Problem Solving

Problem solving is an agent based system that finds sequence of actions that lead to desirable states from the initial state.

- Problem solving agent is a kind of goal based agent which always tries to achieve goals.
- searching is the most commonly used technique of problem solving in AI.

Four steps of problem solving are :

1. Goal formulation :

A goal is a state that the agent is trying to reach. This step formulates the goal based on current situation and the agent's performance.

2. Problem formulation :

This is the process of deciding what actions and states to consider, given goal.

- Problem formulation must follow goal formulation.

3. Search method :

Determine the possible sequence of actions that leads to the states of known values and then choosing the best sequence.

4. Execute :

Once the search algorithm returns a solution to the problem, the solution is then executed by the agent.

A problem can be defined formally by 4 components:

1. Initial state:
It is the state from which agent start solving the problem.

2. State description / successor function:
A description of the possible actions available to the agent.

3. Goal test:
Determine whether the current state is goal state or not.

4. Path cost:
sum of cost of each path from initial state to the given state.

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

E.g.
8-puzzle problem

7	3	
6	8	4
5	2	1

Start state

	1	2
3	4	5
6	7	8

Goal state

Problem formulation of 8-puzzle problem is given by;

1. states:

Description of the eight tiles and location of the blank tile.

2. Goal test:

- checks whether the state matches the goal configuration.

3. Successor function /Actions:

Generate the legal states from trying the four actions {left, right, top, down}.

4. Path cost:

Each step cost 1.

Q. Why problem formulation must follow goal formulation?

Sol^n

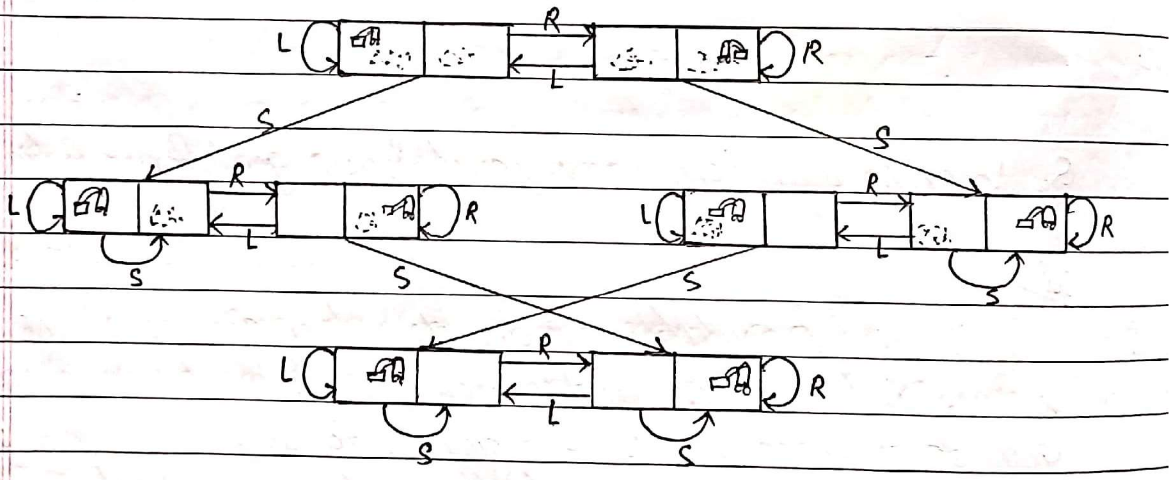
In goal formulation, we decide which aspects of the world we are interested in and which aspects can be ignored. In goal formulation process, the goal is to be set and we should assess those states in which the goal is satisfied. In problem formulation, we decide how to manipulate the important aspects and ignore the others. So without doing goal formulation if we do the problem formulation, we would not know what to include in our problem & what to leave and what should be achieved. so problem formulation must follow goal formulation.

State space Representation

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

- state space is the set of all states reachable from the initial states.

state space representation of vacuum world problem:



States:
two locations with or without dirt = 8 states
Initial state:
Any state can be initial
Actions:
{ left, Right, suck }
Goal test:
check whether squares are clean.
Path cost:
number of actions to reach goal.

Searching

→ Searching is the process of finding the required states or nodes.

Search problem
↳ is to find path from initial state to goal state.

A search problem consists of:
- A state space
- A start state
- A goal state

* Types of search method

1. Uniformed (Blind) search method:

This type of search does not use any domain knowledge. This means it does not use any information to judge where the solution is likely to lie. Uniformed search methods use only the information available in the problem definition.

E.g.
- Breadth first search
- Depth first search
- Uniform cost search
- Depth limited search
- Iterative deepening search
- Bidirectional search

2. Informed (Heuristic) search method:

Heuristic search uses domain-dependent (heuristic) information in order to search the space more efficiently. It uses a heuristic function h(n) that estimates how close we are to a goal. This function is used to estimate the cost from a state n to the closest goal.

E.g.

- Greedy search
- A* search
- Hill climbing search
- simulated annealing etc.

* Performance measures:

We will evaluate the performance of a search algorithm in four ways:

1. Completeness:

Wheather the search is complete or not. An algorithm is said to be complete if it definitely finds solution to the problem, if exist.

2. Time complexity:

Time required to get search solution. Usually measured in terms of the number of nodes expanded.

3. space complexity:

Maximum number of nodes to be visited (stored in memory).

4. Optamility:

If a solution is found, is it guaranteed to be an optimal one? For e.g. is it the one with minimum cost?

d → depth of sol^n
m → max^m depth of search space
b → branching factor
↳ max^m no. of nodes (children) generated by parent node.

} Time & space Complexity are measured in terms of d, m & b.

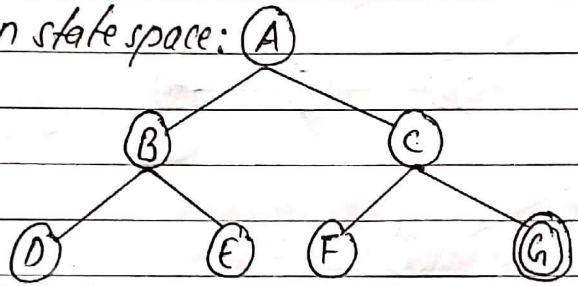
Breadth first search (BFS)

→ Nodes having lowest depth.

It expands the shallowest unexpanded node first. Starting from the root node (initial state) explores all children of the root node, left to right. If no solution is found, expands the first (leftmost) child of the root node, then expands the second node at depth 1 and so on until a solution is found.

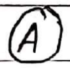
E.g.

1. Given state space:

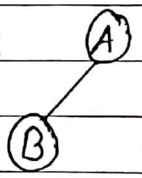


start state = A , Goal state = G

step 1:

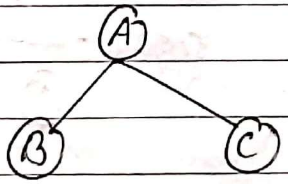


step 2:

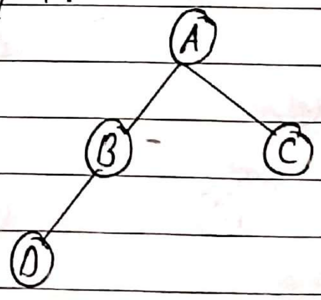


} fringe ↳ collection of nodes.

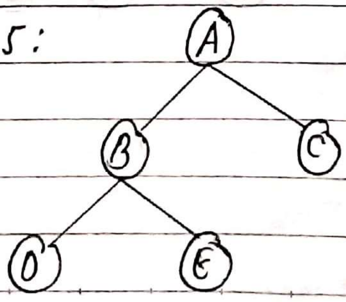
step 3:



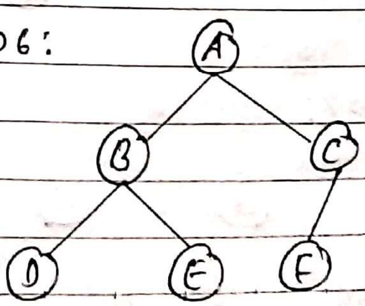
step 4:



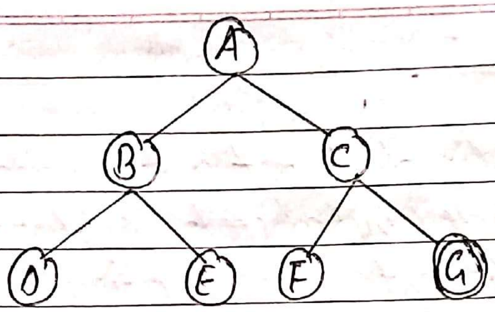
step 5:



step 6:

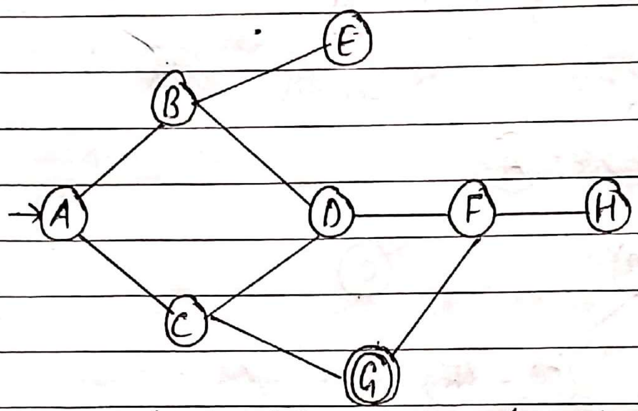


step 7 :



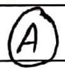
Goal achieved

2

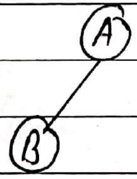


let A be the start state and G be the goal state.

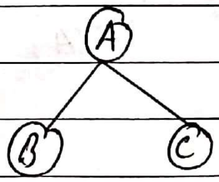
step 1 :



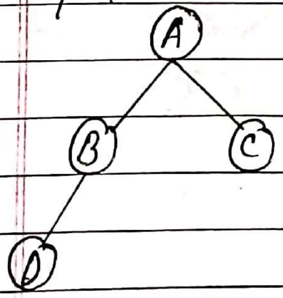
step 2 :



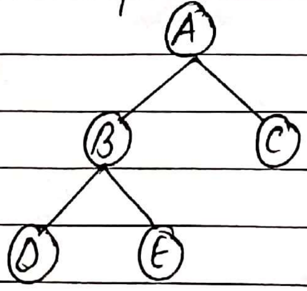
step 3 :



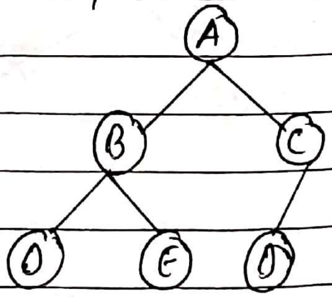
step 4 :



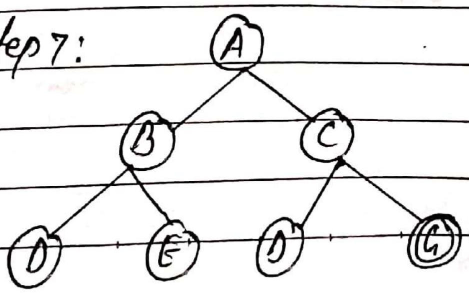
step 5 :



step 6 :



step 7 :



Goal node found.

✓ BFS is known as complete.
↳ It guaranteed to get solution if infinite nodes (depth) with finite final nodes.

BFS evaluation / properties

1. Completeness:
Complete if the goal node is at finite depth.

2. Optamility:
It is guaranteed to find the shortest path.

3. Time complexity:
No. of nodes to be visited.
For branching factor b and depth level d.

$$b + b^2 + b^3 + \cdots\cdots + b^d + (b^{d+1} - b) = O(b^{d-1})$$

4. Space complexity: It stores the entire fringe.

Total no. of nodes in memory $= 1 + b + b^2 + \cdots + b^d + (b^{d+1} - b)$.
$= O(b^{d+1})$

Advantages

→ If any solution exists then BFS guarantees to find it.
→ If there are many solutions, BFS will always find the shortest path solution.

Disadvantages

→ All nodes are to be generated at any level. so even unwanted nodes are to be remembered. Memory wastage.
→ High time

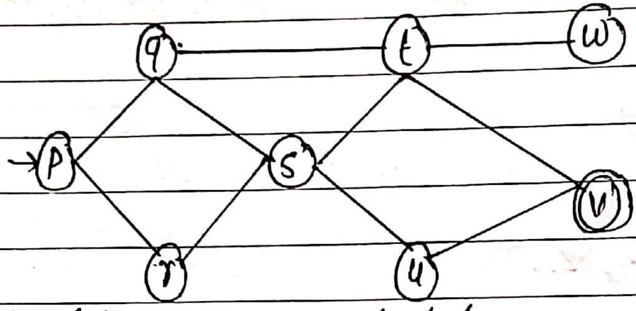
Depth First search (DFS)

It expands the deepest unexpanded node first.

→ It expands the root node, then the leftmost child of the root node, then the leftmost child of that node and so on. Only when the search hits dead end does the search backtrack and expands nodes at higher levels.

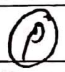
E.g.

r

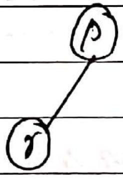


let start state = p and goal state = v

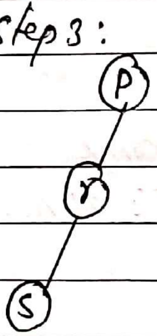
step 1:



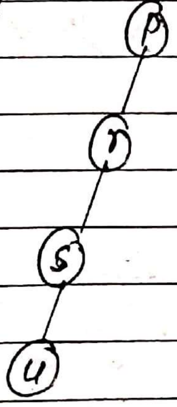
step 2:



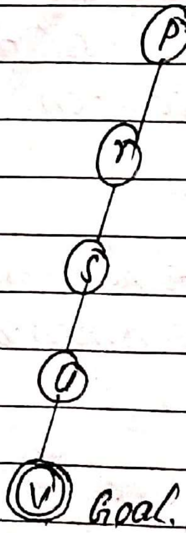
step 3:



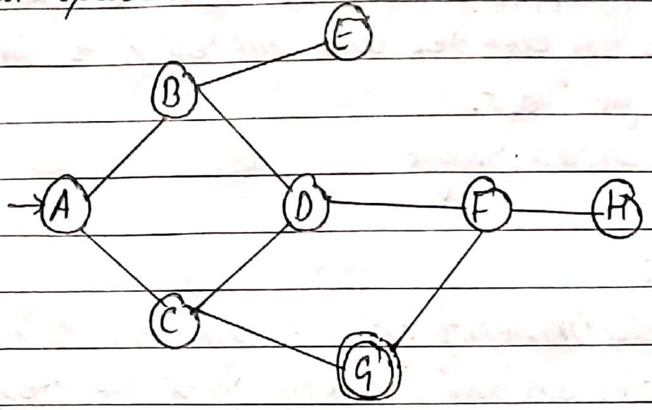
step 4:



step 5:

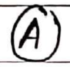


② Given state space :

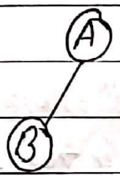


start state = A, Goal state = G

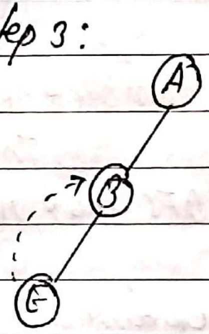
step 1 :



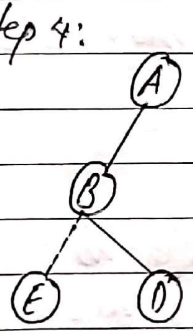
step 2 :



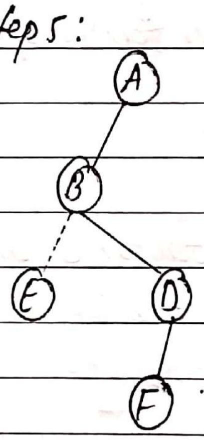
step 3 :



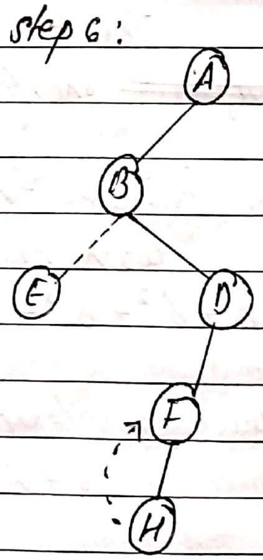
step 4 :



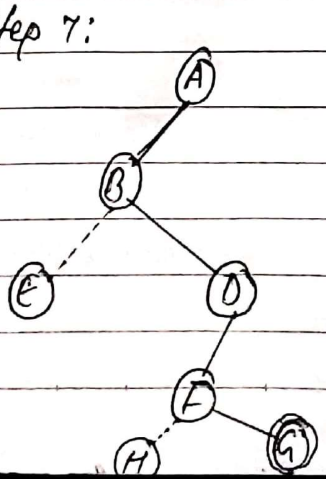
step 5 :



step 6 :



step 7 :



Goal node found.

DFS is not complete if there is infinite structure. This search can go on deeper and deeper into the search space and thus can get lost.

↳ Disadvantage.

Advantages:

→ Memory requirements in DFS are less compared to BFS as only nodes on the current path are stored.

DFS Evaluation:

1. Completeness:

Incomplete if there is infinite structure then DFS may not find solution.

2. optamility:

The first solution found by the DFS may not be ~~complete~~. shortest.

3. Time complexity:

with branching factor b and depth as m

$$b+b^2+b^3+\ldots\ldots+b^m = O(b^m) \quad m\geq d$$

4. Space complexity:

$$1+\underbrace{b+b+\ldots\ldots+b}_{\text{upto } m} = O(bm)$$

Depth limited search

In order to avoid the infinite loop condition arising in DFS, in depth limited search technique, depth-first search is carried out with a predetermined depth limit l.

If $l \geq d$, it gives solution

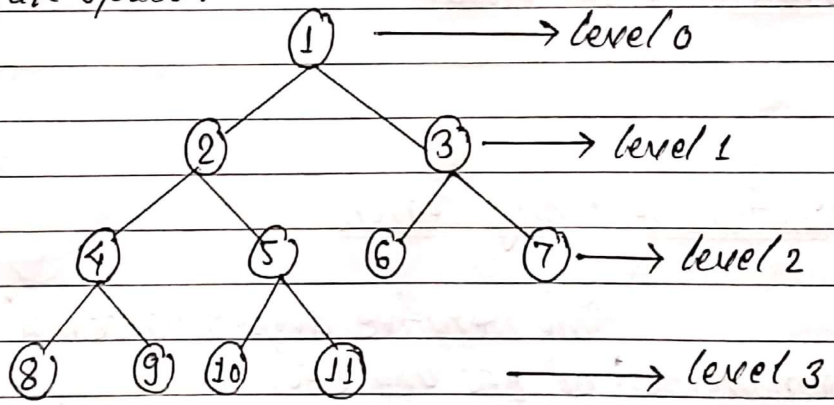
If $l < d$, it will be incomplete i.e. there is no sol^n on the given depth limit.

where, d is the depth of the solution.

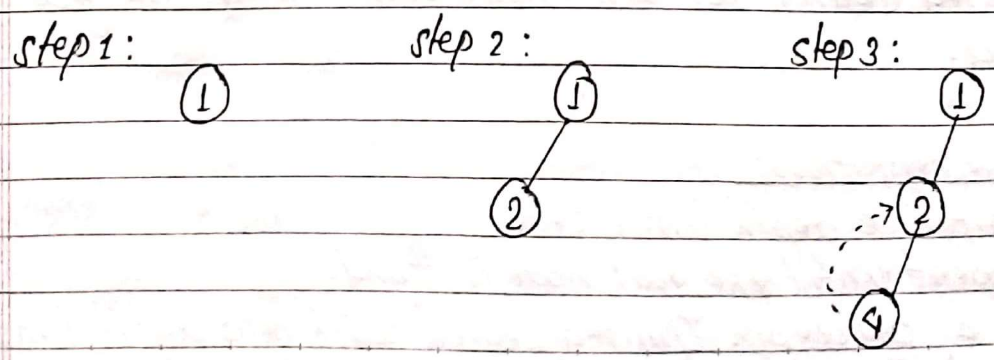
- The level of each node needs to be calculated to check whether it is within the specified depth limit.

E.g.

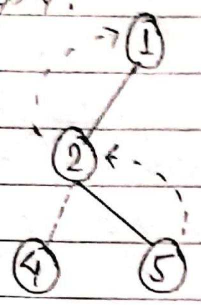
Given state space:



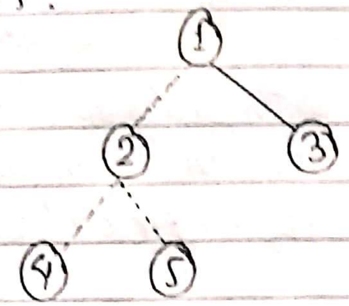
let depth limit = 2 and goal node = 11.



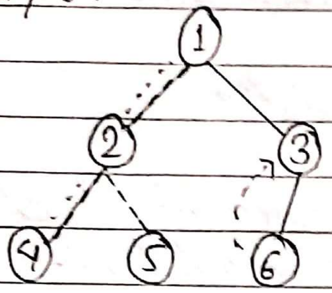
step 4:



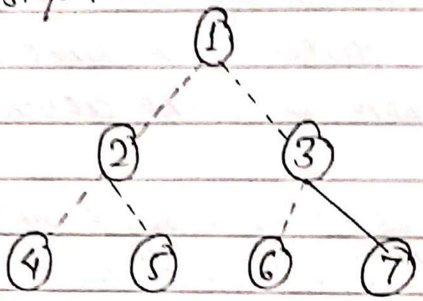
step 5:



step 6:



step 7:



Since depth limit we taken is 2 but goal node at level 3 so searches unsuccessful.

Iterative Deepening search

The iterative deepening search algorithm is a combination of DFS and BFS.

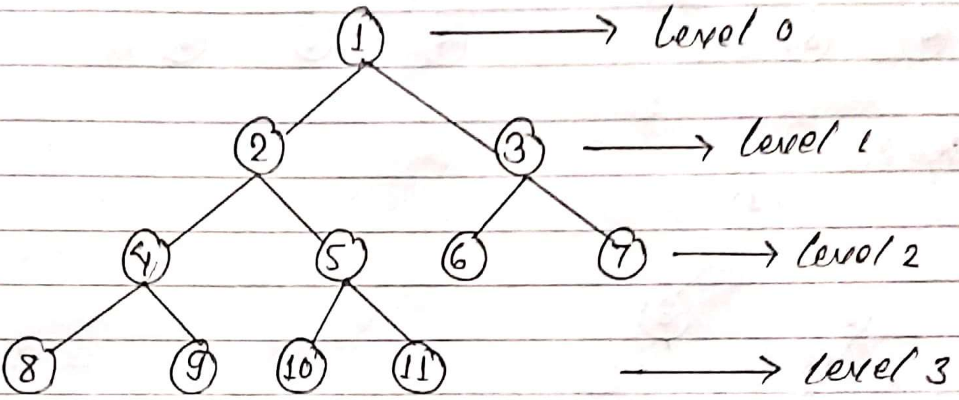
In this algorithm, starting at depth limit $d=0$, we iteratively increase the depth limit performing a depth limited search for each depth limit until the goal is found.

Implementation:

1. Initialize depth limit zero.
2. Repeat until the goal node is found.
    a. call depth limited search with new depth limit.
    b. Increment depth limit to next level.

→ It combines the advantages of both BFS & DFS, taking Completeness and optimality of BFS and the modest memory requirements of DFS.

E.g.



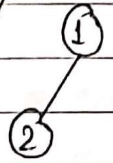
Let goal node = 11

Limit = 0

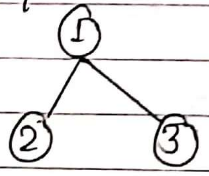
Limit = 1

step 1:

step 2:



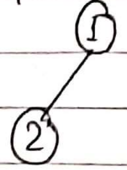
step 3:



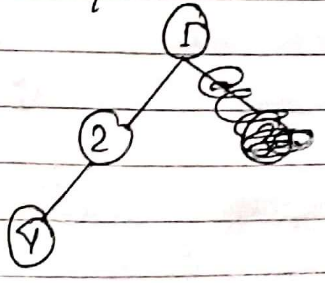
Limit = 2

step 1:

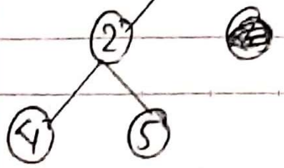
step 2:



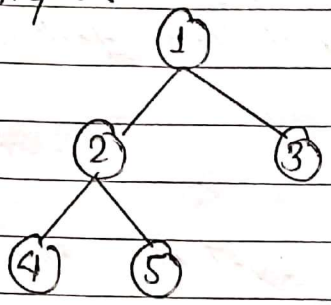
step 3:



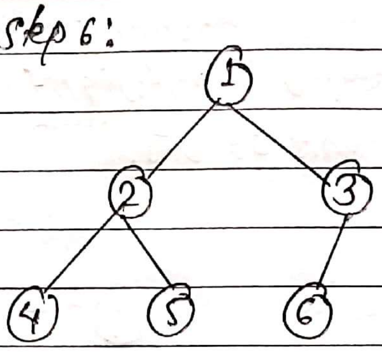
step 4:



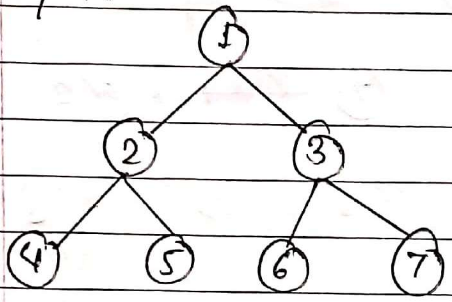
step 5:



step 6:

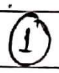


step 7:

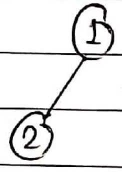


limit = 3

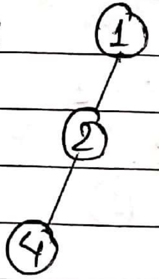
step 1:



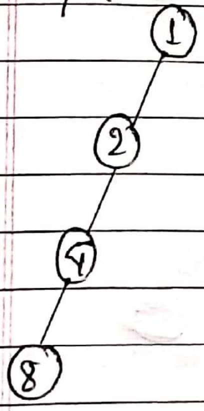
step 2:



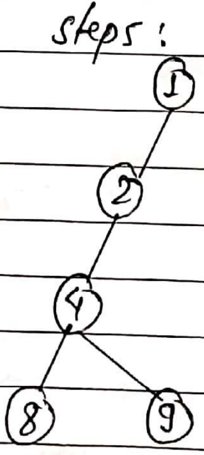
step 3:



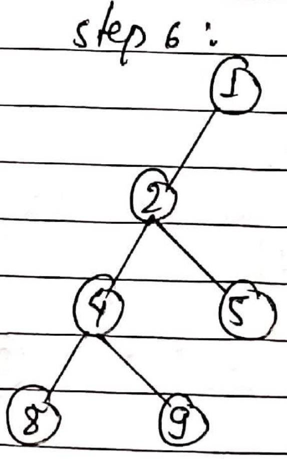
step 4:



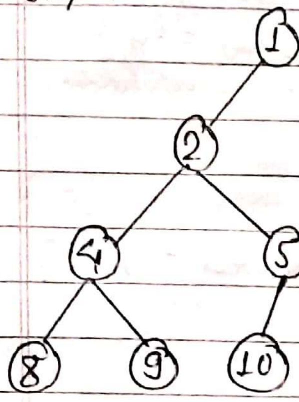
step 5:



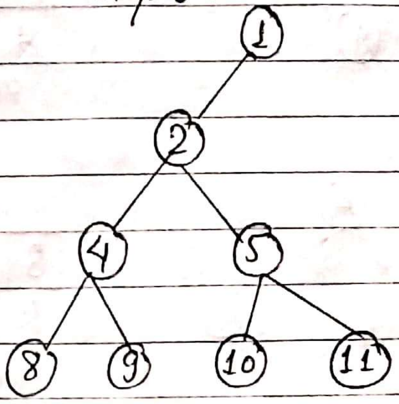
step 6:



step 7:



Step 8:



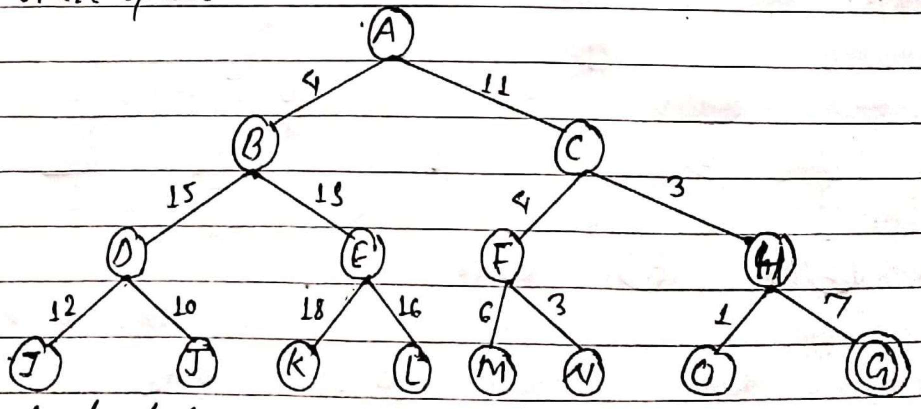
Goal node found.

Uniform cost search

- Searching algorithm used for weighted state space.
- Priority queue for storing nodes in state space is maintained, where least cost paths are given higher priority.
- Node at head of the queue is expanded first.
- The queue is updated at each expansion of nodes. (deletion of node visited & insertion of node to be visited).

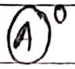
Eg.

1 Given state space



let start state = A & Goal state = G

Step1:

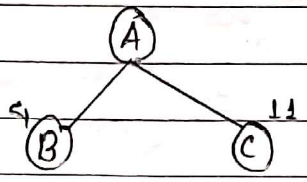


A

queue

visited := {A}

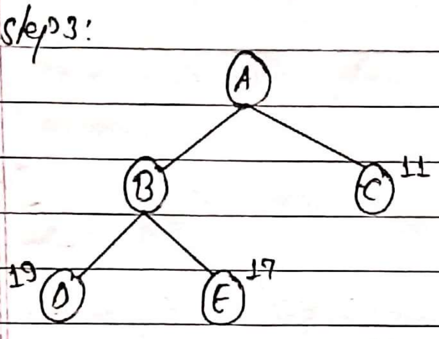
step2:



B	C
↑	

visited = {A, B}

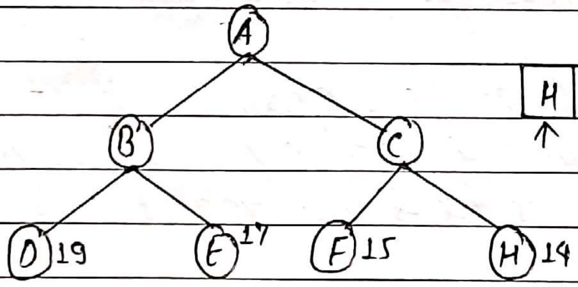
step3:



C	E	D
↑		

visited = {A, B, C}

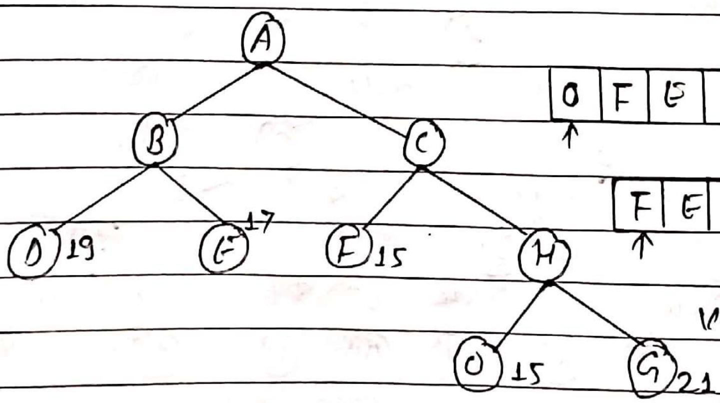
step 4:



H	F	E	D
↑			

visited = {A, B, C, H}

step5:

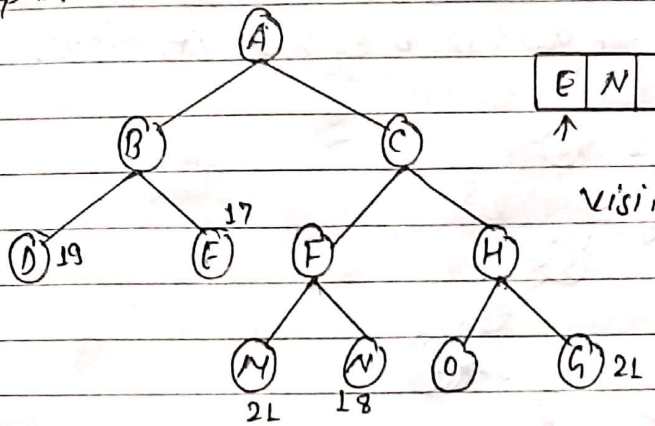


O	F	E	D	G
↑				

F	E	D	G
↑			

visited:{A, B, C, H, O, F}

step 6 :

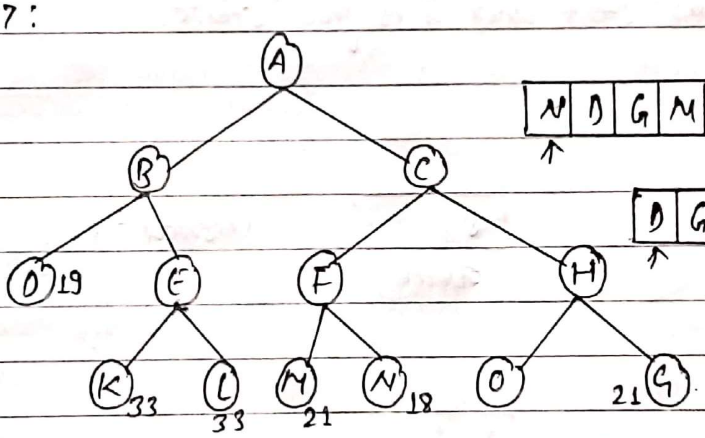


E	N	D	G	M

↑

visited = {A, B, C, H, O, F, E}

step 7 :



N	D	G	M	K	L

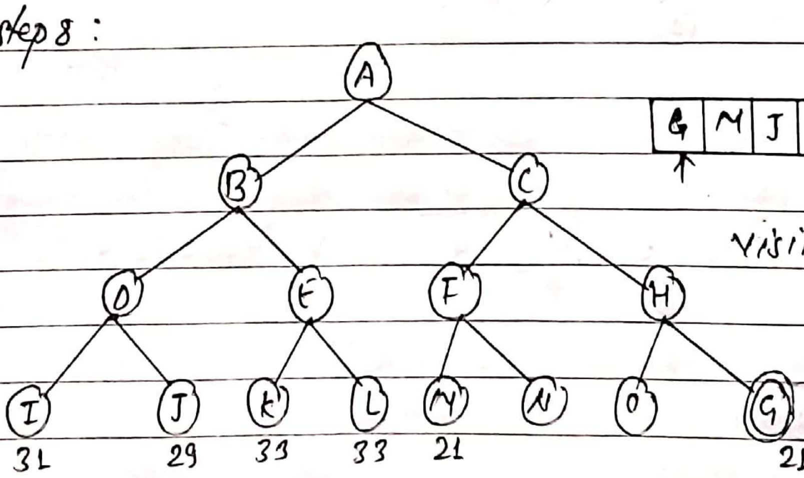
↑

D	G	M	K	L

↑

visited = {A, B, C, H, O, F, E, N, D}

step 8 :



G	M	J	I	K	L

↑

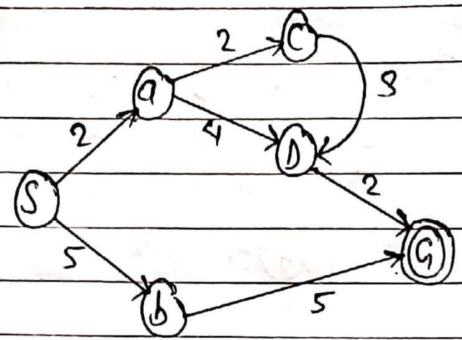
visited = {A, B, C, H, O, F, E, N, D, G}

↓

Goal node visited/found

074

Q. Given following state-space, use uniform cost search algorithm to find the goal. show each of iterations.



Here s is start state and G is goal state.

Soln

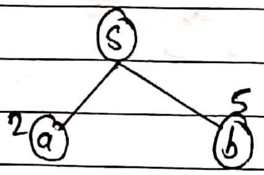
step 1:

s	

queue

visited = {s}

step 2 :

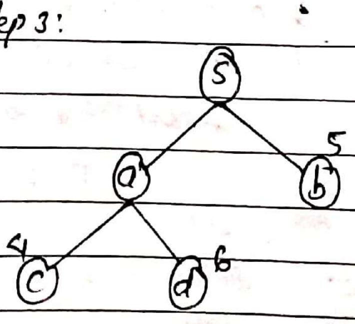


a	b

↑

visited = {s, a}

step 3:

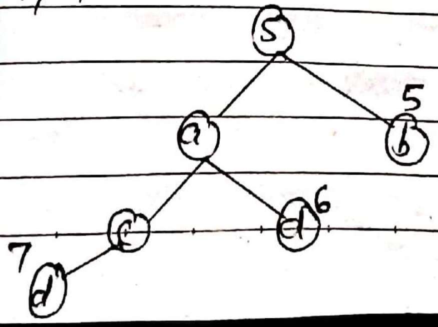


c	b	d

↑

visited = {s, a, c}

step 4:

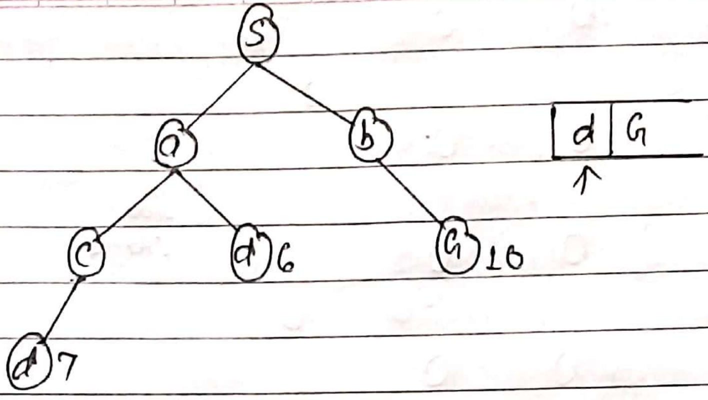


b	d

↑

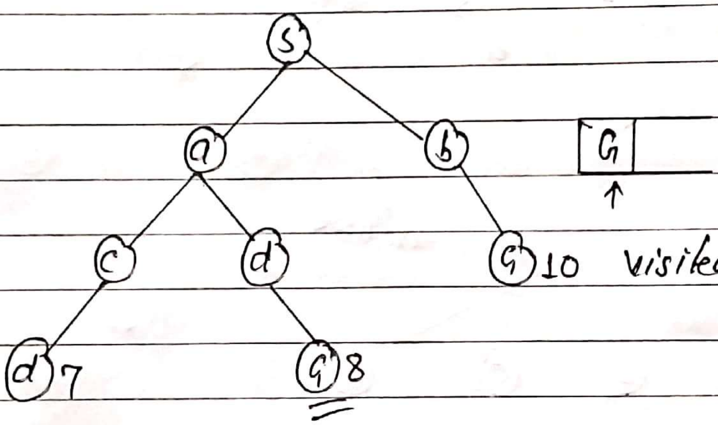
visited = {s, a, c, b}

step 5:



visited = {s, a, c, b, d}

step 6:



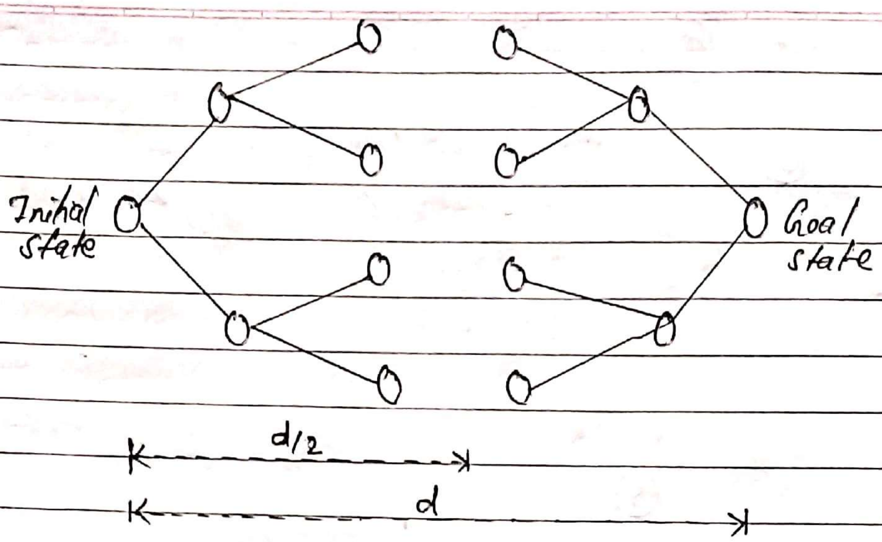
visited = {s, a, c, b, d, G}

Here the value of d is the smallest but it is already visited so search goes to G having value 8. Therefore goal node found.

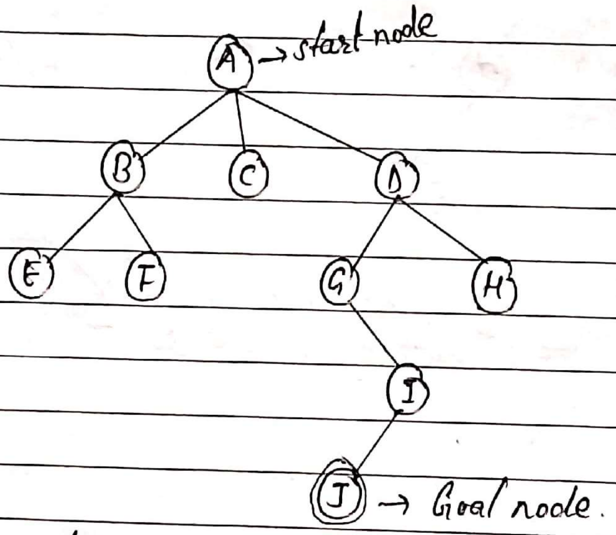
Bidirectional search

Bidirectional search algorithm runs two simultaneous searches, one from initial state called as forward-search and other from goal node called as backward-search. The search process terminates when the searches meet a common node of the search tree.

→ Bidirectional search can use search technique such as BFS, DFS etc.



Example



let us consider we will apply BFS from starting node and we will apply DFS from goal node.

Forward-search : A → B → C → D
Backward-search : J → I → G → D

Here, we can see that both searching method meets a point at D. That's why we have to stop searching here. and we can say that we have a path from start node to goal node. i.e.

A → D → G → I → J

Heuristic / Informed search

- → Best first search
  - → Greedy best first
  - → A*
- → Hill climbing
- → simulated annealing.

Best first search

Nodes are expanded based on the value of evaluation function. Nodes having minimal value of evaluation function are expanded first.

1) Greedy Best first search

It expands the node that appears to be closest to the goal.

The evaluation function is defined by

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

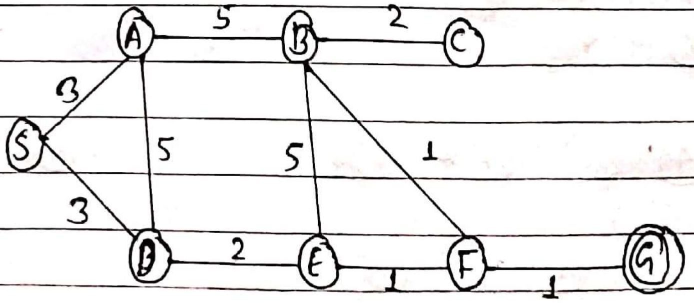
where, $h(n)$ is an estimate of cost from node n to goal node.

$h(n) = 0$ for goal node.

↳ (Heuristic value).

E.g.

1



Consider a heuristic function h(n) for above state space is defined as;

$h(S) = 10$ $h(E) = 4$

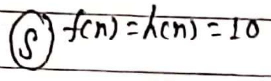
$h(D) = 7$ $h(F) = 2$

$h(A) = 6$ $h(C) = 3$

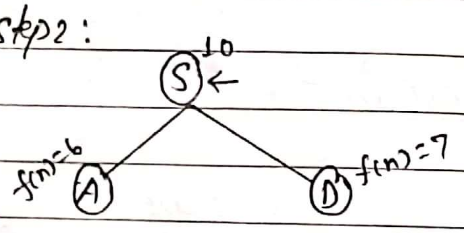
$h(B) = 4$ $h(G) = 0$

The search will be:

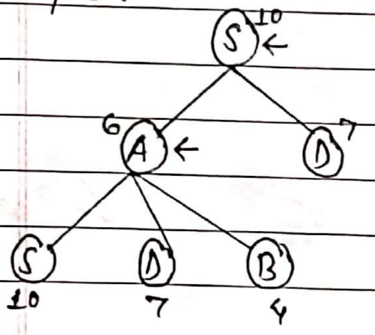
step 1:



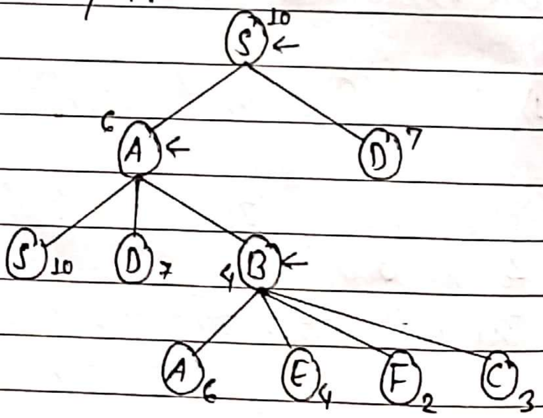
step 2:



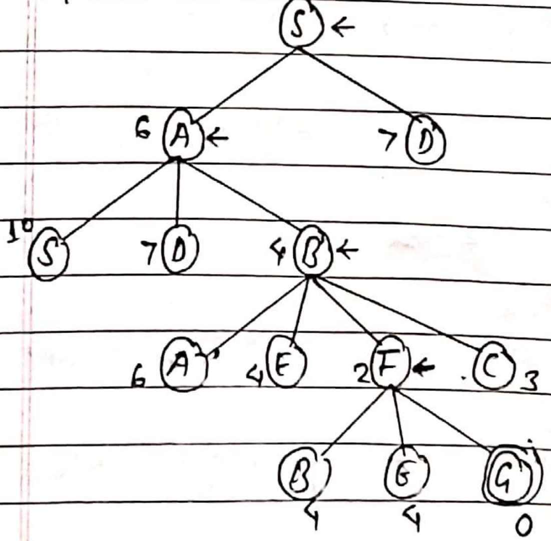
step 3:



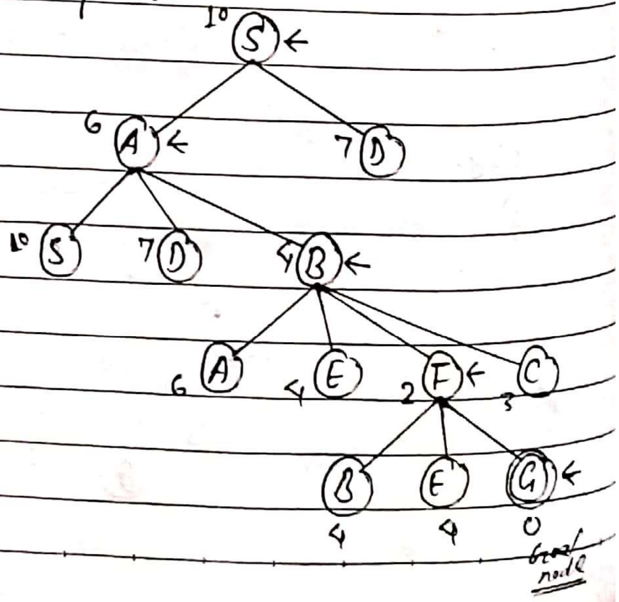
step 4:



step 5:



step 6:



If we change the heuristic of c as h(c) = 2. It stucks in loop. when search reaches at node c it stucks in loop so we can't reach at goal node.

∴ Greedy best first search is not complete & optimal.

→ GBFs doesn't guarantee optimal solⁿ
→ It can get stuck in loop. It is not optimal

2) A* search

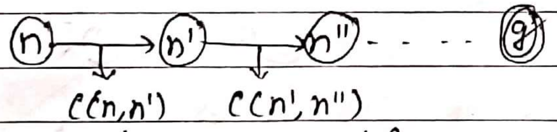
The evaluation function is defined by

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

where, $g(n)$ = sum of actual costs incurred while travelling from root node (start node) to node n.

$h(n)$ = estimate of cost from node n to goal node.

$f(n)$ = estimated total cost of path through n to goal.



↳ actual cost from node n to n' ($n \to n'$).

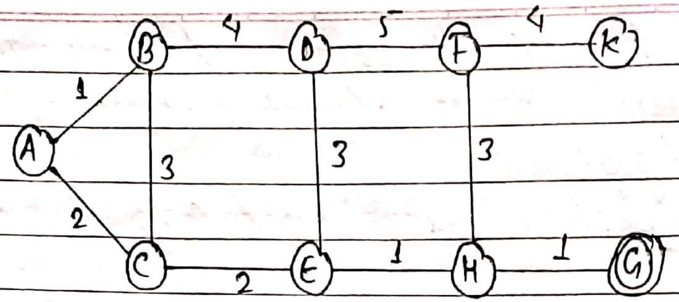
$g(n') = c(n,n')$

$g(n'') = c(n,n') + c(n',n'')$

→ A* search guaranteed to complete.
↳ It definitely finds the solution.

→ A* search is both complete and optimal if an admissible heuristic is involved.

E.g.

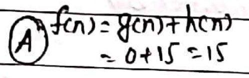


Given h(n) as

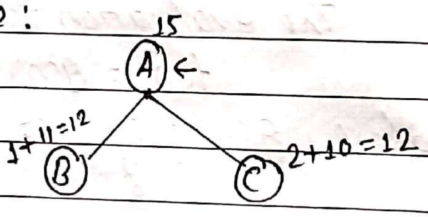
$h(A)=15$	$h(D)=9$	$h(F)=3$
$h(B)=11$	$h(E)=6$	$h(K)=2$
$h(C)=10$	$h(H)=2$	$h(G)=0$

Thus the search will be,

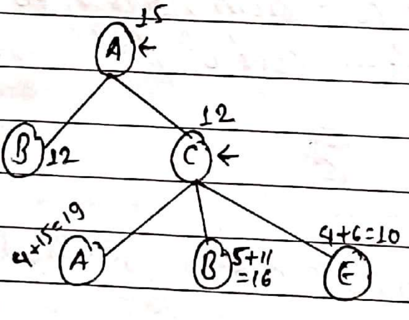
step 1:



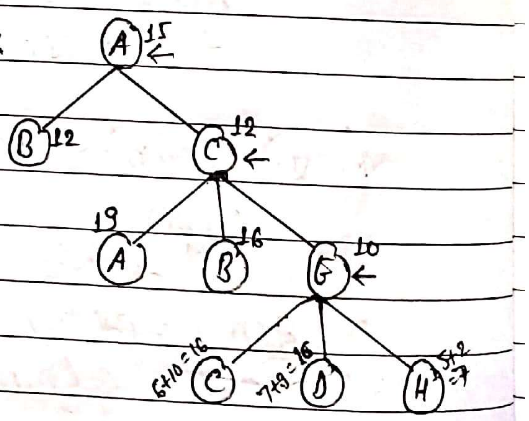
step 2:



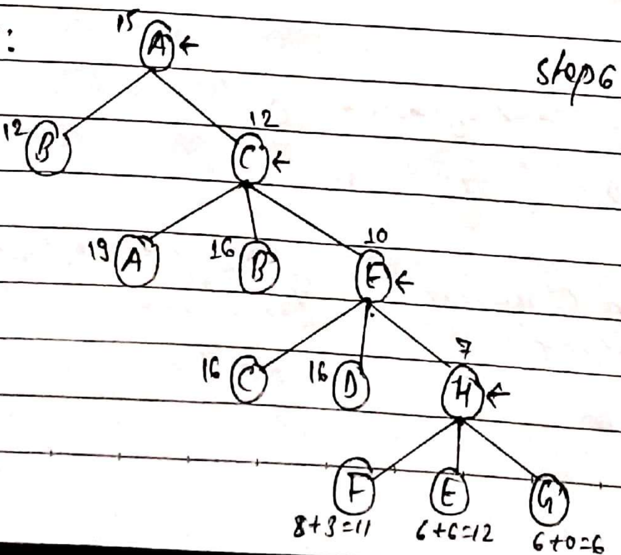
step 3:



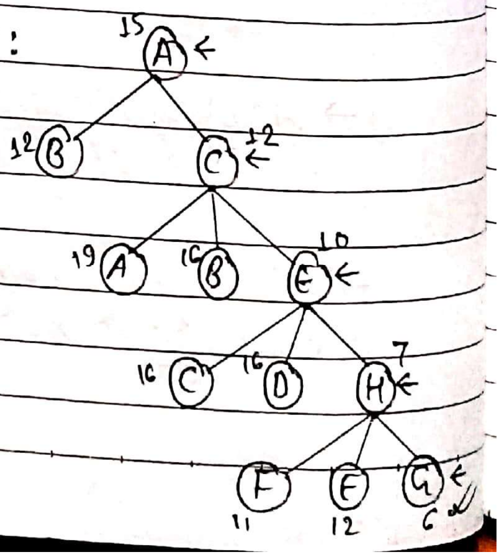
step 4:



step5:



step6:



# If $h(n) = \min\left(\sum \frac{c(n,n')}{2}\right)$ is given then heuristic function is calculated as

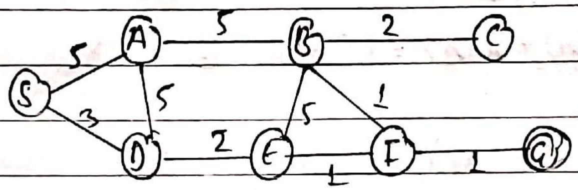
$h(S) = \min(7/2, 14/2, 12/2, 17/2, \ldots) = 7/2 = 3.5$

$h(A) = \min(7/2, 9/2, 12/2, 13/2, \ldots) = 7/2 = 3.5$

$h(B) = \min(2/2, 7/2, \ldots) = 2/2 = 1$

$h(C) = \min(4/2, 9/2, \ldots) = 2$

⋮



* Admissible Heuristic

A heuristic function is said to be admissible if for all node n, $h(n) \leq c(n)$

where, $h(n)$ is heuristic of node n to goal &

$c(n)$ is actual cost to reach goal from node n.

* Consistency (Monotonicity)

A heuristic is said to be consistent if for any node n, $h(n) \leq c(n,n') + h(n')$.

where, n' is successor of n

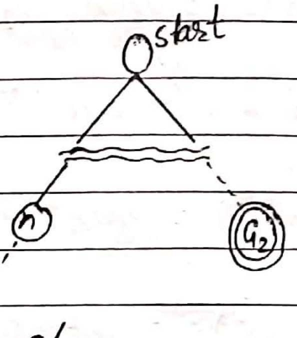
$h(n)$ is estimated cost from node n to goal.

$c(n,n')$ is actual cost from n to n'.

Imp.

Theorem:

A* is optimal if h(n) is admissible.



suppose G_2 is suboptimal goal.
Let n be unexpanded node on shortest path to optimal goal G.
let C^* be cost of optimal goal node.

$f(G_2) = h(G_2) + g(G_2) = g(G_2)$ $[\because h(G_2)=0]$

$f(G_2) > C^*$ ---- ① since G_2 is suboptimal.

since h(n) is admissible;

$$f(n) = g(n) + h(n) \le C^* \text{ ---- ②}$$

from ① & ②

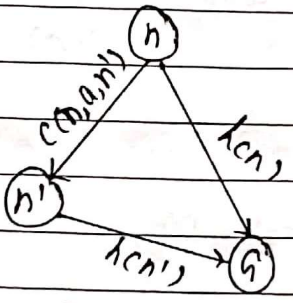
$$f(n) \le C^* < f(G_2)$$

$\therefore$ $f(G_2) > f(n)$

Thus A* will never go for suboptimal goal G_2 if h(n) is admissible.

Theorem:

If h(n) is consistent, then the values of f(n) along the path are non-decreasing.



Suppose n' is successor of n, then

$$g(n') = g(n) + c(n,a,n')$$

we know that,

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

$$f(n') = g(n) + c(n,a,n') + h(n') \text{ —— ①}$$

A heuristic is consistent if

$$h(n) \le c(n,a,n') + h(n') \text{ —— ②}$$

from ① & ②

$$f(n') = g(n) + c(n,a,n') + h(n') \ge g(n) + h(n) = f(n)$$

$$f(n') \ge f(n)$$

f(n) is non-decreasing along any path.

Hill climbing search

In hill climbing the basic idea is to always head towards a state which is better than the current state.

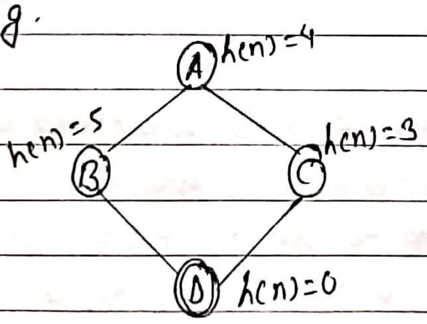
→ Hill climbing search is local search because it just try to evaluate only current node's successor without thinking ahead about where to go next.

Algorithm

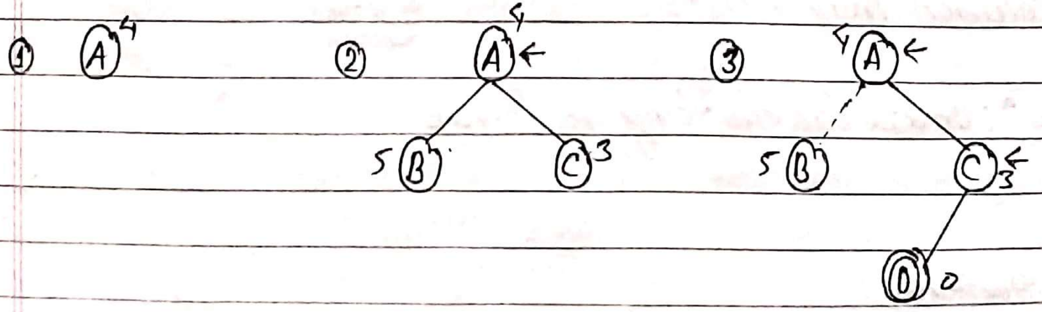
1. set current_state = initial_state
2. Until current_state = goal_state OR there is no change in current state do:
   - i) Get the successors of the current state and use the evaluation function to assign a score to each successor.
   - ii) If one of the successors has a better score than the current state then make it new current state.

E.g.

1

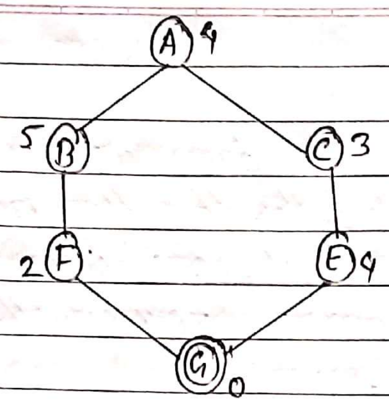


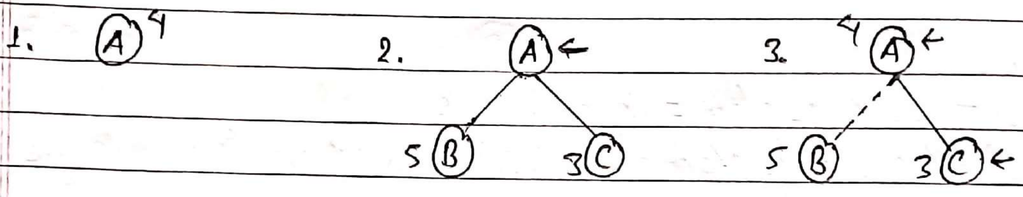
let f(n) = h(n)



→ Hill climbing search is not complete.

2

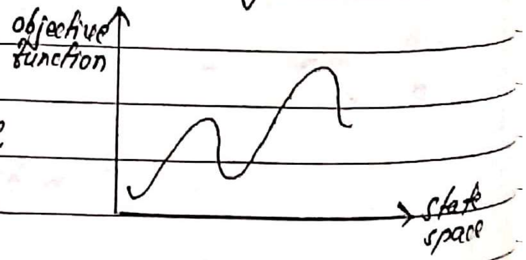




Here, the search will stop at node C. Node C's successor has no better score than C so it will stop here.
∴ It is not complete.

Problems with Hill climbing

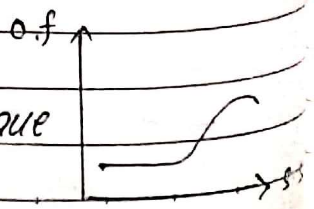
1. It gets stuck at local maxima. At local maxima all neighboring states have a values which is worse than the current state. It will not move to the worse state and terminate itself. The process will end even though a better solution may exist.



sol^n: Utilize backtracking technique

2. Plateaue:

An area of the search ~~fo~~ space where evaluation function is flat. On plateaue all neighbors have same value.



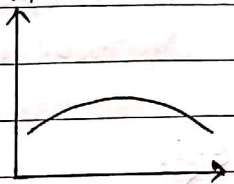
classmate Date Page

Hence it is not possible to select the best direction.

Soln: Make a big jump.

3. Ridge:

where there are steep slopes and any point on the ridge can look like ~~peack~~ peak because movement in all direction is downward. Hence the algorithm stops when it reaches this state.



Soln: Move in several direction at once.

Simulated Annealing (SA)

→ allows downward steps. (Compared to hill climbing search)

→ a move is selected at random and then decides wheather to accept it. If the move is better than its current position then simulated annealing will always take it. If the move is worse then it will be accepted based on some probability. The probability of accepting a worse state is given by the eqn

$$P = e^{-c/t} > r$$

where,

c = change in the evaluation function

t = current value

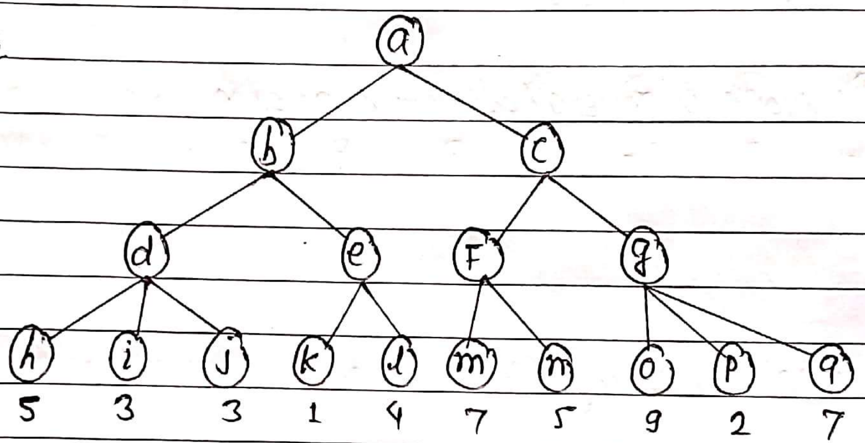
r = a random number betn 0 & 1.

Q. what is local maxima problem in hill-climbing? How simulated annealing handles the problem of local maxima in hill climbing search?

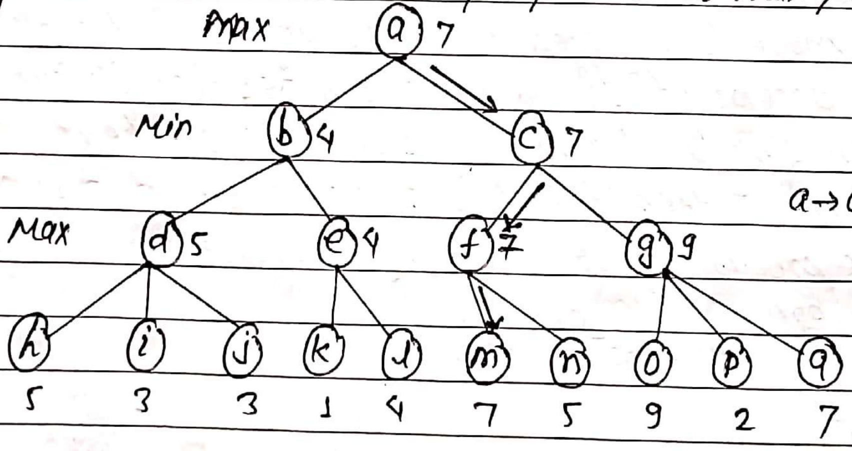
Mini-Max search

- → has two agents min & max.
- → max player maximizes its utility value. Thus choose successor with highest utility.
- → min player minimizes its utility value. Thus choose a successor with lowest utility.

E.g.

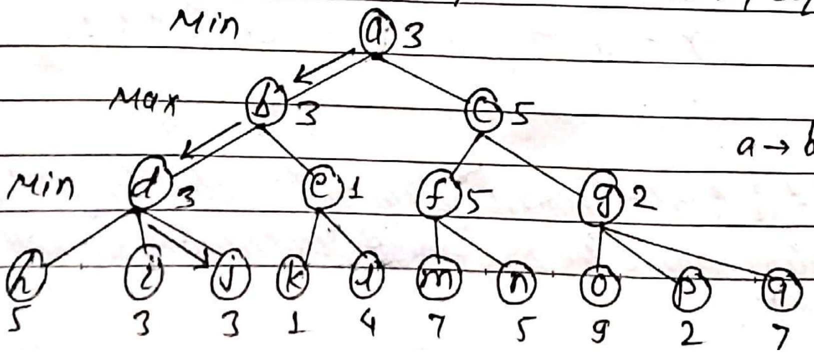


using mini-max search from perspective of max player:



a → c → f → m

using mini-max search from perspective of min player:



a → b → d → j
→ i

Alpha-Beta Pruning

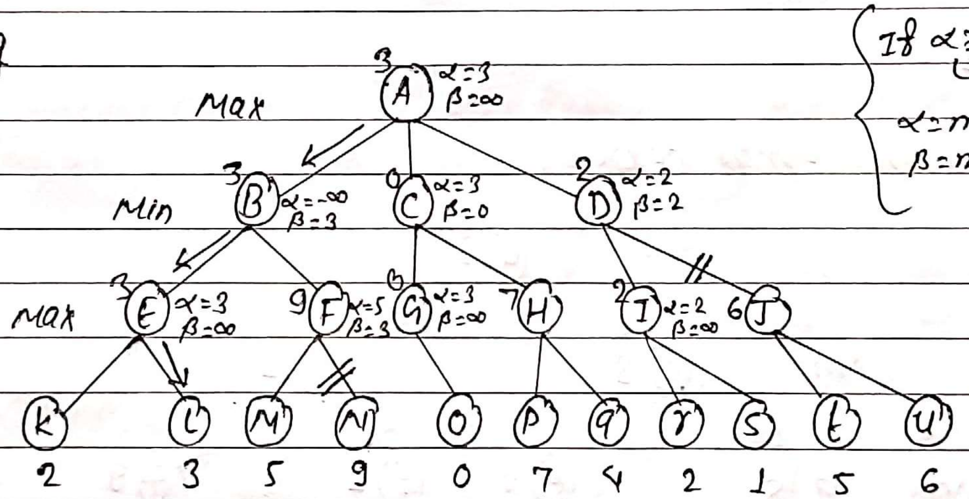
→ is a technique of eliminate unnecessary nodes from state space.

→ has two values alpha and beta.

Alpha → is best (i.e. max^m) value found so far at any choice point along the path for MAX.

Beta → is best (i.e. min^m) value found so far at any choice point along the path for MIN.

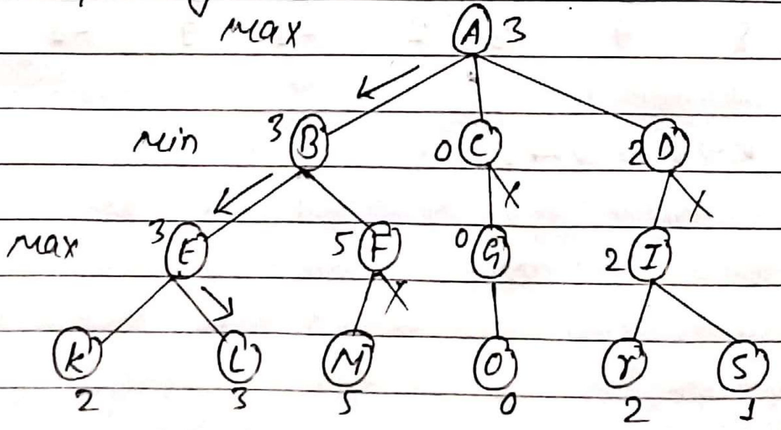
E.g



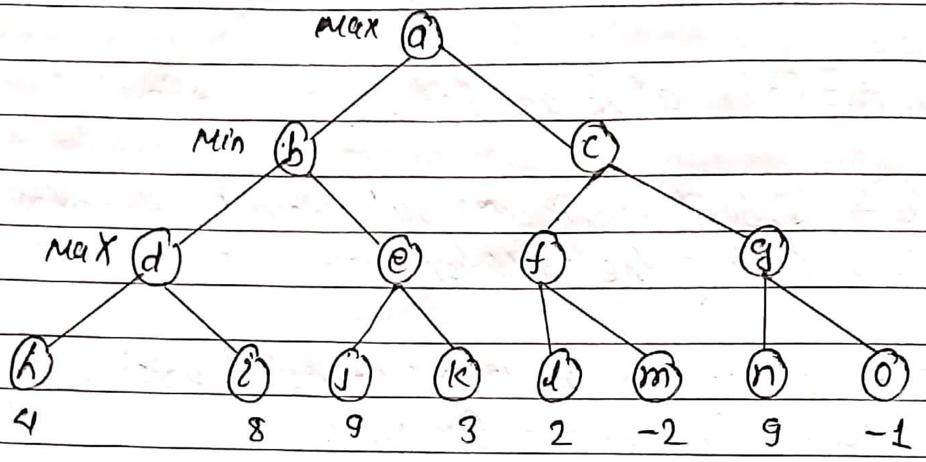
If α ≥ β
↳ Prune

α = max = -∞
β = min = ∞

After pruning:

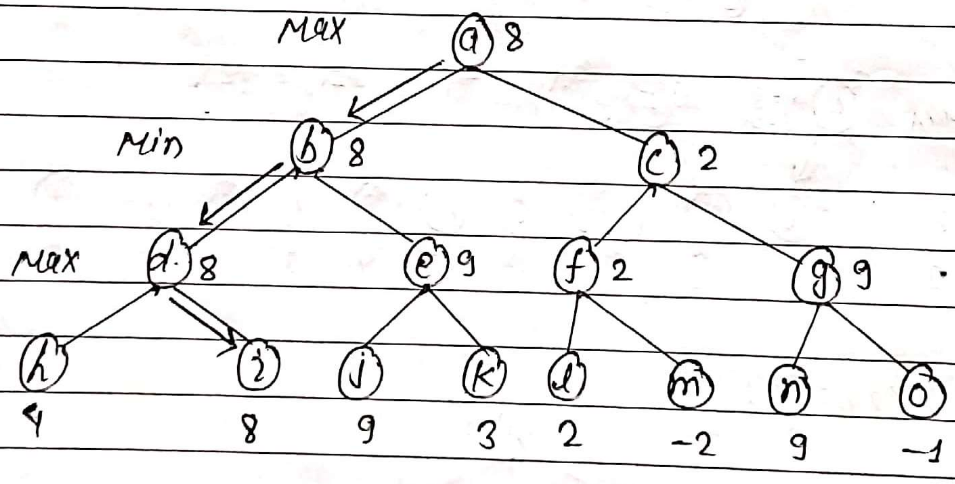


Q. Given following search space with utility, perform mini-max search and identify alpha-beta cutoff if any.



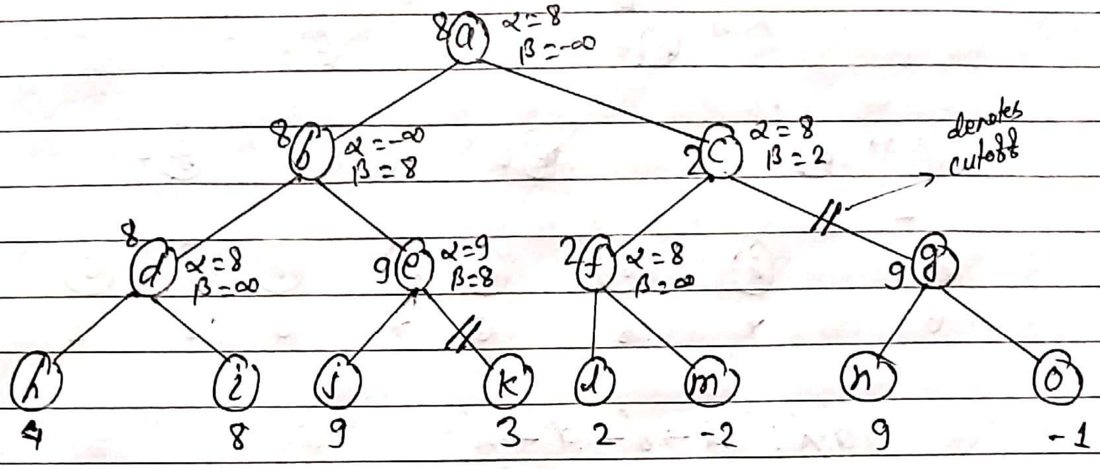
Solⁿ

The mini-max search:

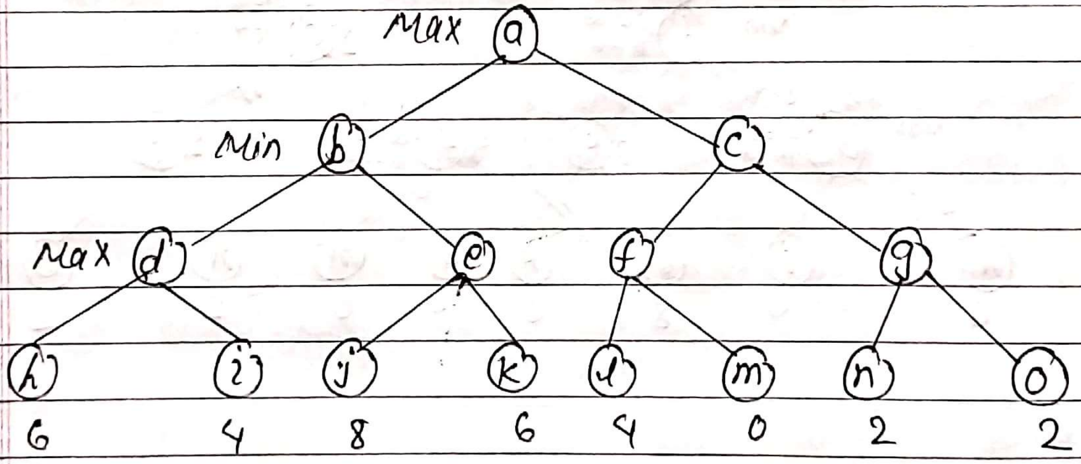


path : a → b → d → i

Alpha-beta cutoff:



Q. Consider the following game tree

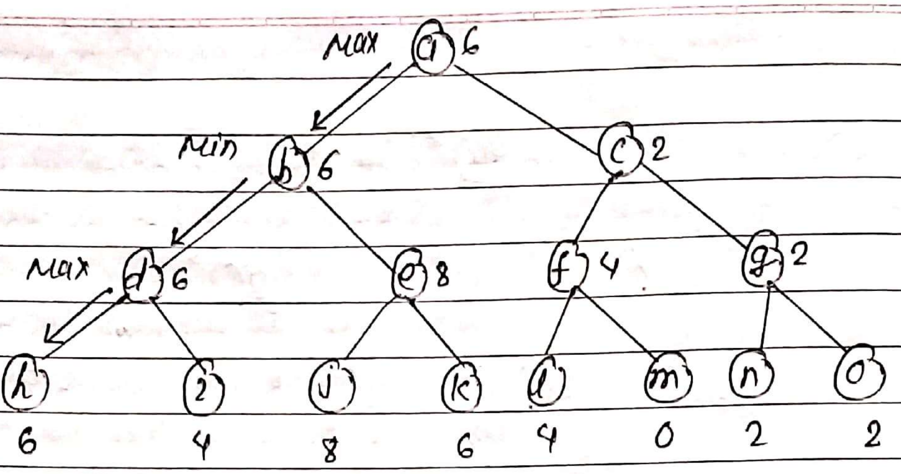


i) use the mini-max procedures and show what moves should be chosen by the two players.

ii) use the alpha-beta pruning procedure and show what nodes would not need to be examined.

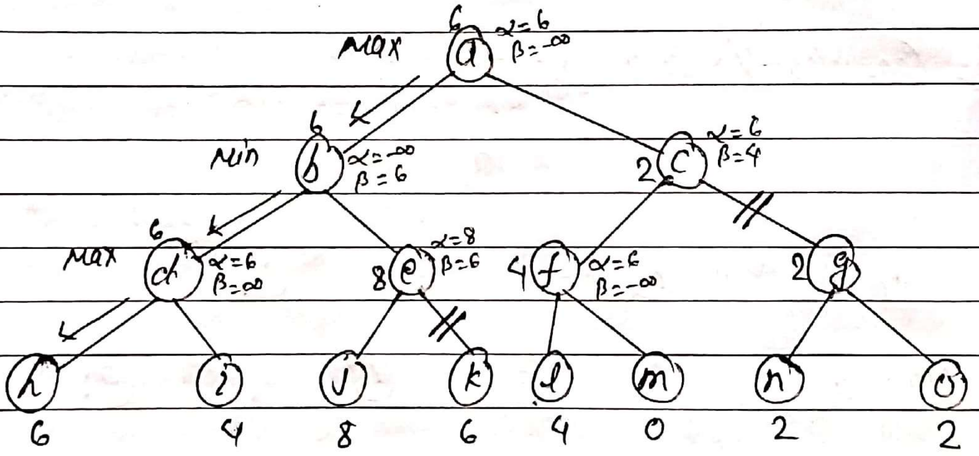
Solⁿ

i)



Path: $a \to b \to d \to h$

ii)



Constraint satisfication Problem (CSP)

CSP are representation where the problems are formulated as constraint between a set of variables. A Constraint satisfication problem (CSP) consists of:

- a set of variables $\{x_1, x_2, ..., x_n\}$
- a finite set of domain D_i associated with each variables.
- a set of constraints $\{C_1, C_2, ..., C_n\}$

A solution is an assignment of a value in D_i to each variable x_i such that every constraints are satisfied.

→ variables represent the entities in the csp.
→ The domain of variable is a set of possible values that can be assigned to the variable.
→ constraint is a set of relations between values (domain) of variables.

E.g.

1) N-Queen Problem

N-queen is a problem where given N-by-N board, you want to place N queens such that no two queens are in the same row, column or diagonal.

CSP definition:

Variables : The row assignment of N queens
Domains : 1 to N
Constraints : No two queens can be on the same row, column or diagonal.

2) Map coloring

Map coloring is a problem where given a map and a set of colors, you color the regions of the map such that no adjacent regions have the same color.

CSP definition:

Variables : The color of each region of the map.
Domains : The list of colors.
Constraints : No two adjacent regions can be colored the same color.