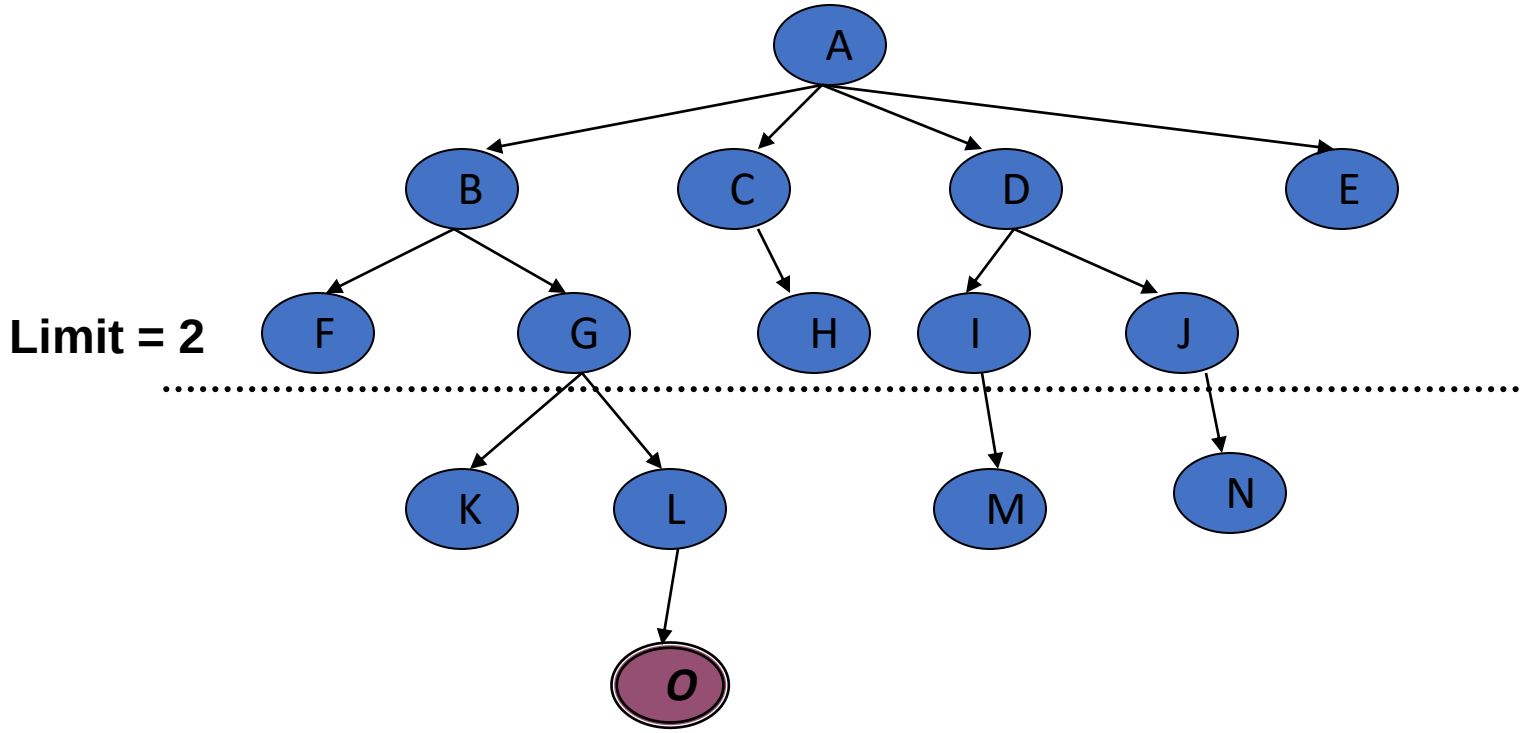
- A,B,F,
- G,
- C,H,
- D,I
- J,
- E, Failure

Limit = 2



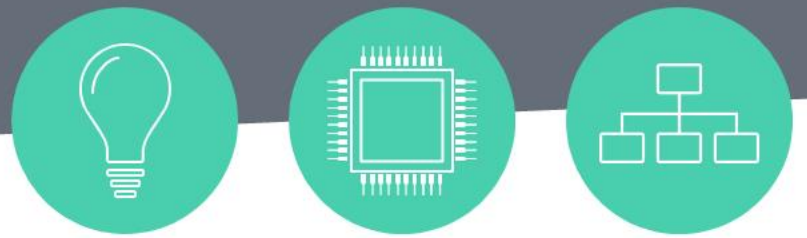
Depth-Limited Search (DLS)

- DLS algorithm returns Failure (no solution)
- The reason is that the goal is beyond the limit (Limit =2): the goal depth is (d=4)



Artificial Intelligence

By
Dr. Manoj Kumar



University School of Automation and Robotics
GGSIP University, East Campus, Delhi, India

Uninformed and Informed Search

- The blind search consists of depth-first and breadth-first
- These two techniques search 'any' or 'a' solution by using brute force
- The number of nodes expanded before reaching a solution may become prohibitively large to nontrivial problems
- Because the order of expanding the nodes is purely arbitrary and does not use any properties of the problem being solved, too many nodes may be expanded
- In such cases we may run out of time or space, or both even in simple problem (causing combinatorial explosion)

Uninformed and Informed Search

- **The techniques for doing so usually require additional information about the properties into the state and operator definitions Of the specific domain which is built**
- **Information about the problem domain can often be invoked to reduce the search.**
- **The technique for doing so usually require additional information about the properties of the specific domain which is built into the state and operator definition.**
- **Information of this sort is called heuristic information and the search methods using this information are called heuristic search or informed search.**

Informed/Heuristic Search

how an informed search strategy—one that uses domain-specific hints about the location of goals—can find solutions more efficiently than an uninformed strategy. The hints come in the form of a heuristic function, denoted h(n):

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

For example, in route-finding problems, we can estimate the distance from the current state to a goal by computing the straight-line distance on the map between the two points.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.

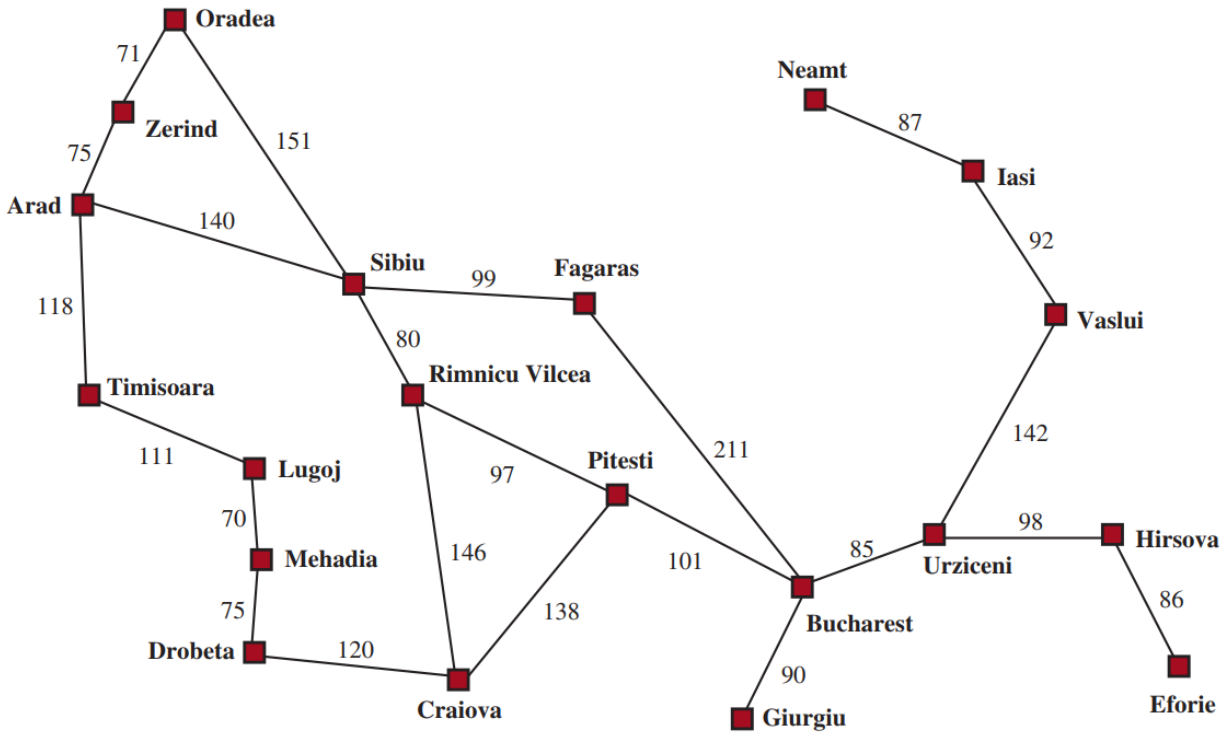


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Heuristic Search

- So far, we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space.

- But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. **It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.** The heuristic method, however, might not always give the best solution, but **it guaranteed to find a good solution in reasonable time**. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. **The value of the heuristic function is always positive.**

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**

Best First Search (Greedy 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
- Best-first search allows us to take the advantages of both algorithms
- 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.

Steps of Best First Search

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.

Advantages and Disadvantages

Advantages:

- **Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.**
- **This algorithm is more efficient than BFS and DFS algorithms.**

Disadvantages:

- **It can behave as an unguided depth-first search in the worst case scenario.**
- **It can get stuck in a loop as DFS.**
- **This algorithm is not optimal.**

– **OPEN** — nodes that have been generated and have had the heuristic function applied to them, but which have not yet been examined (i.e., had their successors generated).

– **CLOSED** — nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been generated before.

Best First Search Example 1

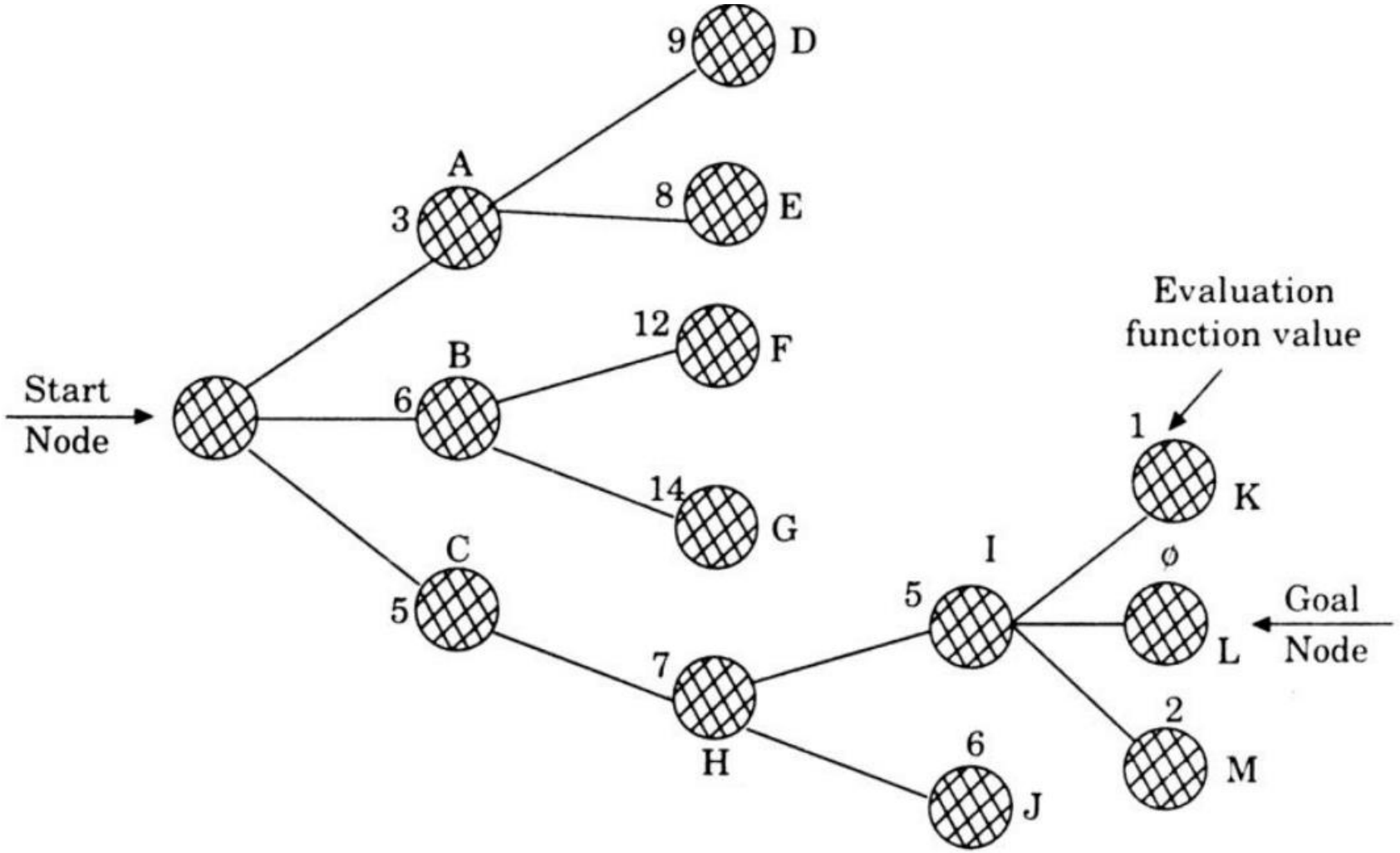


Fig. (12). Search process for best-first search.

Best First Search Example1

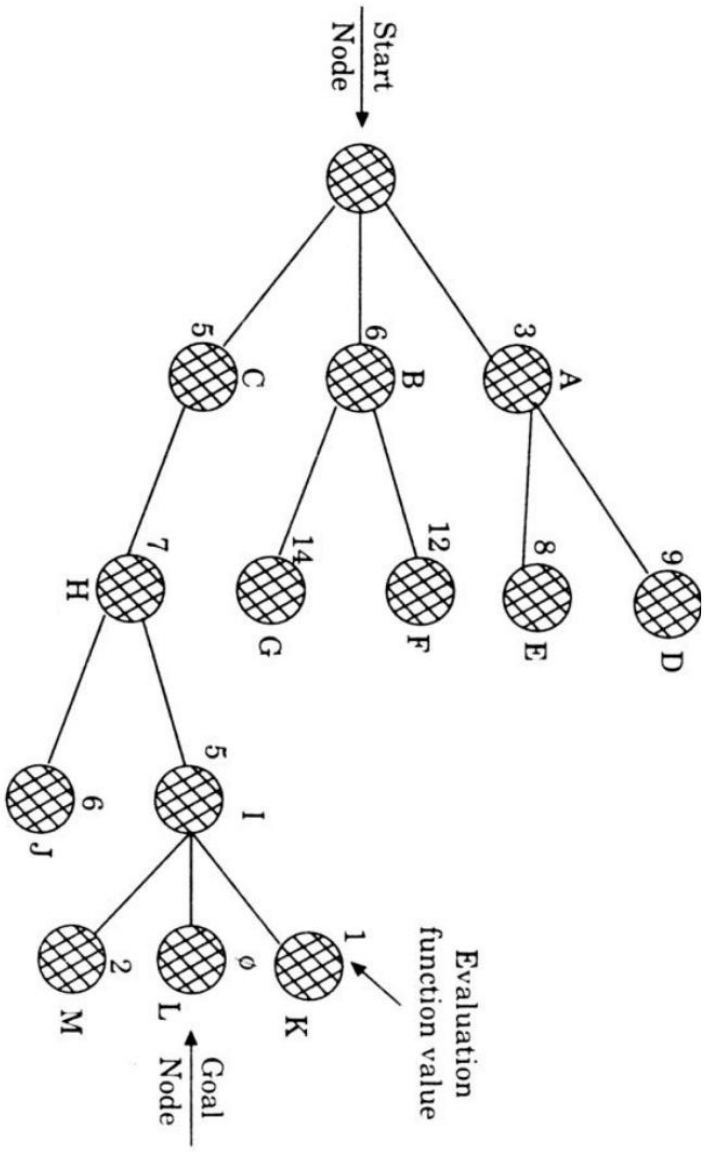


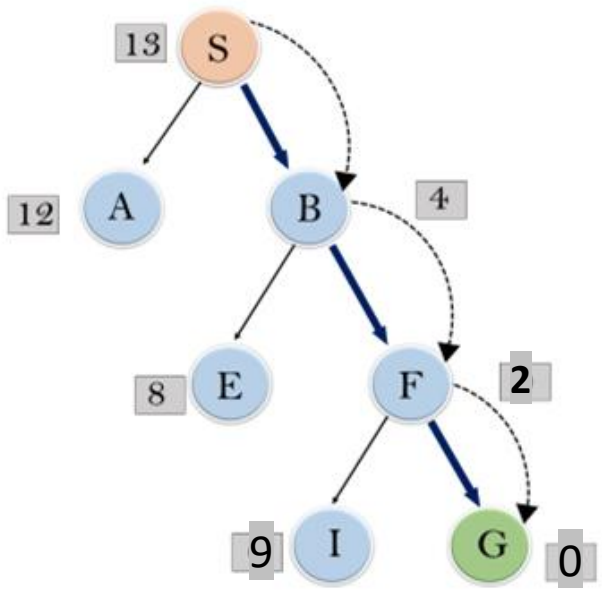
Table 3.2 Search process of best-first search

Step #	Node being expanded	Children	Available nodes	Node chosen
1	S	(A : 3), (B : 6), (C : 5)	(A : 3), (B : 6), (C : 5)	(A : 3)
2	A	(D : 9), (E : 8)	(B : 6), (C : 5), (D : 9), (E : 8)	(C : 5)
3	C	(H : 7)	(B : 6), (D : 9), (E : 8), (H : 7)	(B : 6)
4	B	(F : 12), (G : 14)	(D : 9), (E : 8), (H : 7), (F : 12), (G : 14)	(H : 7)
5	H	(I : 5), (J : 6)	(D : 9), (E : 8), (F : 12), (G : 14), (I : 5), (J : 6)	(I : 5)
6	I	(K : 1), (L : 0), (M : 2)	(D : 9), (E : 8), (H : 7), (F : 12), (G : 14), (J : 6), (K : 1), (L : 0), (M : 2)	Search stops as goal is reached

Best First Search Example2

Example:

Consider the below search problem, and we will traverse it using 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

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]

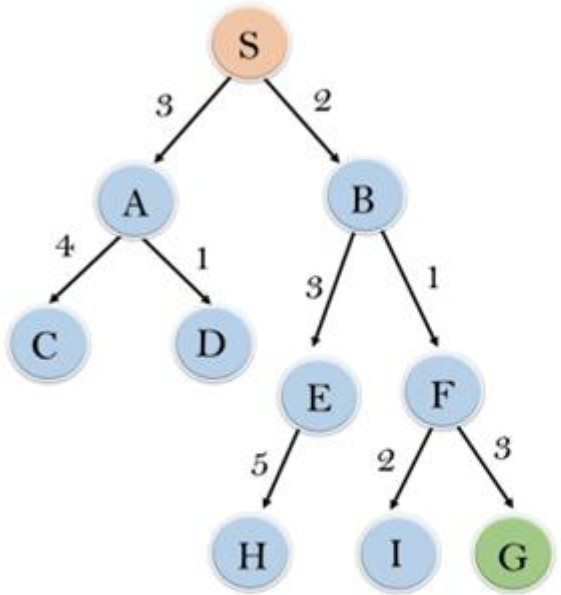
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Best First Search Example2

Example:

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.

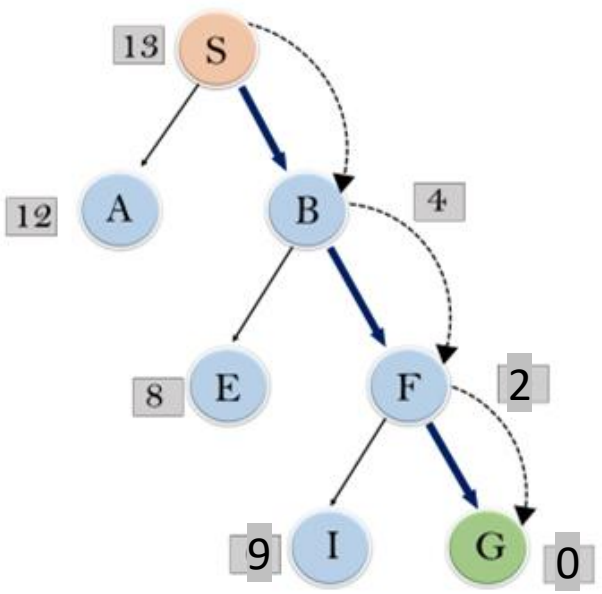


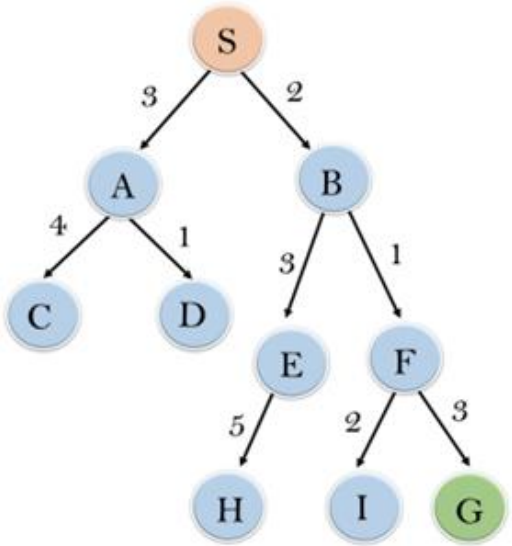
node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are **OPEN** and **CLOSED** Lists. Following are the iteration for traversing the above example.

Least heuristic value state will be in close list.

Best First Search Example2





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

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]

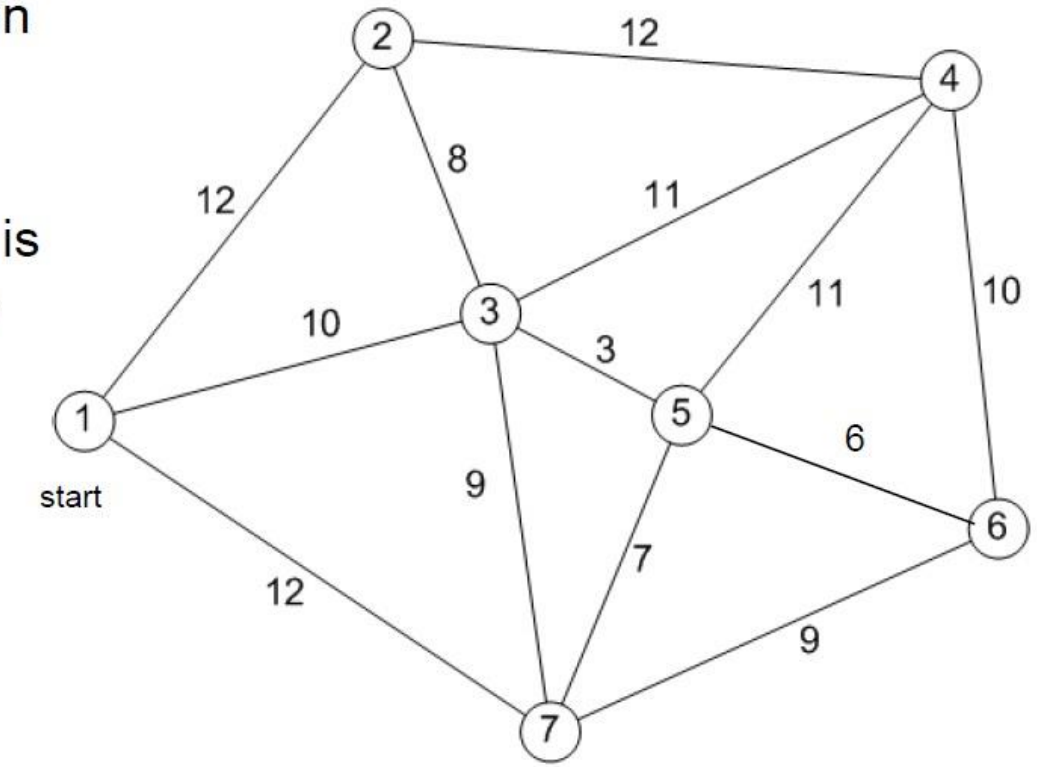
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Best First Search Example 3

The Traveling Salesman Problem

- Starting from city 1, the salesman must travel to all cities once before returning home
- The distance between each city is given, and is assumed to be the same in both directions
- Only the links shown are to be used
- Objective - Minimize the total distance to be travelled

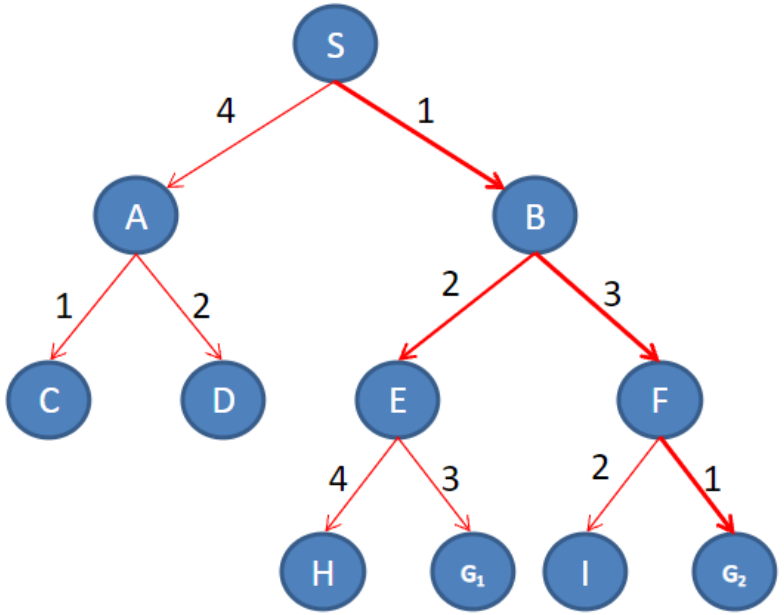


Best First Search Example

EXAMPLE on Best-First Search

open=[S_0]; closed=[ ]
open=[B_1 , A_4]; closed=[S_0]
open=[E_3 , A_4, F_4]; closed=[S_0 , B_1]
open=[A_4 , F_4 , $G1_6$, , H_7]; closed=[S_0 , B_1 , E_3]
open=[F_4 , C_5 , $G1_6$, D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4]
open=[C_5 , $G2_5$, $G1_6$, I_6 , D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4 , F_4]
open=[$G2_5$, $G1_6$, I_6 , D_6 , H_7]; closed=[S_0 , B_1 , E_3 , A_4 , F_4 , C_5]

NOTE: similar to Hill climbing but
WITH revising or backtracking
(keeping track of visited nodes).



SEARCH PATH = [S_0 , B_1 , E_3 , A_4 , F_4 , C_5 , $G2_5$]

Cost =5=1+3+1

Best First Search –Greedy Search

Greedy best-first search is a form of best-first search that expands first the node with the lowest h(n) value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So, the evaluation function f (n) = h(n).

GREEDY BEST FIRST SEARCH

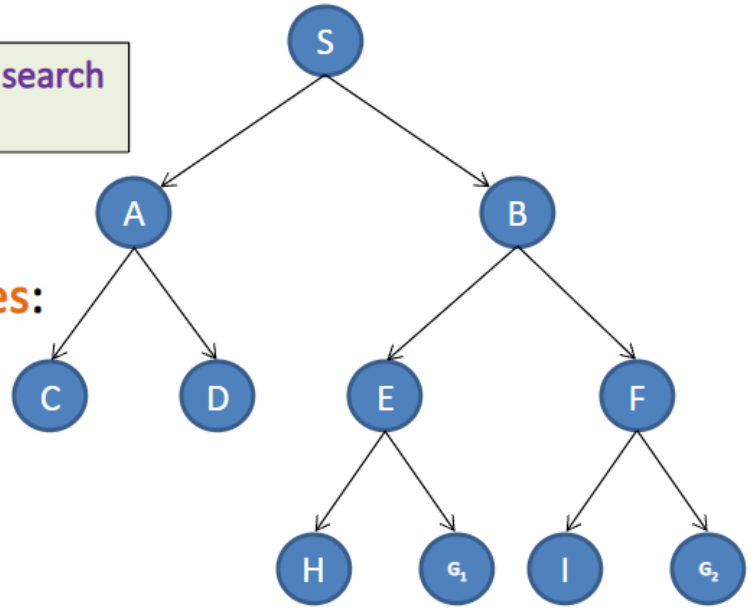
- Greedy best-first search is a best first search but uses heuristic estimate h(n) rather than cost function.
- Both follow the same process but Greedy uses heuristic function whereas best first uses cost function.
- EXAMPLE:

NOTE: similar to Best first search but uses heuristic values.

S: Initial state, G_1,G_2: goal.

Table shows the heuristic estimates:

node	h(n)	node	h(n)	node	h(n)
A	11	D	8	H	7
B	5	E	4	I	3
C	9	F	2		



Best First Search –Greedy Search

GREEDY BEST FIRST SEARCH

edy best-first search is a best first search but uses
ristic estimate h(n) rather than cost function.

n follow the same process but Greedy uses
ristic function whereas best first uses cost function.

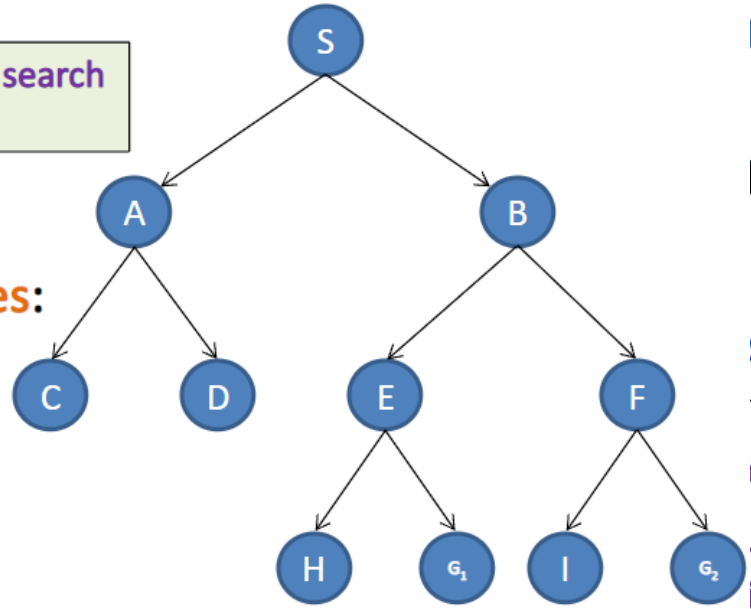
MPLE:

NOTE: similar to Best first search but uses heuristic values.

l state, G_1,G_2: goal.

nows the heuristic estimates:

(n)	node	h(n)	node	h(n)
	D	8	H	7
	E	4	I	3
	F	2		



EDY BEST FIRST SEARCH (cont.)

node	h(n)	node	h(n)
D	8	H	7
E	4	I	3
F	2		

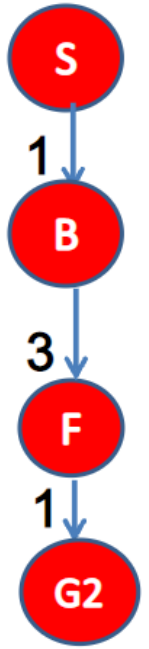
=[ ]
losed=[S]
]; closed=[S , B]
, A_{11}]; closed=[S , B , F]
]; closed=[S , B , F , G2]

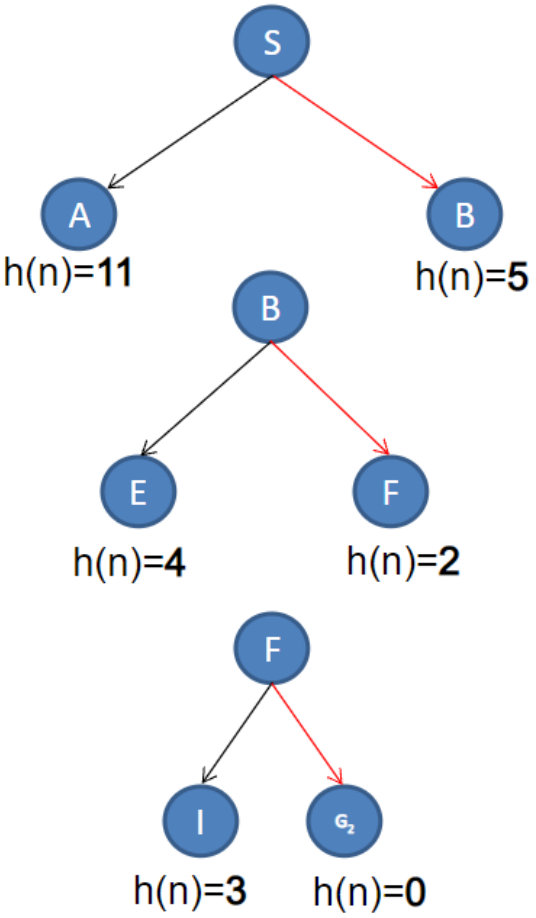
1=5

lution

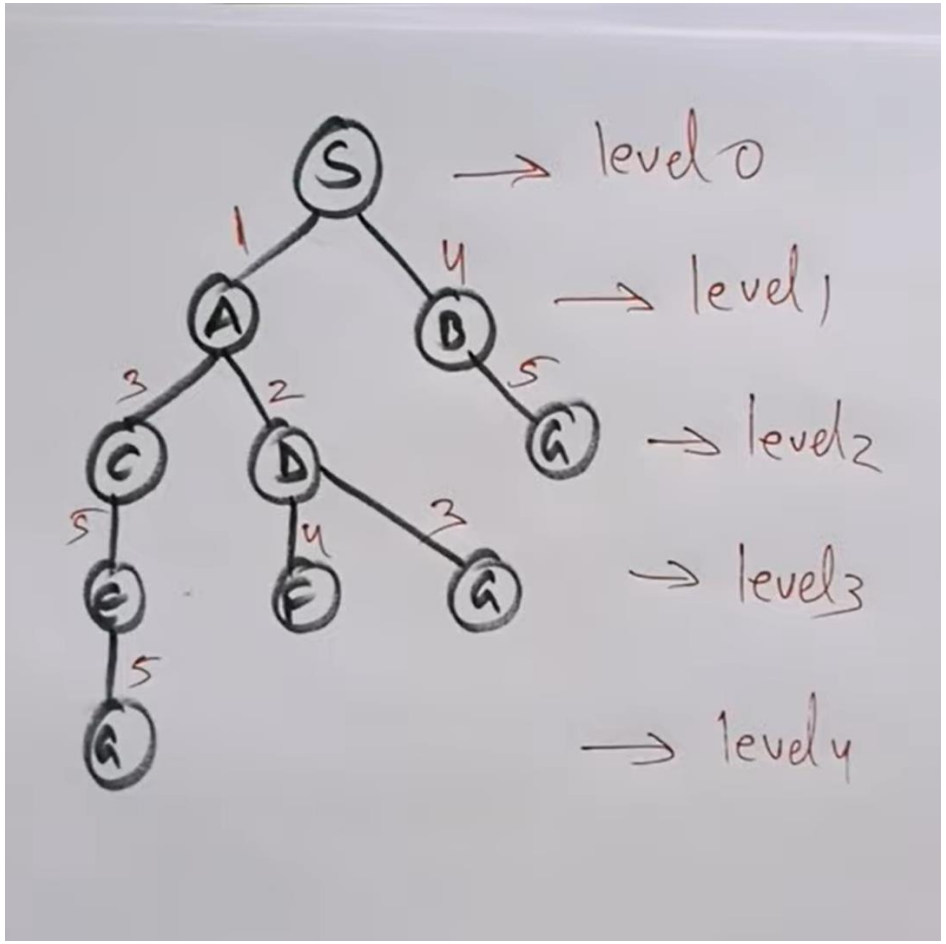
nteed the
ion

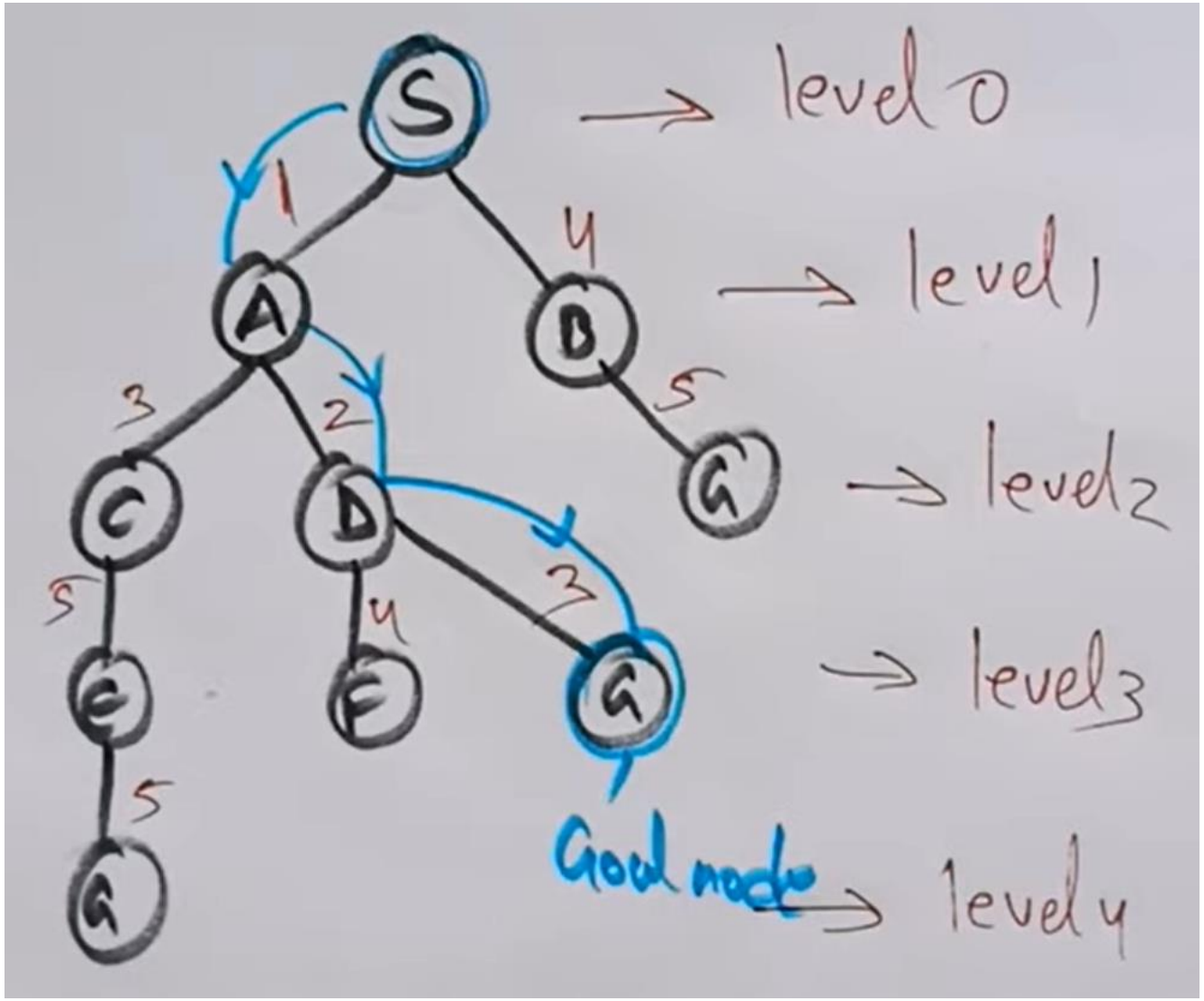
Search Path





Uniform cost search

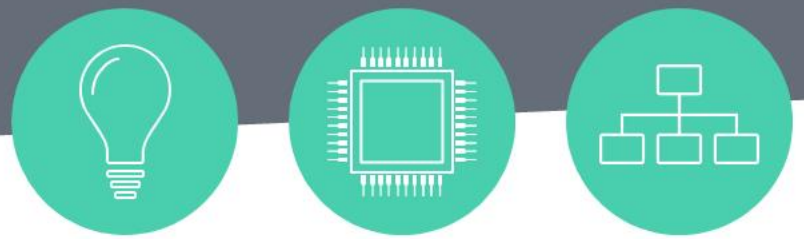




Artificial Intelligence

By

Dr. Manoj Kumar



University School of Automation and Robotics

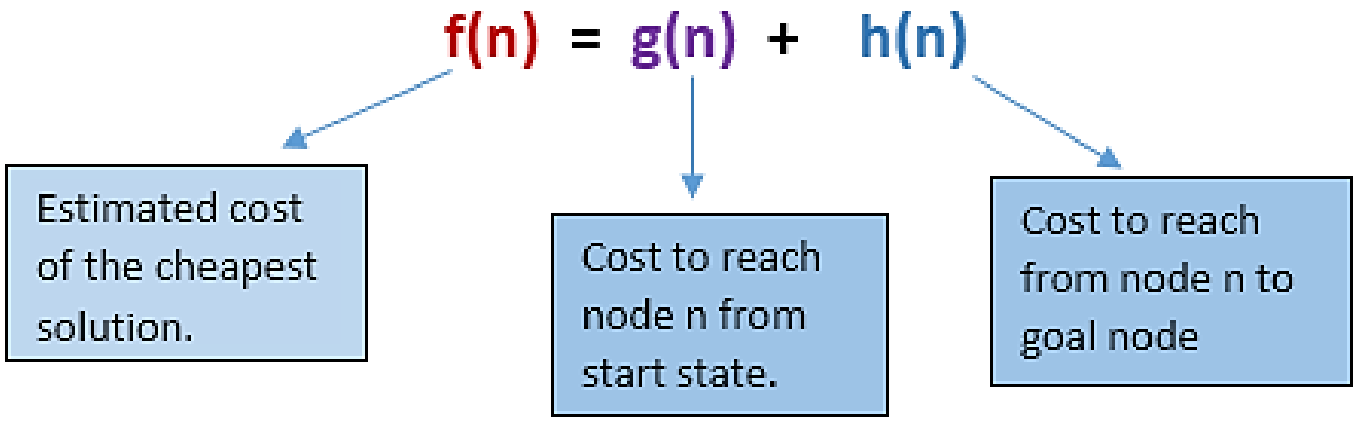
GGSIP University, East Campus, Delhi, India

A* 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).
- It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function.
- This search algorithm expands less search tree and provides optimal result faster.

A* Search

- A* algorithm uses g(n)+h(n) instead of g(n)



- where g(n) is the path cost from the initial state to node n, and h(n) is the estimated cost of the shortest path from n to a goal state, so we have
- f (n) = estimated cost of the best path that continues from n to a goal.
- In A* search algorithm, we use search heuristic as well as the cost to reach the node.
- Hence, we can combine both costs as following, and this sum is called as a fitness number.

A* Search Algorithm

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.

Advantages & Disadvantages

Advantages:

- **A* search algorithm is the best algorithm than other search algorithms.**
- **A* search algorithm is optimal and complete.**
- **This algorithm can solve very complex problems.**

Disadvantages:

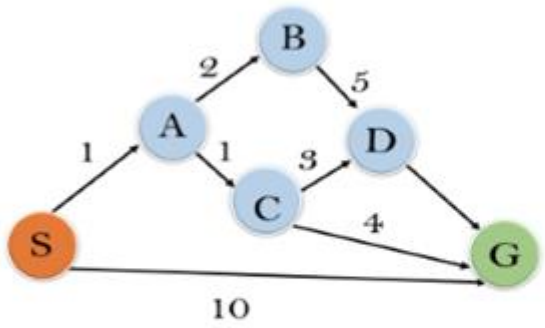
- **It does not always produce the shortest path as it mostly based on heuristics and approximation.**
- **A* search algorithm has some complexity issues.**
- **The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.**

Heuristic Search

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.

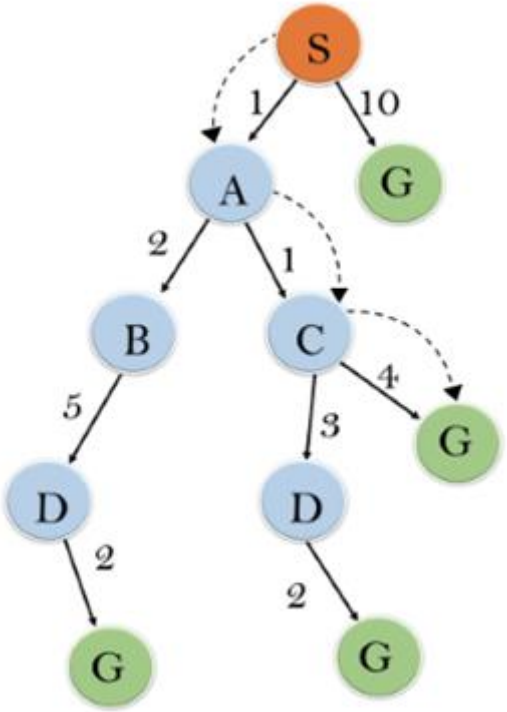
Here we will use OPEN and CLOSED list.



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

Pure Heuristic Search

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



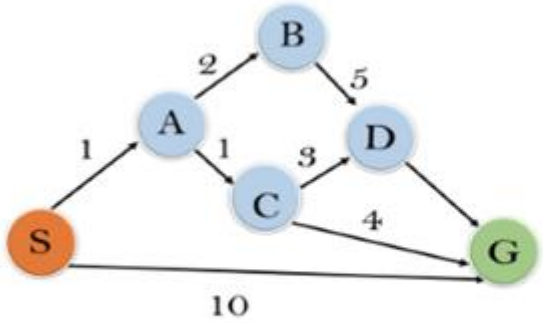
Initialization: {(S, 5)}

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.



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

Points to remember:

•A* algorithm returns the path which occurred first, and it does not search for all remaining paths.

•The efficiency of A* algorithm depends on the quality of heuristic.

•A* algorithm expands all nodes which satisfy the condition f(n)

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:

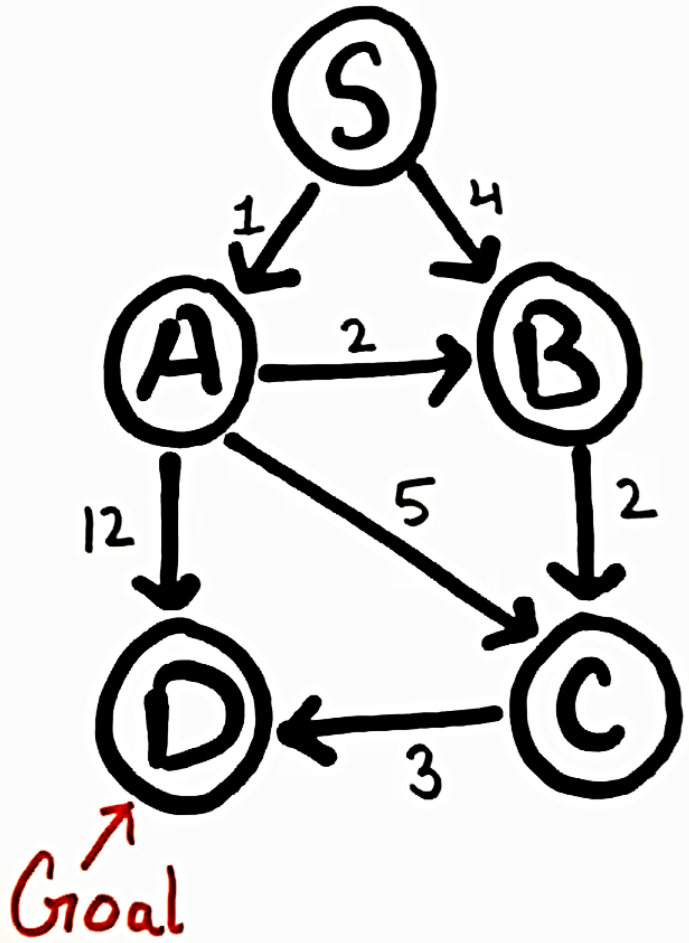
•**Admissible:** the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

•**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.

Space Complexity: The space complexity of A* search algorithm is **O(b^d)**



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

for S,

0+7=7

S→A,

1+6=7

S→B,

4+2=6

S→B→C

6+1=7

S→A→B

3+2=5

S→A→C

6+1=7

S→A→D

13+0=13

(S→A→B)→C

5+1=6

(S→A→B→C→D)

8+0=8

(S→B→C)→D

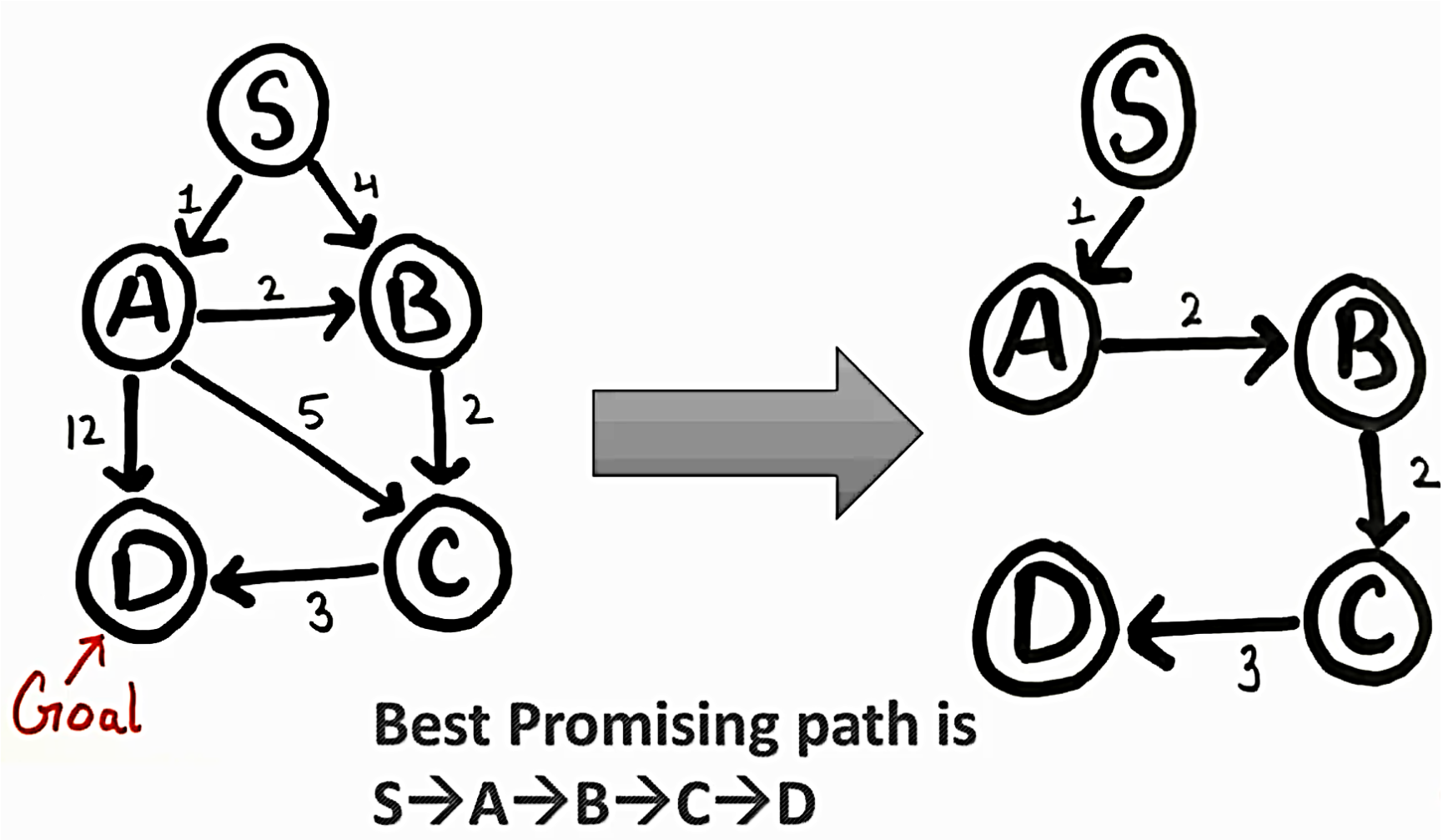
9+0=9

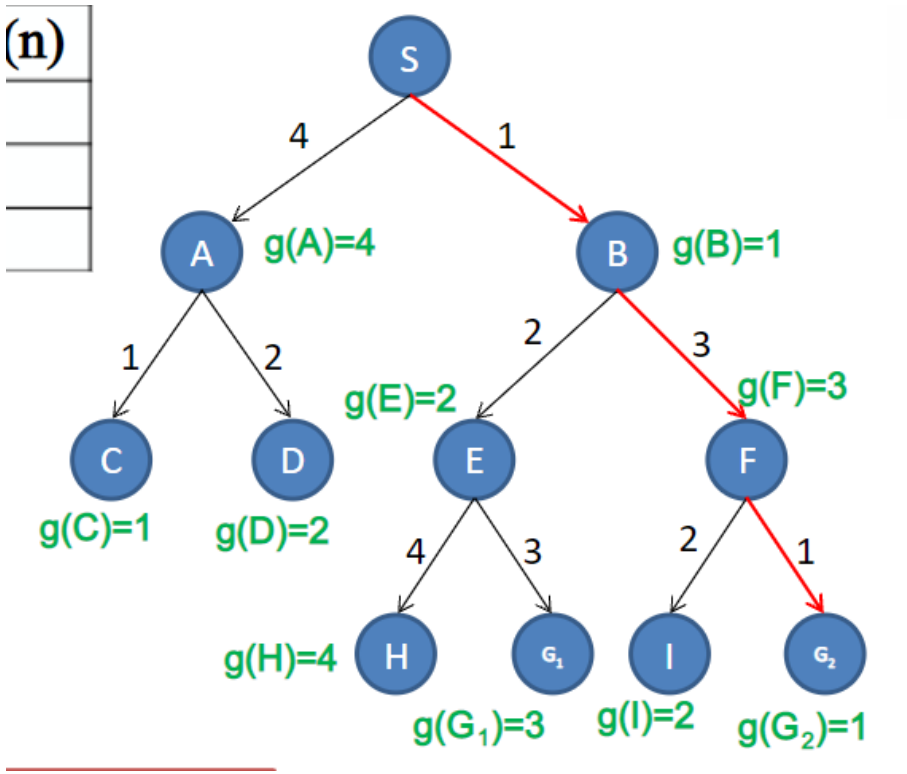
(S→A→C)→D

9+0=9

Heuristic Value	
S	7
A	6
B	2
C	1
D	0

S→A=7 ✓
S→B=6 ✓
S→B→C=7 ✓
S→A→B=5 ✓
S→A→C=7 ✓
S→A→D=13
S→A→B→C→=6 ✓
S→A→B→C→D=8
S→B→C→D=9
S→A→C→D=9

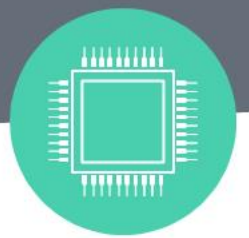

S
A
B
C
D
1
4
2
12
5
2
3
Goal
Best Promising path is
S→A→B→C→D


(n)
S
4
1
A
g(A)=4
B
g(B)=1
1
2
2
3
g(E)=2
g(F)=3
C
D
E
F
g(C)=1
g(D)=2
4
3
2
1
g(H)=4
H
G1
I
G2
g(G1)=3
g(I)=2
g(G2)=1

Artificial Intelligence

By

Dr. Manoj Kumar



University School of Automation and Robotics

GGSIP University, East Campus, Delhi, India

Problem Reduction

- we have considered search strategies for OR graphs through which we want to find a single, path to a goal.
- Such structures represent the fact that we will know how to get from a node to a goal state if we can discover how to get from that node to a goal state along any one of the branches leaving it.

- **Problem reduction is a problem-solving technique in artificial intelligence** (AI) that involves breaking down a complex problem into smaller, more manageable subproblems.

- **It is closely related to the use of And-Or graphs** in AI, which are graphical representations of a problem-solving process that help organize and visualize the decomposition of a problem into subproblems.

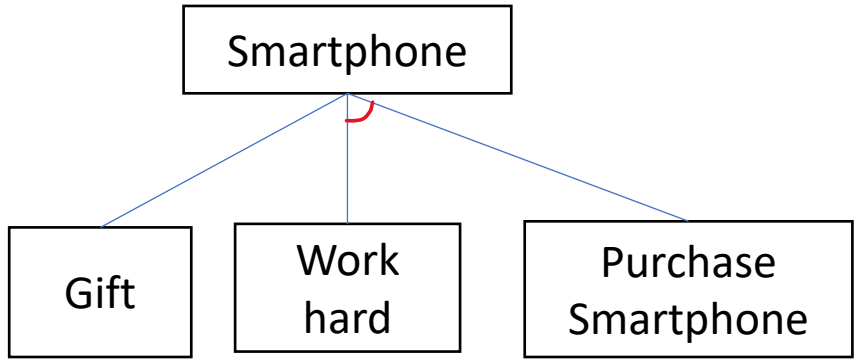
Problem Reduction: AND-OR Graph

And-Or graphs are a graphical representation used to model the decomposition of a problem into subproblems, where nodes represent states or subproblems, and edges represent the relationships or conditions between them.

- They consist of "AND" nodes and "OR" nodes, which serve different purposes: AND nodes represent situations where all child nodes must be satisfied or explored.
- OR nodes represent situations where at least one of the child nodes must be satisfied or explored.

Fig: AND / OR Graph

- This decomposition or reduction generates arcs that we call AND arcs.
- One AND arc may point to any numbers of successor nodes. All of which must then be solved in order for the arc to point solution.
- In order to find solution in an AND-OR graph we need an algorithm similar to best-first search but with the ability to handle the AND arcs appropriately.



Problem Reduction: Futility

We define FUTILITY, if the estimated cost of solution becomes greater than the value of FUTILITY then we abandon the search, FUTILITY should be chosen to correspond to a threshold.

Algorithm: AO*

1. Let *GRAPH* consist only of the node representing the initial state. (Call this node *INIT.)* Compute $h'(INIT)$
2. Until *INIT* is labeled *SOLVED* or until *INIT's* h' value becomes greater than *FUTILITY,* repeat the following procedure:
   (a) Trace the labeled arcs from *INIT* and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node *NODE.*
   (b) Generate the successors of *NODE.* If there are none, then assign *FUTILITY* as the h' value of *NODE.* This is equivalent to saying that *NODE* is not solvable. If there are successors, then for each one (called *SUCCESSOR)* that is not also an ancestor of *NODE* do the following:
      (i) Add *SUCCESSOR* to *GRAPH.*
      (ii) If *SUCCESSOR* is a terminal node, label it *SOLVED* and assign it an h' value of 0.
      (iii) If *SUCCESSOR* is not a terminal node, compute its h' value.
   (c) Propagate the newly discovered information up the graph by doing the following: Let *S* be a set of nodes that have been labeled *SOLVED* or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to *NODE.* Until *S* is empty, repeat the, following procedure:
      (i) If possible, select from *S* a node none of whose descendants in *GRAPH* occurs in *S.* If there is no such node, select any node from *S.* Call this node *CURRENT,* and remove it from *S.*
      (ii) Compute the cost of each of the arcs emerging from *CURRENT.* The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as *CURRENT'S* new h' value the minimum of the costs just computed for the arcs emerging from it.
      (iii) Mark the best path out of *CURRENT* by marking the arc that had the minimum cost as computed in the previous step.
      (iv) Mark *CURRENT SOLVED* if all of the nodes connected to it through the new labeled arc have been labeled *SOLVED.*
      (v) If *CURRENT* has been labeled *SOLVED* or if the cost of *CURRENT* was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of *CURRENT* to *S.*

Algorithm 1: Pseudocode of AO* Algorithm

```
Data: Graph, StartNode
Result: The minimum cost path from StartNode to GoalNode
CurrentNode ← StartNode
while There is a new path with lower cost from StartNode to the
 GoalNode do
    calculate the cost of path from the current node to the goal
     node through each of its successor nodes;
    if the successor node is connected to other successor nodes by
     AND-ARCS then
        sum up the cost of all paths in the AND-ARC;
        return the total cost;
    else
        calculate the cost of the single path in the OR side;
        return the single cost;
    end
    find the minimum cost path
    CurrentNode ← SuccessorNodeOfMinimumCostPath
    if CurrentNode has no successor node then
        do the backpropagation and correct the estimated costs;
        CurrentNode ← StartNode
        return CurrentNode, New estimated costs;
    else
        return null;
    end
    return The minimum cost path;
end
```

AO* Algorithm

- AO* algorithm is a heuristic search algorithm in AI.
- AO* algorithm uses the concept of AND-OR graphs to decompose any complex problem given into smaller set of problems which are further solved.

How AO* algorithm works

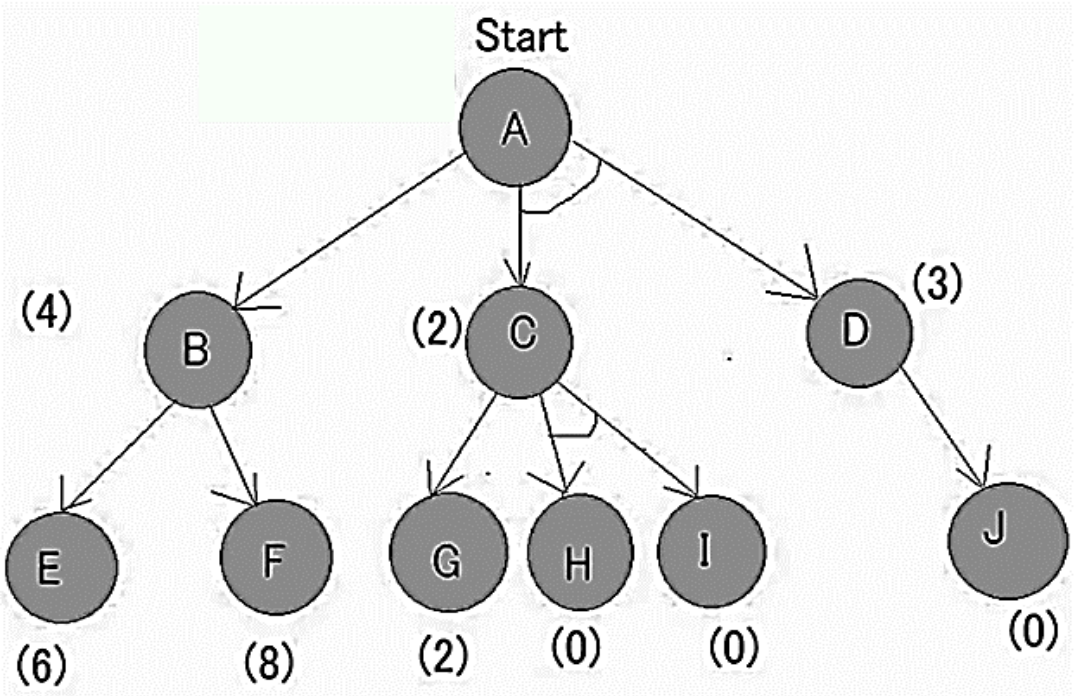
The AO* algorithm works on the formula given below : f (n) = g(n) + h(n)

Where

- g(n):The actual cost of traversal from initial state to the current state.
- h(n): The estimated cost of traversal from the current state to the goal state.
- f(n): The actual cost of traversal from the initial state to the goal state.

AND OR Graph

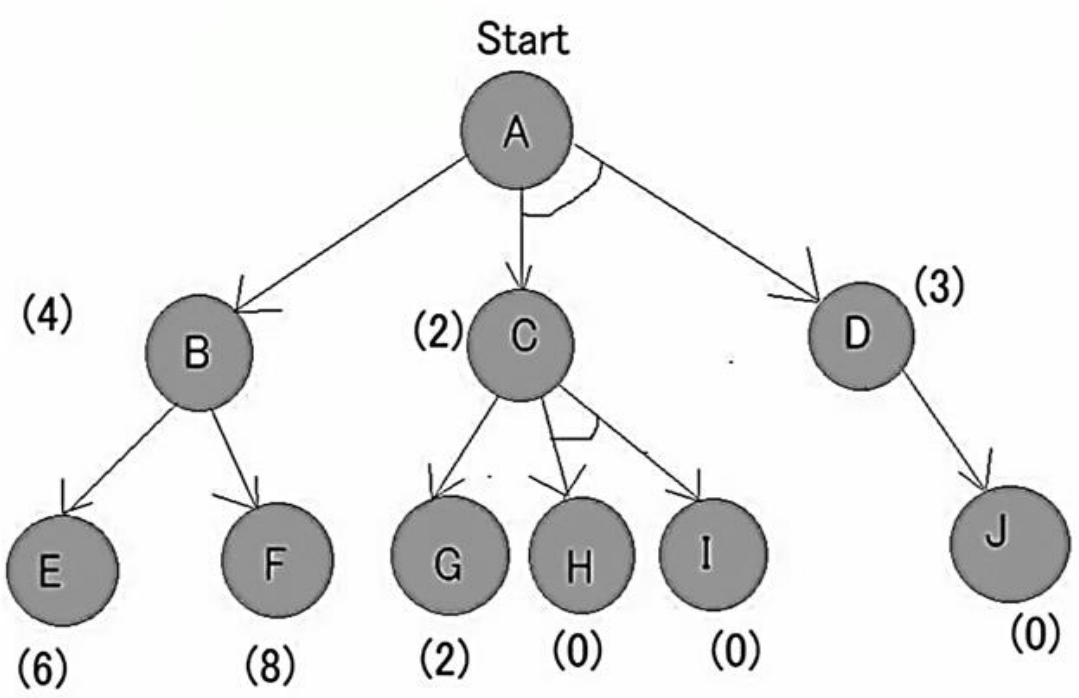
- Here, in the above example all numbers in brackets are the heuristic value i.e h(n).
- Each edge is considered to have a value of 1 by default.



Example 1

Step-1

- Starting from node A, we first calculate the best path.
- **f(A-B) = g(B) + h(B) = 1+4= 5** , where 1 is the default cost value of travelling from A to B and 4 is the estimated cost from B to Goal state.
- **f(A-C-D) = g(C) + h(C) + g(D) + h(D) = 1+2+1+3 = 7**, here we are calculating the path cost as both C and D because they have the AND-Arc.
- The default cost value of travelling from A-C is 1, and from A-D is 1, but the heuristic value given for C and D are 2 and 3 respectively hence making the cost as 7.



Example 1

Step-2

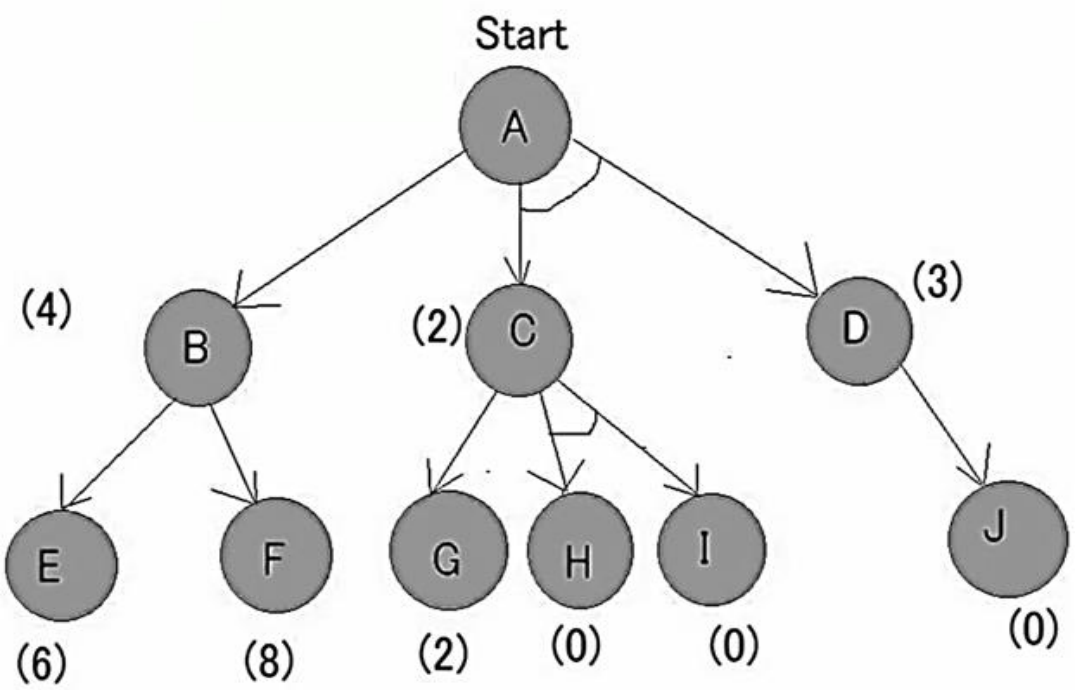
- From the B node,

  f(B-E) = 1 + 6 = 7

  f(B-F) = 1 + 8 = 9

- Hence, the B-E path has lesser cost. Now the heuristics have to be updated since there is a difference between actual and heuristic value of B.
- The minimum cost path is chosen and is updated as the heuristic , in our case the value is 7.
- And because of change in heuristic of B there is also change in heuristic of A which is to be calculated again.

  f(A-B) = g(B) + updated((h(B)) = 1+7=8



Example 1

Step-3

- Now the current node becomes C node and the cost of the path is calculated,

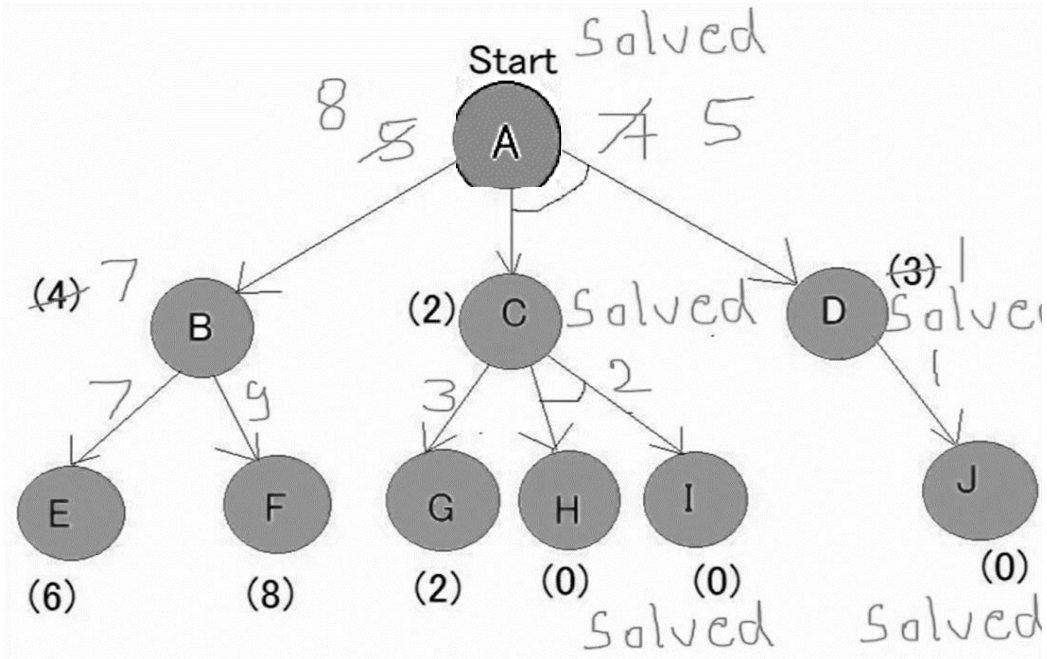
  f(C-G) = 1+2 = 3

  f(C-H-I) = 1+0+1+0 = 2

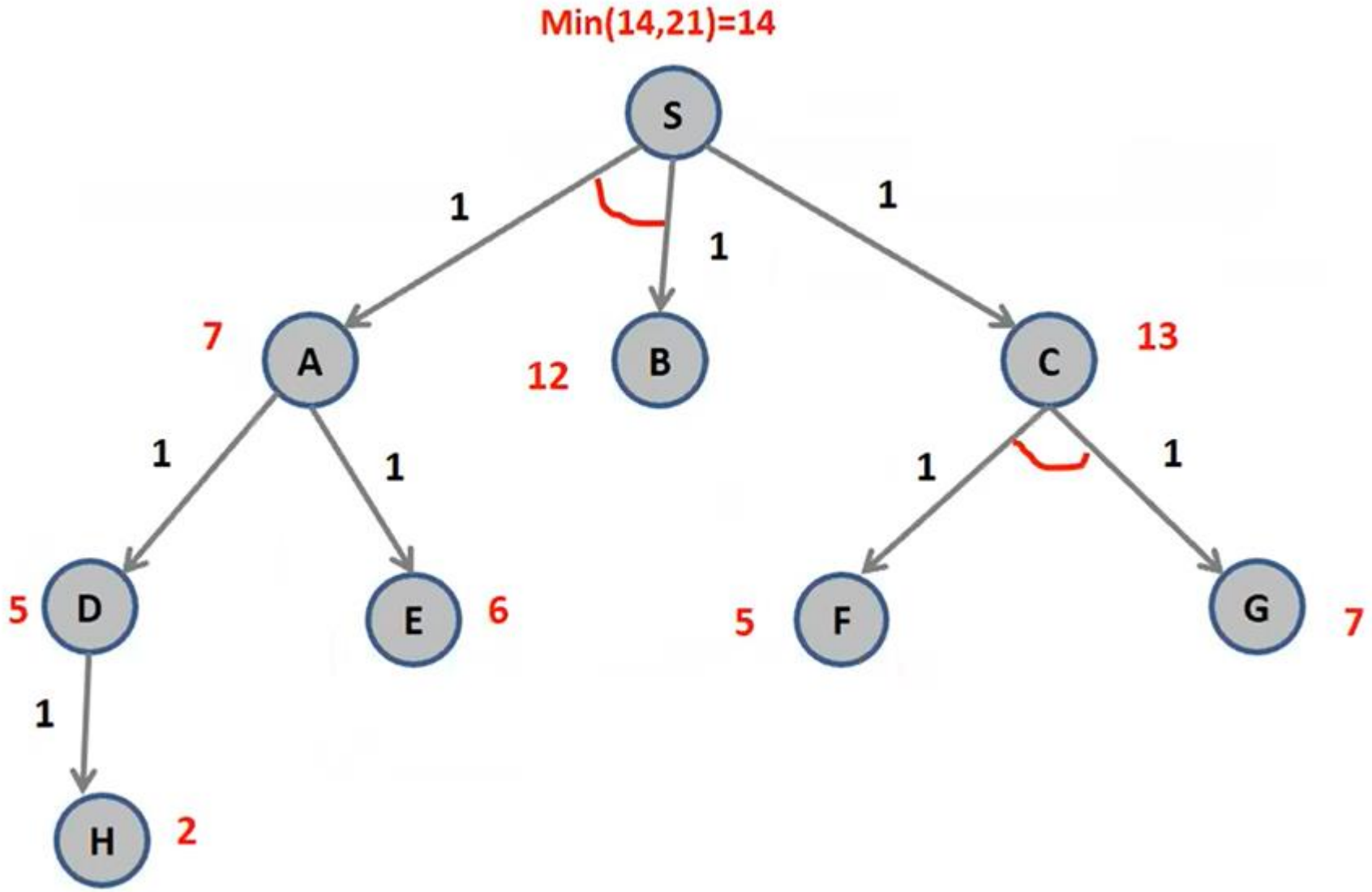
  f(C-H-I) is chosen as minimum cost path.
- Heuristic of path of H and I are 0 and hence they are solved,
- But Path A-D also needs to be calculated , since it has an AND-arc.
- **f(D-J) = 1+0 = 1**, hence heuristic of D needs to be updated to 1.
- And finally the f(A-C-D) needs to be updated.

  f(A-C-D) = g(C) + h(C) + g(D) + updated((h(D))

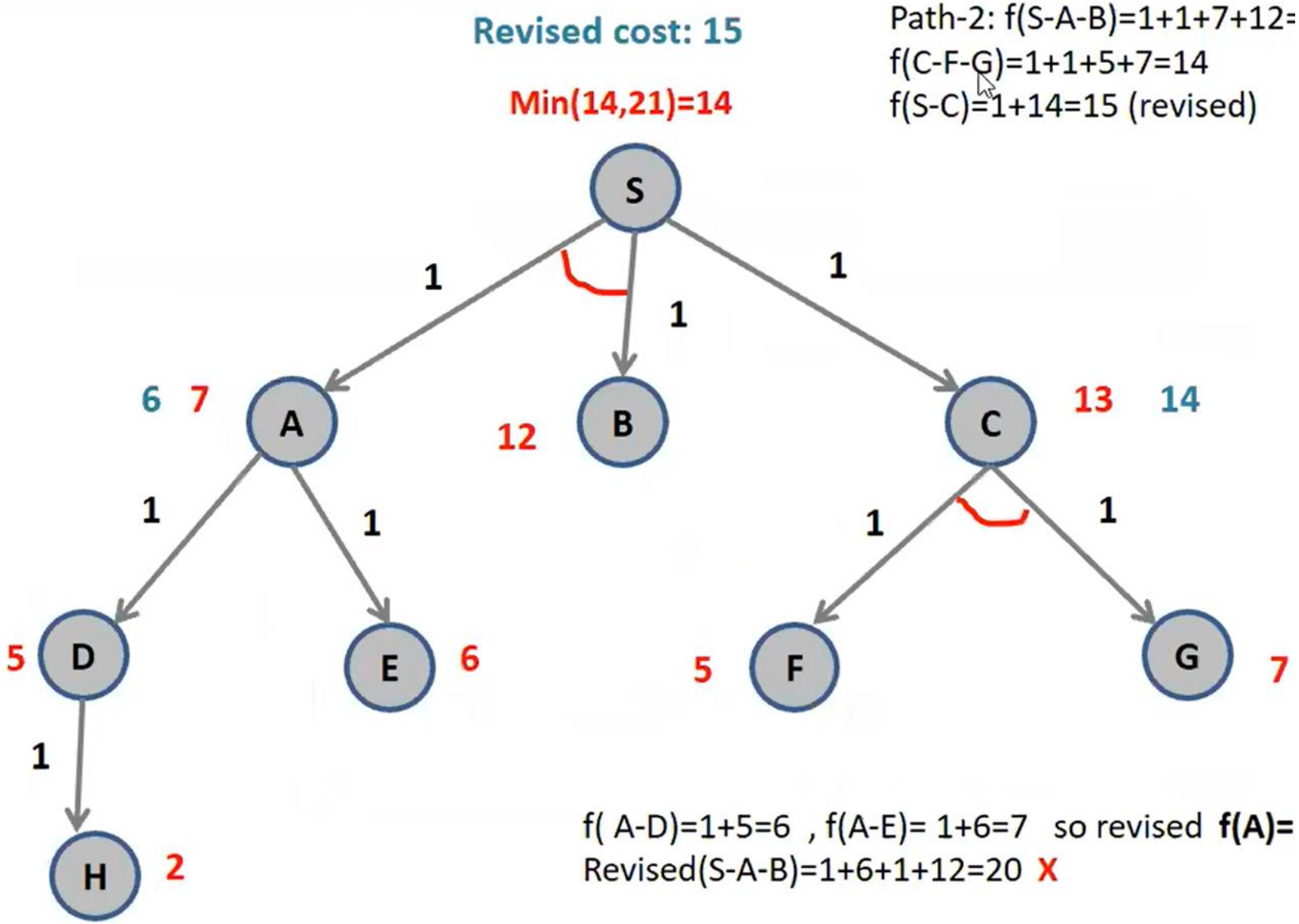
  = 1+2+1+1 =5.



Example 2



Example 2


Revised cost: 15
Min(14,21)=14
Path-2: f(S-A-B)=1+1+7+12=
f(C-F-G)=1+1+5+7=14
f(S-C)=1+14=15 (revised)
S
A
B
C
D
E
F
G
H
1
1
1
1
1
1
1
1
1
6
7
12
13
14
5
6
5
7
2
f(A-D)=1+5=6 , f(A-E)= 1+6=7 so revised f(A)=
Revised(S-A-B)=1+6+1+12=20 X

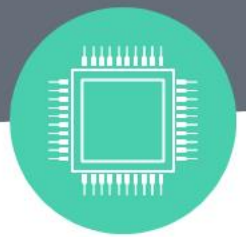
Artificial Intelligence

By

Dr. Manoj Kumar

University School of Automation and Robotics

GGSIP University, East Campus, Delhi, India



Game Search

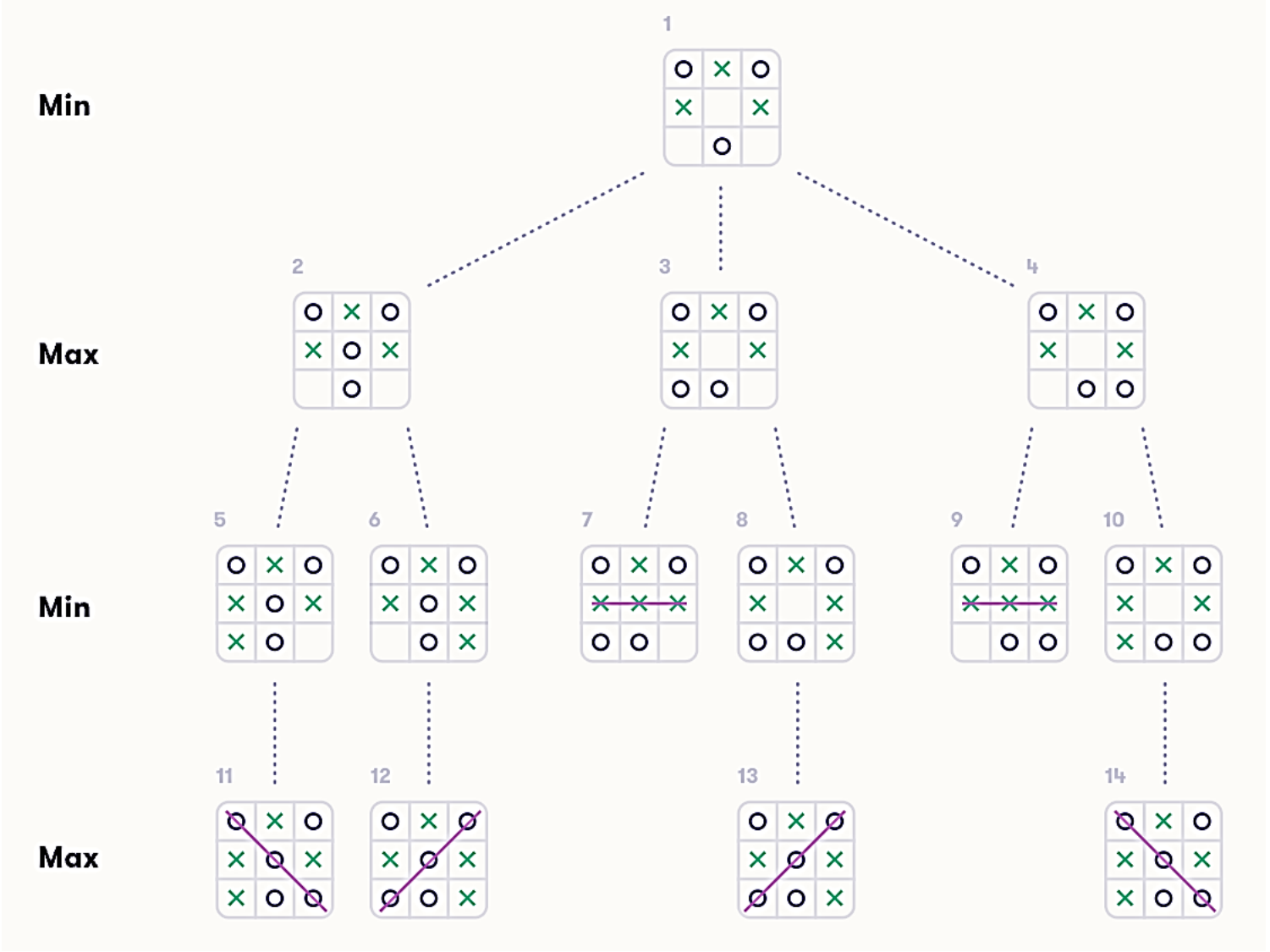
ADVERSARIAL SEARCH : in which two or more agents have conflicting goals, giving rise to adversarial search problems.

- **Rather than deal with the chaos of real-world clash, we will concentrate on games, such as chess, Go, and poker.**
- **For AI researchers, the simplified nature of these games is a plus: the state of a game is easy to represent, and agents are usually restricted to a small number of actions whose effects are defined by precise rules.**
- **Physical games, such as croquet and ice hockey, have more complicated descriptions, a larger range of possible actions, and rather imprecise rules defining the legality of actions.**
- **With the exception of robot soccer, these physical games have not attracted much interest in the AI community.**

Game Search

ADVERSARIAL SEARCH : in which two or more agents have conflicting goals, giving rise to adversarial search problems.

Two or more agents (players) compete against each other with opposing objectives.



Game Search

Two or more agents (players) compete against each other with opposing objectives.

5.1 Game Theory

There are at least three stances we can take towards multi-agent environments. The first stance, appropriate when there are a very large number of agents, is to consider them in the aggregate as an **economy**, allowing us to do things like predict that increasing demand will cause prices to rise, without having to predict the action of any individual agent.

Economy

Second, we could consider adversarial agents as just a part of the environment—a part that makes the environment nondeterministic. But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn't, we miss the idea that our adversaries are actively trying to defeat us, whereas the rain supposedly has no such intention.

The third stance is to explicitly model the adversarial agents with the techniques of adversarial game-tree search. That is what this chapter covers. We begin with a restricted class of games and define the optimal move and an algorithm for finding it: minimax search, a generalization of AND–OR search (from Figure 4.11). We show that **pruning** makes the search more efficient by ignoring portions of the search tree that make no difference to the optimal move. For nontrivial games, we will usually not have enough time to be sure of finding the optimal move (even with pruning); we will have to cut off the search at some point.

Pruning

For each state where we choose to stop searching, we ask who is winning. To answer this question we have a choice: we can apply a heuristic **evaluation function** to estimate who is winning based on features of the state (Section 5.3), or we can average the outcomes of many fast simulations of the game from that state all the way to the end (Section 5.4).

Section 5.5 discusses games that include an element of chance (through rolling dice or shuffling cards) and Section 5.6 covers games of **imperfect information** (such as poker and bridge, where not all cards are visible to all players).

Imperfect information

Game Search/Min-Max Algorithm

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory.
- It provides an optimal move for the player assuming that opponent is also playing optimally.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various two-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game; one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Games

Adversarial search is a key concept in artificial intelligence and game theory, where two or more agents (players) compete against each other with opposing objectives. This concept is often studied in the context of various games to develop strategies and algorithms for making optimal decisions. Here are some common games studied in the context of adversarial search, along with their importance:

Checkers (Draughts): Checkers is another classic board game with a smaller search space compared to chess. It serves as an excellent introductory game for teaching adversarial search algorithms due to its simpler rules and manageable complexity.

Chess: Chess is one of the most famous and complex board games that involve perfect information. It's often used to study adversarial search because of its vast branching factor and deep game tree. Chess has been extensively studied to develop sophisticated AI algorithms, including minimax with alpha-beta pruning and neural network-based approaches.

The chief difference between chess and checkers lies in the fact that in the latter, **the primary objective is to get rid of all your opponent's pieces. In chess, however, the target is to capture or 'check' the rival king**.

Games

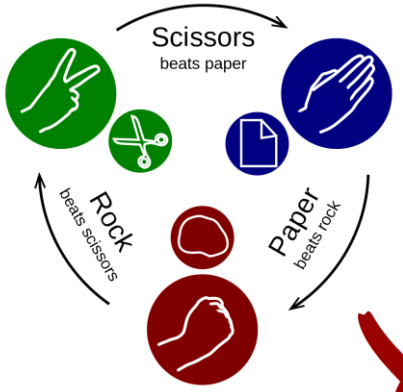
Go: Go is an ancient board game from East Asia that is known for its immense complexity. The number of possible game states in Go is significantly larger than in chess, making it a challenging domain for AI research. The development of AlphaGo by DeepMind was a groundbreaking achievement in adversarial search, showcasing the power of deep learning and Monte Carlo Tree Search (MCTS) in solving complex games.

Backgammon: Backgammon is a dice-based board game with elements of chance and strategy. It has been used as a testbed for reinforcement learning algorithms and is an interesting game to study due to the combination of stochastic elements and strategy.

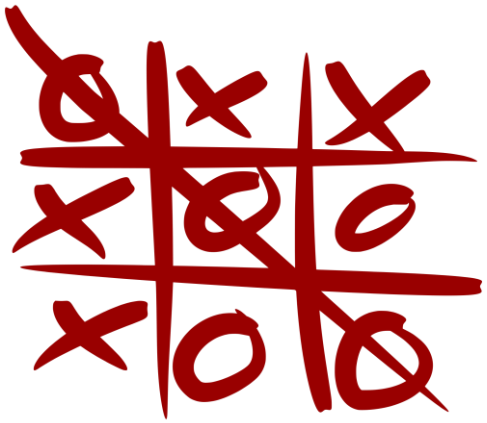
Games

Poker: Poker is a card game that involves incomplete information (hidden cards) and bluffing. It's an important game to study in adversarial search because it models scenarios where players have imperfect knowledge of the game state. AI research in poker has led to the development of algorithms for strategic reasoning under uncertainty.

Rock-Paper-Scissors: While simple, Rock-Paper-Scissors is often used to teach the basic concepts of adversarial search. It's an example of a zero-sum game with a small strategy space.



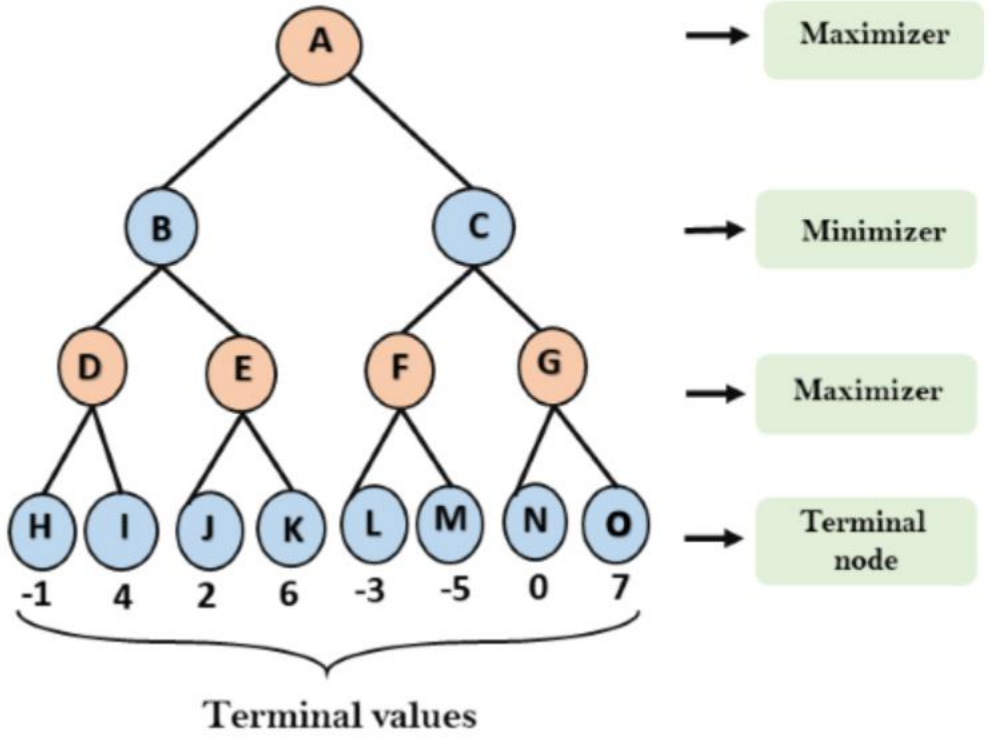
Tic-Tac-Toe: Tic-Tac-Toe is a straightforward game that is often used as an introductory example of adversarial search algorithms like minimax. It helps students understand the basics of game trees, terminal states, and evaluation functions.



Min-Max Algorithm

- **This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.**
- **Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.**
- **At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:**

Step-1: 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 the tree diagram, let's take A is the initial state of the tree. Suppose **maximizer** takes first turn which has **worst-case initial value =- infinity**, and **minimizer** will take next turn which has **worst-case initial value = +infinity.**

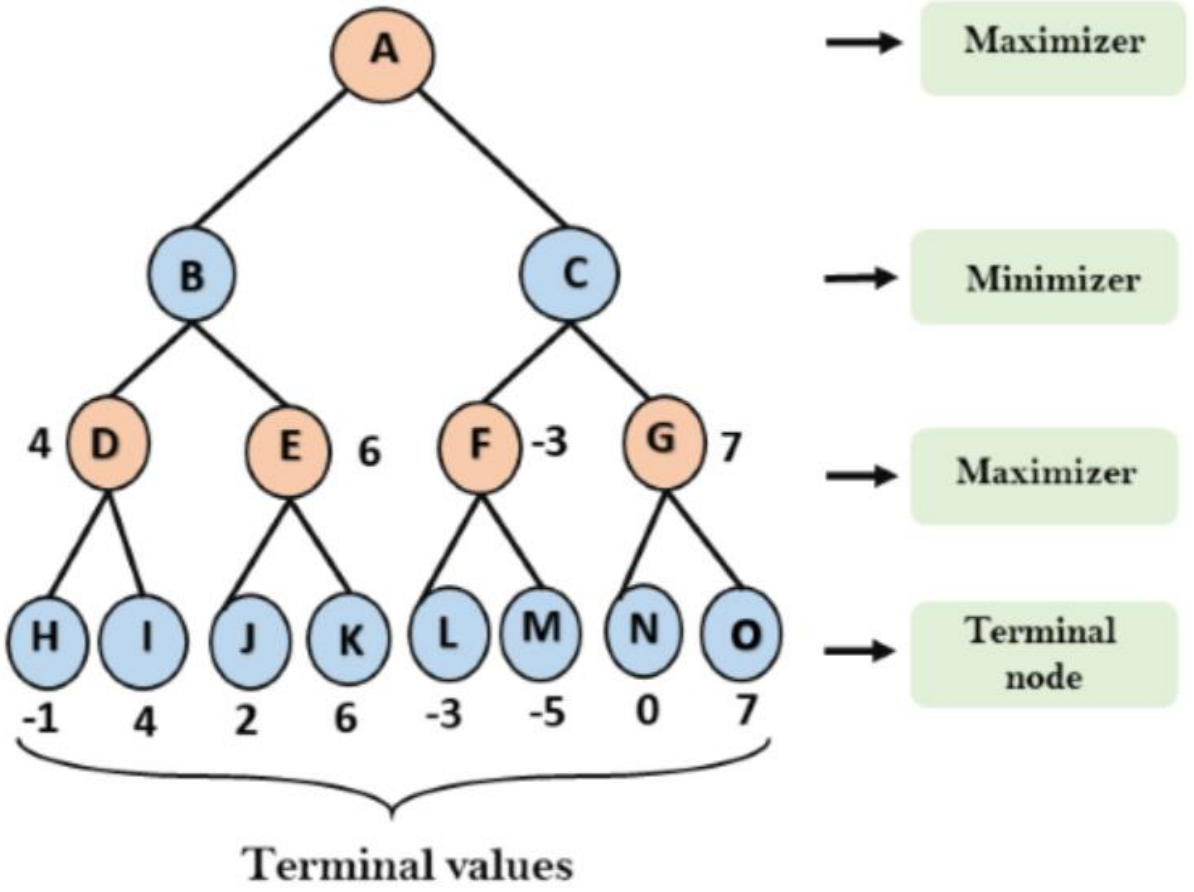


- **Max will try to maximize its utility (best move)**
- **Min will try to minimize opponent utility (worst move for opponent)**

Min-max search algorithm

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.

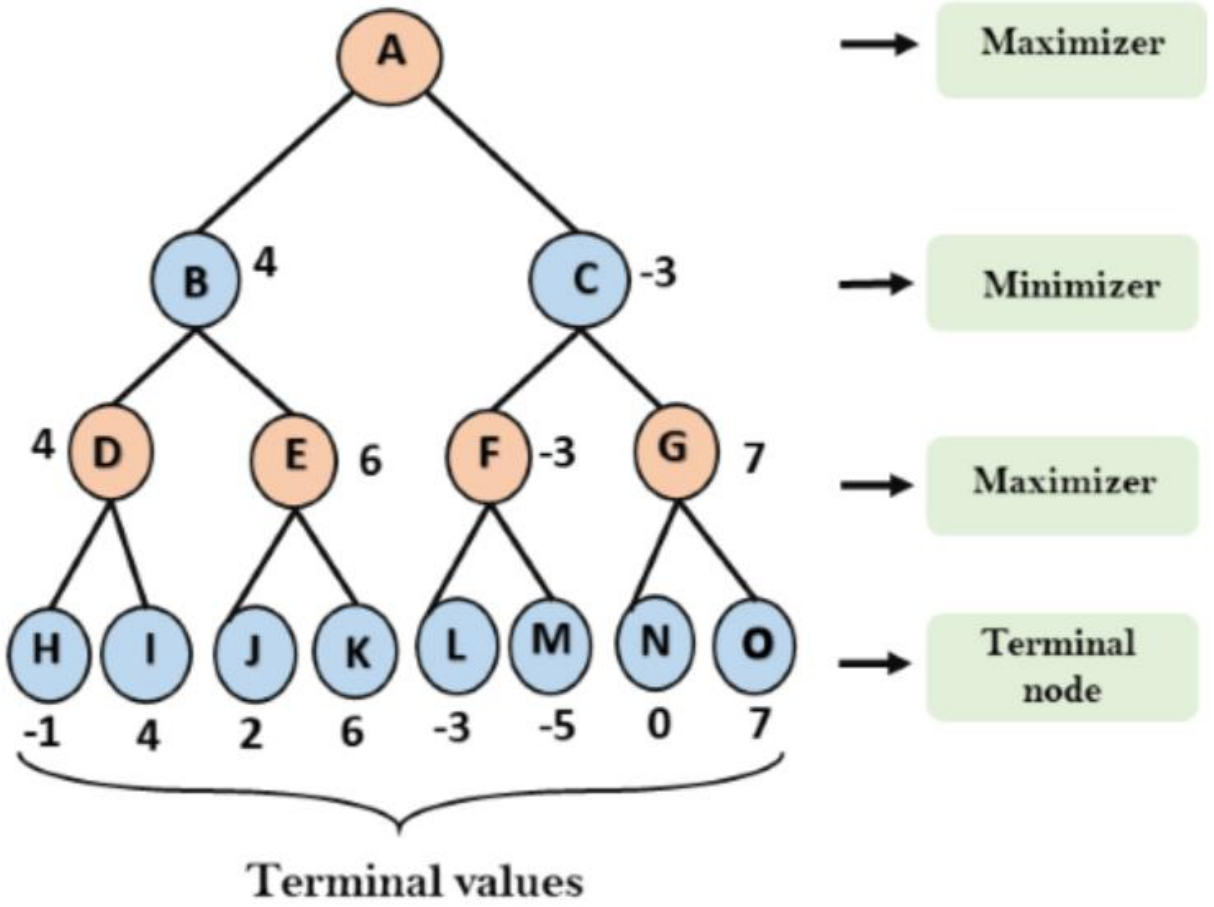
- **For node D** **max(-1, -∞) => max(-1,4)= 4**
- **For Node E** **max(2, -∞) => max(2, 6)= 6**
- **For Node F** **max(-3, -∞) => max(-3,-5) = -3**
- **For node G** **max(0, -∞) = max(0, 7) = 7**



Min-max search algorithm

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

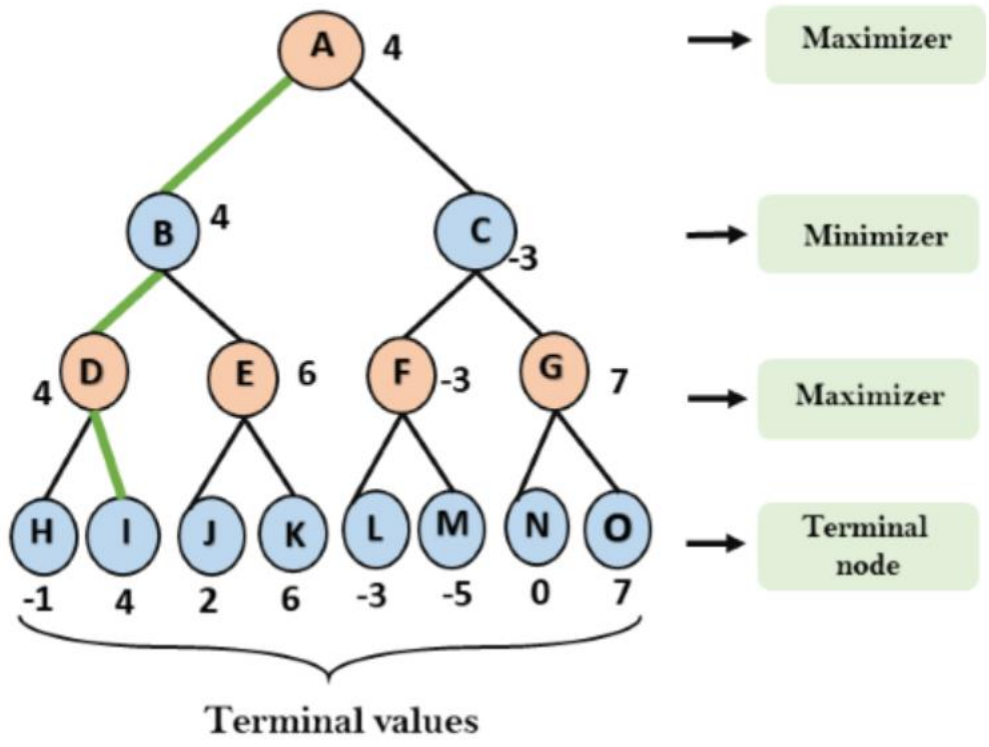


Min-max search algorithm

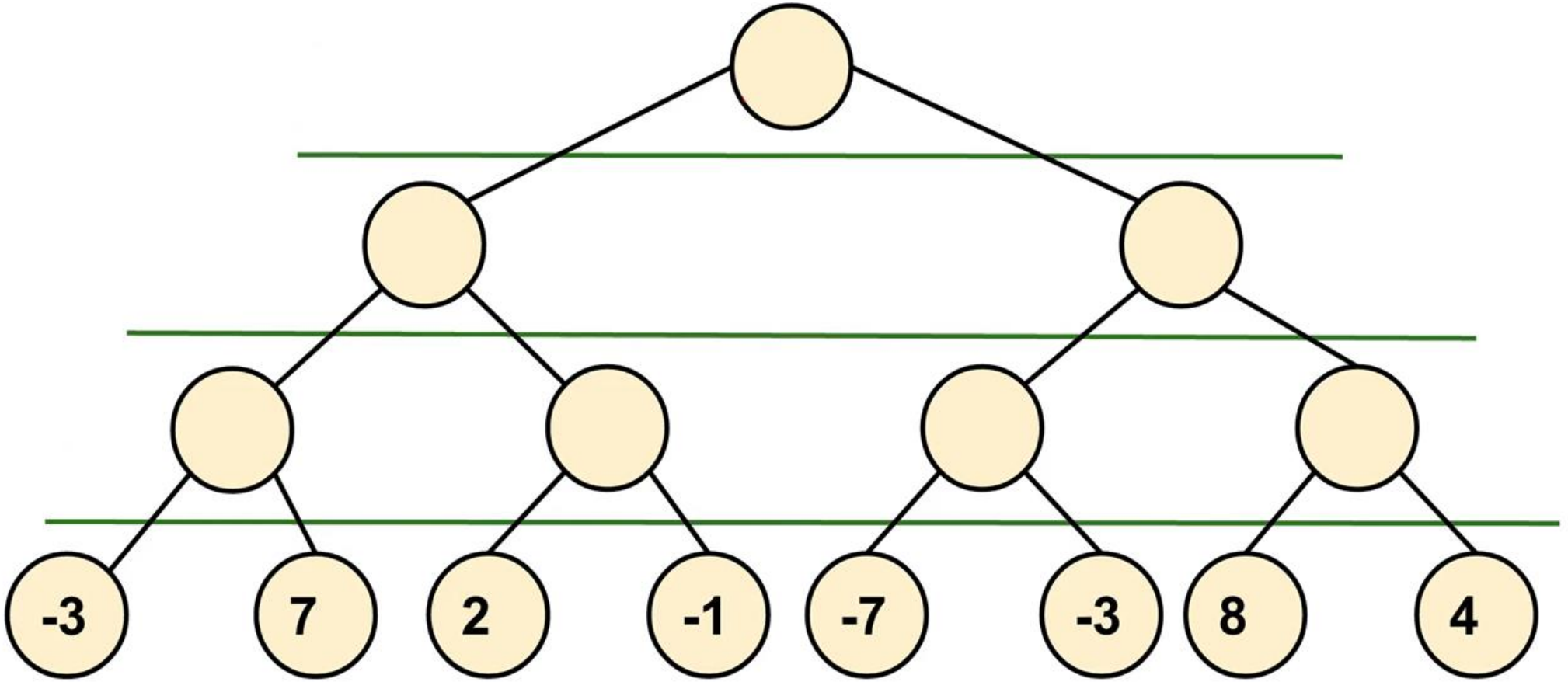
Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

•For node A max(4, -3)= 4

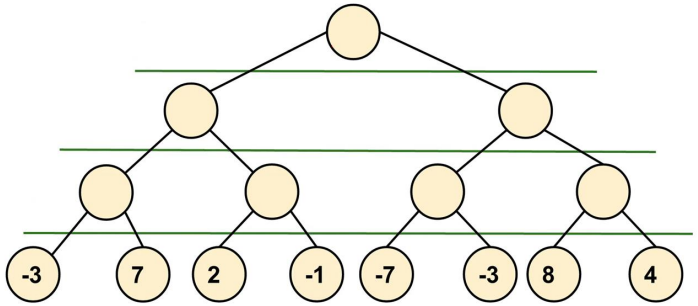
That was the complete workflow of the minimax two player game.

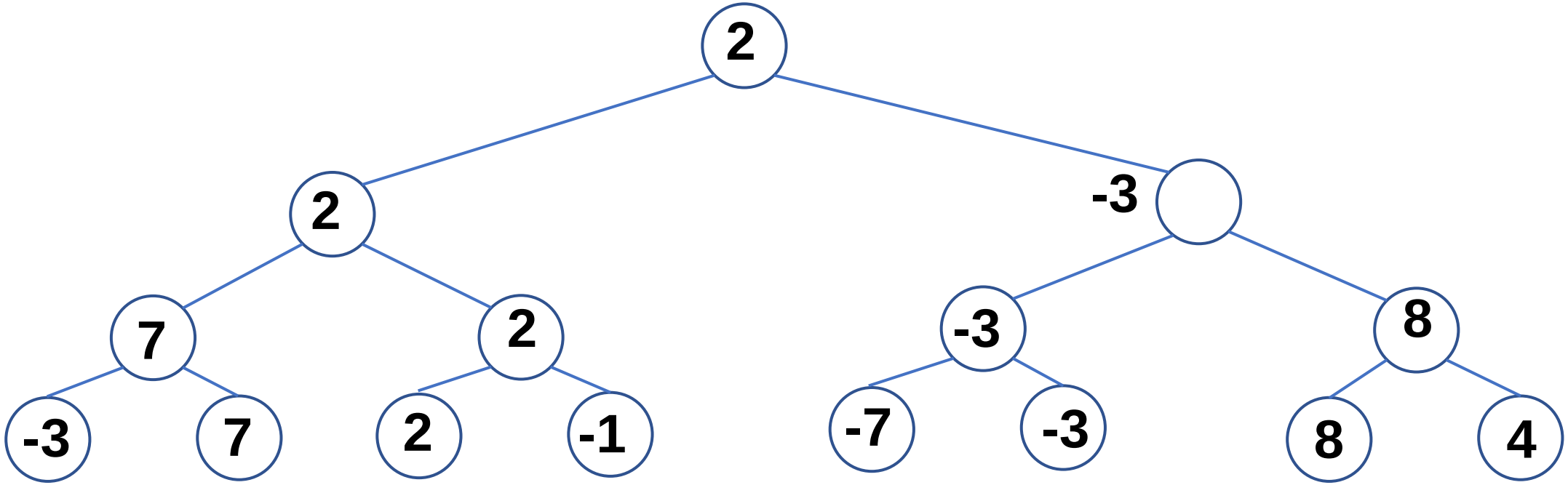


Min-max search algorithm

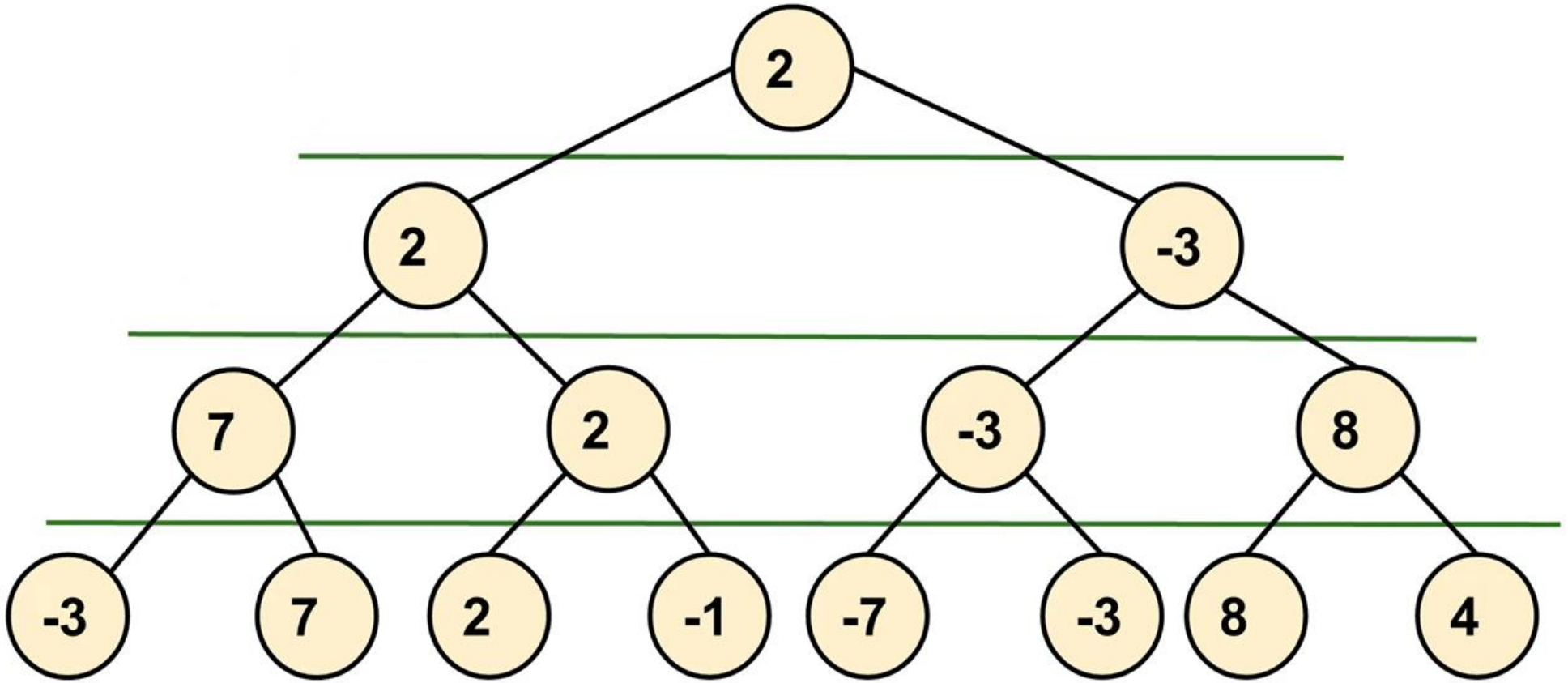


Min-max search algorithm





Min-max search algorithm



Min-max search algorithm

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)**.

Min-max search algorithm

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from alpha-beta pruning which we have discussed in the next topic.

Artificial Intelligence

By

Dr. Manoj Kumar

University School of Automation and Robotics

GGSIP University, East Campus, Delhi, India

MiniMax Algorithm

Algorithm: MINIMAX(Position, Depth, Player)

1. If DEEP-ENOUGH(*Position, Depth*), then return the structure

   VALUE = *STATIC*(*Position, Player*);
   PATH = nil

   This indicates that there is no path from this node and that its value is that determined by the static evaluation function.
2. Otherwise, generate one more ply of the tree by calling the function MOVE-GEN(Position Player) and setting SUCCESSORS to the list it returns.
3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned if DEEP-ENOUGH had returned true.
4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This is done as follows.
   Initialize BEST-SCORE to the minimum value that STATIC can return. It will be updated to reflect the hest score that can be achieved by an element of SUCCESSORS.
   For each element SUCC of SUCCESSORS, do the following:
   (a) Set RESULT-SUCC to
      MINIMAX(SUCC, *Depth* + 1, OPPOSITE(*Player*))
      This recursive call to MINIMAX will actually carry out the exploration of SUCC.
   (b) Set NEW-VALUE to - VALUE(RESULT-SUCC). This will cause it to reflect the merits of the position from the opposite perspective from that of the next lower level.
   (c) If NEW-VALUE > BEST-SCORE, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:
      (i) Set BEST-SCORE to NEW-VALUE.
      (ii) The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined hy the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESULT-SUCC).
5. Now that all the successors have been examined, we know the value of Position as well as which path to take from it. So return the structure
   VALUE = BEST-SCORE
   PATH = BEST-PATH

Optimal Decisions in multiplayer games

5.2.2 Optimal decisions in multiplayer games

Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues.

First, we need to replace the single value for each node with a *vector* of values. For example, in a three-player game with players A, B, and C, a vector $\langle v_A, v_B, v_C \rangle$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. (In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure 5.4. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Hence, the backed-up value of X is this vector. In general, the backed-up value of a node n is the utility vector of the successor state with the highest value for the player choosing at n.

Anyone who plays multiplayer games, such as Diplomacy or Settlers of Catan, quickly becomes aware that much more is going on than in two-player games. Multiplayer games usually involve **alliances**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. How are we to understand such behavior? Are alliances a natural consequence of optimal strategies for each player in a multiplayer game? It turns out that they can be.

A **utility function** is a mathematical function used in Artificial Intelligence (AI) to represent a system's preferences or objectives. It assigns a numerical value, referred to as utility, to different outcomes based on their satisfaction level.

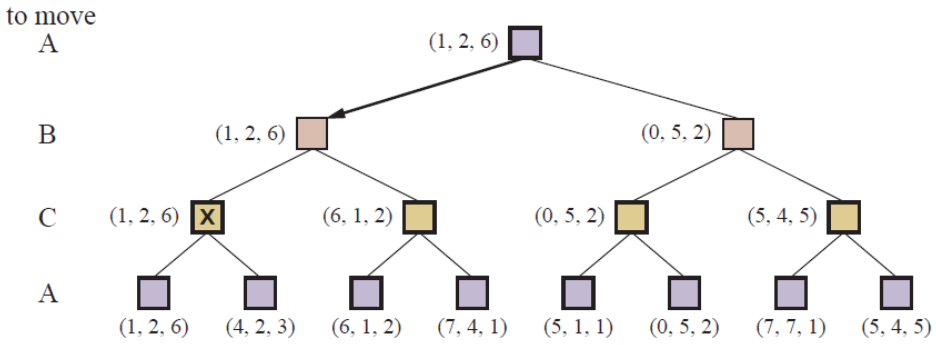


Figure 5.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Heuristic Alpha–Beta Tree Search

5.3 Heuristic Alpha–Beta Tree Search

To make use of our limited computation time, we can cut off the search early and apply a heuristic **evaluation function** to states, effectively treating nonterminal nodes as if they were terminal. In other words, we replace the UTILITY function with EVAL, which estimates a state's utility. We also replace the terminal test by a **cutoff test**, which must return true for terminal states, but is otherwise free to decide when to cut off the search, based on the search depth and any property of the state that it chooses to consider. That gives us the formula H-MINIMAX(s, d) for the heuristic minimax value of state s at search depth d:

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s,\text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a),d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a),d+1) & \text{if TO-MOVE}(s) = \text{MIN}. \end{cases}$$

Optimal Decisions in multiplayer games

5.3.1 Evaluation functions

A heuristic evaluation function $\text{EVAL}(s, p)$ returns an *estimate* of the expected utility of state s to player p, just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal. For terminal states, it must be that $\text{EVAL}(s, p) = \text{UTILITY}(s, p)$ and for nonterminal states, the evaluation must be somewhere between a loss and a win: $\text{UTILITY}(\mathit{loss}, p) \leq \text{EVAL}(s, p) \leq \text{UTILITY}(\mathit{win}, p)$.

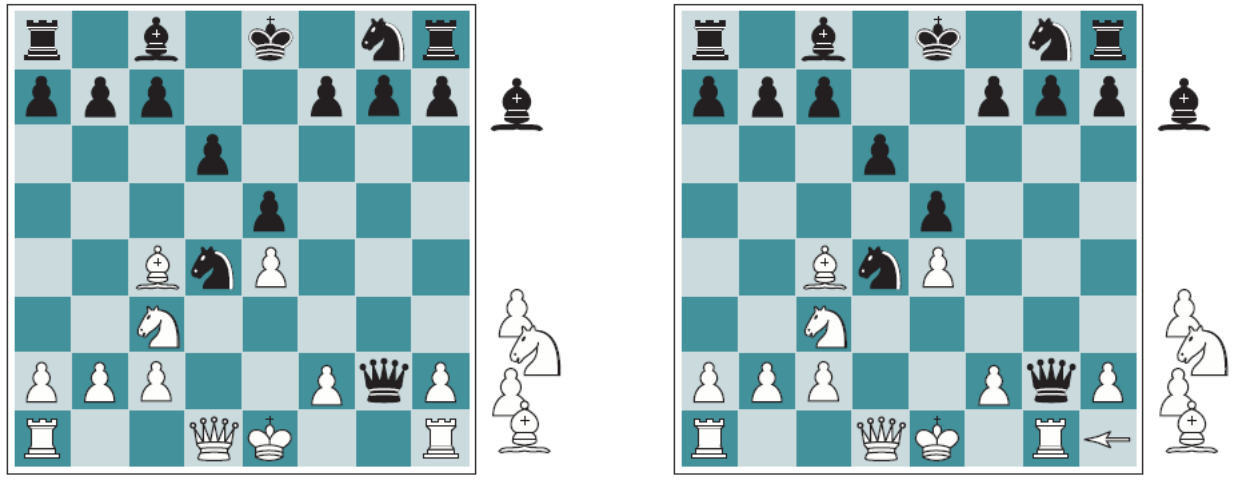
Beyond those requirements, what makes for a good evaluation function? First, the computation must not take too long! (The whole point is to search faster.) Second, the evaluation function should be strongly correlated with the actual chances of winning. One might well wonder about the phrase "chances of winning." After all, chess is not a game of chance: we know the current state with certainty, and no dice are involved; if neither player makes a mistake, the outcome is predetermined. But if the search must be cut off at nonterminal states, then the algorithm will necessarily be *uncertain* about the final outcomes of those states (even though that uncertainty could be resolved with infinite computing resources).

Let us make this idea more concrete. Most evaluation functions work by calculating various **features** of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on. The features, taken together, define various *categories* or *equivalence classes* of states: the states in each category have the same values for all the features. For example, one category might contain all two-pawn versus one-pawn endgames. Any given category will contain some states that lead (with perfect play) to wins, some that lead to draws, and some that lead to losses.

Optimal Decisions in multiplayer games

The evaluation function does not know which states are which, but it can return a single value that estimates the *proportion* of states with each outcome. For example, suppose our experience suggests that 82% of the states encountered in the two-pawns versus one-pawn category lead to a win (utility +1); 2% to a loss (0), and 16% to a draw (1/2). Then a reasonable evaluation for states in the category is the **expected value**: $(0.82 \times +1) + (0.02 \times 0) + (0.16 \times 1/2) = 0.90$. In principle, the expected value can be determined for each category of states, resulting in an evaluation function that works for any state.

In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities. Instead, most evaluation functions compute separate numerical contributions from each feature and then *combine* them to find the total value. For



(a) White to move (b) White to move

Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Alpha-Beta Pruning algorithm

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

Cut off the search by exploring less number of nodes

Alpha-Beta Pruning algorithm

•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 **+∞**.

•The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, **but it removes all the nodes which are not really affecting the final decision** but making algorithm slow. ***Hence by pruning these nodes, it makes the algorithm fast.***

Condition for Alpha-beta pruning

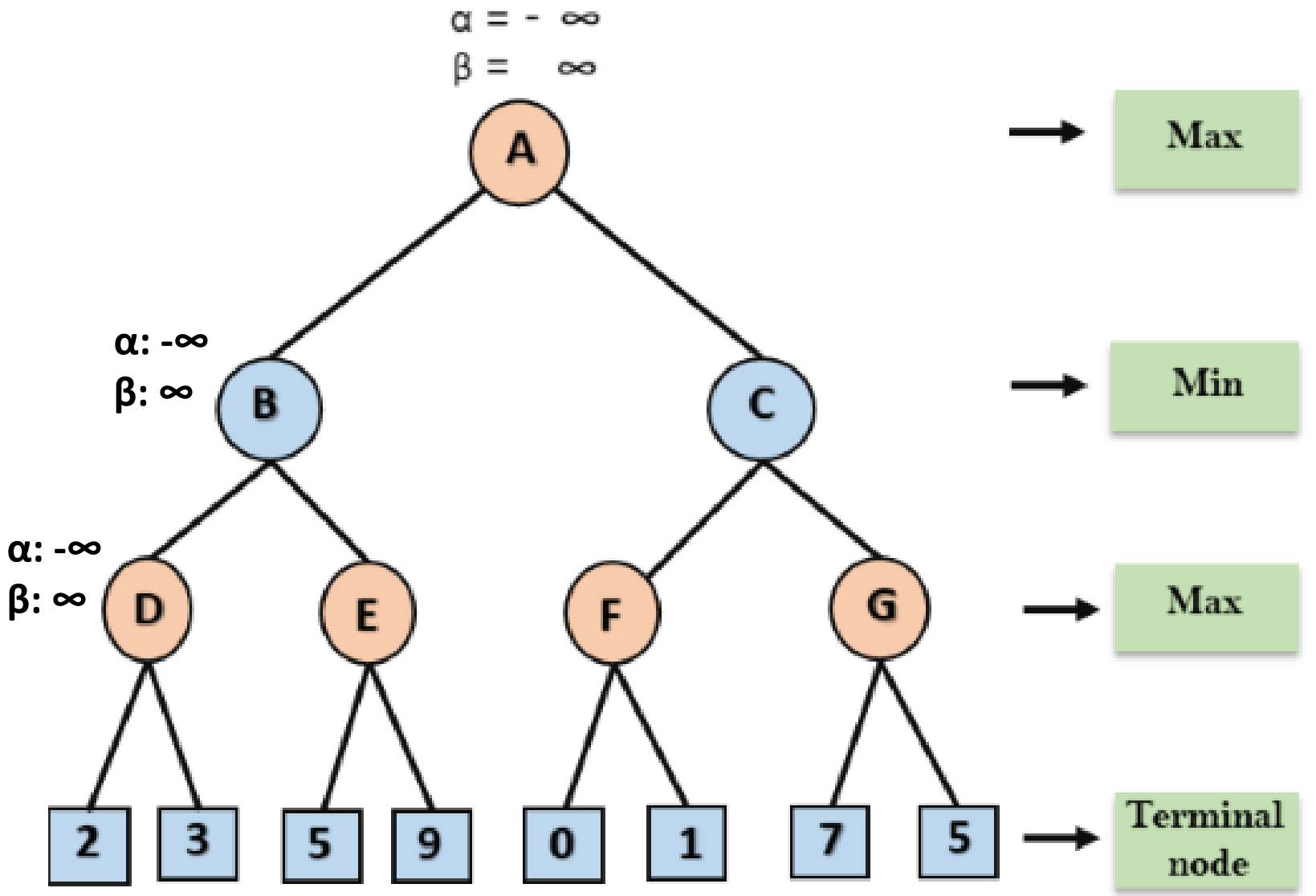
- **The main condition which required for alpha-beta pruning is:** $\alpha \geq \beta$

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

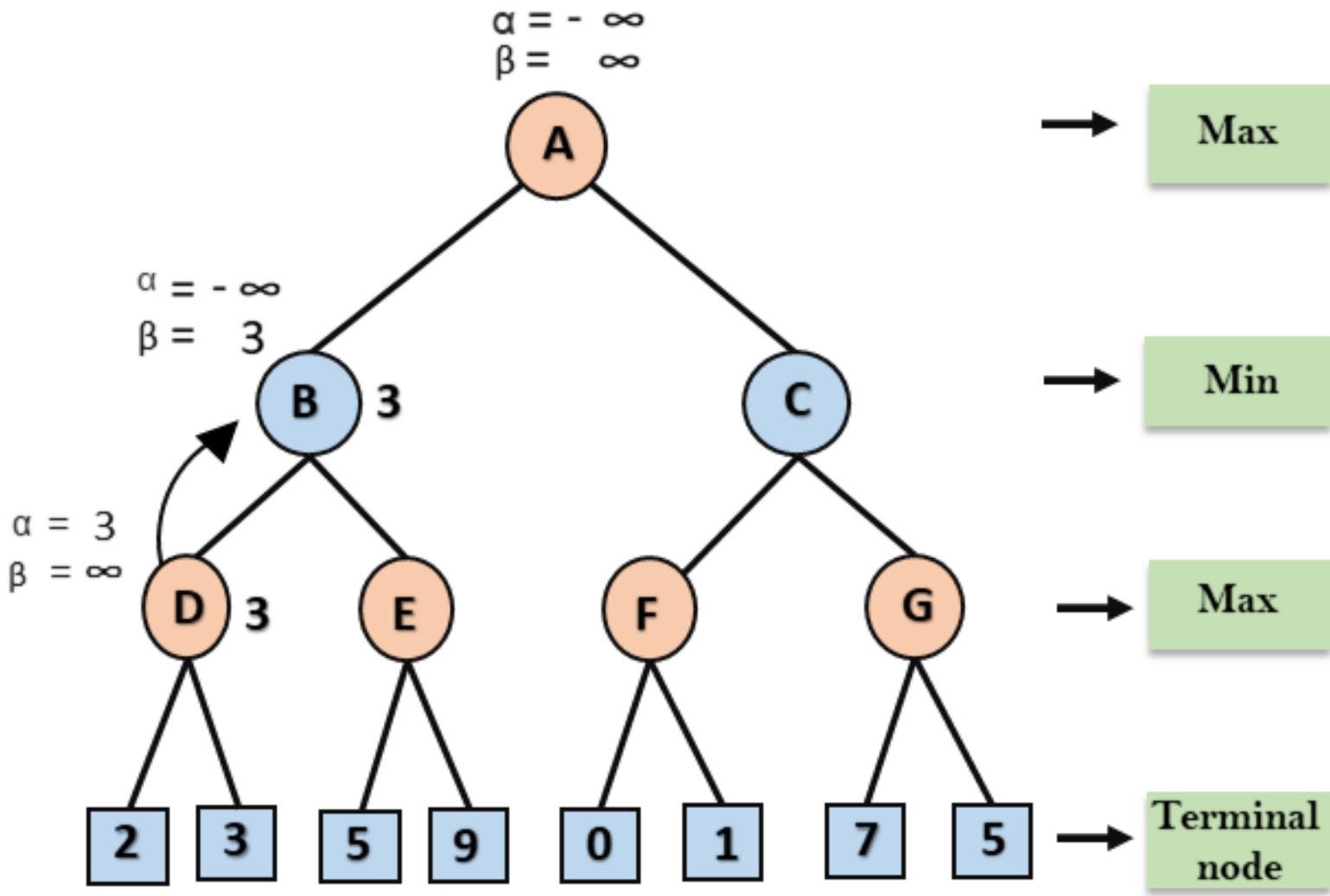
Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



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.

α: 3
β: ∞

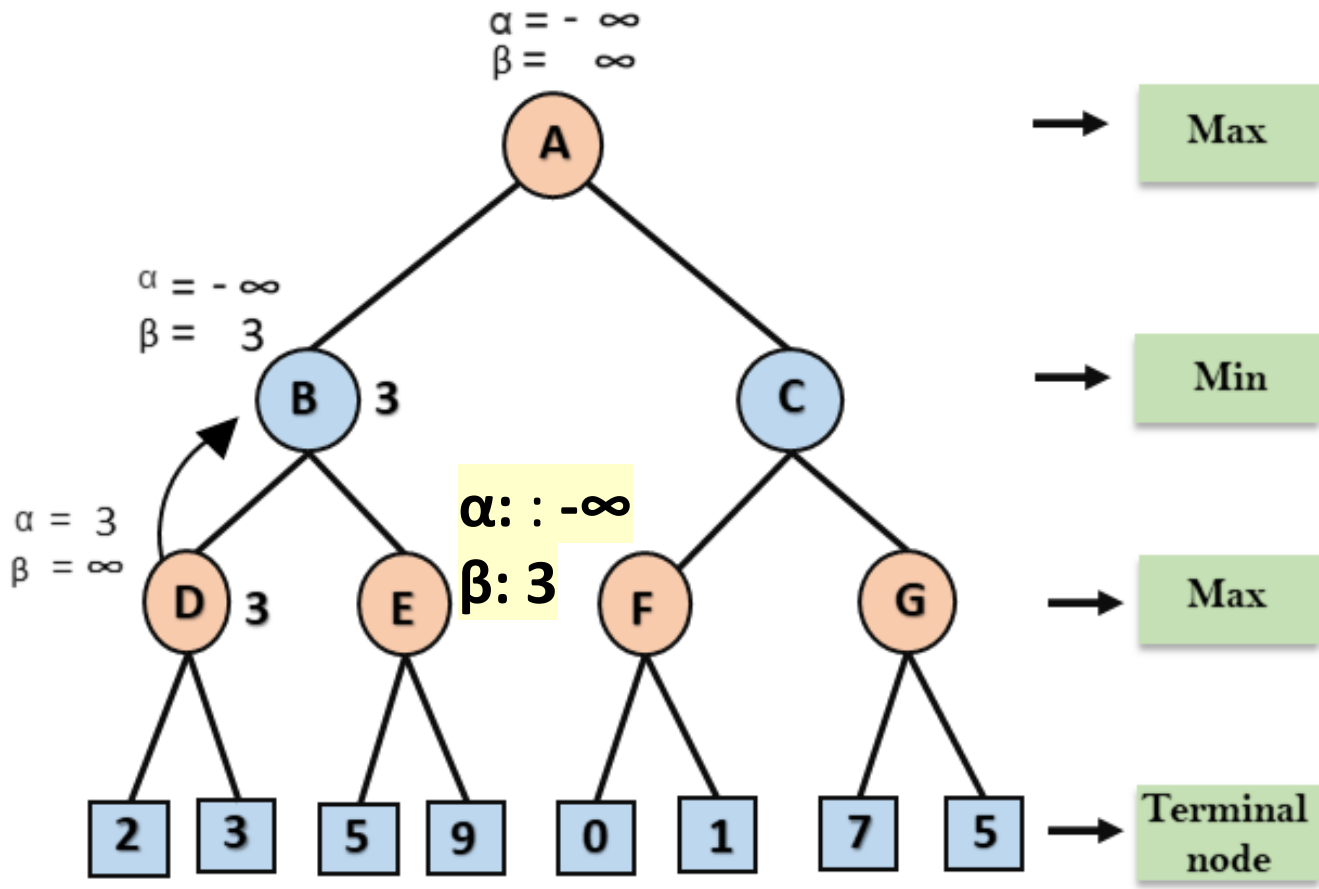


α: -∞
β: ∞

Working of Alpha-Beta Pruning

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**.

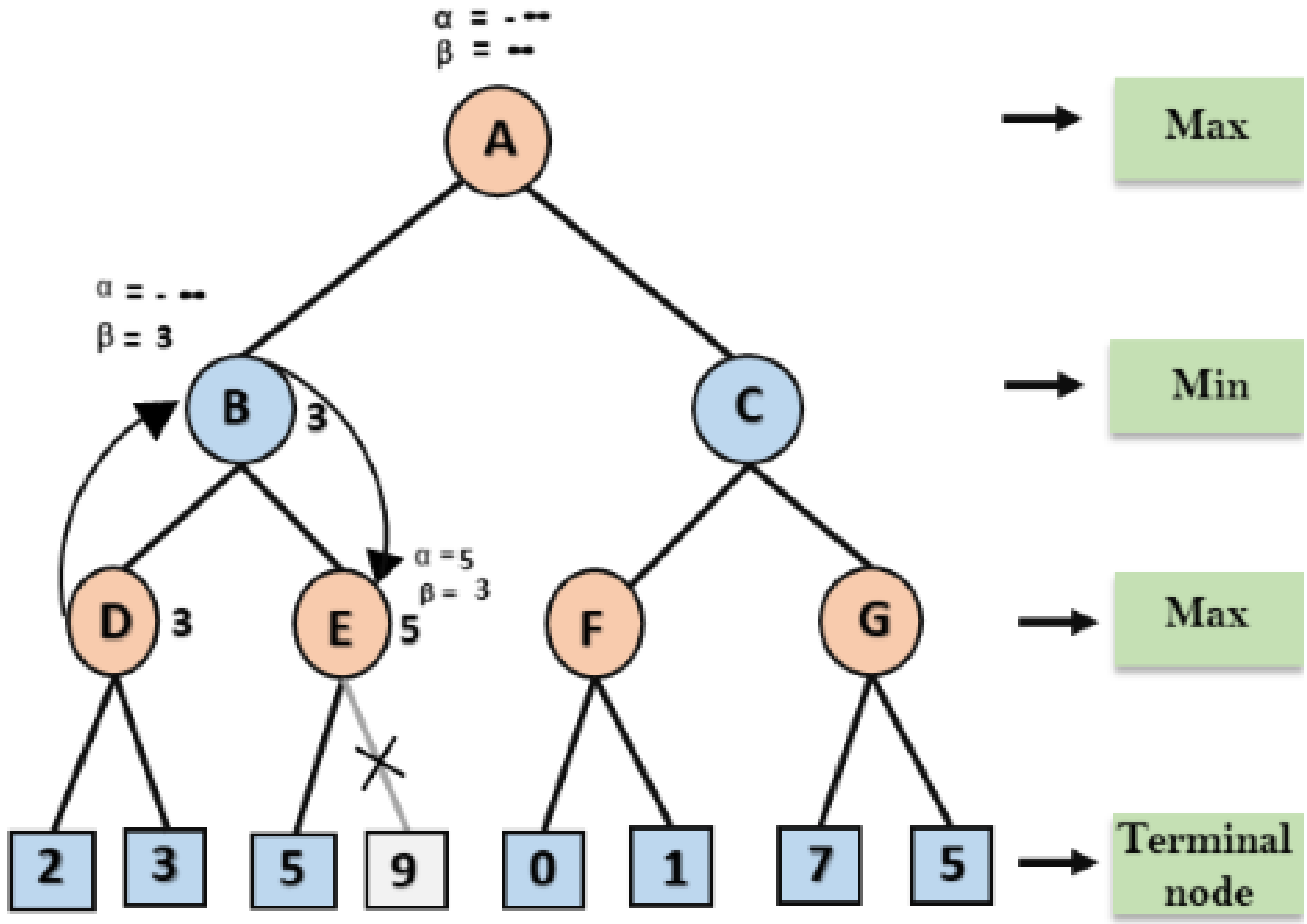
In the next step, algorithm traverse the next successor of Node B which is node **E, and the values of α= -∞, and β= 3** will also be passed.



α: -∞
β: ∞

Working of Alpha-Beta Pruning

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.**



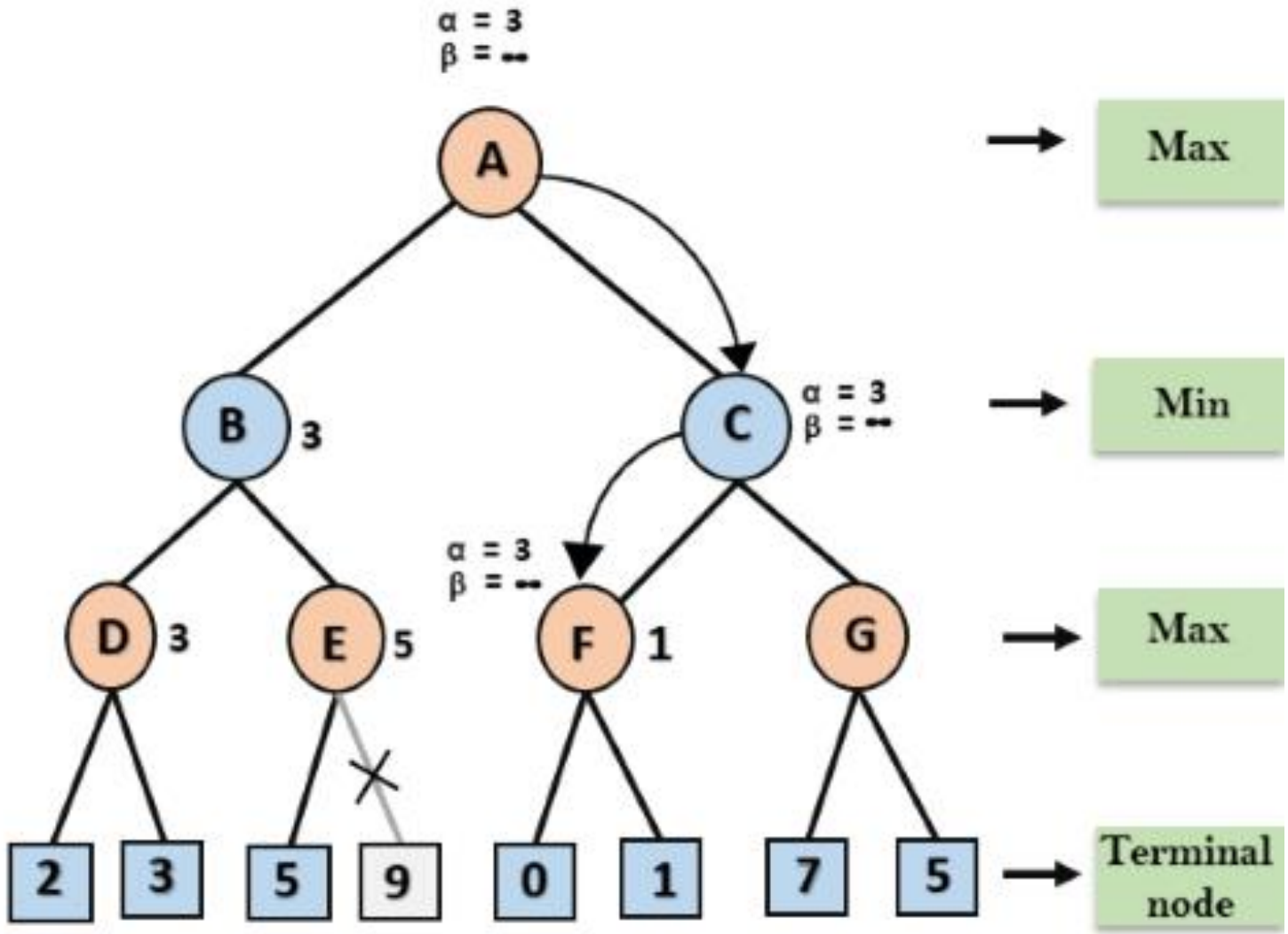
α: -∞
β: ∞

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.**

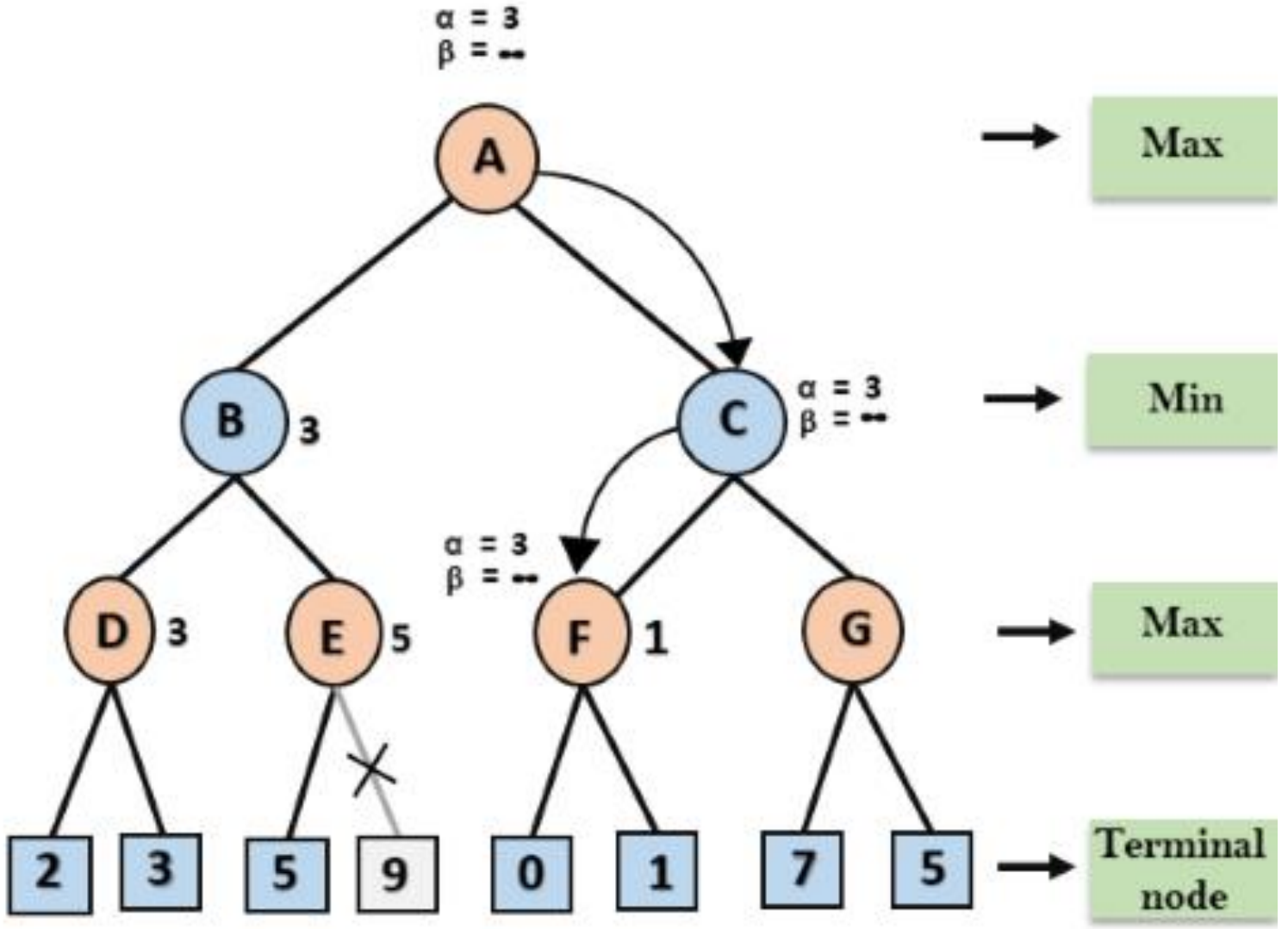
α: -∞
β: ∞



α: -∞
β: ∞

Working of Alpha-Beta Pruning

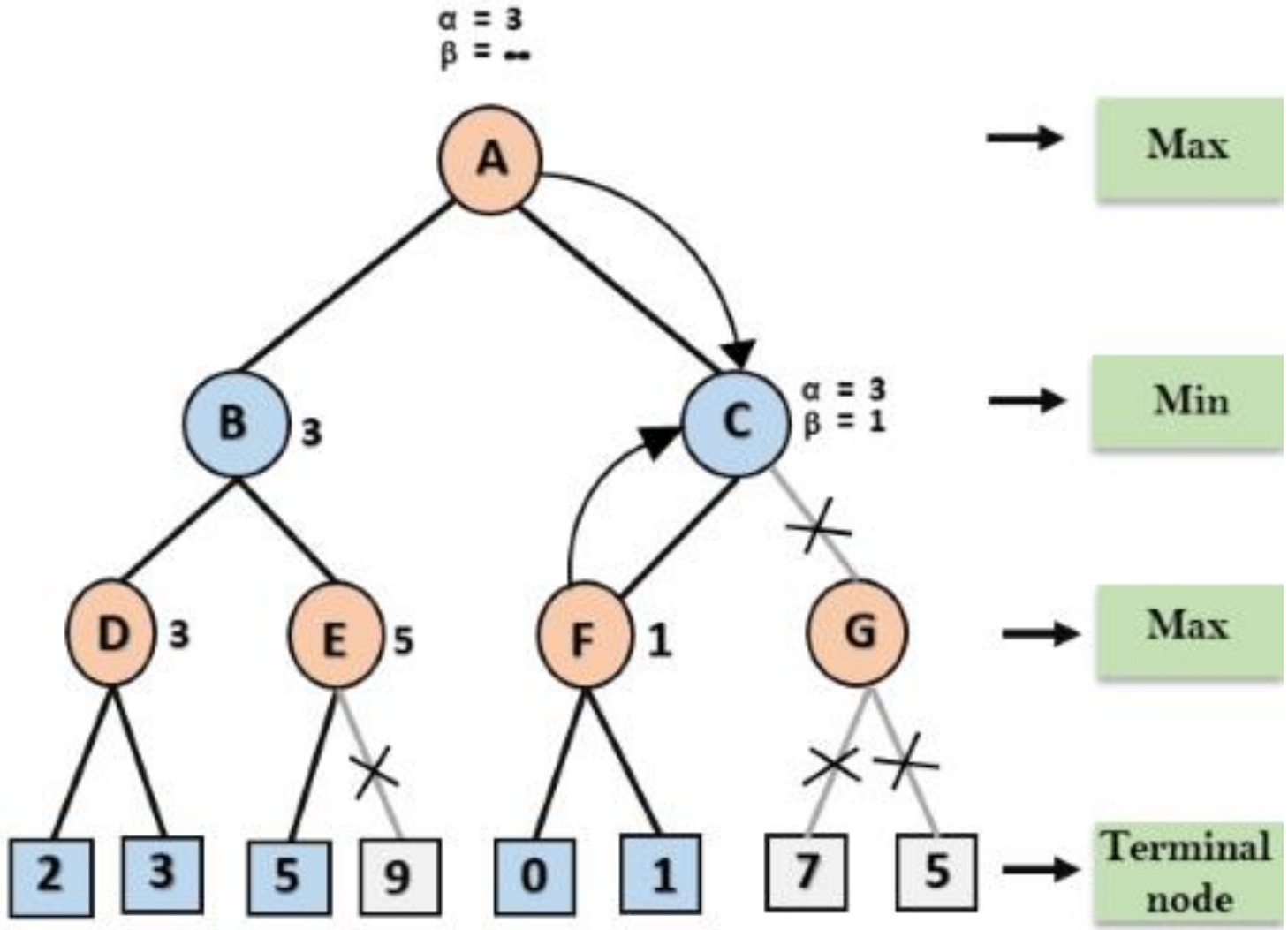
Step 6: At node F, again the value **of α will be compared with left child which is 0**, and **max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3**, but the node **value of F will become 1.**



α: -∞
β: ∞

Working of Alpha-Beta Pruning

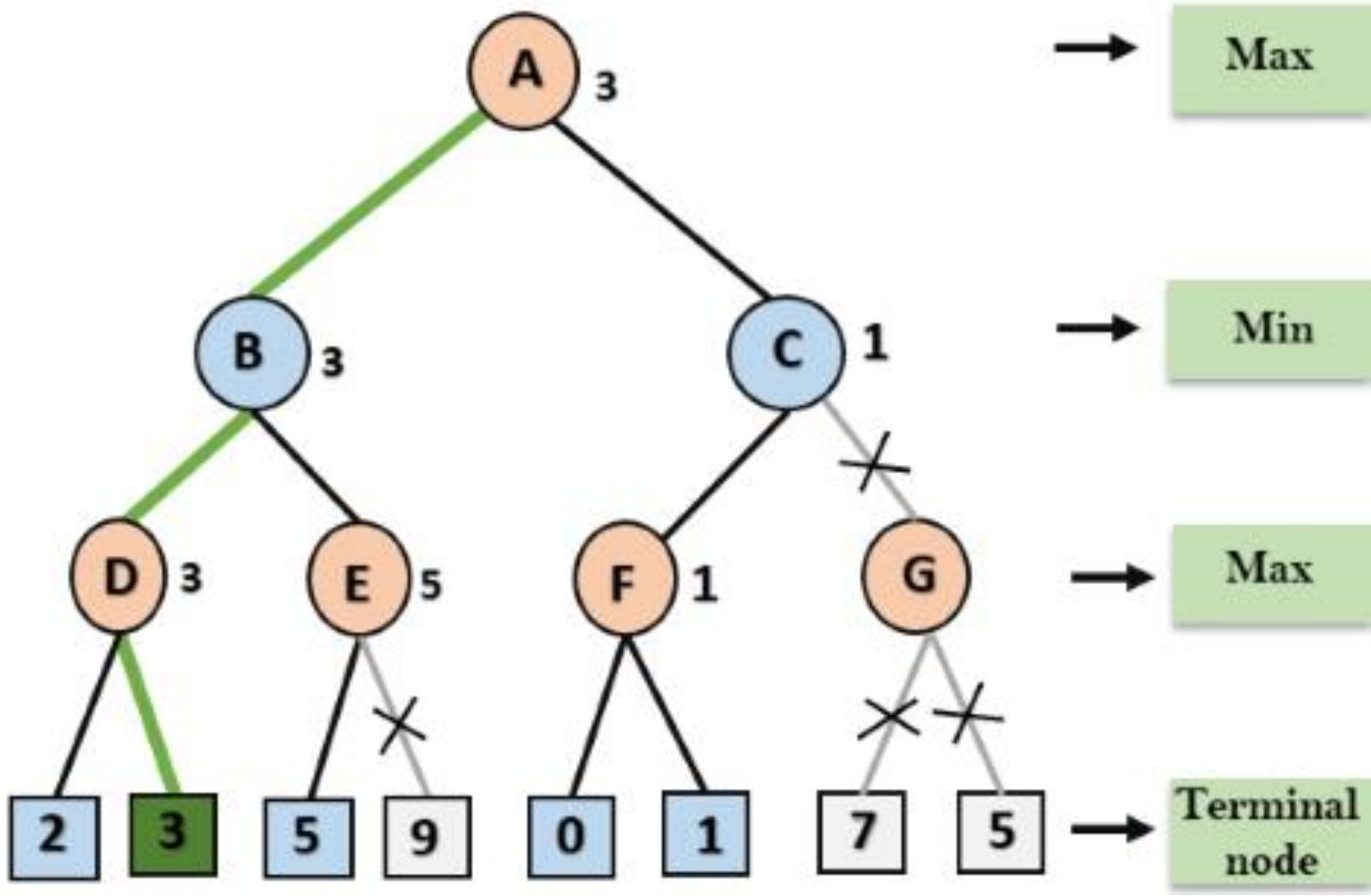
Step 7: Node **F returns the node value 1 to node C**, at **C α= 3 and β= +∞**, here the value of beta will be changed, **it will compare with 1 so min (∞, 1) = 1**. Now at C, α=3 and β= 1, and again it satisfies the condition **α>=β**, so the next child of C which is G will be pruned, and the algorithm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



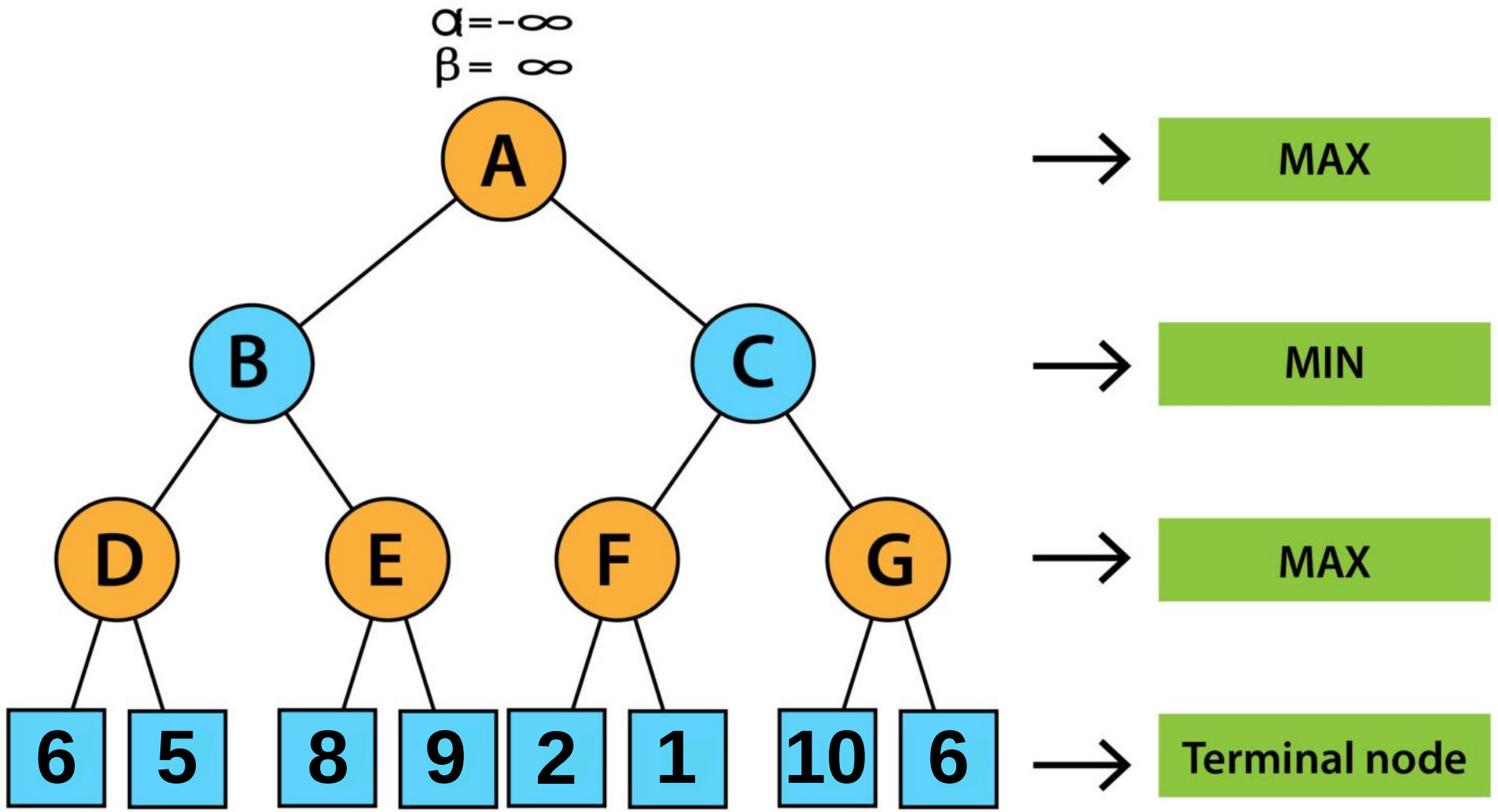
α: -∞
β: ∞

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.**

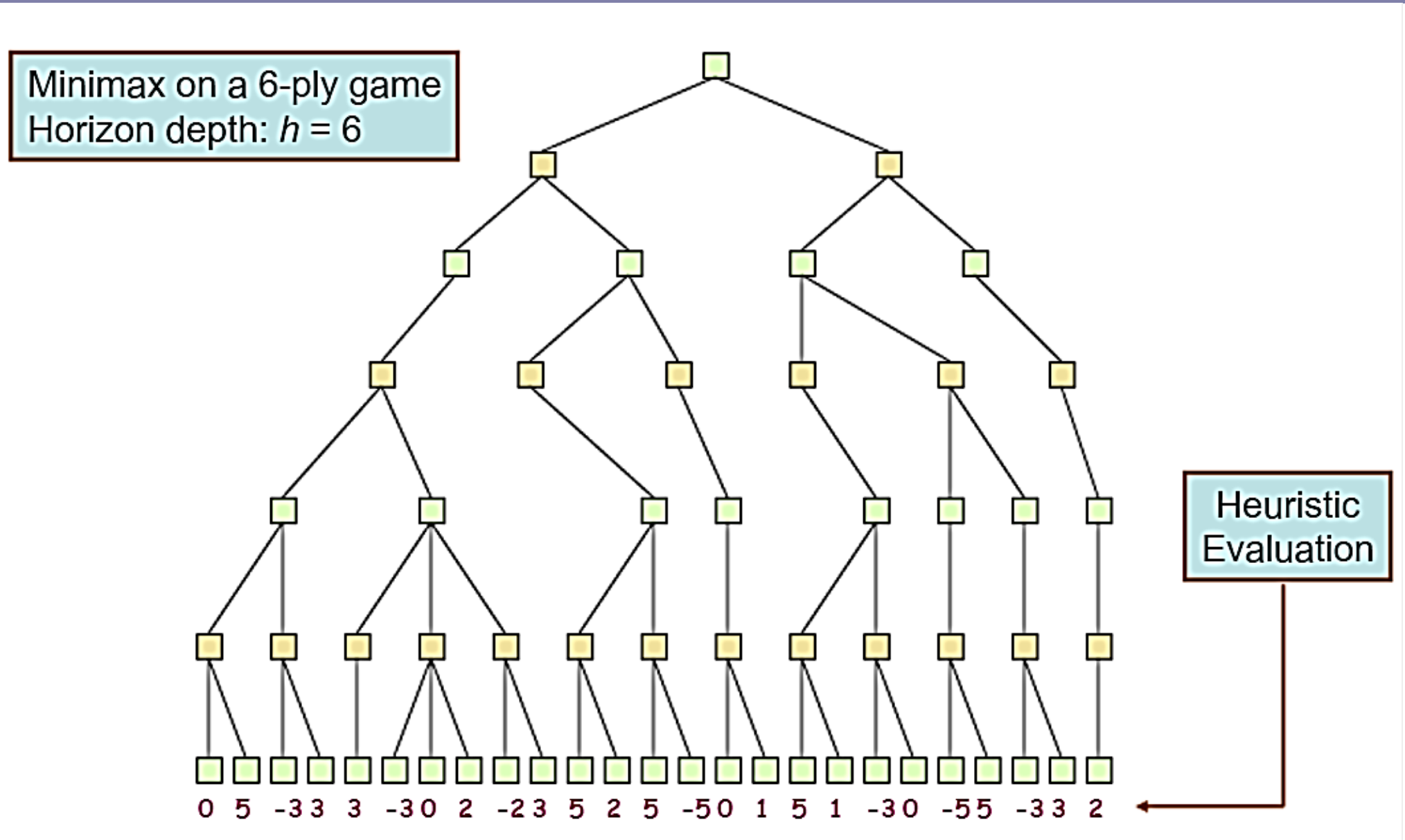


Example 1 of Alpha-Beta Pruning

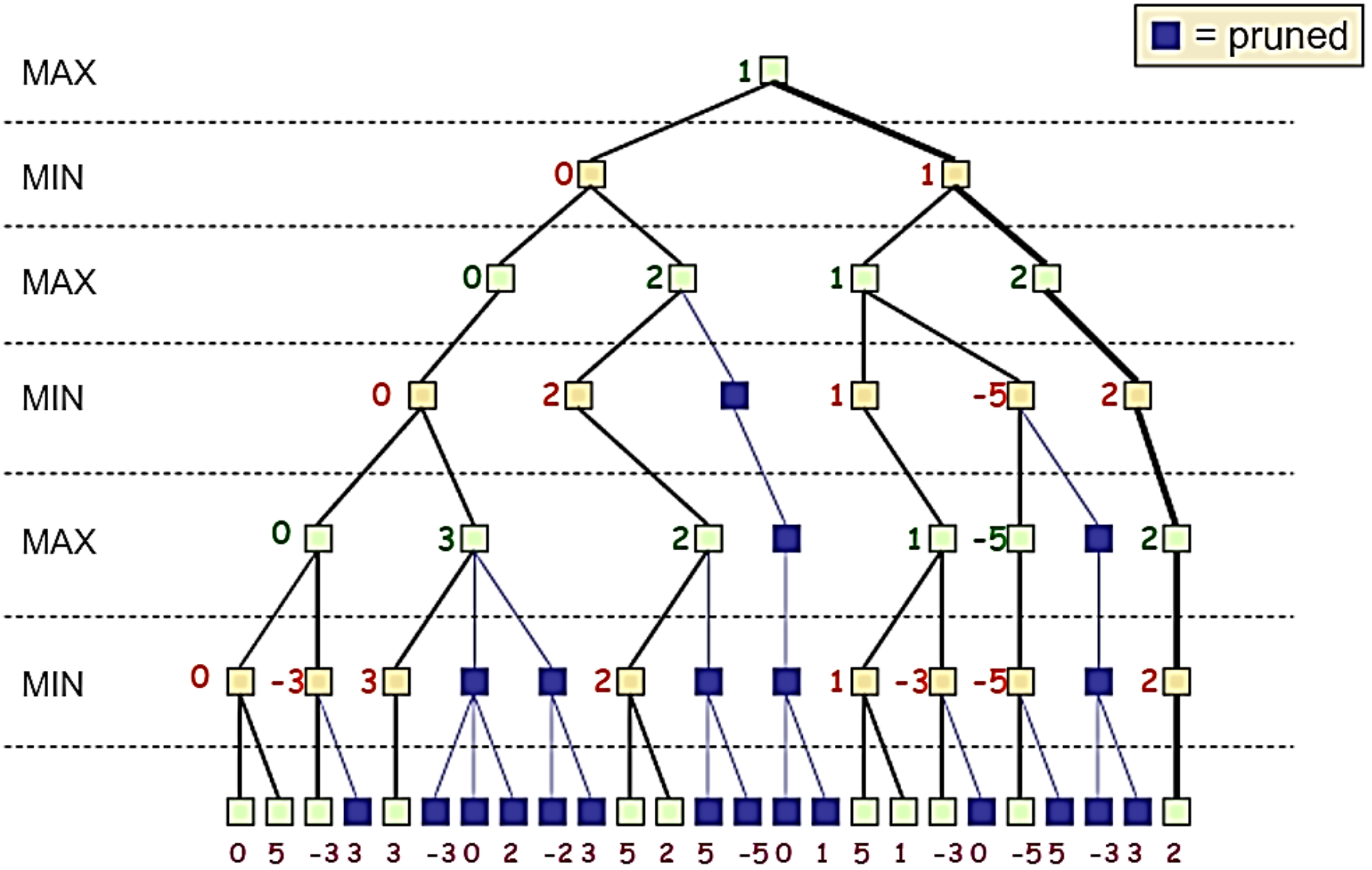


Example 1 of Alpha-Beta Pruning

Example 2 of Alpha-Beta Pruning

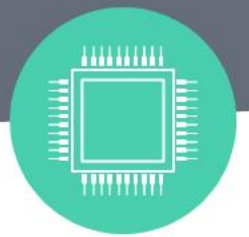


Example2 of Alpha-Beta Pruning



Artificial Intelligence

By
Dr. Manoj Kumar



University School of Automation and Robotics
GGSIP University, East Campus, Delhi, India

What is Knowledge?

- facts, information, and skills acquired through experience or education; the theoretical or practical understanding of a subject.
- Knowledge = information + rules
- EXAMPLE
- Doctors, managers.

What is Knowledge?

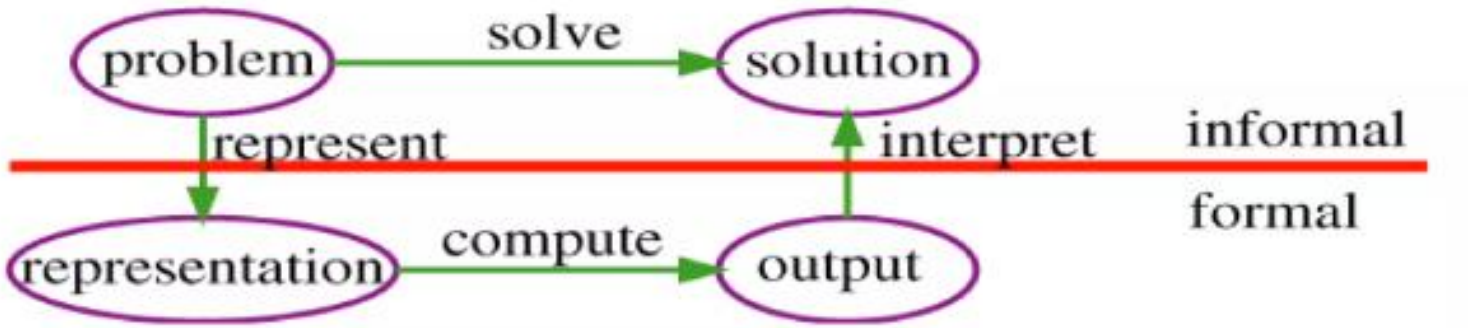
- **Importance of Knowledge**
  - Knowledge can be defined as the body of facts and principles accumulated by human-kind or the act, fact or state of knowing.
  - Actuality it is more than this, it also includes having a familiarity with language, concepts, procedures, rules, ideas, abstractions , places, customs, facts associations along with ability to use these notions effectively in modeling different aspects of the world.

- **How is knowledge stored in biological organisms and computers?**
  - Human brain weighs 3.3 pounds - estimated number of neurons 10^{12} - potential storage – 10^{14}
  - In computers, knowledge is also stored as symbolic structures, in the form of collections of magnetic spots and voltage states.

Knowledge representation

What is Knowledge representation?

- Knowledge representation is a relationship between two domains.
- **Knowledge representation**(KR) is the field of **artificial intelligence** (**AI**) that representing information about the world in a form of computer system, that can solve complex tasks, such as diagnosing a medical condition.



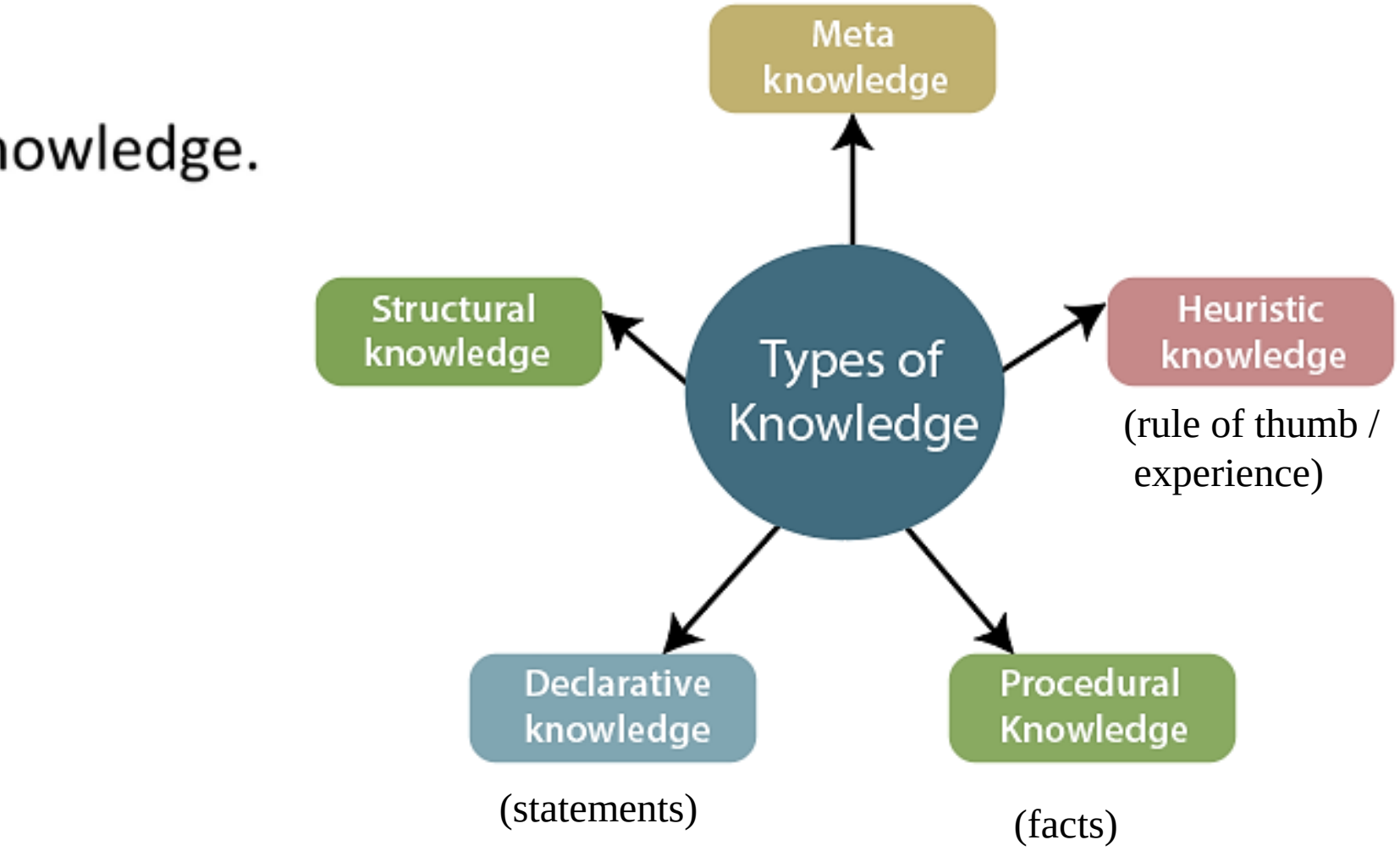
Knowledge representation

- Let’s take the following examples
  - A is tall.
  - A Loves B. ☺
  - C has learned to use recursion to manipulate linked lists in several programming languages.

  - 1st one represents a fact, an attribute possessed by a person.
  - 2nd expresses a complex binary relation between two persons.
  - 3rd is most complex, expressing relations between a person and more abstract programming concepts.

Types of Knowledge

- There are 5 types of knowledge.
- 1) Procedural k.
- 2) Declarative k.
- 3) Meta k.
- 4) Heuristic k.
- 5) Structural k.



Procedural Knowledge

- Procedural knowledge refers to knowledge about **how to do something** or perform a particular task.
- It involves **knowing the steps, actions, and sequences required** to complete a specific activity.

- Gives information/ knowledge about how to achieve something.
- Describes how to do things provides set of directions of how to perform certain tasks.
- **Procedural knowledge**, also known as imperative **knowledge**, is the **knowledge** exercised in the performance of some task.
- It depends on targets and problems.
- Example
- How to drive a car?

Example: Riding a Bicycle
Procedural Knowledge: Knowing how to ride a bicycle involves a series of steps and actions, such as:
1.Balancing on two wheels.
2.Pedaling to move forward.
3.Steering to control direction.
4.Using the brakes to stop.
5.Starting and stopping smoothly.
This knowledge is acquired through practice and experience, and once you have learned how to ride a bicycle, you can apply this procedural knowledge to ride any bike effectively.

Declarative Knowledge

- Declarative knowledge refers to knowledge that can be **explicitly stated or declared**, typically in the form of statements or facts.
- **It is knowledge about "what is" or "what exists".**
- Declarative knowledge is the opposite of procedural knowledge, which is knowledge about "how to do" something.

- Its about statements that describe a particular object and its attributes , including some behavior in relation with it.
- *"Can this knowledge be true or false?"*
- It is non-procedural, independent of targets and problem solving.
- Example
- It is sunny today and chemise are red.

Declarative Knowledge Example

Statement: "Water boils at 100 degrees Celsius (212 degrees Fahrenheit) at standard atmospheric pressure.“

In this example, the statement provides a clear, factual piece of information about the temperature at which water boils. It is declarative knowledge because it states a fact without describing how to boil water or perform any specific actions. Instead, it simply presents a piece of information about the natural world.

Declarative knowledge is commonly found in textbooks, encyclopedias, databases, and other sources of factual information. It forms the basis for understanding various subjects, and it can be used as building blocks for problem-solving and decision-making in various fields, including science, mathematics, and everyday life.

Metacognitive/Meta Knowledge

- Meta-knowledge, also known as metacognitive knowledge, refers to knowledge about one's own cognitive processes, strategies, and abilities.
- It is knowledge about what you know or don't know and how you approach learning, problem-solving, and decision-making.
- Meta-knowledge allows individuals to monitor and control their cognitive activities, which can lead to more effective learning and decision-making.

- It’s a knowledge about knowledge and how to gain them.
- Example
- The knowledge that blood pressure is more important for diagnosing a medical condition then eyes color.

Meta Knowledge Example

Example: Imagine you are a student studying for an exam. You have a piece of meta-knowledge that says, "I learn best when I take notes during lectures."

1.The knowledge about "learning best when taking notes" is the declarative aspect of meta-knowledge. It's a statement about your preferred learning strategy.

2.The meta-cognitive aspect comes into play when you consciously recognize that you learn best through note-taking. You are aware of your own learning preferences and have a strategy in place to maximize your learning efficiency.

Meta-knowledge is knowledge about how you learn, think, and make decisions. It allows individuals to adapt their strategies and approaches to better suit their cognitive abilities and preferences, ultimately improving their learning and problem-solving outcomes.

Heuristic Knowledge

- Heuristic knowledge refers to a set of rules, strategies, or shortcuts that people use to make decisions or solve problems more efficiently, even when the solution is not guaranteed to be optimal.
- Heuristics are practical problem-solving approaches that help simplify complex situations by providing approximate solutions or guiding decision-making based on past experience or common sense.

- Representing knowledge of some expert in a field or subject.
- Rules of thumb.
- Heuristic Knowledge are sometimes called shallow knowledge.
- Heuristic knowledge are empirical as opposed to deterministic.

Heuristic Knowledge Example

Imagine you're trying to find a parking spot in a busy downtown area. You know from experience that it's often challenging to find parking during certain times of the day. So, you decide to use the "follow the person" heuristic. This means you watch for someone walking toward a parked car and follow them, assuming they are leaving and freeing up a parking spot. **This is a heuristic strategy** because it simplifies your search for a parking space by relying on the observed behavior of others, even though it's not guaranteed to work every time.

In this example:

- **The "follow the person" strategy is a heuristic.**
- **You're using your past experience and common sense to guide your decision-making.**
- **While it may not always result in a parking spot, it's a practical approach to increase your chances of finding one without spending excessive time searching.**

- Heuristic knowledge is commonly used in various aspects of daily life, problem-solving, and decision-making.
- It allows people to make reasonably good decisions quickly and efficiently, even when facing complex or uncertain situations.

Structural Knowledge

- Structural knowledge, in the context of cognitive psychology and learning, refers to knowledge that is organized and interconnected in a meaningful way.
- It involves understanding the relationships, hierarchies, and connections between different pieces of information or concepts.
- Structural knowledge allows individuals to not only possess discrete pieces of information but also to see how they relate to one another, leading to a deeper and more integrated understanding.

- Describes what relationship exists between concepts/ objects.
- Describe structure and their relationship.
- Example
- How to various part of car fit together to make a car, or knowledge structures in term of concepts, sub concepts and objects.

Knowledge Example

Imagine you are learning about the solar system in school. You could have two different levels of knowledge:

1.Declarative Knowledge: You know individual facts about the solar system, such as the names of the planets, their order from the sun, and some key characteristics of each planet.

2.Structural Knowledge: In addition to the individual facts, you have a deeper understanding of the structure and relationships within the solar system. You know that the planets are organized in a specific order based on their distance from the sun (e.g., Mercury, Venus, Earth, Mars, etc.), and you understand the concept of orbits. You also grasp that the planets closer to the sun have shorter orbits and that the outer planets have longer orbits.

In this example:
•Declarative knowledge involves knowing isolated facts about the solar system.
•Structural knowledge goes beyond that by organizing these facts into a coherent framework. It involves understanding the hierarchy and relationships between the planets, the concept of orbits, and how these elements fit together.

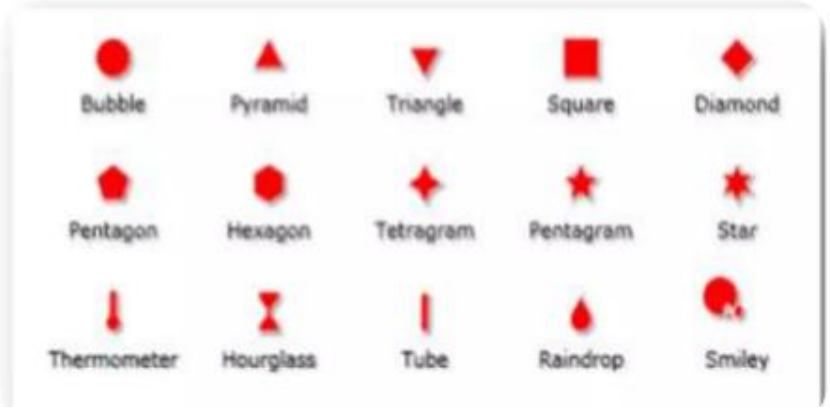
Knowledge representation

- There are multiple approaches and scheme that comes to mind when we begin to think about representation.
- 1)Pictures and symbols
- 2)Graphs and network
- 3)Numbers

Knowledge representation

1)Pictures and symbols

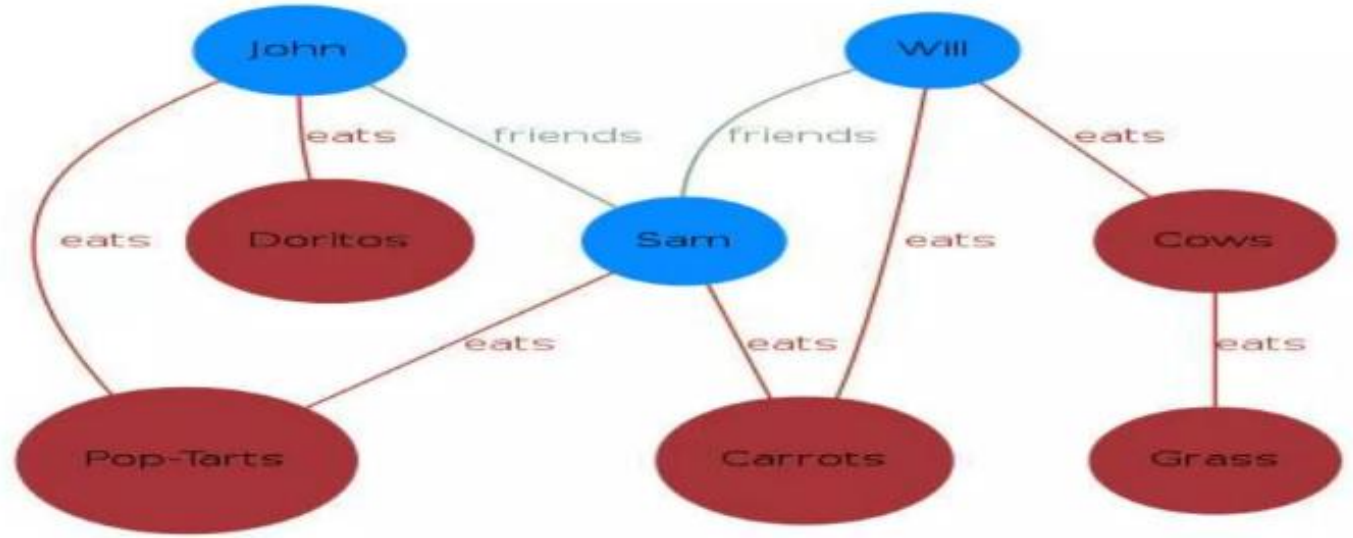
- Pictorial representation are not easily translate to useful information is computer because computer can’t interpret pictures directly with out complex reasoning.
- Through pictures are useful for human understanding.



Knowledge representation

2)Graph and network

- Allows relationship between objects to be incorporated.
- We can represent procedural knowledge using graphs.



Knowledge representation

3)Numbers

- Numbers are an integral part of knowledge representation used by humans.
- Numbers translate easily to computer representation.