Artificial Intelligence & Expert System

UNIT I

Overview & Search Techniques

Introduction to AI

- Artificial Intelligence is concerned with the design of intelligence in an artificial device.
- The term was coined by McCarthy in 1956.
- Artificial intelligence (AI) is wide-ranging branch of computer science concerned with building smart machines capable of performing tasks that typically require human intelligence.

Introduction to AI

Different interpretations have been used by different researchers as defining the scope and view of Artificial Intelligence:

- One view is that artificial intelligence is about designing systems that are as intelligent as humans.

- The second approach is best embodied by the concept of the Turing Test.

Introduction to AI: Turing Test

Turing put forward the idea of an 'imitation game', in which a human being and a computer would be interrogated under conditions where the interrogator would not know which was which, the communication being entirely by textual messages.

Turing argued that if the interrogator could not distinguish them by questioning, then it would be unreasonable not to call the computer intelligent.

Introduction to AI: Typical AI problems

While studying the typical range of tasks that we might expect an “intelligent entity” to perform, we need to consider both “common-place” tasks as well as expert tasks.

Examples of common-place tasks include:

- Recognizing people, objects.
- Communicating (through natural language).
- Navigating around obstacles on the streets

Introduction to AI: Typical AI problems

Expert tasks include:

- Medical diagnosis
- Mathematical problem solving
- Playing games like chess

These tasks cannot be done by all people, and can only be performed by skilled specialists.

Introduction to AI: Intelligent Behavior

This discussion brings us back to the question of what constitutes intelligent behavior. Some of these tasks and applications are:

- Perception involving image recognition and computer vision
- Reasoning
- Learning
- Natural Language Processing
- Robotics

Problem Solving: State space search

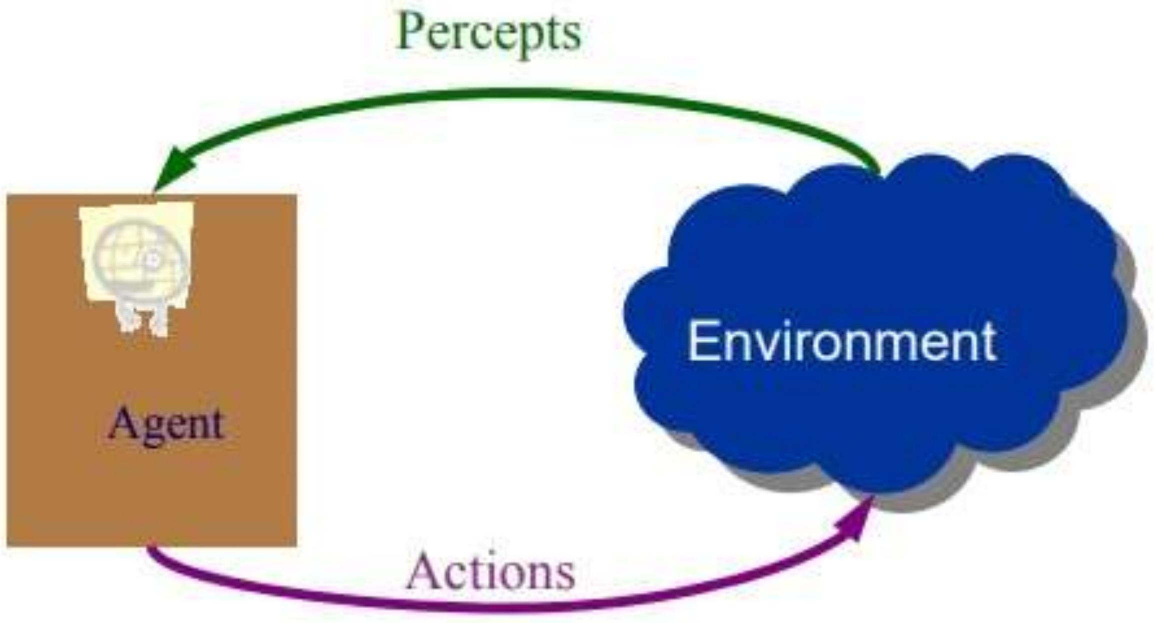
- Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states.

- A state is defined by the specification of the values of all attributes of interest in the world.

- An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator.

Problem Solving: State space search

- The initial state is where you start.
- The goal state is the partial description of the solution.



State space search: Describing a search problem

The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state g ∈ G.

A search problem is represented by a 4-tuple {S, S_0, A, G}.

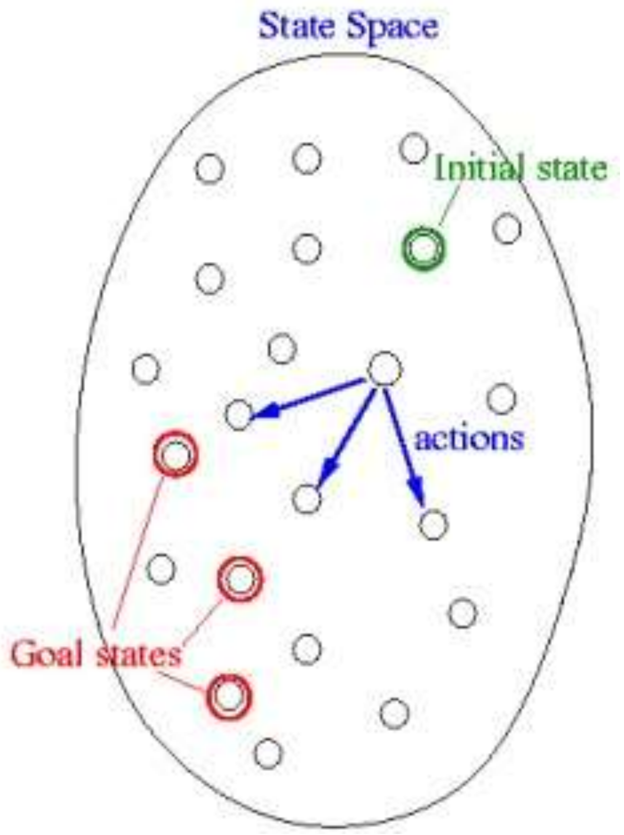
S: the full set of states

S_0 : the initial state

A:S→S is a set of operators

G is the set of final states.

Note that G ⊆S



State space search: Describing a search problem

The sequence of actions is called a solution plan. It is a path from the initial state to a goal state. A plan P is a sequence of actions.

$P = \{a_0, a_1, \ldots , a_N\}$ which leads to traversing a number of states $\{S_0, S_1, \ldots , S_{N+1} \in G\}$.

The cost of a path is a positive number.

In many cases the path cost is computed by taking the sum of the costs of each action.

State space search:
Describing a search problem

A search problem is represented using a directed graph:

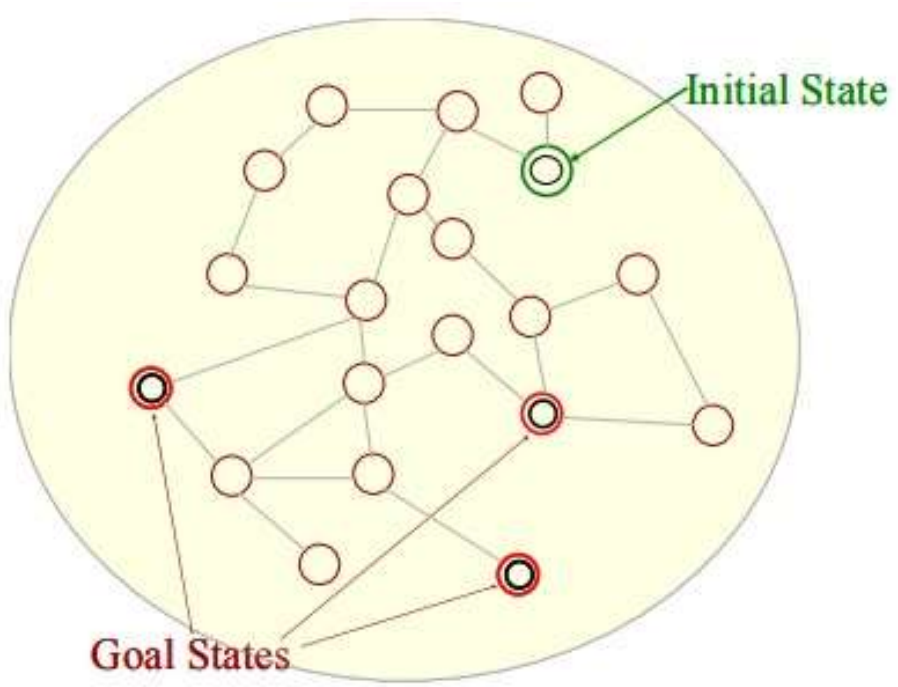
– The states are represented as nodes.
– The allowed actions are represented as arcs.

State space search: Describing a search problem

The generic searching process can be very simply described in terms of the following steps:

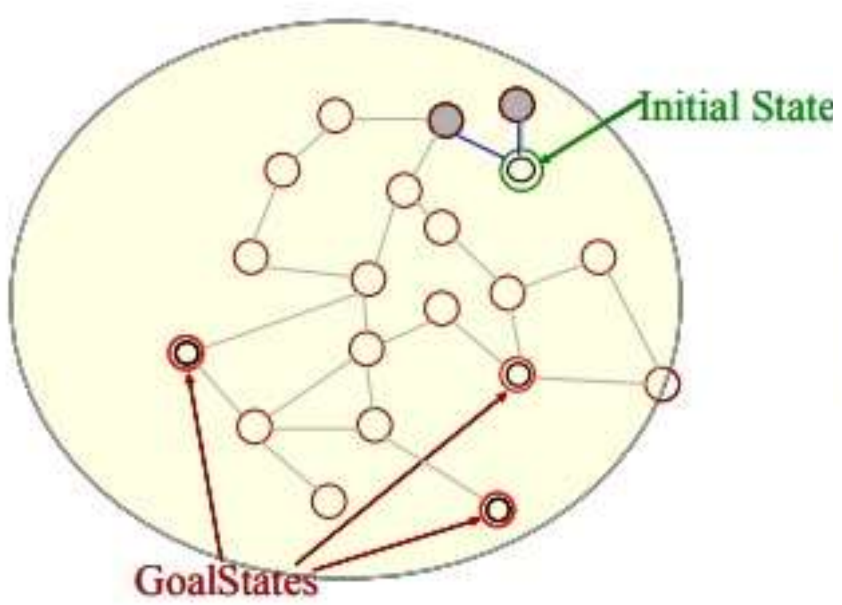
1. Check the current state.
2. Execute allowable actions to find the successor states.
3. Pick one of the new states.
4. Check if the new state is a solution state.
5. If it is not, the new state becomes the current state and the process is repeated.

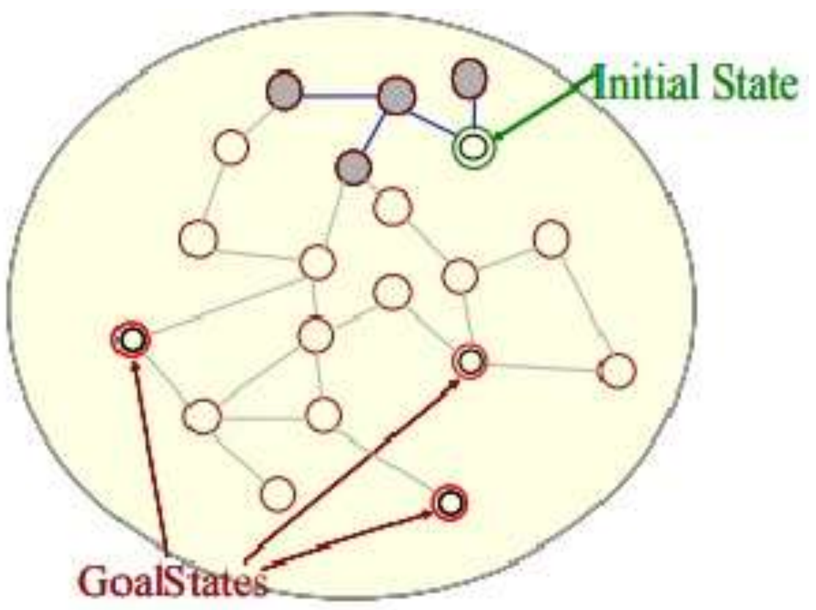
State space search: Example



- S_0 is the initial state.
- The successor states are the adjacent states in the graph.
- There are three goal states.

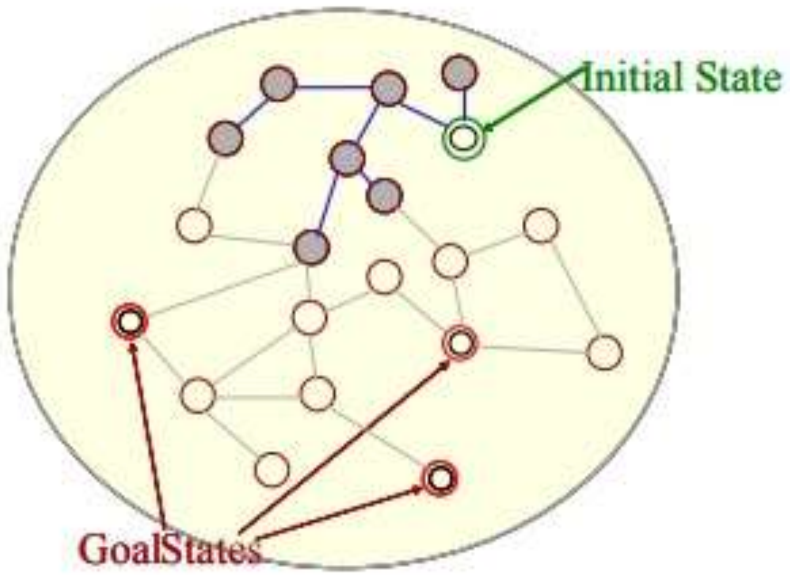
State space search: Example

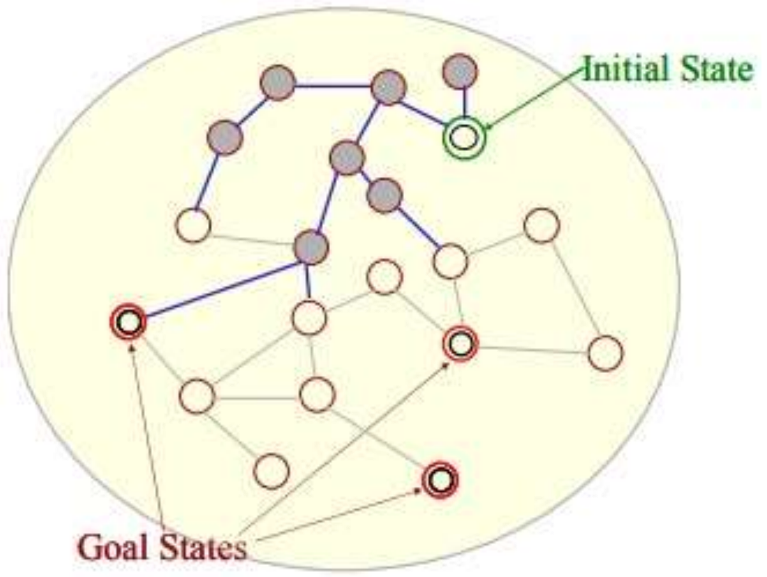




- The two successor states of the initial state are generated.
- The successors of these states are picked and their successors are generated.

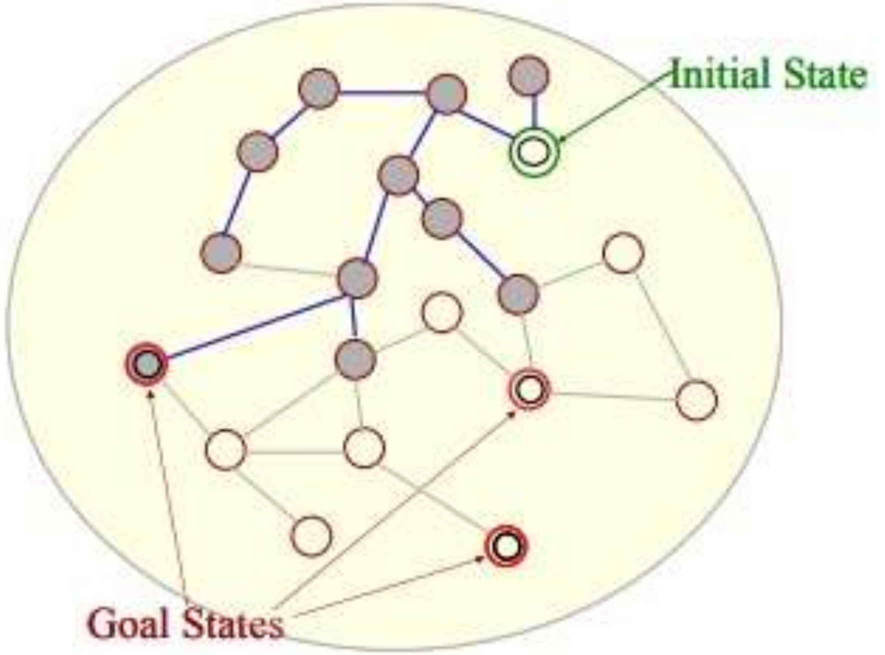
State space search: Example





- Successors of all these states are generated.
- The successors are generated.

State space search: Example



A goal state has been found.

The grey nodes define the search tree.

Usually the search tree is extended one node at a time.

The order in which the search tree is extended depends on the search strategy.

Blind Search

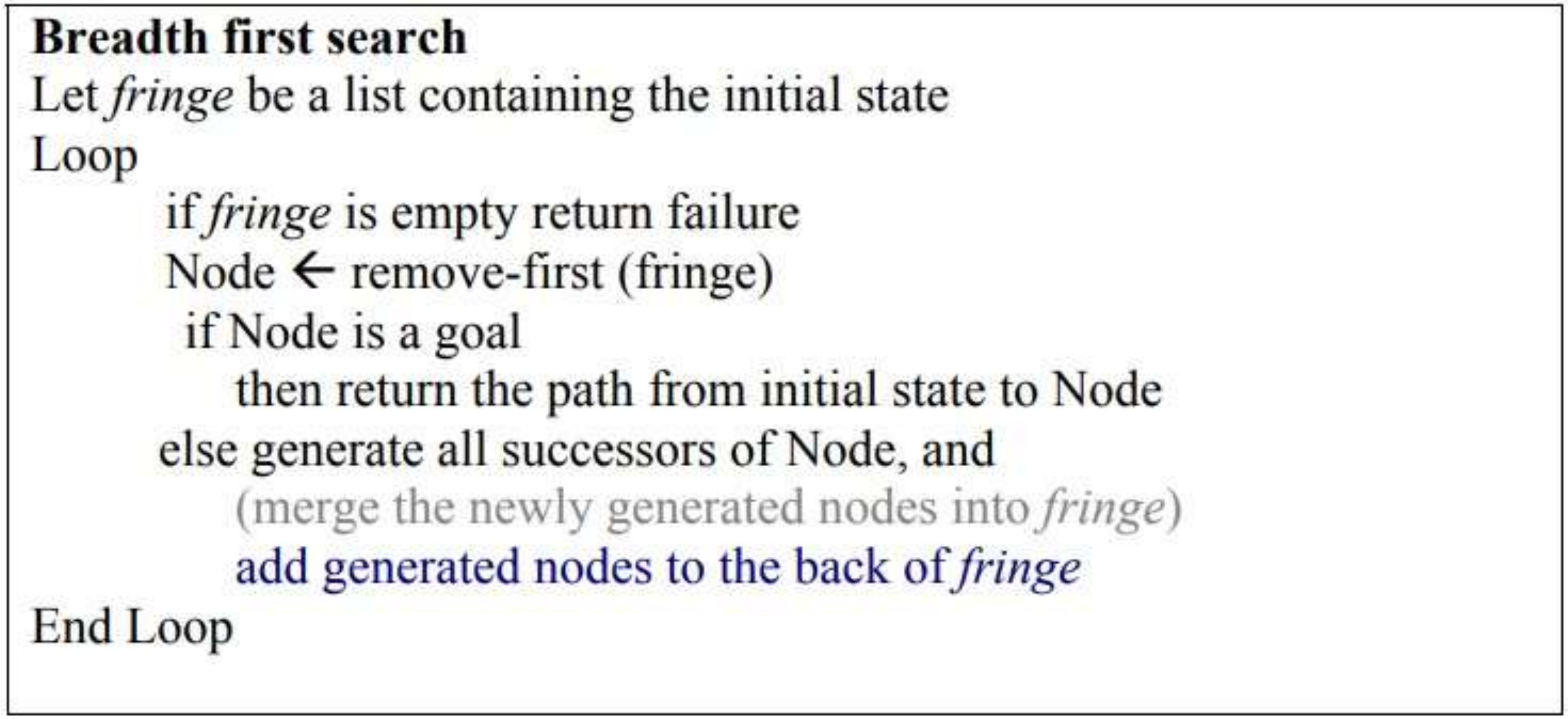
Here we will talk about blind search or uninformed search that does not use any extra information about the problem domain.

The two common methods of blind search are:

- BFS or Breadth First Search
- DFS or Depth First Search

Breadth First Search

Algorithm:



Breadth first search

```
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else generate all successors of Node, and
            (merge the newly generated nodes into fringe)
            add generated nodes to the back of fringe
End Loop
```

Breadth First Search

Note that in breadth first search the newly generated nodes are put at the back of fringe or the OPEN list.

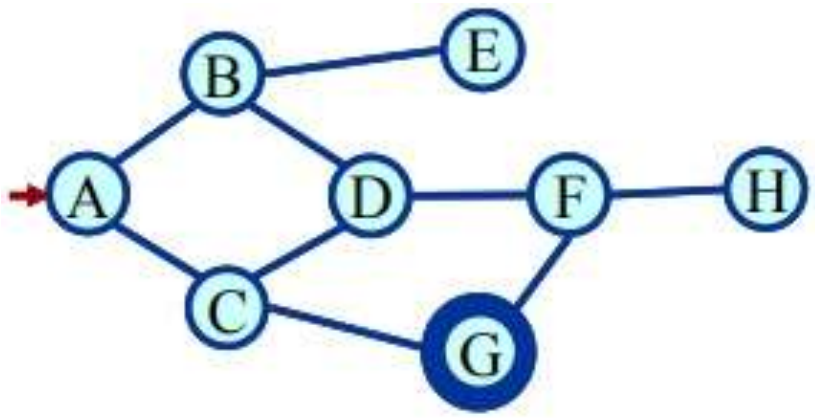
What this implies is that the nodes will be expanded in a FIFO (First In First Out) order.

The node that enters OPEN earlier will be expanded earlier.

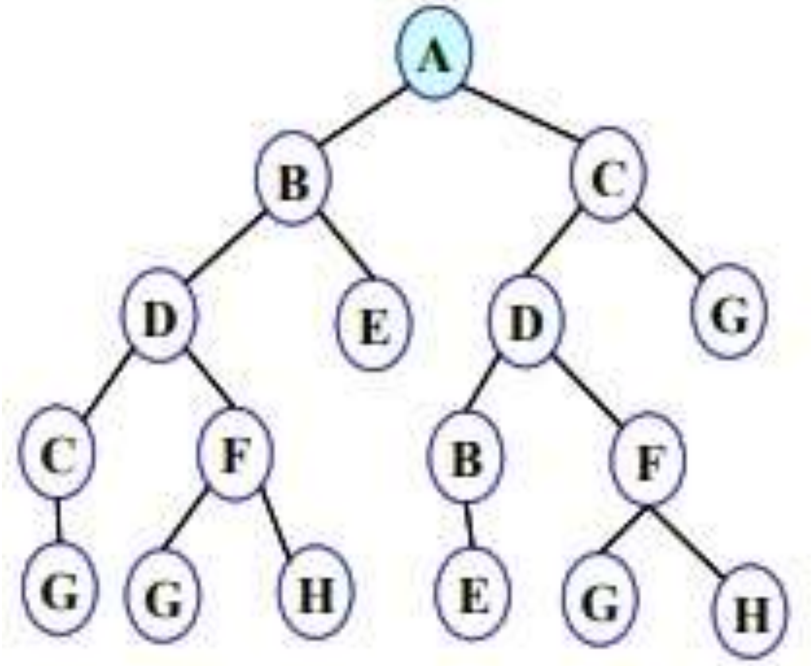
This amounts to expanding the shallowest nodes first.

Breadth First Search: Example

We will perform BFS on the graph shown below:



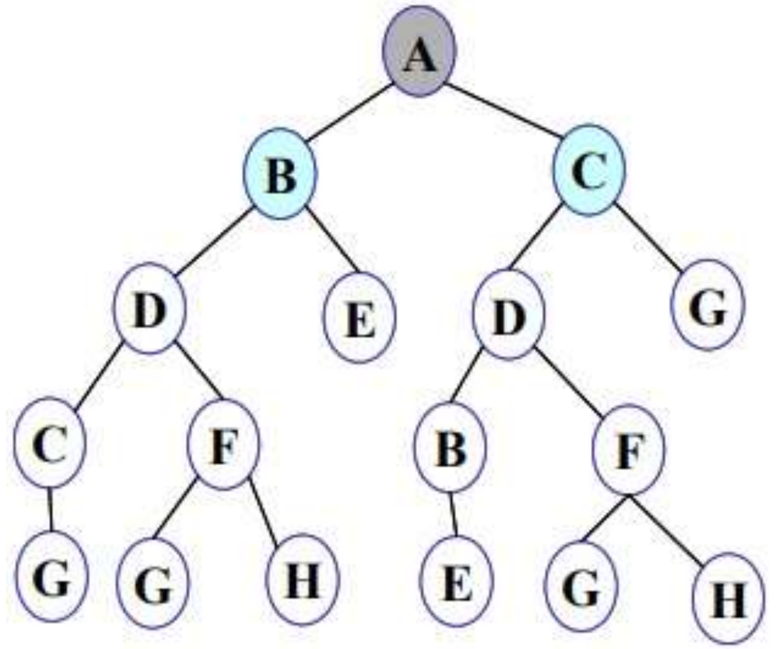
Step 1: Initially fringe contains only one node corresponding to the source state A.



Breadth First Search: Example

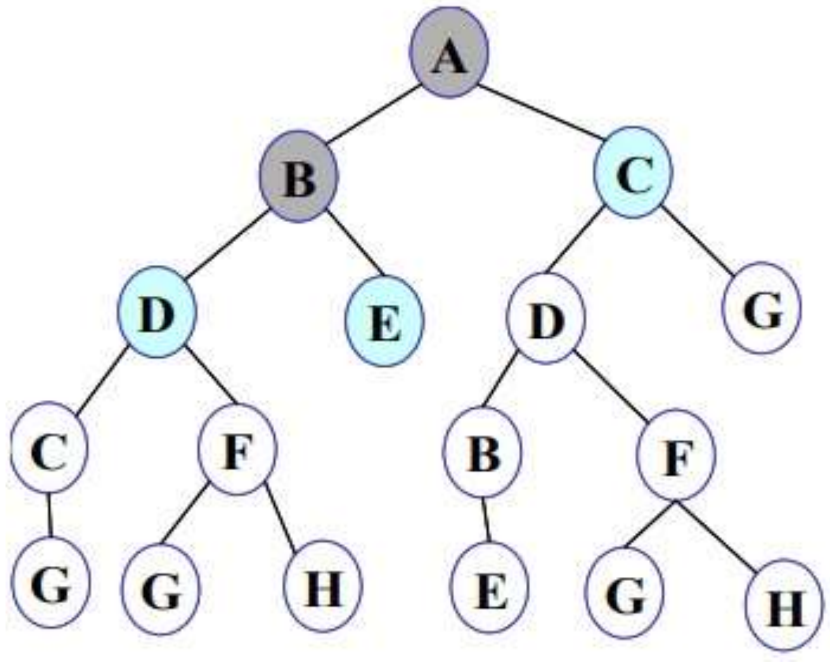
Step 2:

A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.



Breadth First Search: Example

Step 3:
Node B is removed from fringe and is expanded.
Its children D, E are generated and put at the back of fringe.

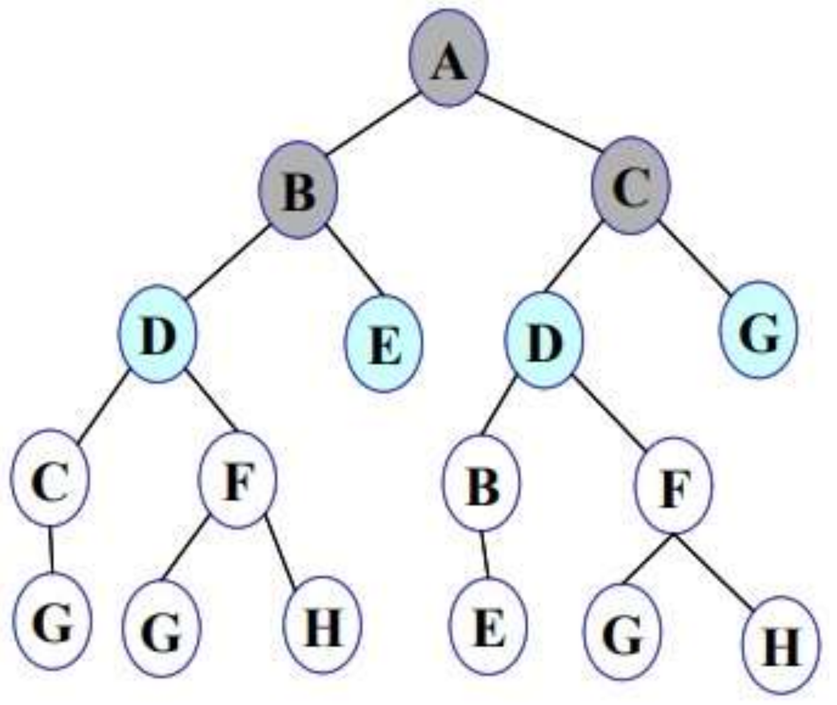


Breadth First Search: Example

Step 4:

Node C is removed from fringe and is expanded.

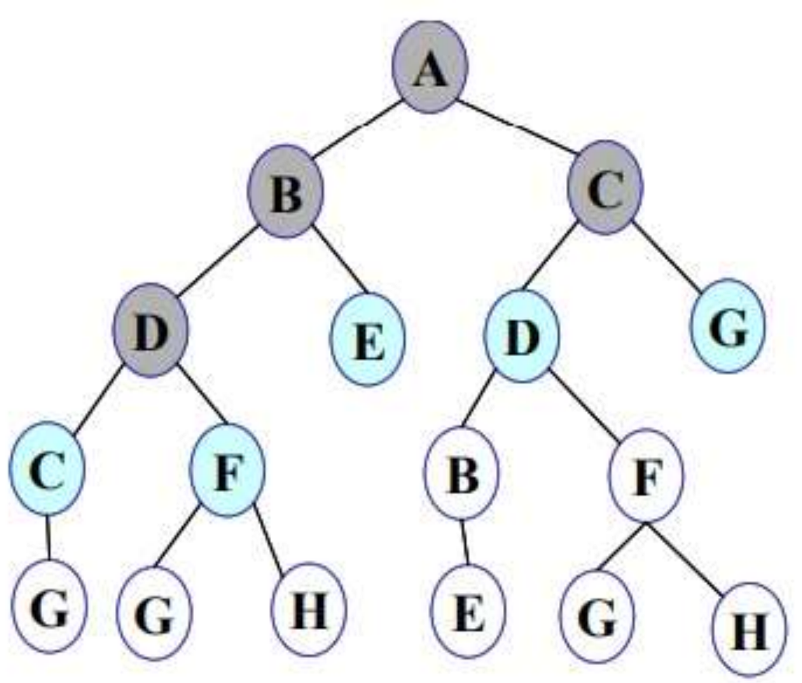
Its children D and G are added to the back of fringe.



Breadth First Search: Example

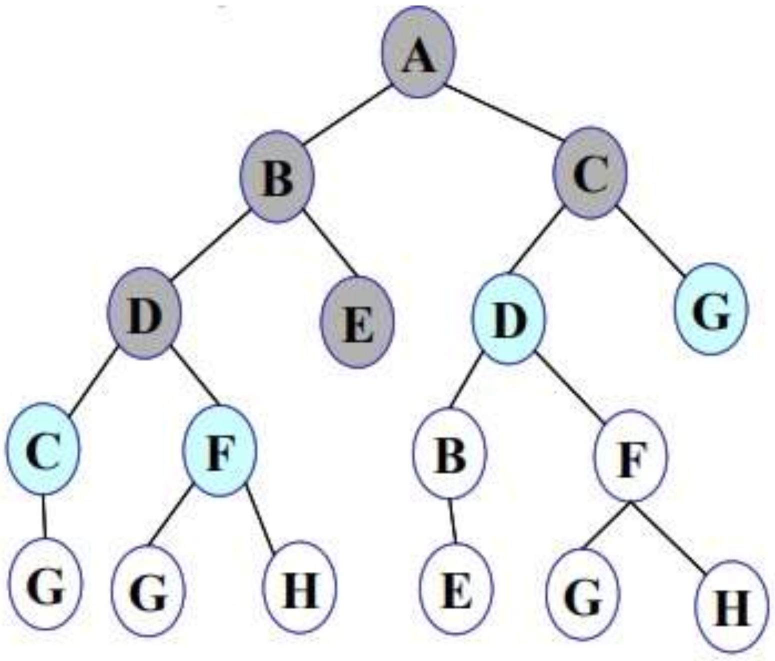
Step 5:

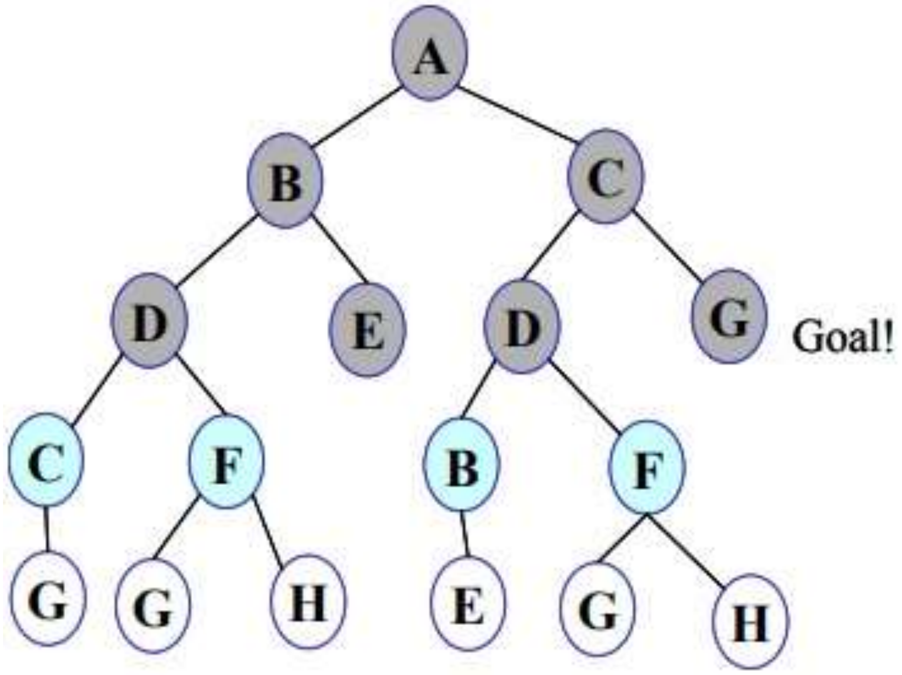
Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.



Breadth First Search: Example

Step 6: Node E is removed from fringe. It has no children.





Step 7: D is expanded, B and F are put in OPEN.

Breadth First Search: Example

Step 8:

G is selected for expansion.

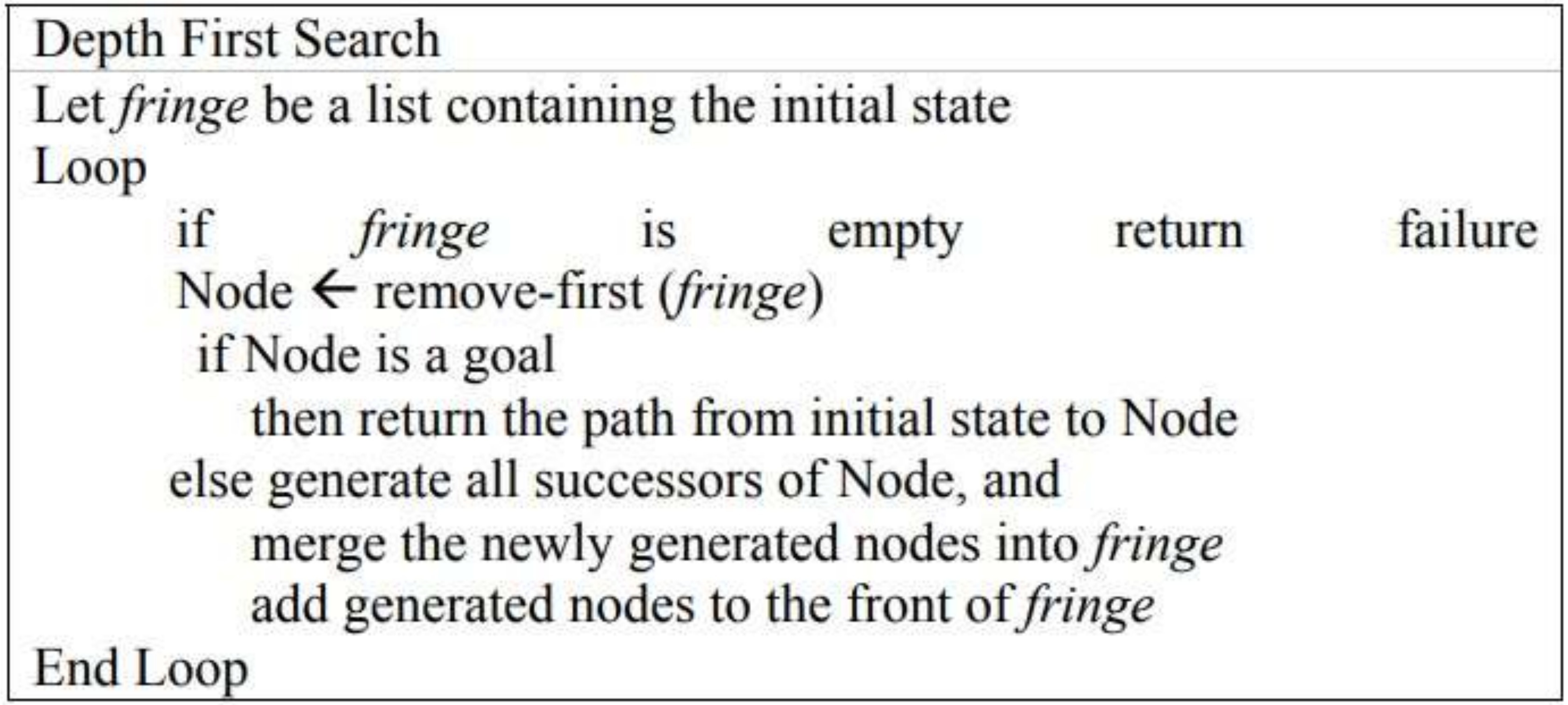
It is found to be a goal node.

So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G.

The algorithm terminates.

Depth First Search

Algorithm:



Depth First Search

```
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else generate all successors of Node, and
            merge the newly generated nodes into fringe
            add generated nodes to the front of fringe
End Loop
```

Depth First Search

The depth first search algorithm puts newly generated nodes in the front of OPEN.

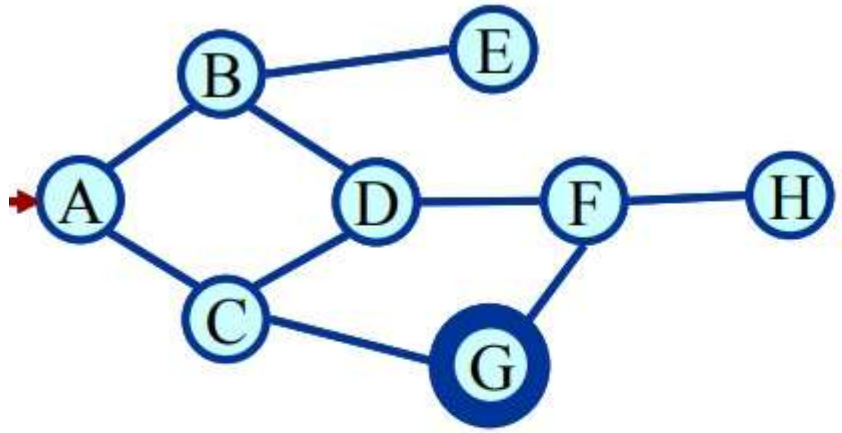
This results in expanding the deepest node first.

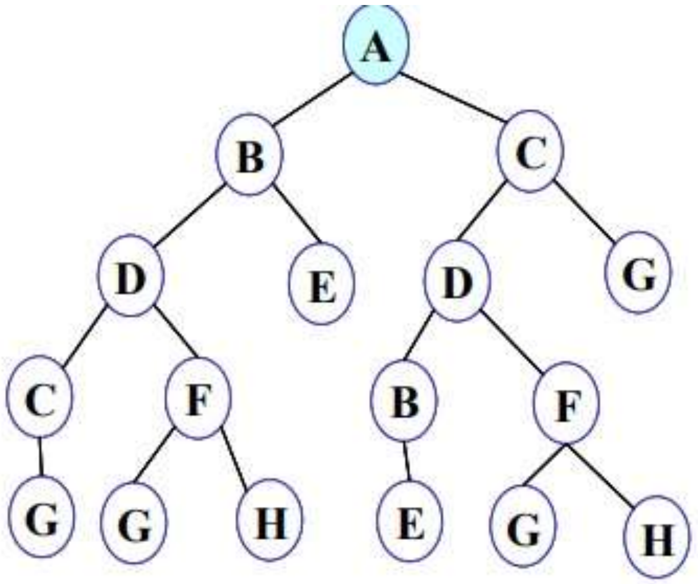
Thus the nodes in OPEN follow a LIFO order (Last In First Out).

OPEN is thus implemented using a stack data structure.

Depth First Search: Example

Consider the graph shown below:





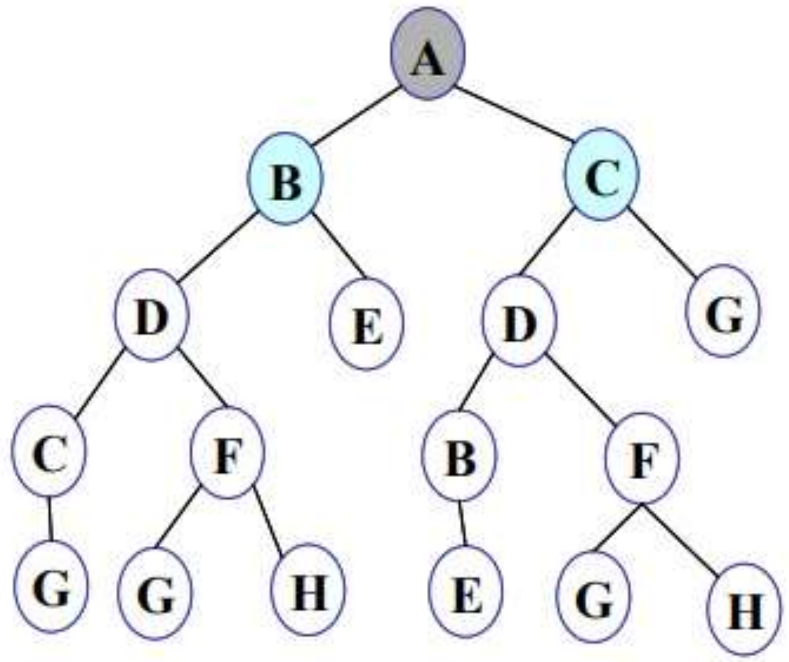
Step 1:

Initially fringe contains only the node A.

FRINGE: A

Depth First Search: Example

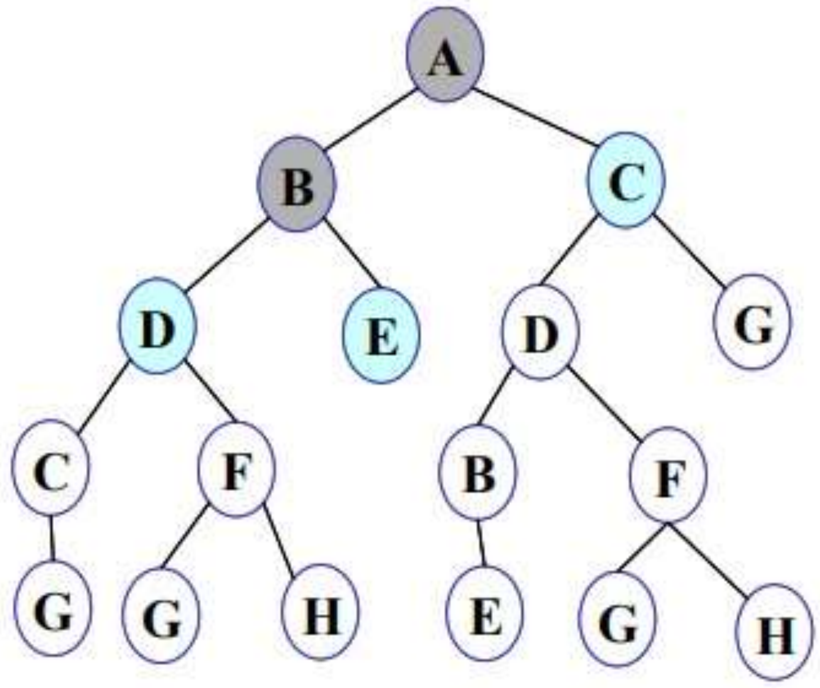
Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



FRINGE: B C

Depth First Search: Example

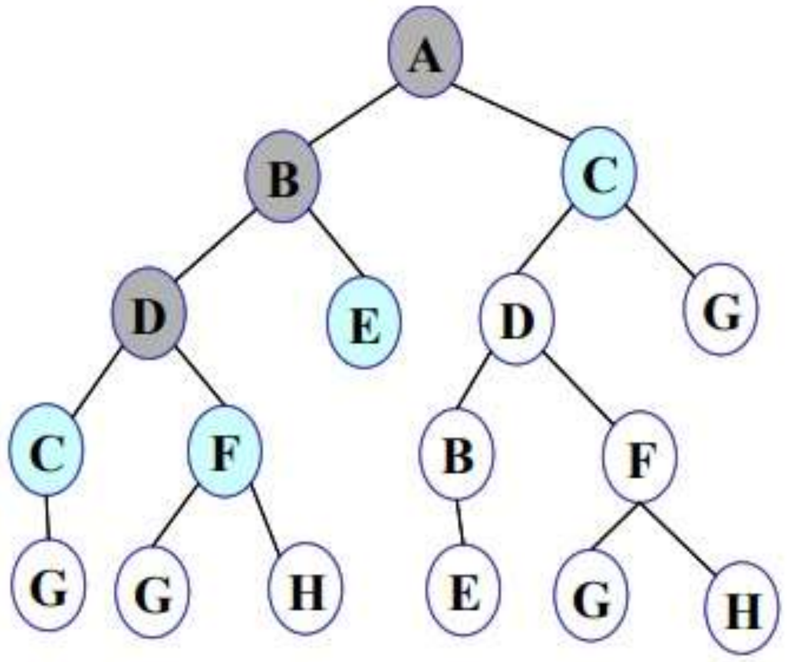
Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



FRINGE: D E C

Depth First Search: Example

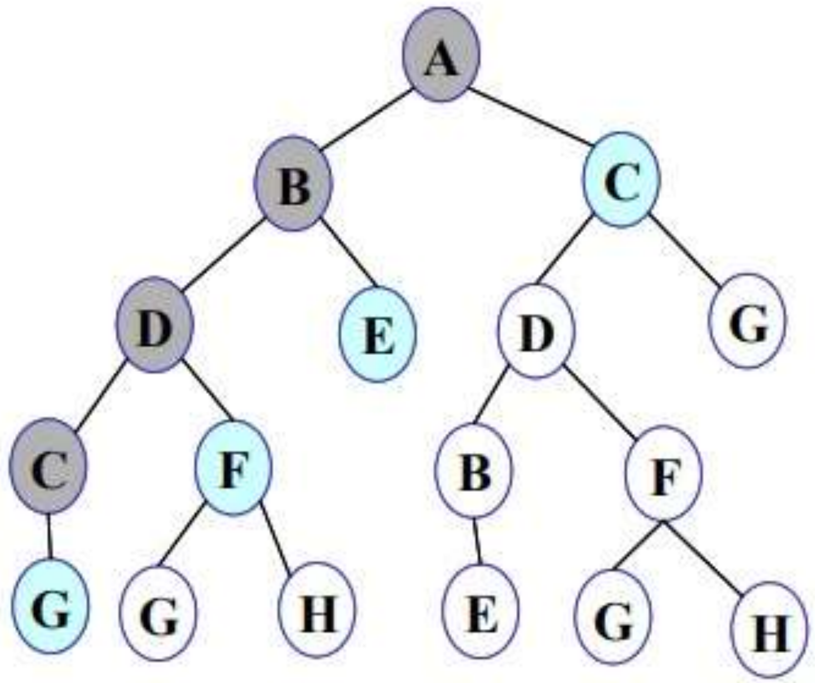
Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



FRINGE: C F E C

Depth First Search: Example

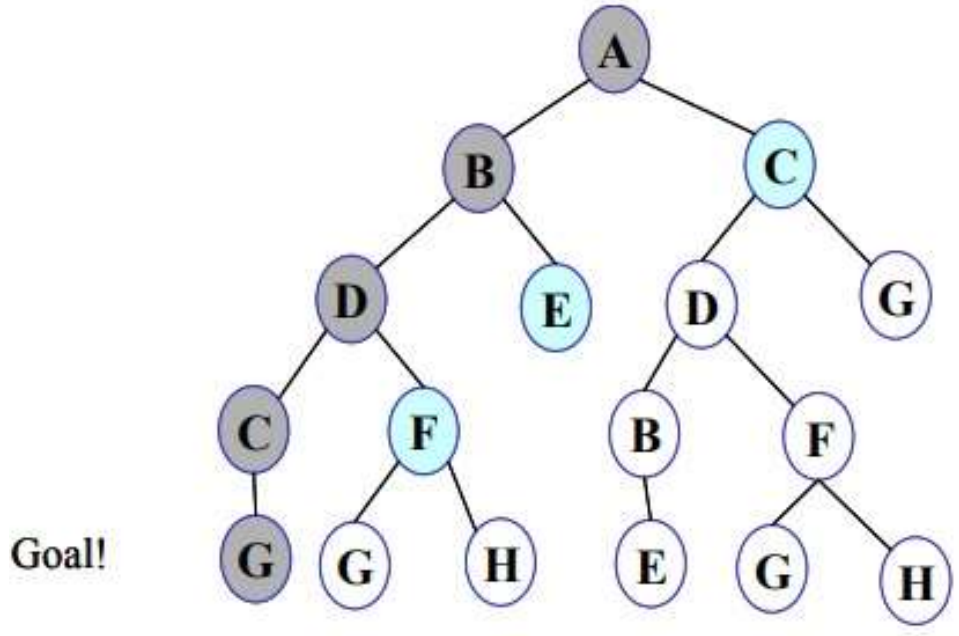
Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



FRINGE: G F E C

Depth First Search: Example

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.



FRINGE: G F E C

Informed Search

We have seen that uninformed search methods that systematically explore the state space and find the goal.

They are inefficient in most cases.

Informed search methods use problem specific knowledge, and may be more efficient.

At the heart of such algorithms there is the concept of a heuristic function.

Heuristic function

Heuristic means “rule of thumb”.

To quote Judea Pearl, “Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal”.

In heuristic search or informed search, heuristics are used to identify the most promising search path.

Heuristic function: Example

A heuristic function at a node n is an estimate of the optimum cost from the current node to a goal.

It is denoted by *h(n)*.

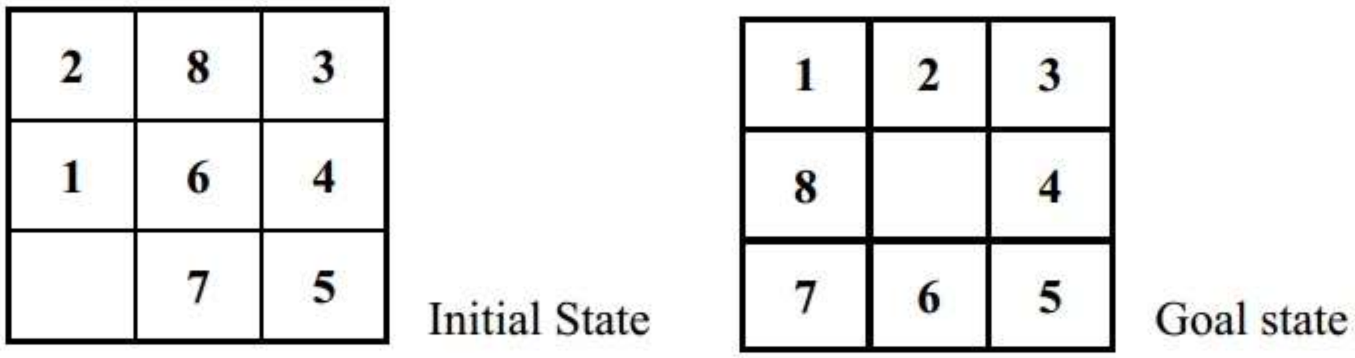
h(n) = estimated cost of the cheapest path from node n to a goal node.

Example 1: We want a path from Kolkata to Guwahati Heuristic for Guwahati may be straight-line distance between Kolkata and Guwahati.

h(Kolkata) = euclideanDistance(Kolkata, Guwahati)

Heuristic function: Example

Example 2: 8-puzzle: Misplaced Tiles Heuristics is the number of tiles out of place.



The first picture shows the current state *n*, and the second picture the goal state.

h(n) = 5 because the tiles 2, 8, 1, 6 and 7 are out of place.

Heuristic function: Example

Manhattan Distance Heuristic:

Another heuristic for 8-puzzle is the Manhattan distance heuristic.

This heuristic sums the distance that the tiles are out of place. The distance of a tile is measured by the sum of the differences in the x-positions and the y-positions.

For the above example, using the Manhattan distance heuristic

$h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6$

Hill climbing search

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem.

It terminates when it reaches a peak value where no neighbor has a higher value.

It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.

Hill climbing search

Features of Hill Climbing Algorithm:

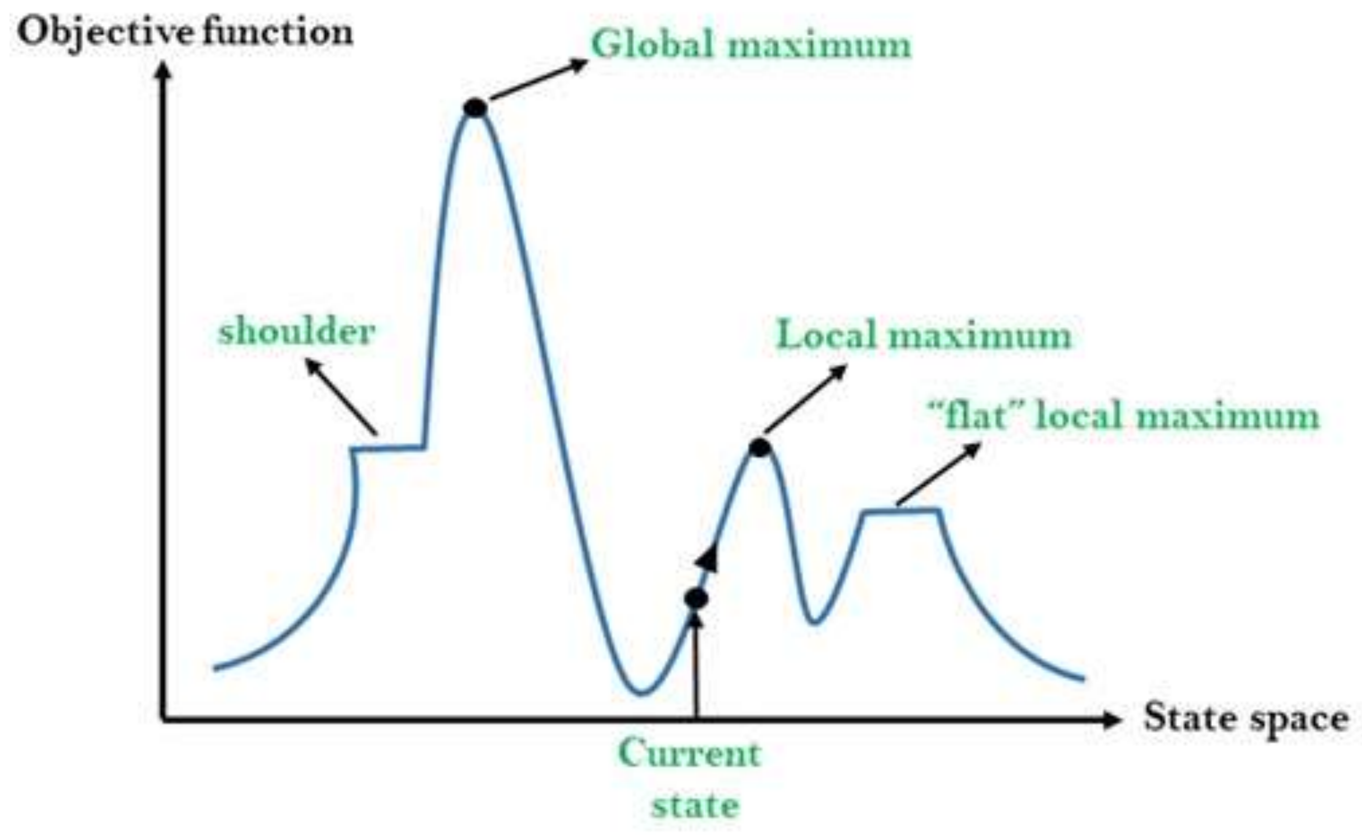
- Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.

- Hill-climbing algorithm search moves in the direction which optimizes the cost.

- It does not backtrack the search space, as it does not remember the previous states.

Hill climbing search

State-space Diagram for Hill Climbing:



Hill climbing search

If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum.

If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Hill climbing search

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Hill climbing search: Algorithm

Step 1: Evaluate the initial state, if it is goal state then return success and Stop.

Step 2: Loop Until a solution is found or there is no new operator left to apply.

Step 3: Select and apply an operator to the current state.

Step 4: Check new state:

If it is goal state, then return success and quit.

Else if it is better than the current state then assign new state as a current state.

Else if not better than the current state, then return to step2.

Step 5: Exit.

Best first search

Uniform Cost Search is a special case of the best first search algorithm.

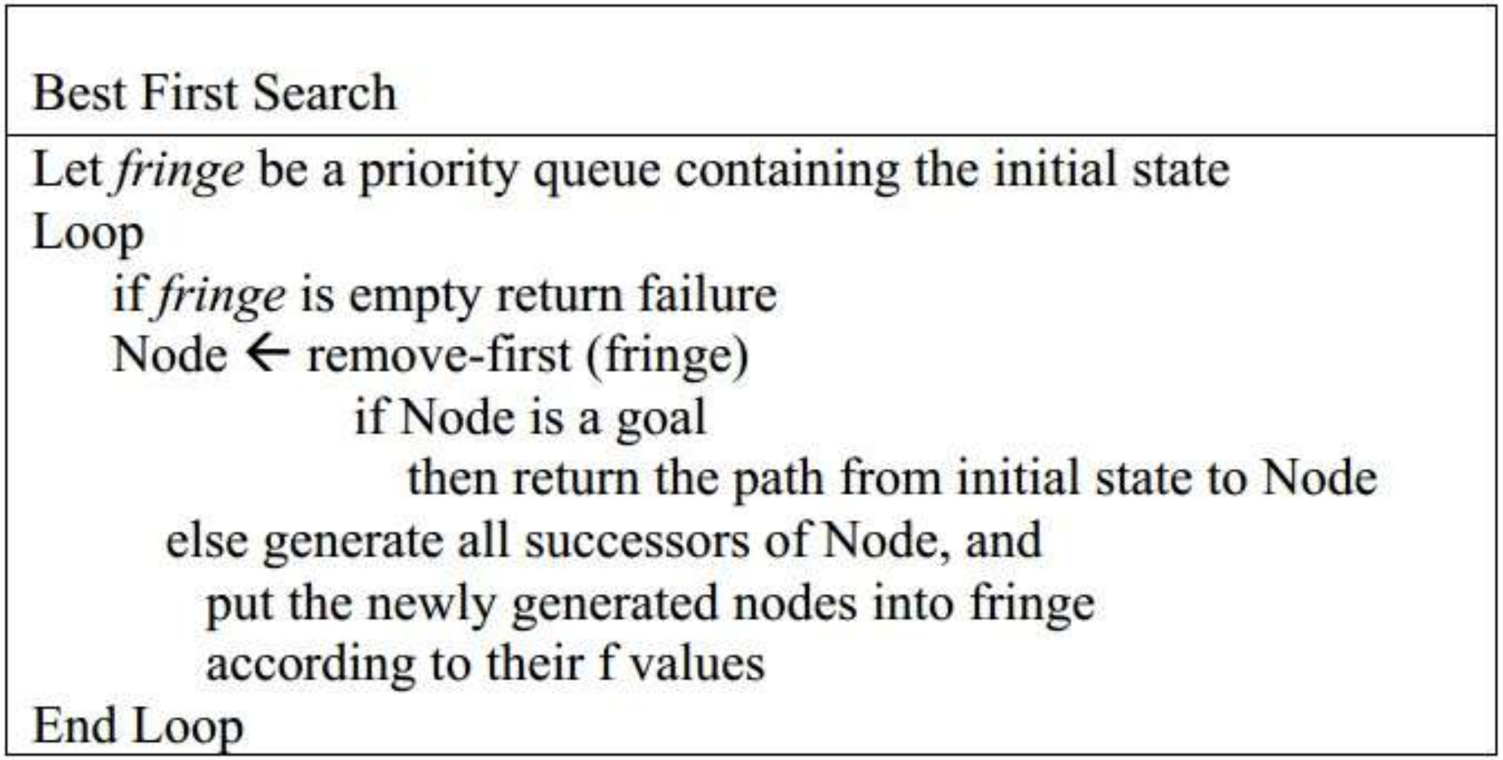
The algorithm maintains a priority queue of nodes to be explored.

A cost function f(n) is applied to each node. The nodes are put in OPEN in the order of their f values.

Nodes with smaller f(n) values are expanded earlier.

Best first search

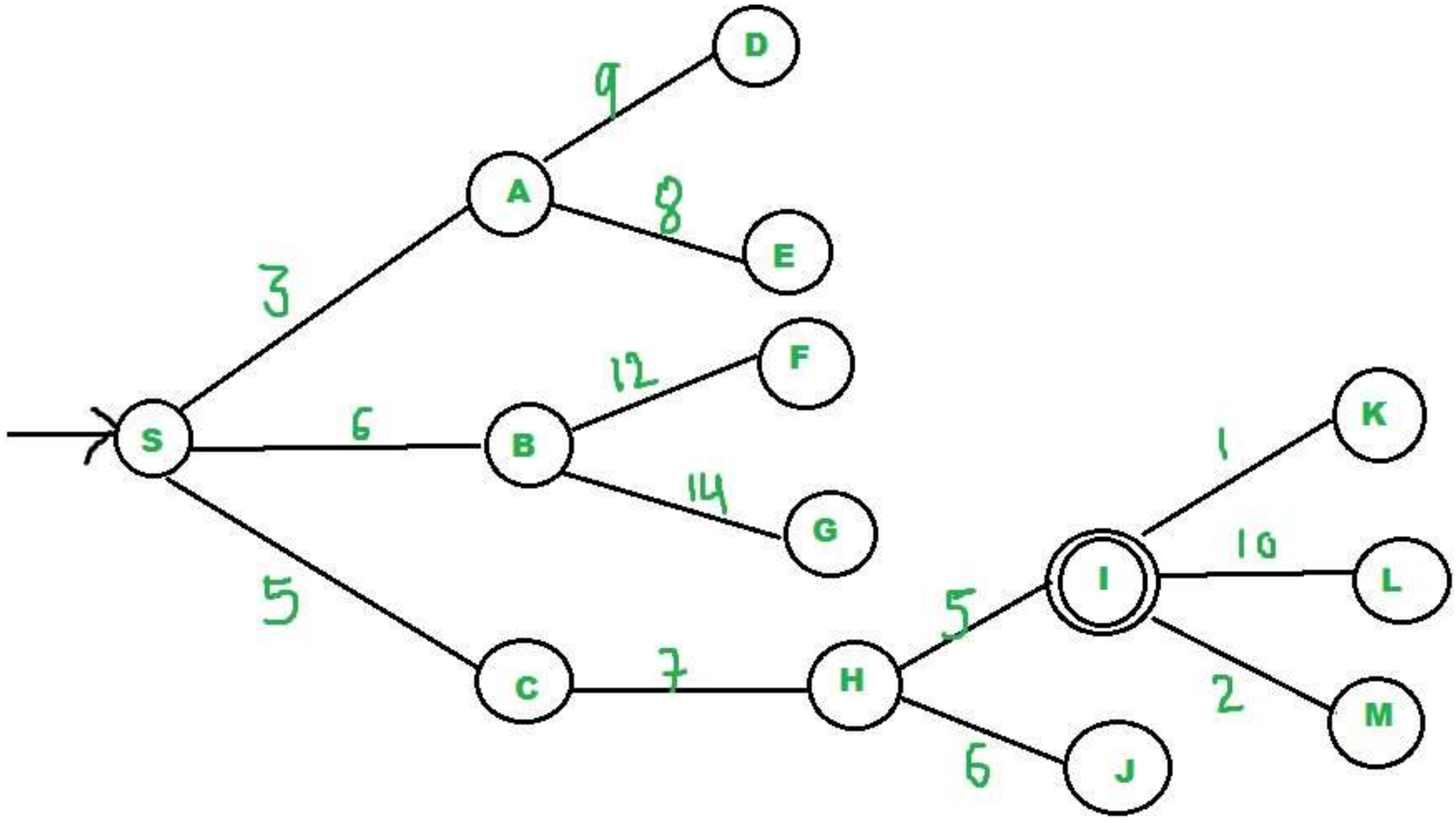
Algorithm:



```
Best First Search
Let fringe be a priority queue containing the initial state
Loop
     if fringe is empty return failure
     Node ← remove-first (fringe)
                    if Node is a goal
                       then return the path from initial state to Node
        else generate all successors of Node, and
          put the newly generated nodes into fringe
          according to their f values
End Loop
```

Best first search

Let us understand Best first search using below example:



Best first search

We start from source "S" and search for goal "I" using given costs and Best First search.

fringe initially contains S. We remove S and process unvisited neighbors of S to fringe.

fringe now contains {A, C, B} (C is put before B because C has lesser cost).

We remove A from *fringe* and process its unvisited neighbors to fringe. *fringe* now contains {C, B, E, D}.

Best first search

We remove C from *fringe* and process unvisited neighbors of C to *fringe*. *fringe* now contains {B, H, E, D}.

We remove B from *fringe* and process unvisited neighbors of B to *fringe*. *fringe* now contains {H, E, D, F, G}.

We remove H from *fringe*. Since our goal "I" is a neighbor of H, we return.

A* search

A* is a best first search algorithm with

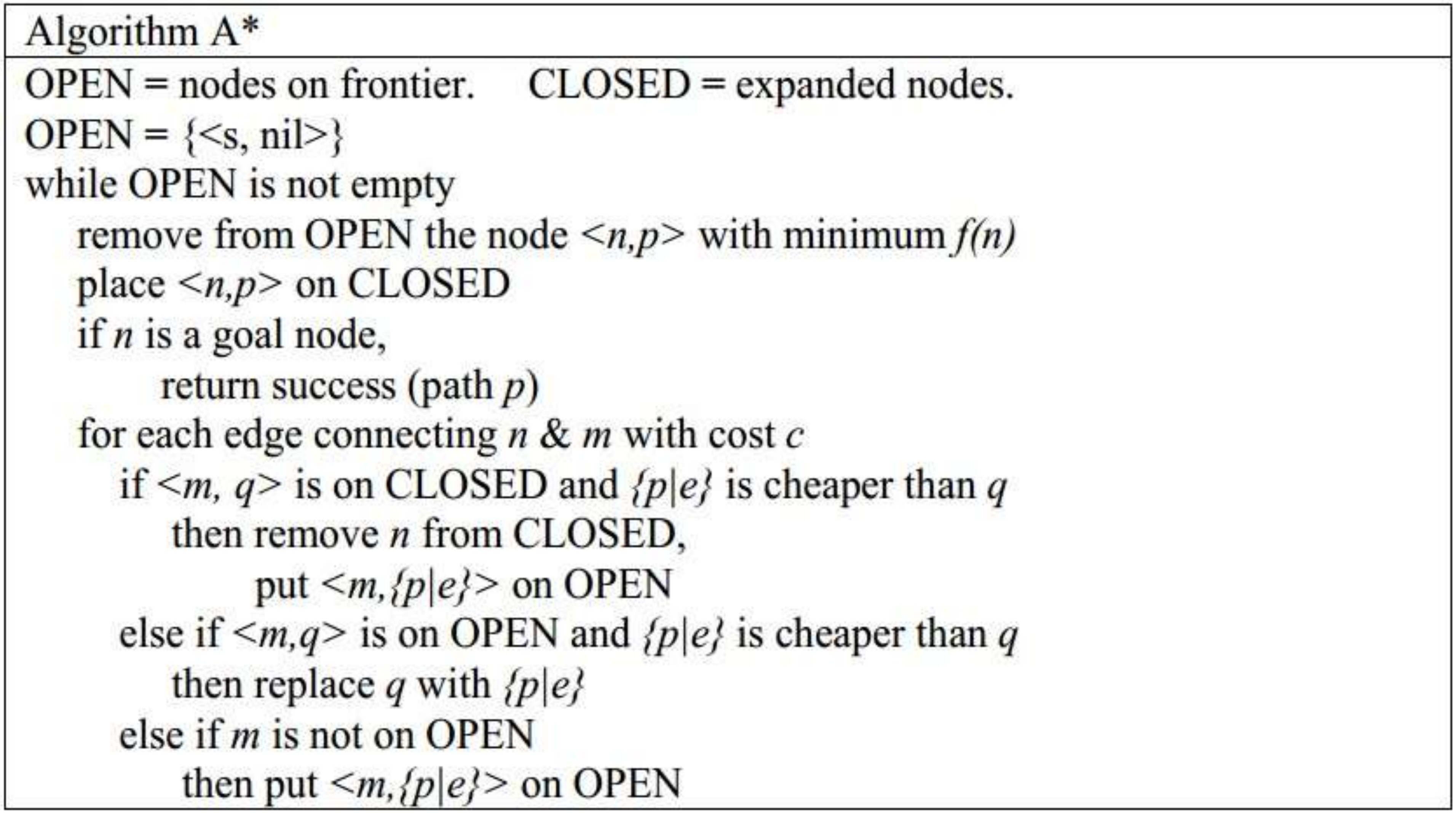
f(n) = g(n) + h(n); where

g(n) = sum of edge costs from start to n

h(n) = estimate of lowest cost path from n to goal

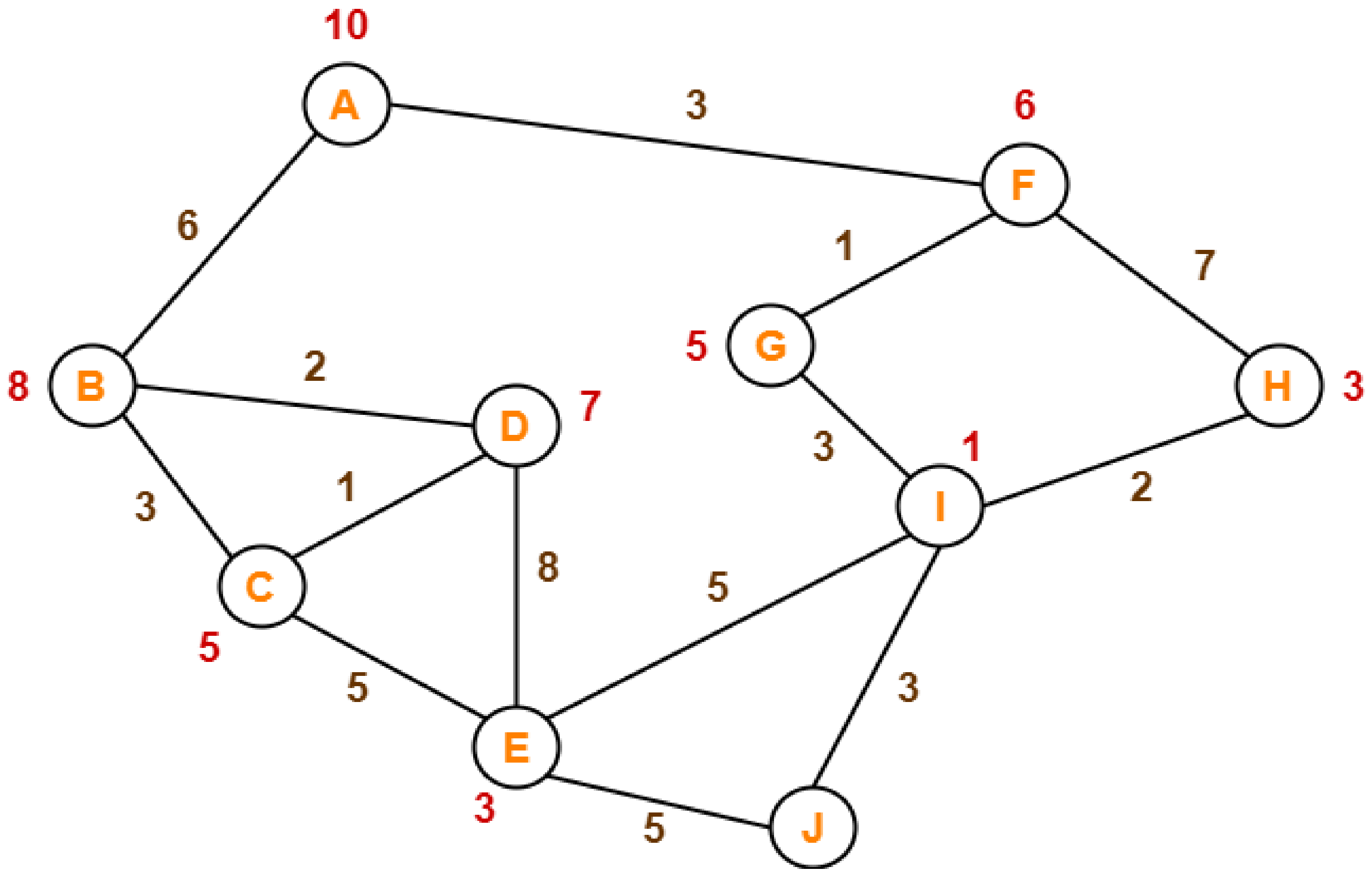
f(n) = actual distance so far + estimated distance remaining

A* search: Algorithm



```
Algorithm A*
OPEN = nodes on frontier.    CLOSED = expanded nodes.
OPEN = {<s, nil>}
while OPEN is not empty
    remove from OPEN the node <n,p> with minimum f(n)
    place <n,p> on CLOSED
    if n is a goal node,
          return success (path p)
    for each edge connecting n & m with cost c
       if <m, q> is on CLOSED and {p|e} is cheaper than q
           then remove n from CLOSED,
                 put <m,{p|e}> on OPEN
       else if <m,q> is on OPEN and {p|e} is cheaper than q
           then replace q with {p|e}
       else if m is not on OPEN
           then put <m,{p|e}> on OPEN
```

A* search: Example



The numbers on edges represent the distance between the nodes and numbers on nodes represent heuristic value.

Start Node – A, Goal Node – J

A* search: Example

Step-01:

We start with node A.

Node B and Node F can be reached from node A.

A* Algorithm calculates f(B) and f(F).

f(B) = 6 + 8 = 14

f(F) = 3 + 6 = 9

Since f(F) < f(B), so it decides to go to node F.

Path- A → F

A* search: Example

Step-02:

Node G and Node H can be reached from node F.

A* Algorithm calculates f(G) and f(H).

f(G) = (3+1) + 5 = 9

f(H) = (3+7) + 3 = 13

Since f(G) < f(H), so it decides to go to node G.

Path- A → F → G

A* search: Example

<u>Step-03:</u>

Node I can be reached from node G.

A* Algorithm calculates f(I).

f(I) = (3+1+3) + 1 = 8

It decides to go to node I.

Path- A → F → G → I

A* search: Example

Step-04:

Node E, Node H and Node J can be reached from node I.

A* Algorithm calculates f(E), f(H) and f(J).

f(E) = (3+1+3+5) + 3 = 15

f(H) = (3+1+3+2) + 3 = 12

f(J) = (3+1+3+3) + 0 = 10

Since f(J) is least, so it decides to go to node J.

Path- A → F → G → I → J Goal reached.

Constraint Satisfaction Problems

Constraint satisfaction problems or CSPs are mathematical problems where one must find states or objects that satisfy a number of constraints or criteria.

A constraint is a restriction of the feasible solutions in an optimization problem.

Such problems are usually solved via search, in particular a form of backtracking or local search.

Constraint satisfaction originated in the field of artificial intelligence in the 1970s

Game Tree

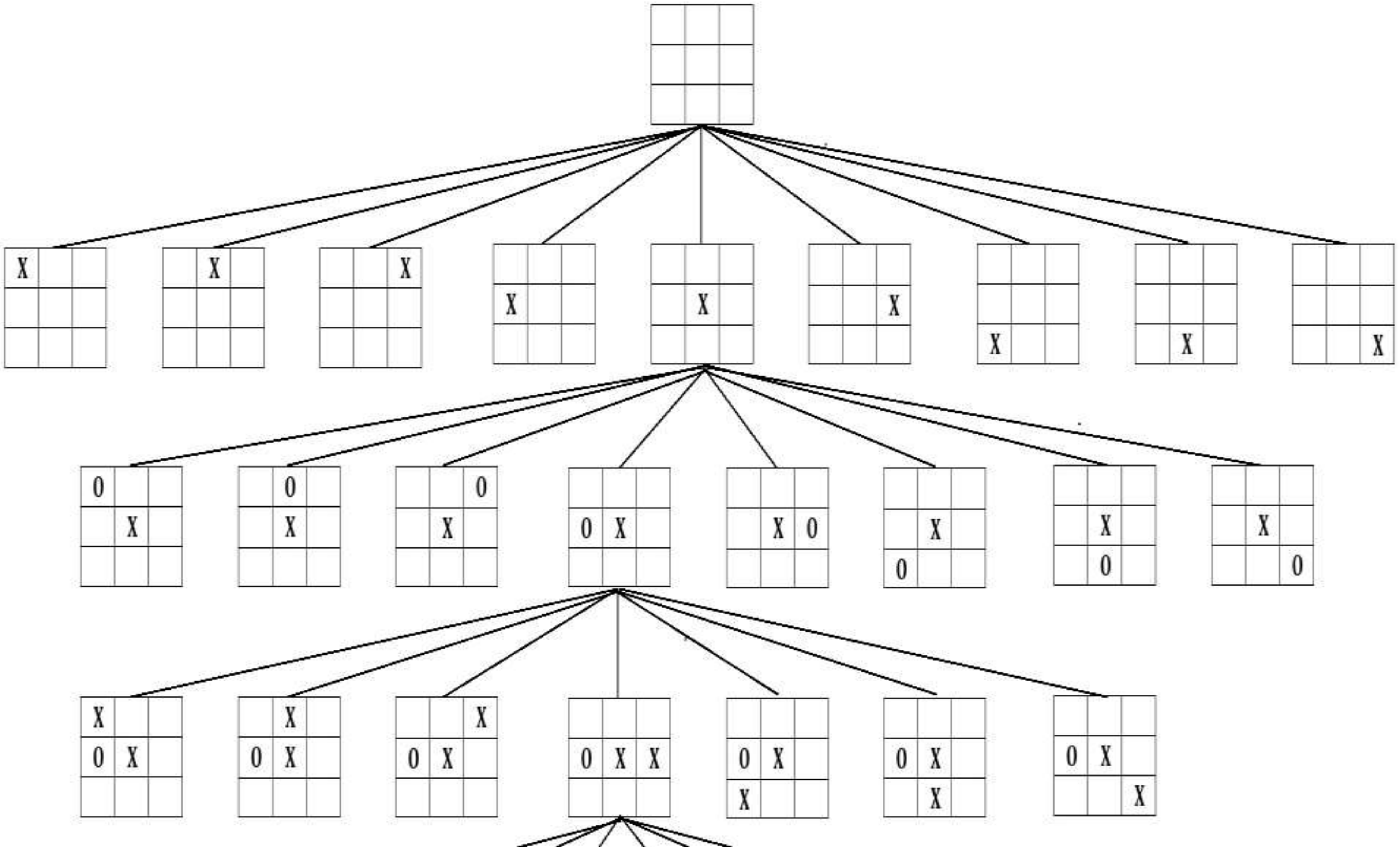
In game theory, a game tree is a directed graph whose nodes are positions in a game and whose edges are moves.

The complete game tree for a game is the game tree starting at the initial position and containing all possible moves from each position.

Game trees are important in artificial intelligence because one way to pick the best move in a game is to search the game tree using any of numerous tree search algorithms.

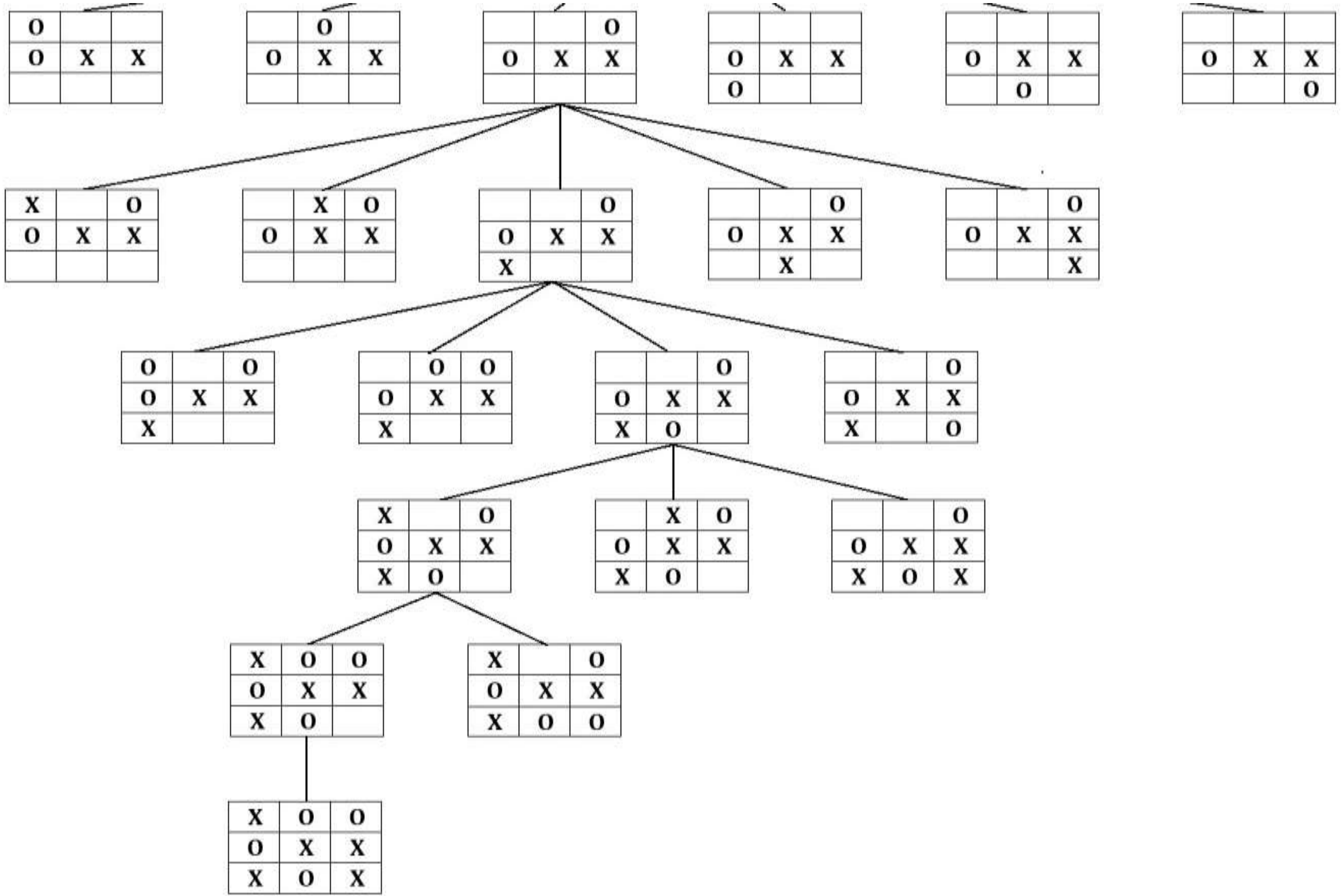
Game Tree

Game Tree for tic tac toe problem (continued in next slide)



Game Tree

Game Tree for tic tac toe problem (continued)



Evaluation Function

An evaluation function, also known as a heuristic evaluation function or static evaluation function

It is a function used by game-playing computer programs to estimate the value or goodness of a position (usually at a leaf or terminal node) in a game tree.

A tree of such evaluations usually returns a particular node and its evaluation as a result of alternately selecting the most favorable move for the side on move at each ply of the game tree.

Evaluation Function

The value is presumed to represent the relative probability of winning if the game tree were expanded from that node to the end of the game.

The function looks only at the current position (i.e. what spaces the pieces are on and their relationship to each other).

It does not take into account the history of the position or explore possible moves forward of the node (therefore static).

Mini-Max search

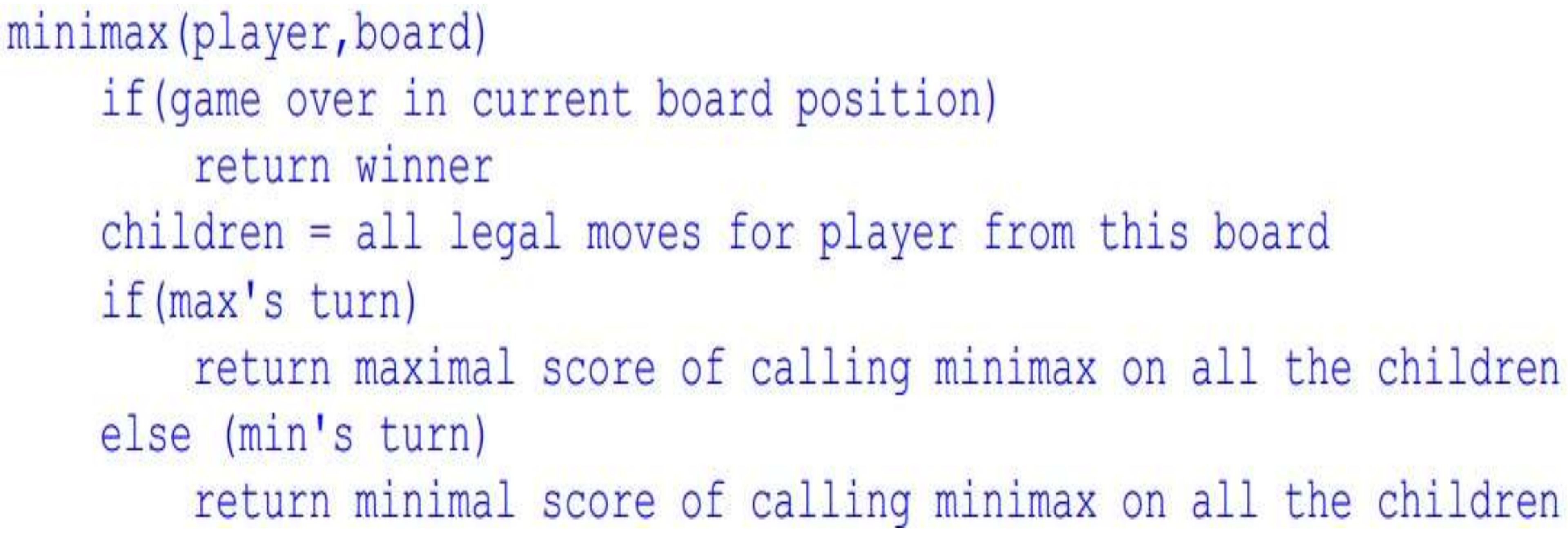
One algorithm for computing the best move in a game is the mini-max algorithm.

We will assume that the opponent is rational; that is, the opponent can compute moves just as well as we can.

The opponent will always choose the optimal move with the assumption that we, too, will play perfectly.

Mini-Max search

Algorithm:



```
minimax(player,board)
    if(game over in current board position)
        return winner
    children = all legal moves for player from this board
    if(max's turn)
        return maximal score of calling minimax on all the children
    else (min's turn)
        return minimal score of calling minimax on all the children
```

Mini-Max search

Algorithm Explanation:

If the game is over in the given position, then there is nothing to compute.

Mini-max will simply return the score of the board.

Otherwise, mini-max will go through each possible child, and (by recursively calling itself) evaluate each possible move.

Mini-Max search

Algorithm Explanation (continued):

Then, the best possible move will be chosen.

where ‘best’ is the move leading to the board:

- with the most positive score for player 1, and
- with the most negative score for player 2.

Alpha-beta pruning

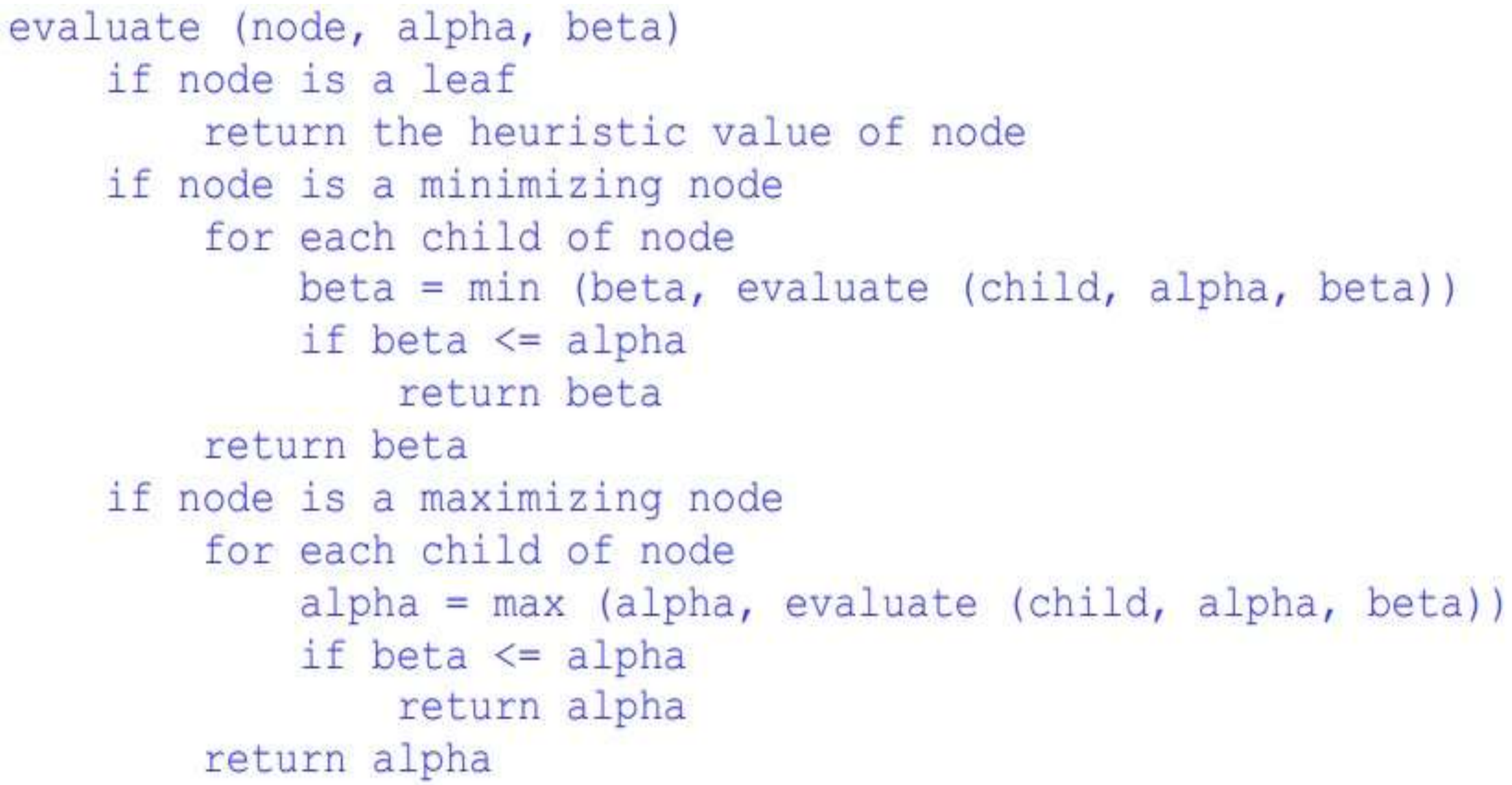
ALPHA-BETA pruning is a method that reduces the number of nodes explored in Mini-max strategy.

It reduces the time required for the search and it must be restricted so that no time is to be wasted searching moves that are obviously bad for the current player.

The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.

Alpha-beta pruning

Pseudo code:



```
evaluate (node, alpha, beta)
    if node is a leaf
        return the heuristic value of node
    if node is a minimizing node
        for each child of node
            beta = min (beta, evaluate (child, alpha, beta))
            if beta <= alpha
                return beta
        return beta
    if node is a maximizing node
        for each child of node
            alpha = max (alpha, evaluate (child, alpha, beta))
            if beta <= alpha
                return alpha
        return alpha
```

Alpha-beta pruning

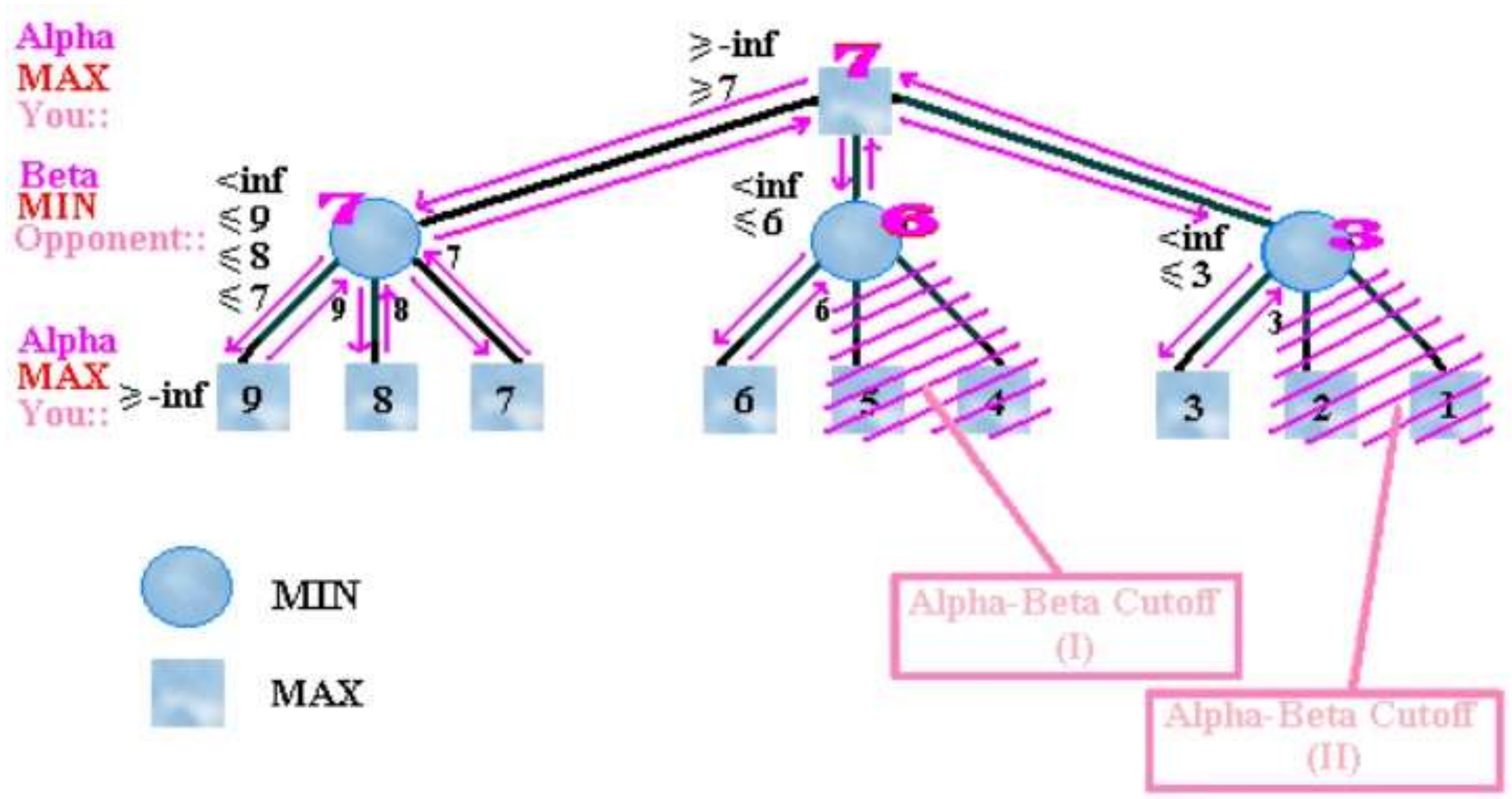
The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively.

Initially alpha is negative infinity and beta is positive infinity.

When beta becomes less than alpha, it means that the current position cannot be the result of best play by both players and hence need not be explored further.

Alpha-beta pruning

Here is an example of Alpha-Beta search:



Games of chance

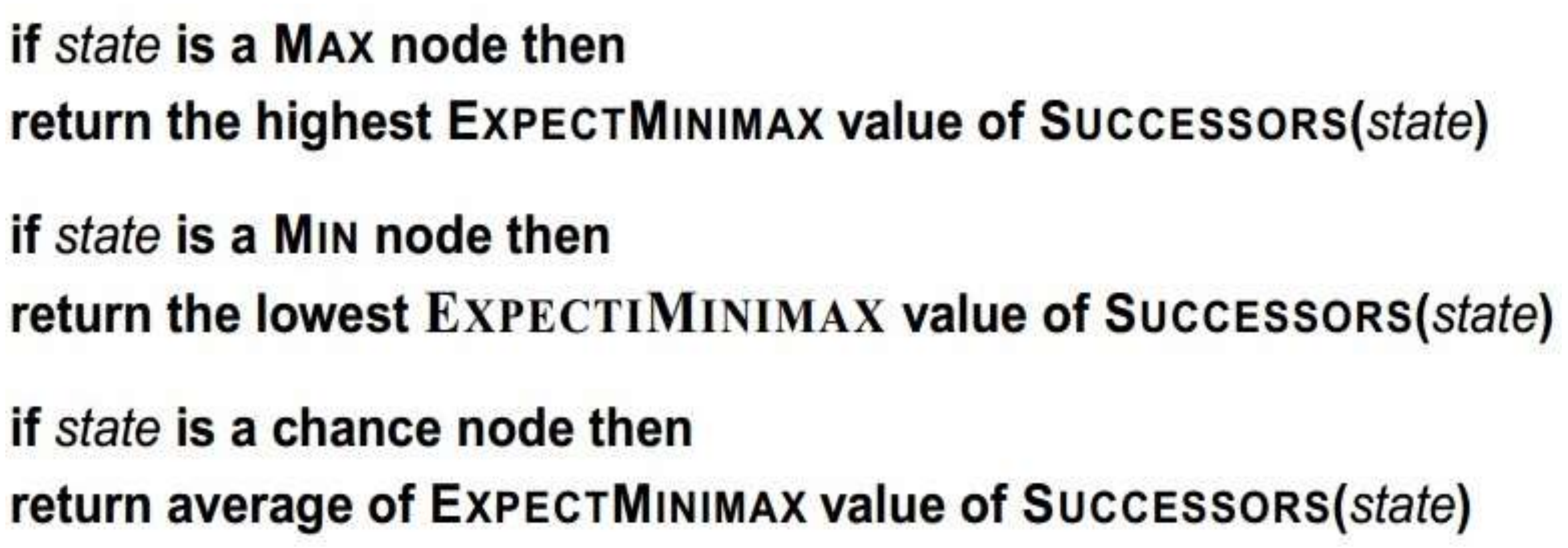
A game of chance is a game whose outcome is strongly influenced by some random event.

Common examples include:

- Dice
- Spinning tops
- Playing cards
- Roulette wheels
- Numbered balls drawn from a container
- Tossing a coin

Games of chance

In such methods, not only MAX and MIN states, but also a CHANCE state is considered as shown in the algorithm:



```
if state is a MAX node then
return the highest EXPECTMINIMAX value of SUCCESSORS(state)

if state is a MIN node then
return the lowest EXPECTIMINIMAX value of SUCCESSORS(state)

if state is a chance node then
return average of EXPECTMINIMAX value of SUCCESSORS(state)
```