UNIT-3

Problem Solving By Searching

Problem solving in AI is a systematic search through a range of possible action in order to reach some predefined goal or solution. There are four general states in problem solving which are as follows:

i) Goal ~~formation:~~ formulation:

A goal is a state that the agent is trying to reach.

ii) Problem formulation:

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

iii) Search Method:

It determines possible sequence of actions and choose the best one.

iv) Execute:

It provides the solution to perform action.

Searching → It is a commonly used method in AI for solving problems. The search technique explores the possible moves that one can make in a space of 'states', called the search space.

Problem Formulation → It involves deciding what actions and states to consider, given the goal. A problem can be defined formally by 4 components as;

i) An initial state: from which agent start.

ii) State Description: Description of possible action available on agent. It is also called successor function.

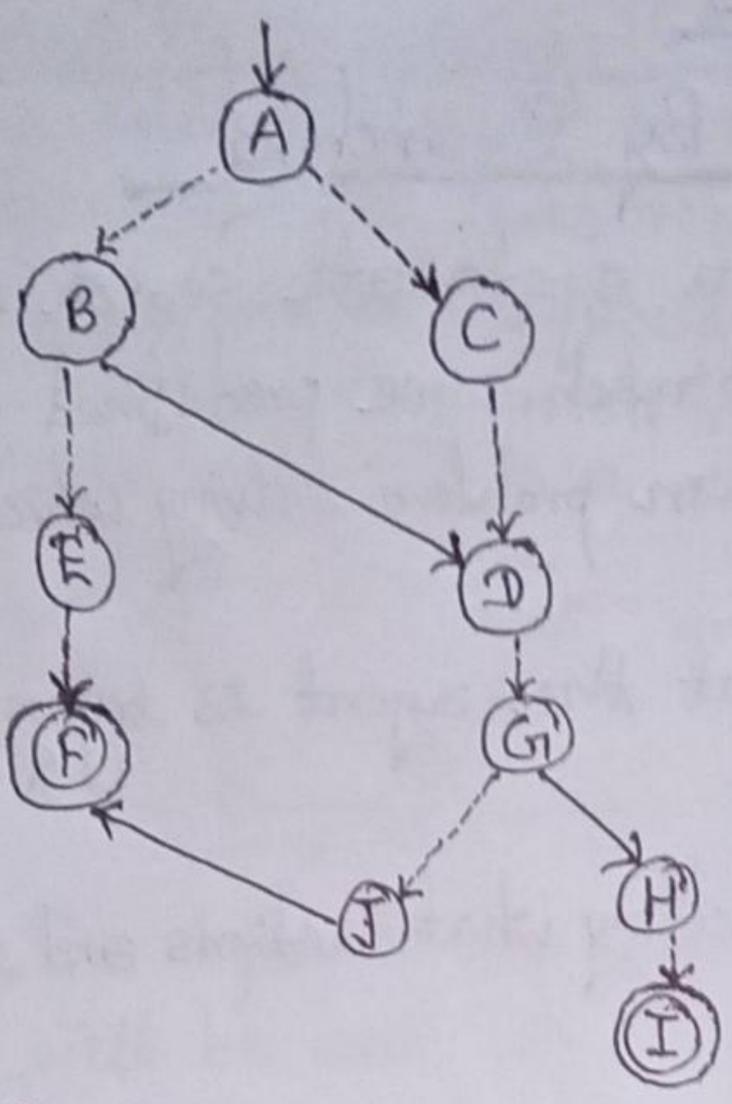
iii) Goal test: Determining the state is goal state or not.

iv) Path cost: Sum of cost of each path from initial to given state.

State Space Representation → The state space is defined as a directed graph with each node is a state and arc represents the application of an operator transforming a state to a successor state. A solution is path from the initial state to goal state consisting of (S, s, O, G).

where, S = Set of states
s = start state
O = Operation (which helps to transfer state to its successor state).
G = Goal state.

Example :-



$S = \{A, B, C, D, E, F, G, H, I, J\}$

$s = A$

$G = \{F, I\}$

Well-Defined Problem: A problem that lacks one or more of the below 5 properties is an ill-defined problem. But a well-defined problem can be defined formally by following five components:

1) Initial state → State from which agent start.
2) State Description (Successor function) → Description of possible actions available on agent.
3) State Space and Path → All states reachable from initial by any sequence of actions is state space. Path is the sequence through state space.
4) Path cost → Sum of cost of each path from initial to given state.
5) Goal test → Determining the state is goal state or not.

Well defined problem
- → Vaccum world state problem.
- → Real world Problem (Used to define the time, money and human resource cost).
- → Toy problem (Used to compare performance of system).

⊛. Solving Problems by Searching: (Concept) ← less imp

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

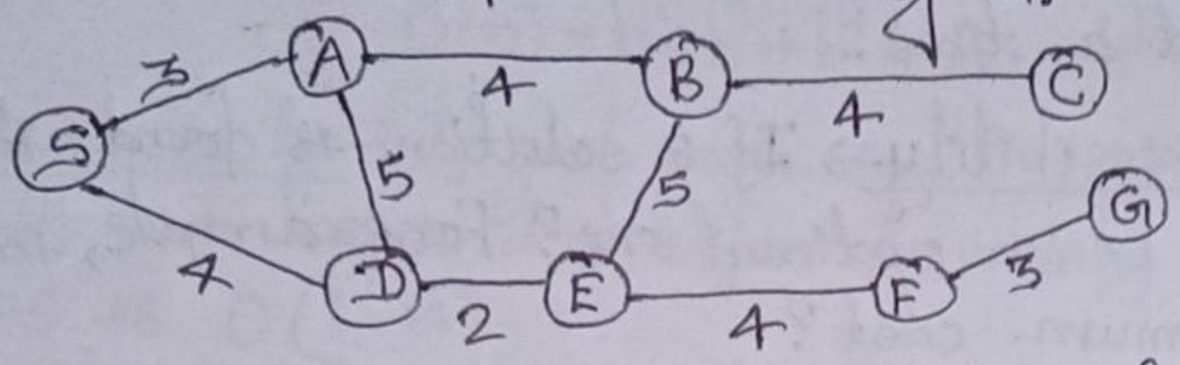


Fig. A graph representation of map.

This problem will be used to illustrate some search methods. Search problems are part of a large number of real ~~word~~ world applications:

→ VLSI Layout
→ Web Search
→ Path Planning
→ Robot navigation etc.

⊛. Search Techniques / Search Strategies:

There are two broad ~~searl~~ search strategies:

1) Uninformed (or blind) search methods → In the method, the order in which potential solution paths are considered is arbitrary, using - no domain specific information to judge where the solution is likely to lie. E.g. Breadth first search, Depth first search, Depth limit search, Bidirectional search etc.

2) Heuristically informed search methods → In this method, one uses domain-dependent (heuristic) information in order to search the space more efficiently. E.g. Greedy best first search, A* search, Hill climbing search etc.

⊛ Performance Evaluation of Search Techniques:-

We can evaluate the performance of search techniques in four ways:

1) Completeness → An algorithm is said to be complete if it definitely finds solution to the problem, if exist.

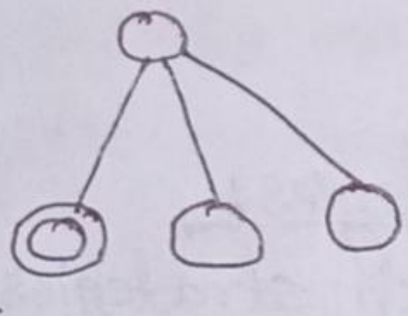
iii) Time Complexity → How long (worst or average case) does it take to find a solution? Usually measured in terms of number of nodes expanded.

iii) Space Complexity → How much space is used by the algorithm? Usually measured in terms of the maximum number of nodes in memory at a time.

iv) Optimality / Admissibility → If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

⇒ Time and Space complexity are measured in terms of:
- ⇒ b → Maximum branching factor (number of successor of any node) of the search tree.
- ⇒ m → Depth of the least-cost solution.
- ⇒ d → Maximum length of any path in the space.

Example:



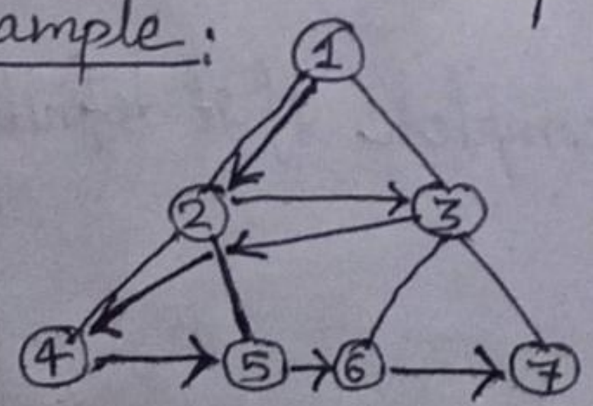
b = 3
d = 1
m = 1

1. Uniformed Search:

@ Breadth First Search (BFS): It is an algorithm for traversing or searching tree or graph data structures. It starts with the tree root and explores all of the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level. It is implemented in FIFO queue in DSA.

In BFS the graph is traversed as follows:
- → First move horizontally and visit all the nodes of the current layer.
- → Move to the next layer.
- → Do not generate as child node if the node is already parent to avoid more loop. (i.e, avoid loop formation).

Example:



The traversing order of given graph in BFS is 1,2,3,4,5,6,7. (as shown by arrow in figure).

Performance Measures of BFS

Completeness → It always finds the solution if exist. If shallowest goal node is at some finite depth d and if b is finite.

Time complexity → Assume a state space where every state has b successors, then

$$O(n) = b + b^2 + b^3 + \ldots + b^d = O(b^d).$$

Space complexity → If goal node is at depth d then the nodes of d depth are expanded before traversing so space complexity of BFS is $O(b^{d+1})$.

Optimal → If all the path has same cost then optimal otherwise not.

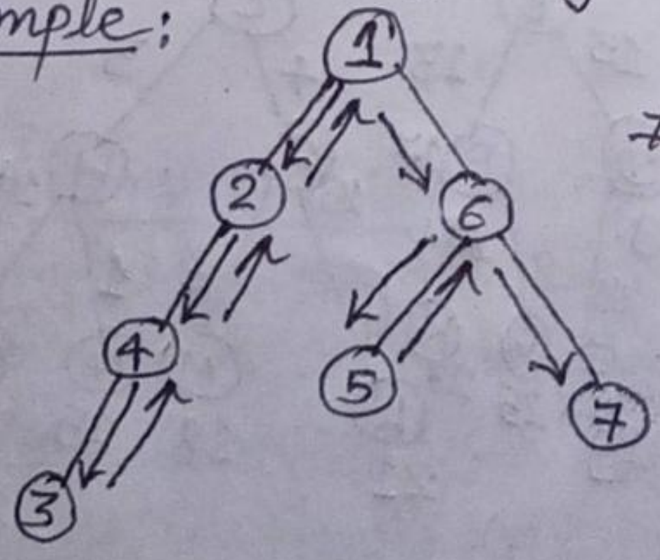
Major lessons from BFS are:
→ Memory management is the biggest issue than it's execution time.
→ Exponential complexity can not be computed in uninformed search.

Applications:
→ Social Networking Websites use BFS.
→ In GPS navigation systems.
→ Broadcasting in network.

(b) Depth First Search (DFS): It is another method for traversing or searching tree or graph data structures. This algorith starts at root node and explores as far as possible along each branch before backtracking. It is implemented in LIFO architecture (stack).

The main strategy of DFS is to explore deeper into the graph whenever possible. When all the edges have been explored, the search backtracks until it reaches an unexplored neighbor. This process continues until all the vertices are reached without forming any loop.

Example:



Traversal of DFS is 1,2,4,3,6,5,7 as shown by arrows in graph.

Performance Measure/Evaluation of DFS:

Completeness → It does not find the solution in all cases (If depth is infinite and DFS contains loops).

Time Complexity → Let m be the maximum depth, then time complexity will be

$$b+b^2+b^3+\ldots+b^m$$
$$= O(b^m).$$

Space Complexity → Need to store single node so;

$$= (b+b+b+\ldots) * m \text{ time}$$
$$= O(bm).$$

Optimality → It gives optimal solution than BFS but it does not give optimal solution if the goal node is in right child of level 2 or 3 of graph having m=10 or more.

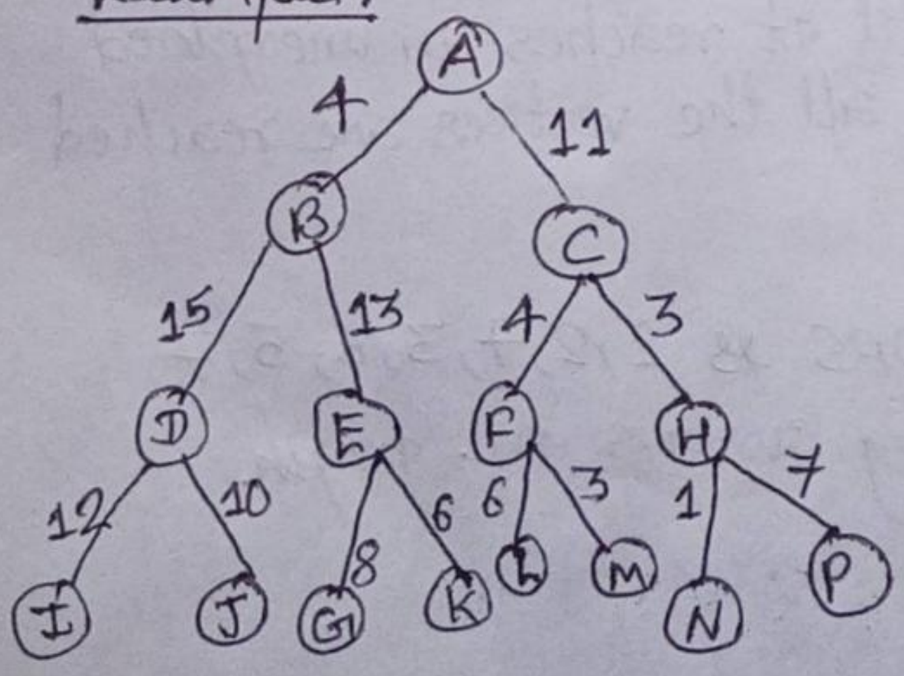
Applications:

→ Detecting cycle in a graph.
→ Path Finding.
→ Topological sorting
→ Solving puzzle with only one solution, such as mazes.

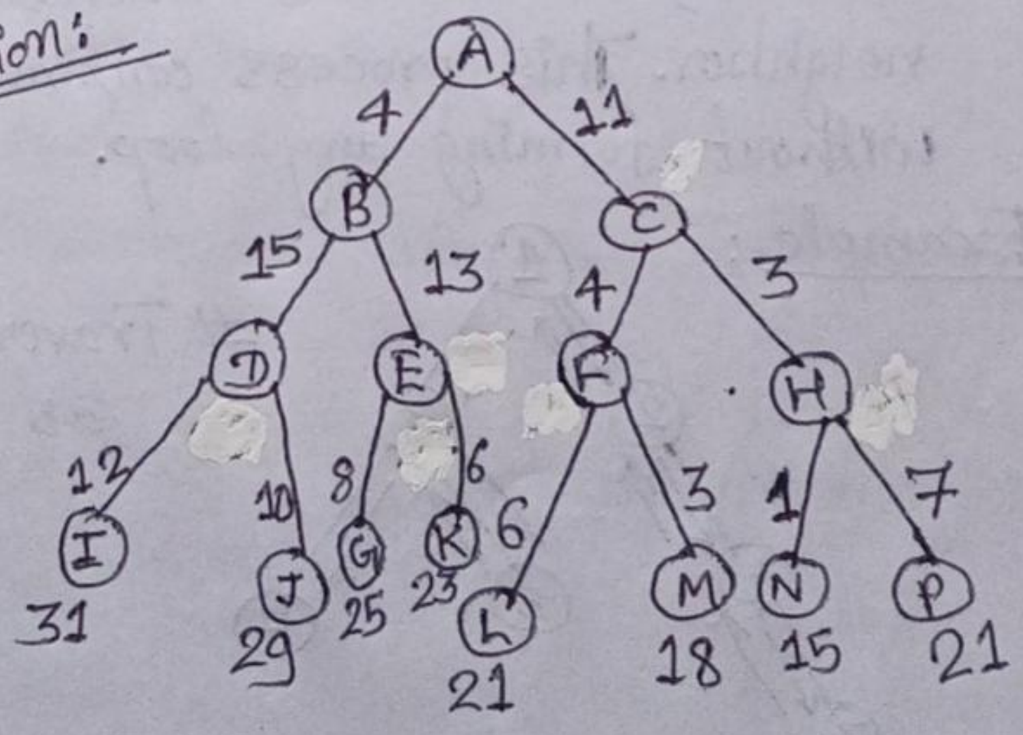
© Uniform Cost Search (UCS):-

It is the modified version of BFS to make optimal. This algorithm is used for traversing or searching a weighted tree or graph. The search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. It does not care about the number of steps, only care about the total cost.

Example:



Solution:



Performance measure of UCS:

Completeness → Yes it satisfies this property, because cost can be definitely calculated for graph with finite depth.

Space Complexity → Maximum of BFS.

~~Time~~ Time Complexity → Maximum of BFS.

Optimal → Yes, because at every step the path with the least cost is chosen.

Conclusion ⇒ UCS can be used instead of BFS in case that path cost is not equal and is guaranteed to be greater than a small positive value e.

④. Depth Limited Search (DLS):

It is the modification of depth-first search. It is an algorithm to explore the vertices of graph. It works exactly like depth-first search but avoids the drawbacks regarding completeness by imposing a maximum limit on the depth of the search. It solves the infinite-path problem of DFS.

Algorithm:

```
depth limit = max depth to search to;
agenda = initial stage;
If initial stage is goal stage then return solution
else
  while agenda not empty do
      take node from front of agenda;
      if depth(node) < depth limit then
      {
        new does = apply operations to node;
        add new nodes to front of agenda;
        if goal stage in new nodes then return solution;
      }
```

Performance measure of DLS

Completeness → In case limit $l < d$, DLS will ~~neg~~ never reach a goal, so in this case ~~incomplete bo~~ DLS is not complete.

Time complexity → $O(b^l)$

Space complexity → $O(b^l)$

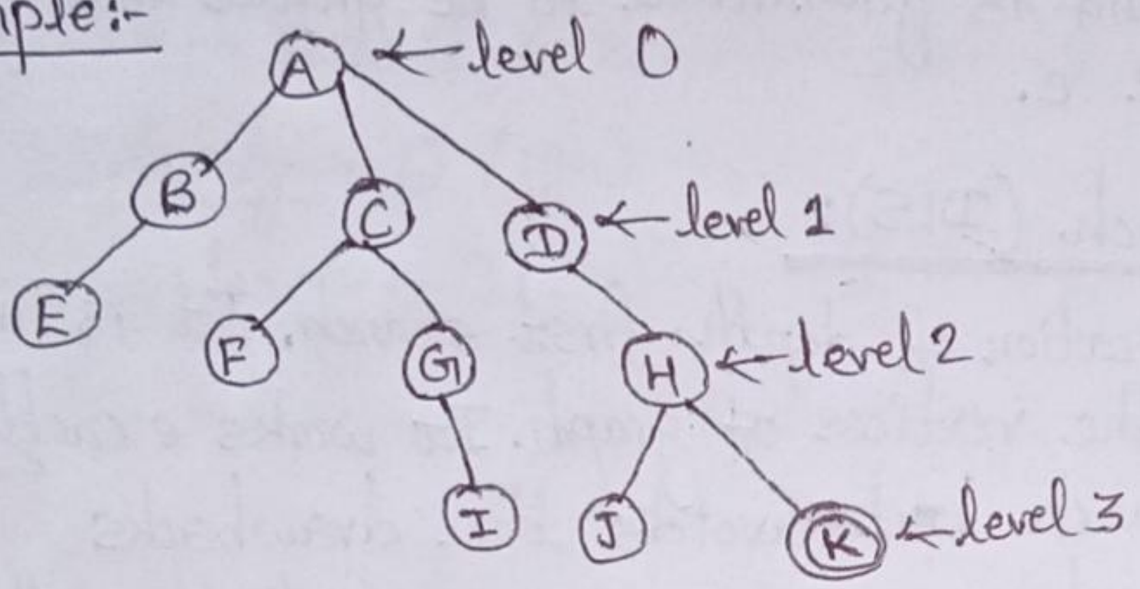
Optimality → If DLS is complete then it is optimal than DFS.

ⓔ Iterative Deeping First Search:

This technique is the combination of BFS and DFS so that the completeness and optimality of BFS and the memory requirement level of DFS could be achieved.

In this strategy, depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches d, the depth of the shallowest goal state. On each iteration this method visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited.

Example:-



Depth level	IDFS	Goal
0	A	No
1	ABCD	No
2	ABCDEFGH	No
3	ABCDEFGHIJK	Yes.

Performance Measure of IDFS:

Completeness → IDFS is like BFS, It is complete when the branching factor b is finite.

Space Complexity → $O(b^{d/2})$

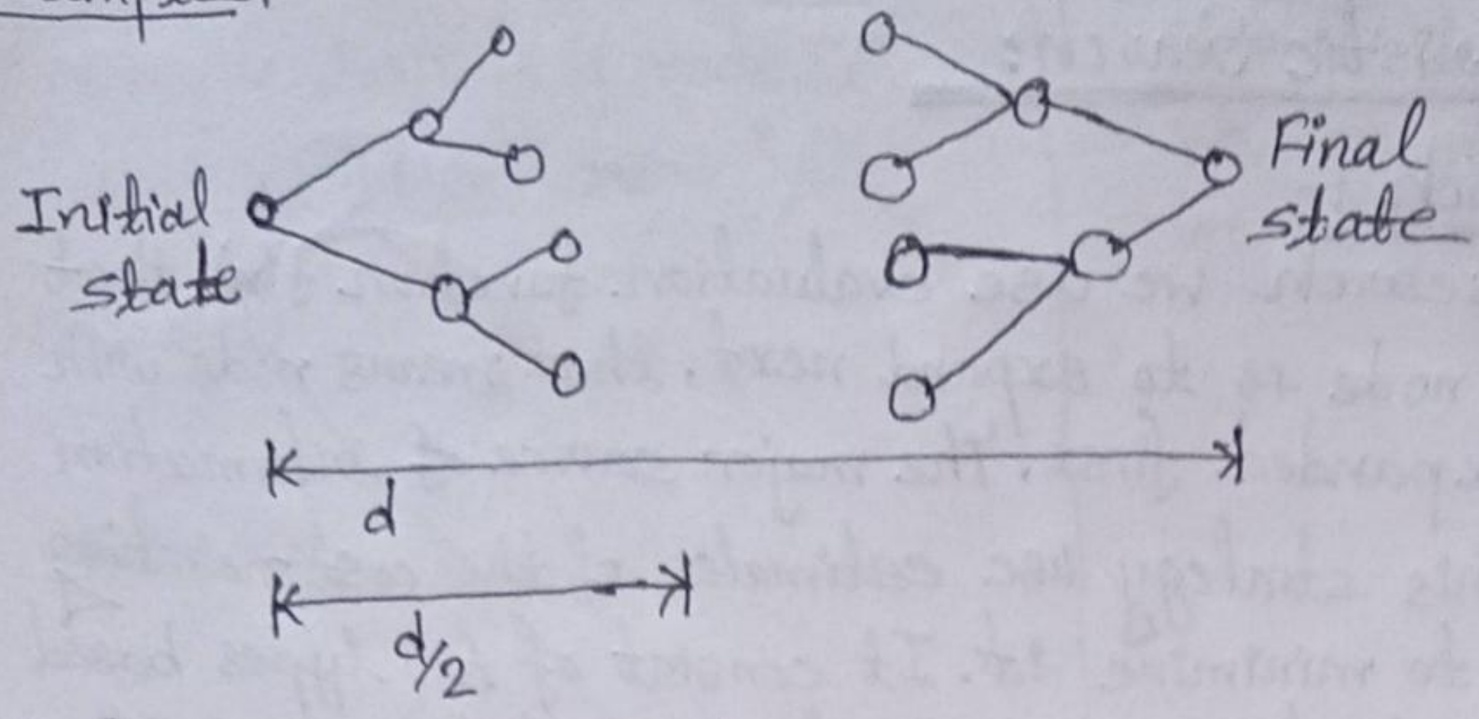
Time Complexity → $O(b^{d/2})$.

Optimality → IDFS is also like BFS, so optimal when the steps are of same cost.

ⓕ. Bidirectional Search:

Bidirectional search suggests to run two simultaneous search graph, one from the initial state and the other from the goal state. It then expands node from the start and goal state simultaneously. Check at each stage if they meet in the middle. If so, the path concatenation is the solution.

Example:



Performance measure of Bidirectional search:

Completeness → Yes it is complete when we use BFS in both searches.

Time Complexity → $O(b^{d/2})$.

Space Complexity → $O(b^{d/2})$

Optimality → Optimal when BFS is used and paths are of a uniform cost.

⊛ Comparision of All Uniformed Searches:

Strategy	Complete	Optimal	Time Complexity	Space Complexity
BFS	Yes	Yes	$O(b^d)$	$O(b^{d+1})$
DFS	No	No	$O(b^d)$	$O(bm)$
DLS	Yes(if $l>d$)	No	$O(b^m)$	$O(bm)$
IDFS	Yes	Yes	$O(b^d)$	$O(b^d)$
BDS	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

→ BFS and BDS are complete and optimal but consumes more space

→ DFS require less memory if the maximum tree depth is limited but cannot guarantee the solution.

→ DLS offers an improvement over DFS.

→ Best algorithm is IDFS or IDS, which is complete, optimal less space consumption but it cannot give solution in exponential time.

⊛. Heuristic function h(n) :-

Heuristic function h(n) estimates the "goodness" of a node n. It is an estimate based on domain-specific information that is computable from current state description, of how close we are to a goal. h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.

2) Informed or Heuristic Search:

@ Best-First Search :-

In this search we use evaluation function f(n) that determines which node is to expand next, that means node with lowest f(n) is expanded first. The major source of information for f(n) is h(n). This strategy use estimates of the cost reaching the goal and try to minimize it. It consist of two types based on the evaluation function with special cases which are as follows:-

1) Greedy Best-First Search:

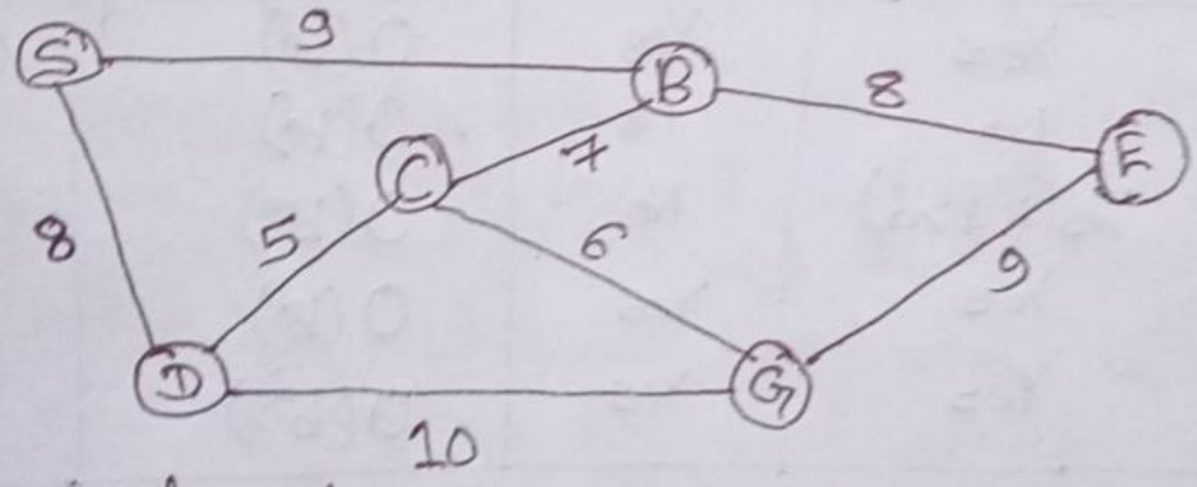
This method uses an evaluation function to select which node is to be expanded next. The node with the lowest evaluation is selected for expansion because it has the closest path to the goal.

Evaluation function $f(n) = h(n)$

$= $ ~~or~~ estimate of cost from n to goal.

This algorithm has $O(b^m)$ space and time complexity. Where b is the average branching factor and m is the maximum depth of the search tree.

Example:



Let G1 be goal node, then distances to node G1 from other node are:

S → G1 = 12 ← given values

B → G1 = 4

E → G1 = 7

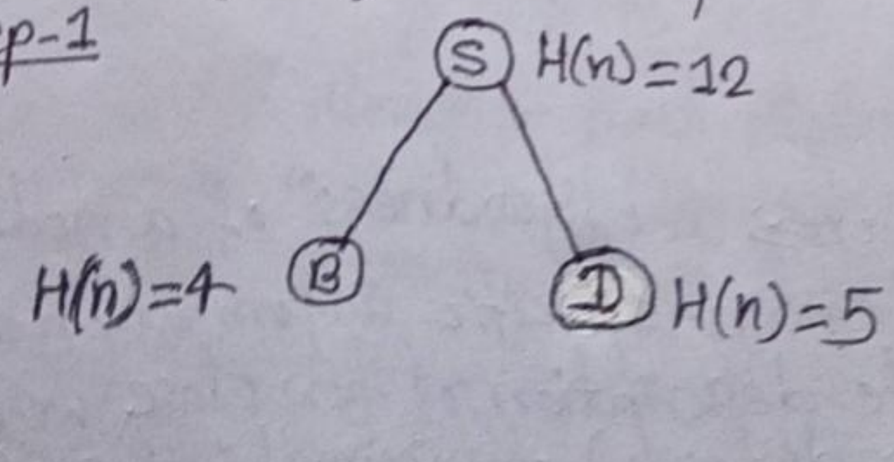
D → G1 = 5

C → G1 = 3

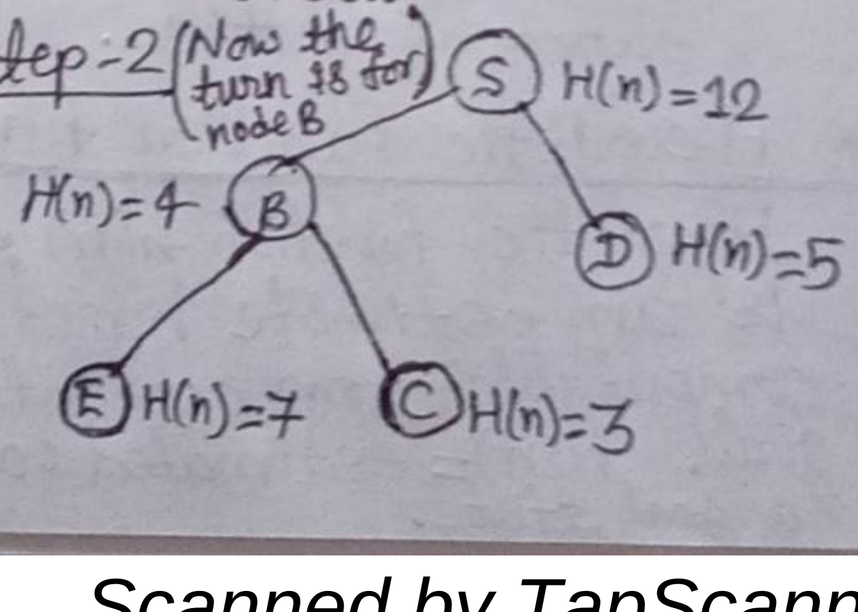
Solution:

Now greedy search operation is done as below:

Step-1

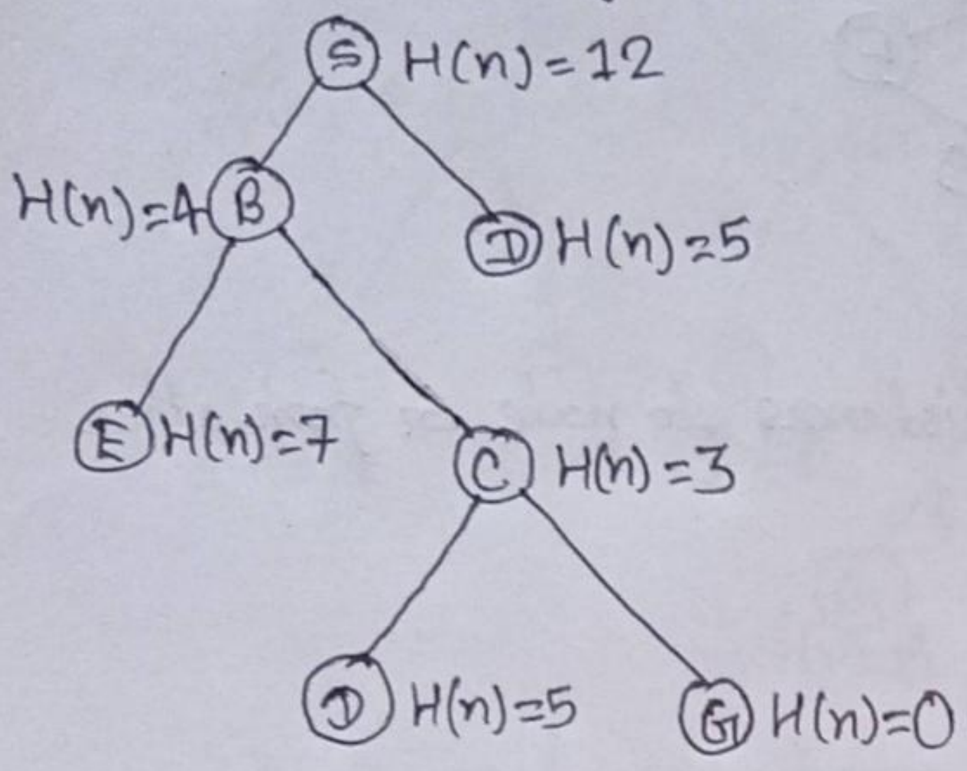


Step-2 (Now the turn is for node B)



Step:-3

Now the turn is of node C



Step:-4

Now the turn is of node G, but we reached our goal node so we stop here.

Performance Measures:

Completeness → It is not complete because while minimizing the value of heuristic greedy may oscillate between two nodes.

Time complexity → $O(b^m)$.

Space complexity → $O(b^m)$.

Optimality → It is not optimal.

2) A* Search:

Admissible Heuristic → A heuristic function is said to be admissible if it is no more than the lowest-cost path to the goal. It is used to estimate the cost of reaching the goal state in an informed search algorithm. The estimated cost must always be lower than the actual cost of reaching the goal state.

A* search uses heuristic function $h(n)$ as well as the path cost to the node $g(n)$ in computing the total cost $f(n)$.

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

where, ~~h(n) within the f(n) & must be admissible heuristic~~.

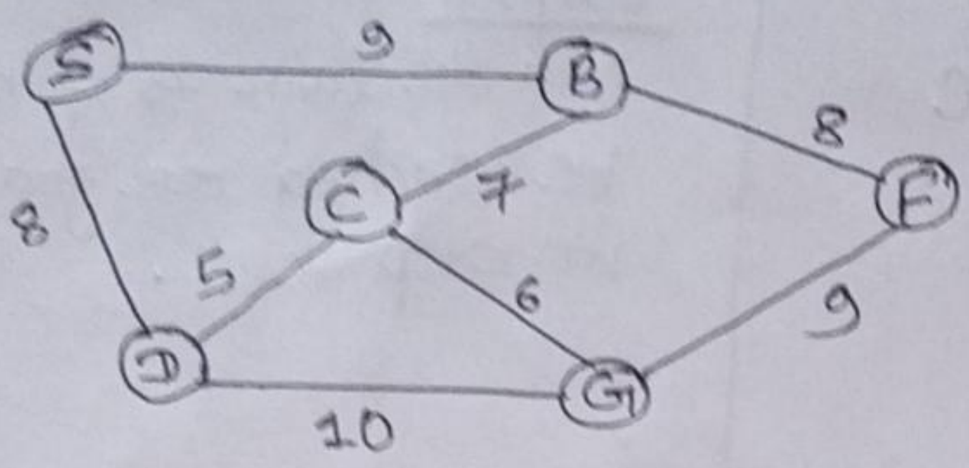
$f(n)$ = the evaluation function.

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

$g(n)$ = the cost from the start node to current node.

This method follows a path of the lowest known path keeping a sorted priority queue of alternate path segments along the way. If path being traversed has higher cost than another path then it neglects the higher cost path and traverses lower cost path segment. This process continues until goal is reached.

Example:

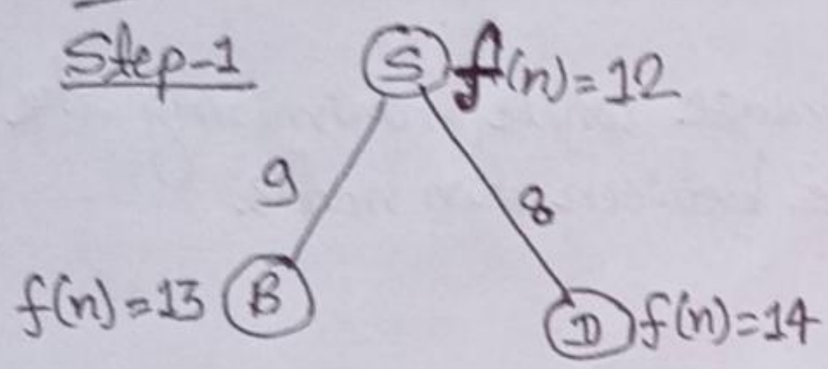


Let G1 be the goal node, then distances to node G1 from other node are:

$S \rightarrow G1 = 12$
$B \rightarrow G1 = 4$
$E \Rightarrow G1 = 7$
$D \rightarrow G1 = 6$
$C \rightarrow G1 = 3$

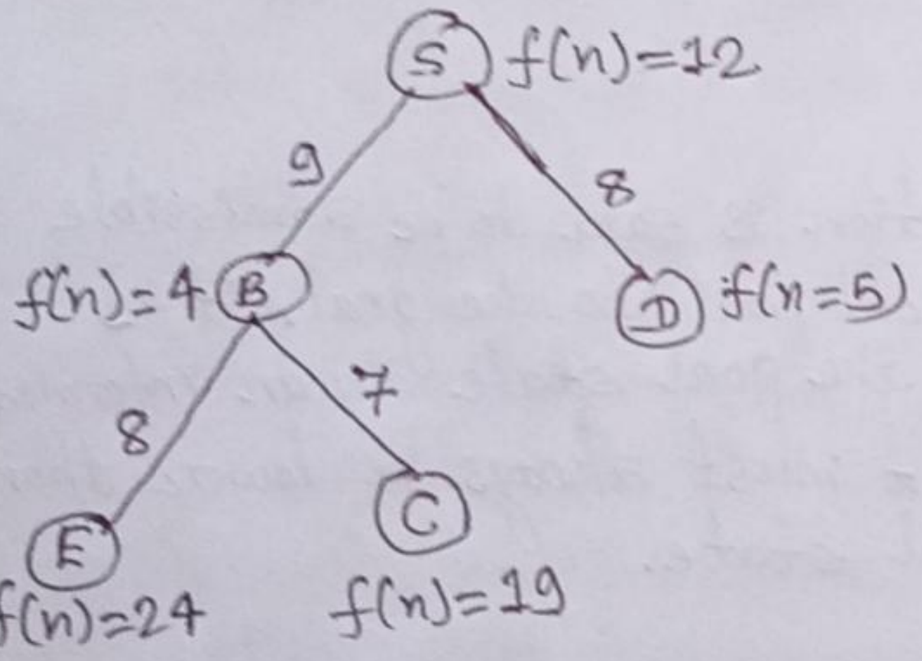
Solution:

Step-1



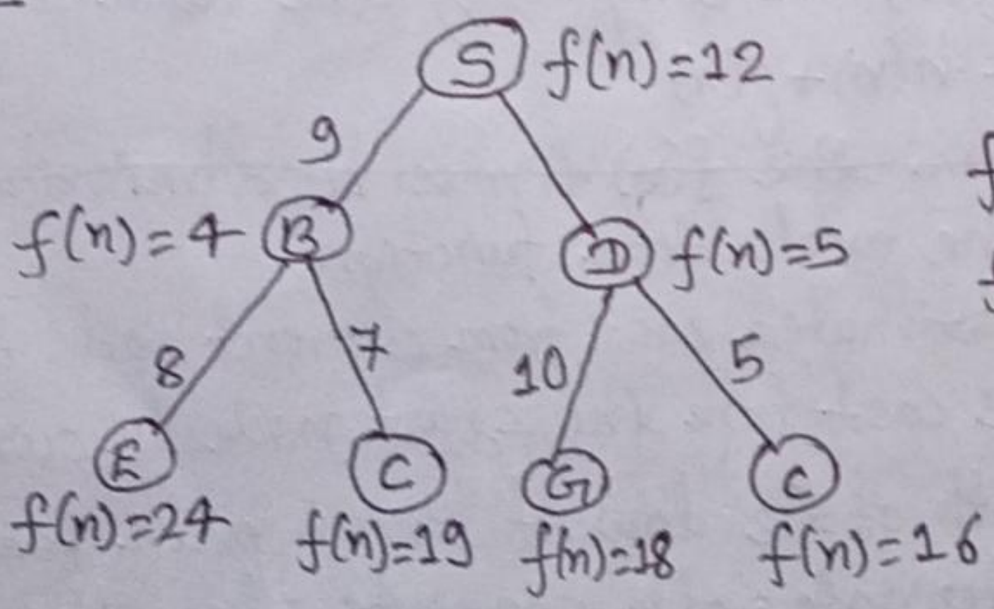
$f(B) = 9+4 = 13$
$f(D) = 8+6 = 14$

Step-2: Now the turn is of node B.



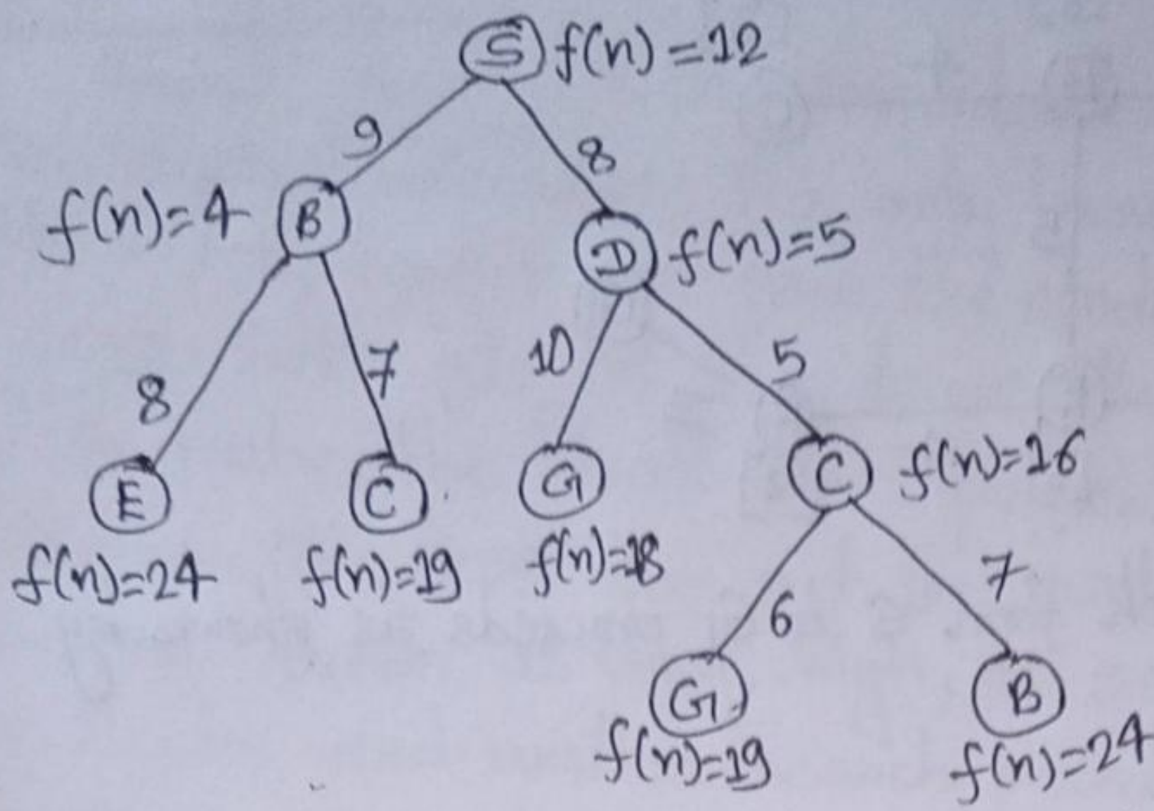
$f(E) = 17+7 = 24$
$f(C) = 16+3 = 19.$

Step-3: Now the turn is of node D.



$f(G1) = 18+0 = 18$
$f(C) = 13+3 = 16$

Step:-4 Now turn is of node C.



$f(G1) = 19 + 0 = 19$
$f(C) = 20 + 4 = 24$

G1 is goal node, hence we are done.

Evaluation/Performance Measures:-

Completeness → Yes A* seach always give us solution.

Time Complexity → $O(b^d)$.

Space Complexity → It keeps all generated nodes in memory. Hence space is the major problem not time.

Optimality → A* gives optimal solution.

Admissibility → A* is admissible and considers fewer nodes than any other admissible search algorithm with the same heuristic.

(b) Hill Climbing Search:-

This search technique can be used to solve problems that have many solutions, some of which are better than others. It starts with a random solution, and iteratively makes small changes to the solution, each time improving a little. When the algorithm cannot see any improvement anymore, it terminates. It can be described as follows:-

i) Evaluate the initial state, if it is goal state then terminate. Otherwise, current state is initial state.

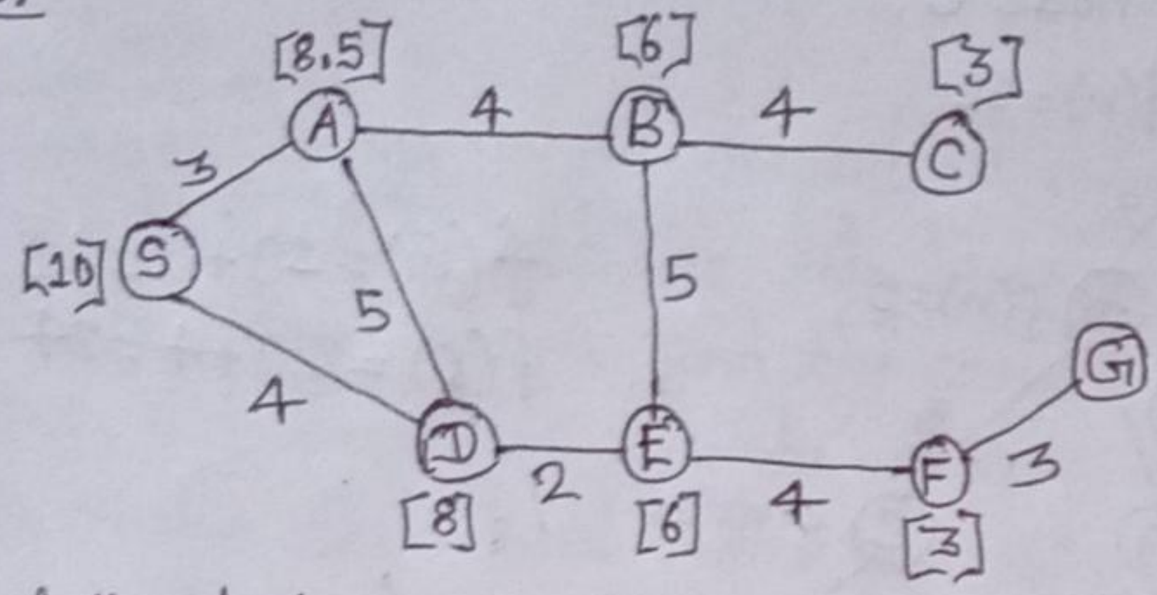
ii) Select a new operator for this state and generate a new state.

iii) Evaluate the new state
   → If it is closer to goal, make it current state.
   → If it is no better, ignore.

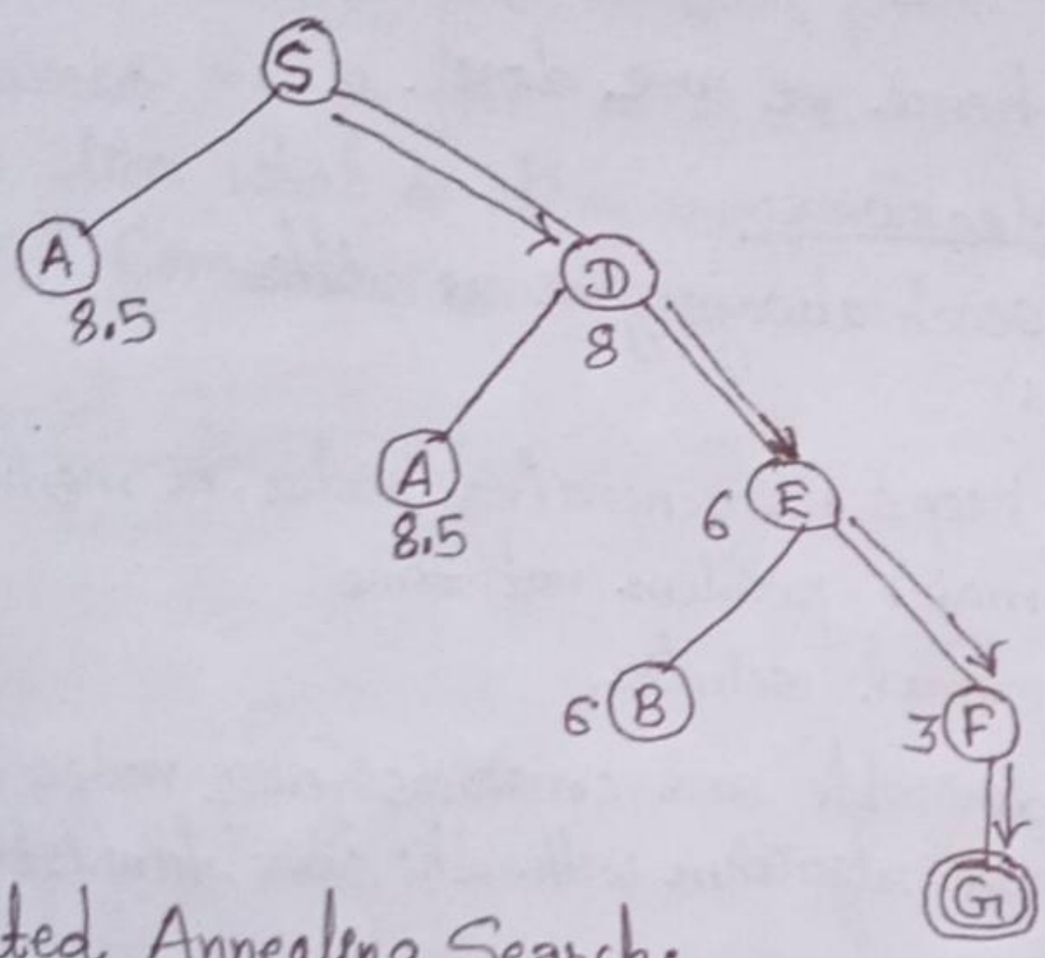
iv) If the current state is goal state then terminate. Otherwise repeat from (ii).

Hill climbing terminates when there are no successors of the current state which are better than the current state itself.

Example:



The hill climbing search from S to G proceeds as shown by arrow as follows:



© Simulated Annealing Search:

Compared to hill climbing the main difference is that simulated annealing allows downwards steps. It also differs from hill climbing that a move is selected at random and decides whether to accept it. If the move is better than its current position then it will be accepted. 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 equation;

$$P = \text{exponential}(-c/t) > r$$

where, c = the change in the evaluation function
t = the current value
r = a random number between 0 and 1.

⊛. Game Playing:

Game Playing is an important domain of AI. Games don't require much knowledge; The only knowledge we need to provide is the rules, legal moves and the conditions of winning or losing the game. Both players try to win the game. So both of them try to make the best move possible at each turn. Searching techniques like BFS are not accurate for this as the branching factor is very high, so searching will take a lot of time. So, we need another search procedures that improves searching time. Games are a form of multi-agent environment and arise questions like what do other agents do and how they affect our success? Competitive multi-agent environments give rise to games.

⊛ Adversarial Search Techniques:

Adversarial search is also known as Minimax search used in calculating the best move in two player games where all the information is available, such as chess or tic tac toe. It consists of navigating through a tree which captures all the possible moves in the game, where each move is represented in terms of loss and gain for one of the players.

Game-adversary

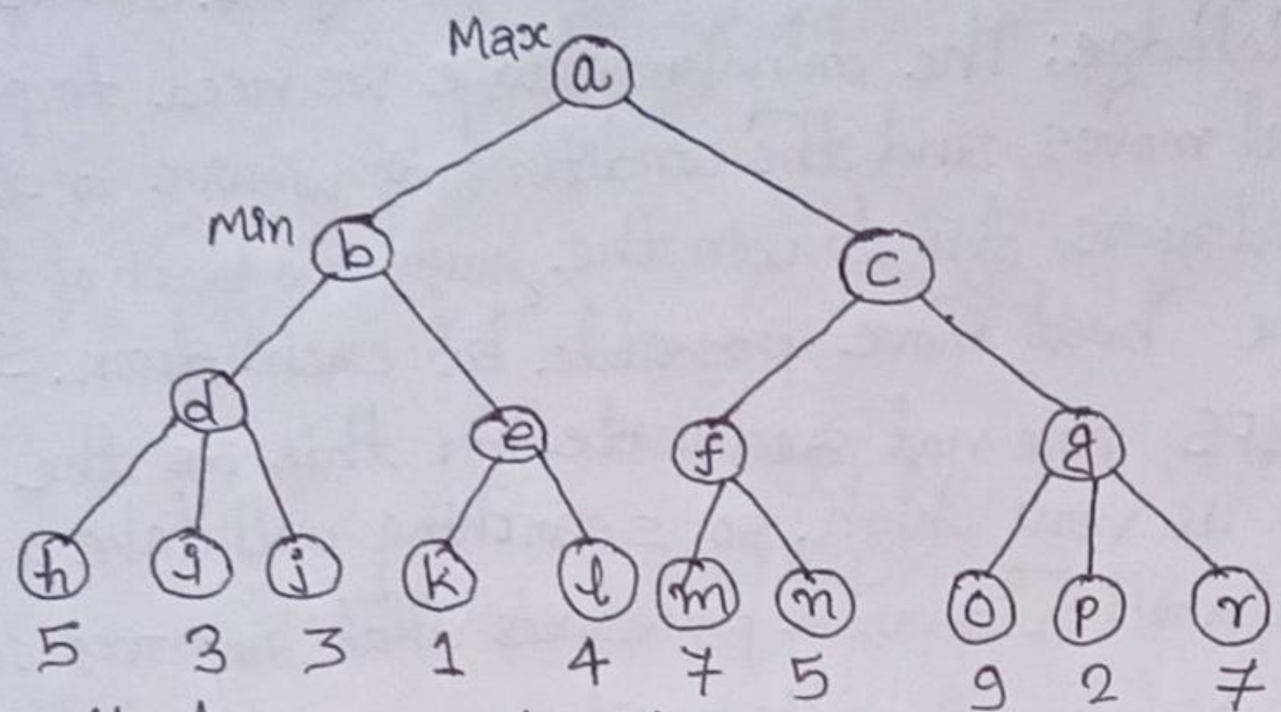
→ Solution is strategy (strategy specifies move for every possible opponent reply).
→ Time limits force an approximate solution.
→ Evaluation function: evaluate "goodness" of game position.
→ Examples: chess, checkers, backgammon.

1> The Mini-max Search:

Let us assign the following value for the game: 1 for win by X, 0 for draw by X, -1 for loss by X. Given the values of the terminal nodes (win for X(1), loss for X(-1), draw for X(0)), now the values of the non-terminal nodes are computed as follows:-

→ The value of a node where it is the turn of player X to move is the maximum of the values of its successors (because X tries to maximize its outcome).
→ The value of a node where it is the turn of player O to move is the minimum of the values of its successors (because O tries to minimize the outcome of X).

Example: Consider the following game tree (drawn from the point of view of the maximizing player):



Show that moves should be chosen by the two players, assuming that both are using mini-max procedure.

Solution:

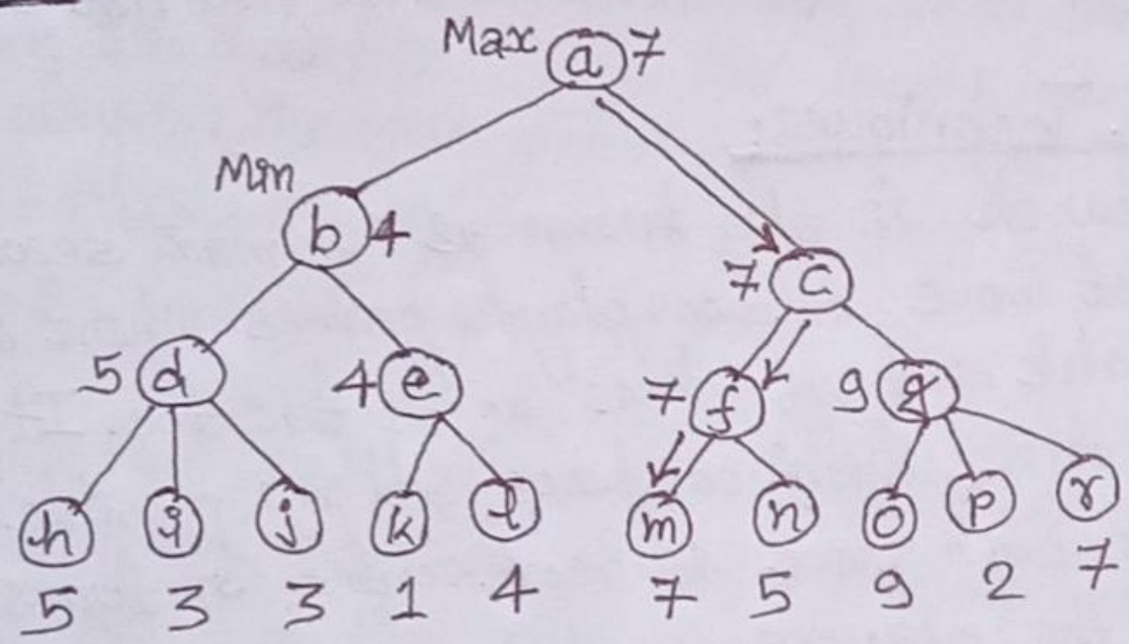


Fig. The min-max path for the game tree

2) Alpha-Beta Pruning:

This method is applied to a standard minmax tree, it returns the same move as minmax would, but prunes away branches that can not possibly influence the final decision. Alpha-Beta ~~tuning~~ Pruning can be applied to trees of any depth and it is often possible to prune entire sub-trees rather than just leaves. This is a technique for evaluating nodes of a game tree that eliminates unnecessary evaluations. It uses two parameters alpha and beta.

Alpha → It is the value of the best (i.e, highest value) choice we have found so far at any choice along the path for MAX.

Beta → It is the value of the best (i.e, lowest value) choice we have found so far at any choice along the path for MIN.

An alpha cutoff → Let us consider that the current board situation corresponds to the node A in the following figure:

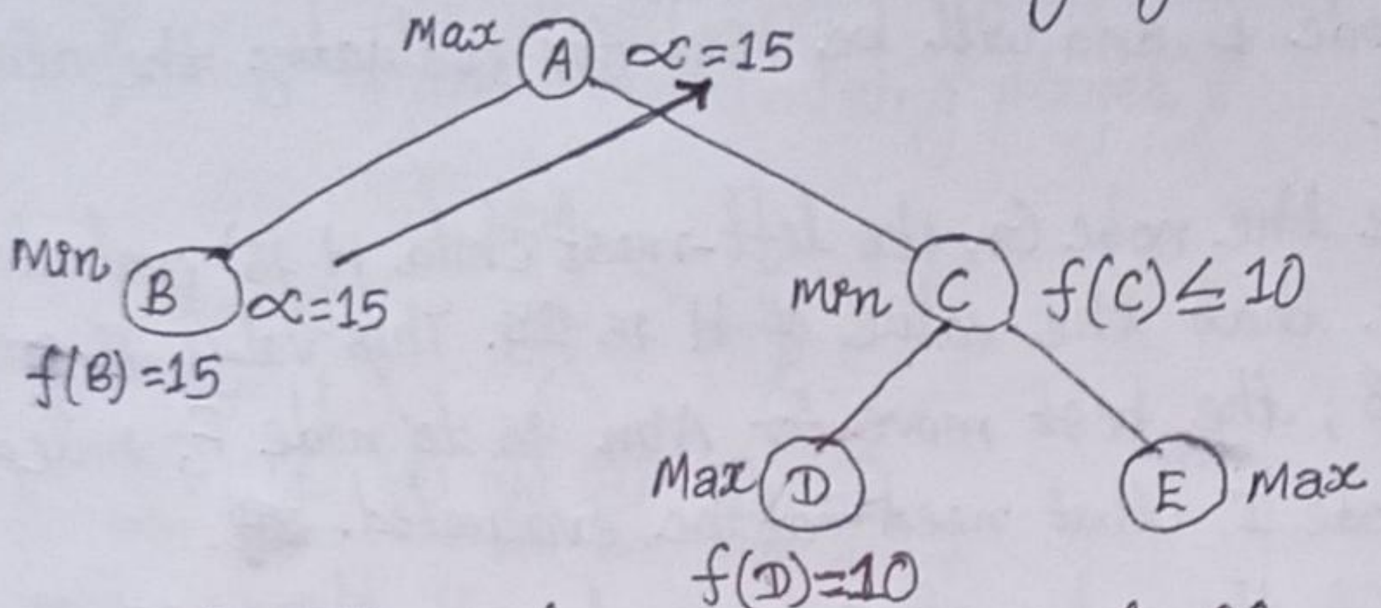


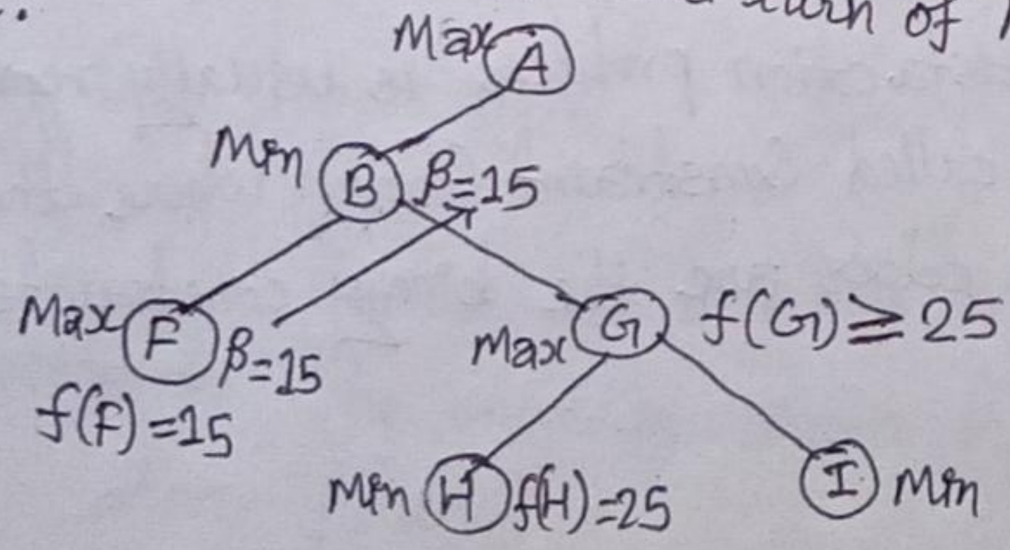
fig. Illustration of the alpha cut-off

The mini-max method uses a depth-first search strategy therefore it will estimate the value of node B first. Let node ~~be~~ B be evaluated 15. If Max will move to B then it is guaranteed to achieve 15. Therefore 15 is the lower bound for the achievement of max player. Therefore value of α at node B is 15 which is transmitted upward to node A and will be used for evaluating the other possible moves from A.

To evaluate the C, it's left-most child D has to be evaluated first. Let us assume that value of D is 10. This value is less than the value of α, the best move for MAX is to node B, independent of the value of node E that need not be evaluated.

If the value of E is greater than 10, Min will move to D which has the value 10 for Max, otherwise, Min will move to E which has a value less than 10. So, if Max moves to C, the best it can get is 10; which is less than the value $\alpha=15$. that we get if Max would move to B. Therefore, the best move for Max is to B, independent of the value of E. The Elimination of node E is an alpha cutoff.

A beta cutoff → It represents an upper bound for the achievement of the Max player. In above tree, the Max player moved to node B. Now it is the turn of Min player to decide where to move:



Let us assume that the value of F has be evaluated to 15. Therefore the value of β at the node F is 15. This value is transmitted upward to node B and will be used for evaluating the other possible moves from B.

To evaluate the node G, it's left-most child H is evaluated first. let us assume that the value of H is 25. This value is greater than the value of β, the best move for Min is to node F, independent of value of node I that need not be evaluated.

If the value of I is greater or equal to 25, then Max(In G) will move to I otherwise Max will move to H. So in both cases the value obtained by Max is at least 25 which is greater than β. Therefore the best move for Min is at F, independent of the value of I. The elimination of the node I is a beta cutoff.

⊛. Constraint Satisfaction Problems: (With Examples):

A Constraint Satisfaction Problem (CSP) consists of a set of variables, a domain of values for each variable and a set of constraints. The objective is to assign a value for each variable such that all constraints are satisfied.
Examples: The n-Queen problem, Crossword puzzle, Sudoku etc.

A Constraint Satisfaction Problem is characterized by:
→ a set of variables $\{x1, x2, \ldots, xn\}$.
→ for each variable xi a domain Di with the possible values for that variable, and a set of constraints that are assumed to hold between the values of variables.

A Constraint Satisfaction Problem is to find, for each q from 1 to n, a value in Di for xi so that all constraints are satisfied.

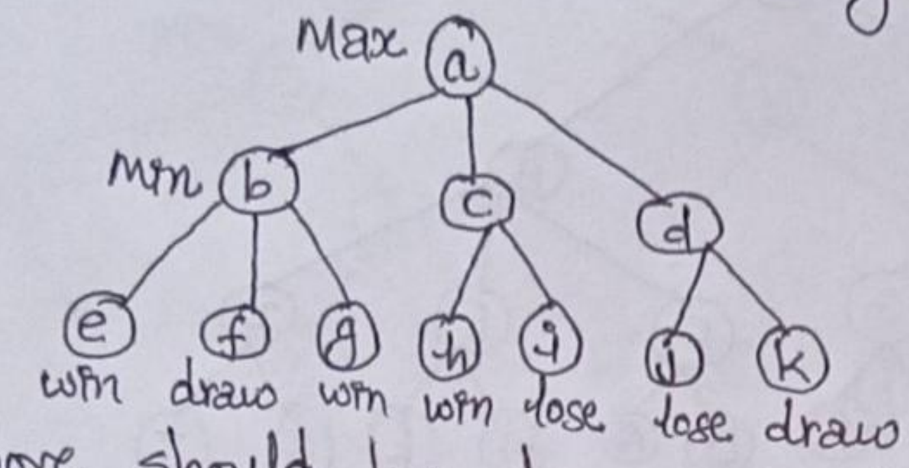
This problem can be easily stated as a sentence in first order logic of the form:

$$(\text{exist } 1) \ldots (\text{exist } xn)\ (D1(x1) \& \ldots Dn(xn)) \Rightarrow (C1 \ldots Cm)$$

A Constraint Satisfaction problem is usually represented as an undirected graph called Constraint Graph where the nodes are the variables and the edges are the binary constraints.

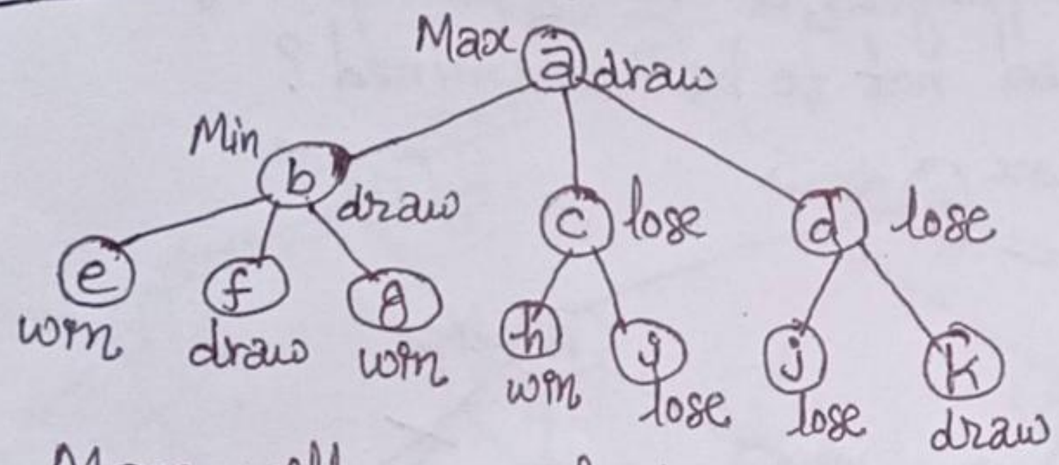
Examples: (Important)

Example 1: Consider the following game tree (drawn from the point of view of the Maximizing player):



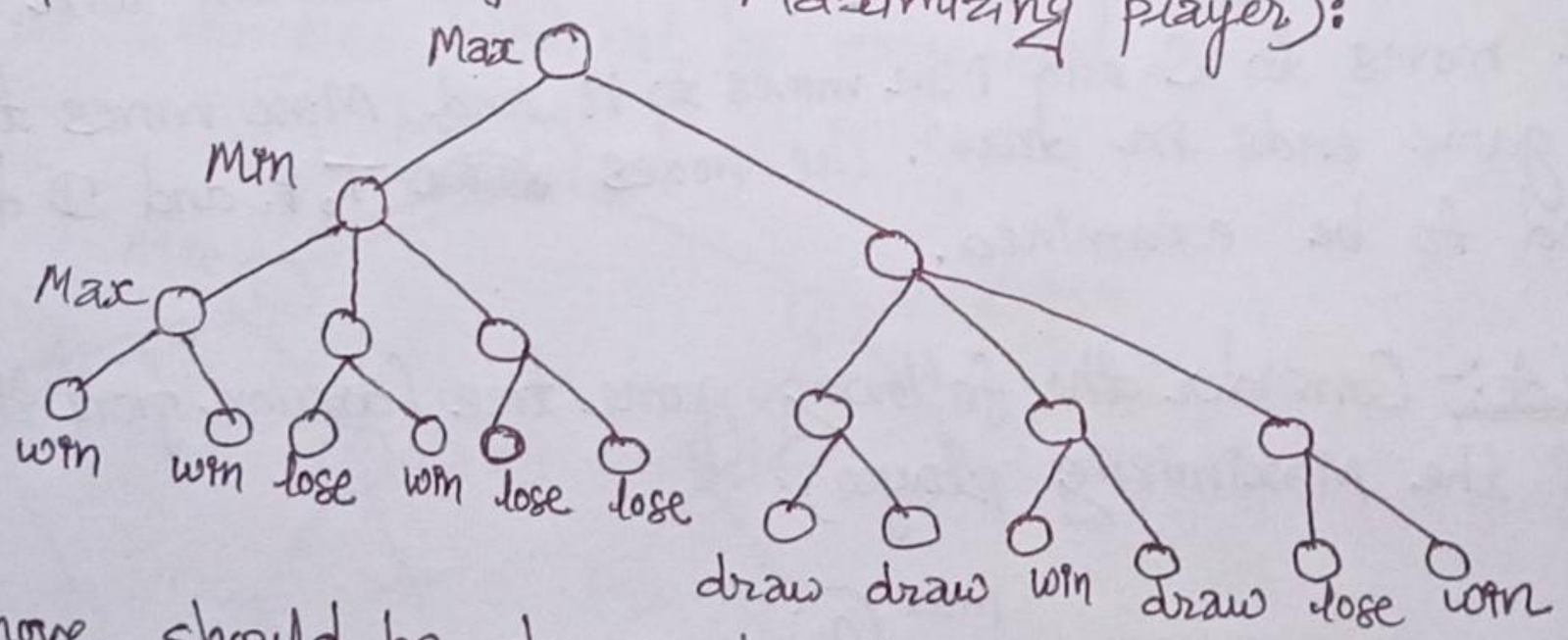
What move should be choosen by the Max player, and what should be the response of the Min player, assuming that both are using the mini-max procedure?

Solution:



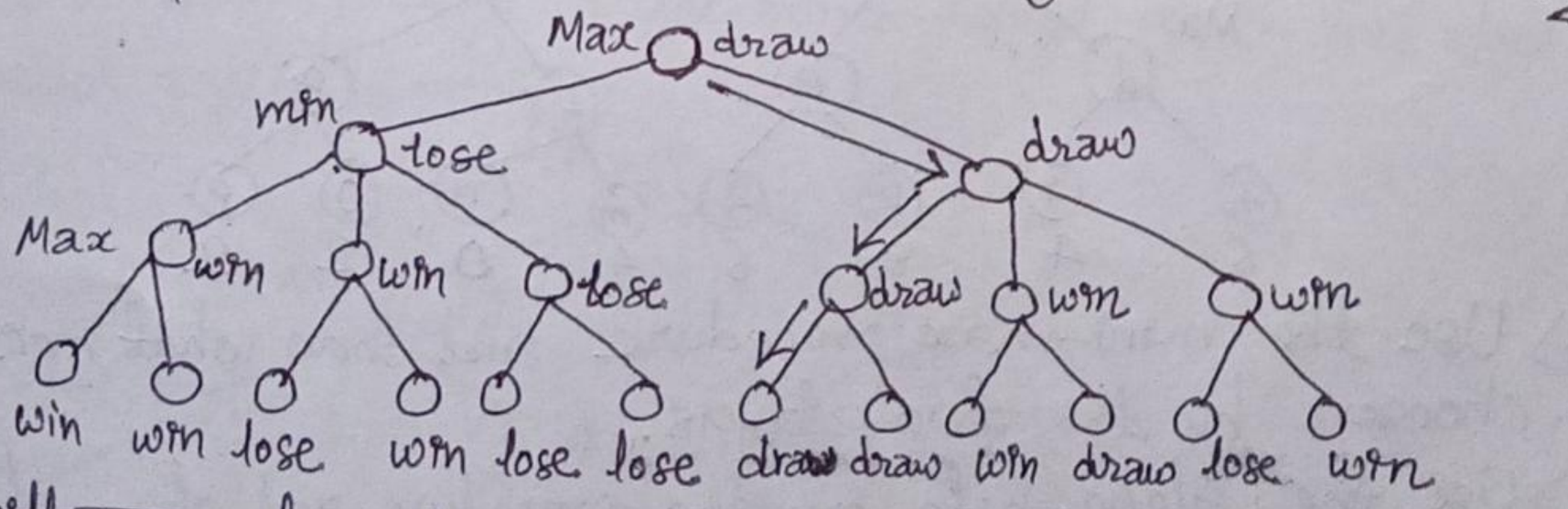
Max will move to b and min will respond by moving to f.

Example 2: Consider the following game tree ~~(done~~ drawn from the point of view of the Maximizing player):



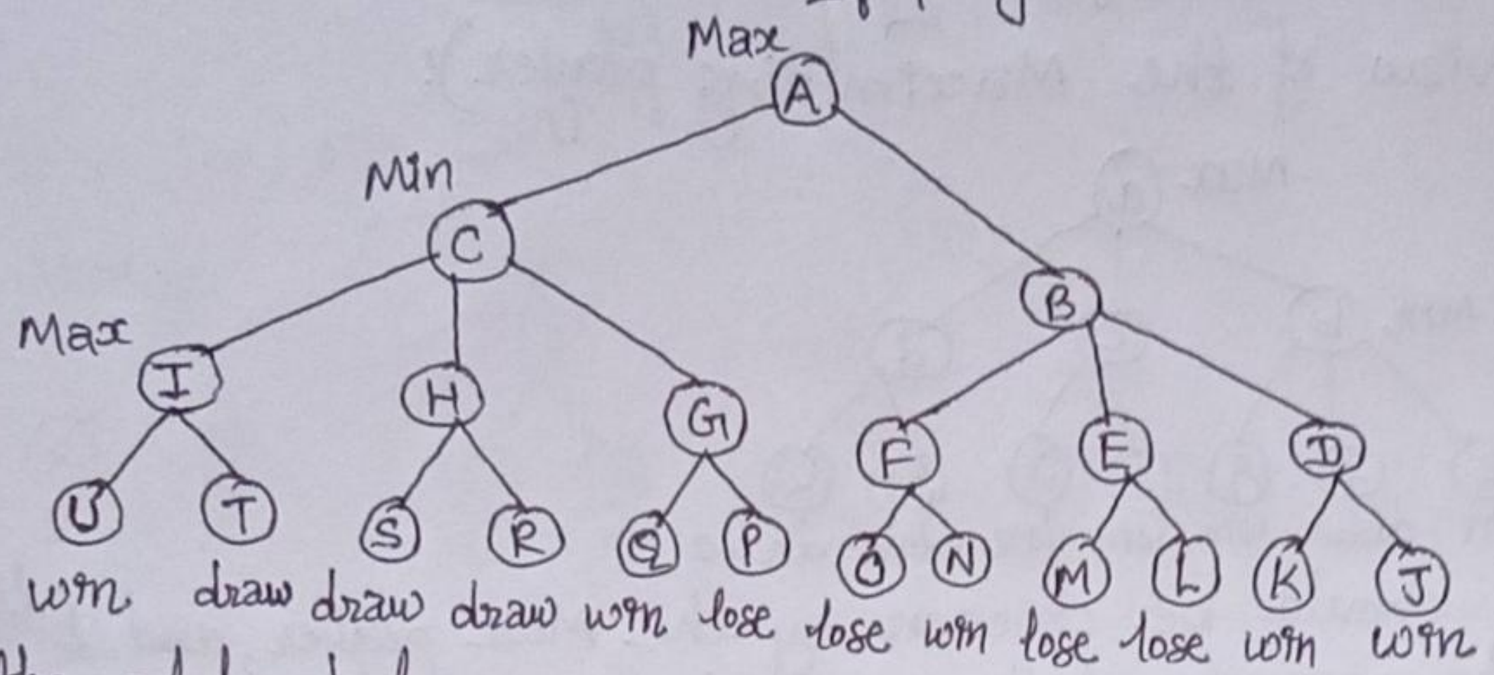
What move should be choosen by the Max player, and what should be the response of the Min player, assuming that both are using the mini-max procedure?

Solution:



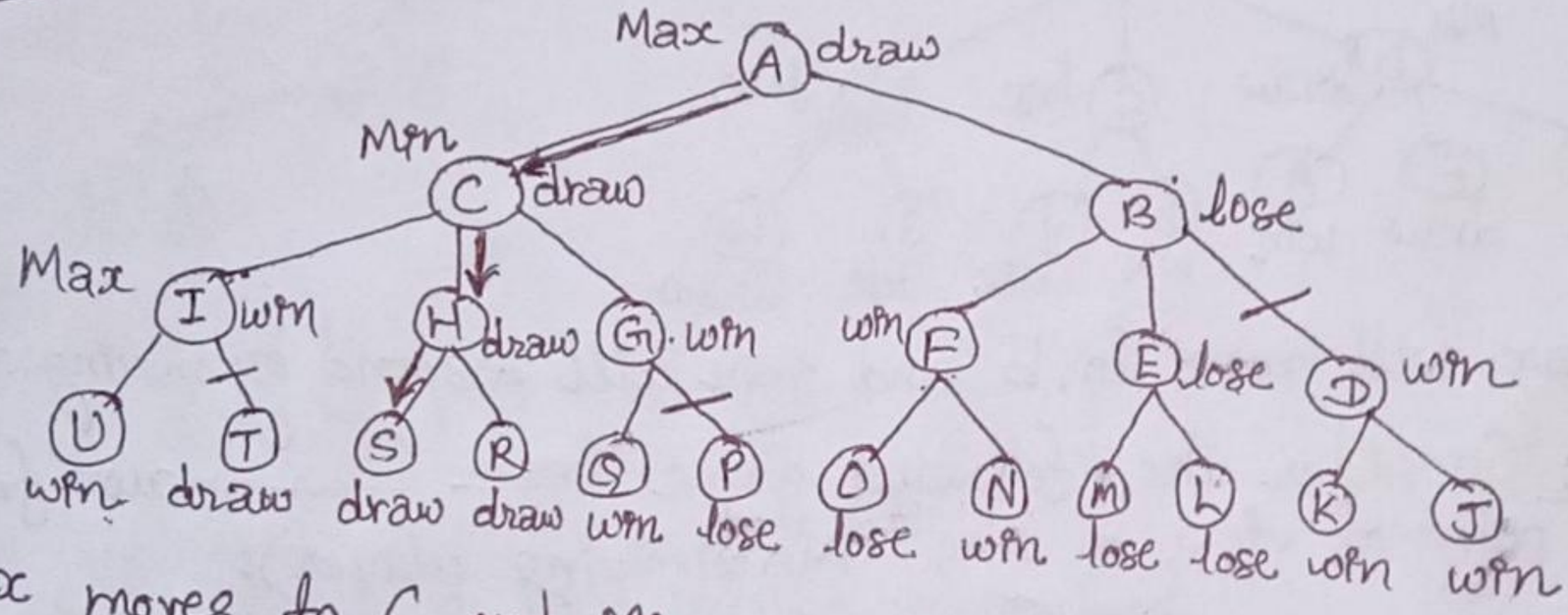
~~Max will move to~~ The move is shown by the arrow in above figure.

Example 3:- Consider the following two-person game tree, drawn from the point of view of the Maximizing player:



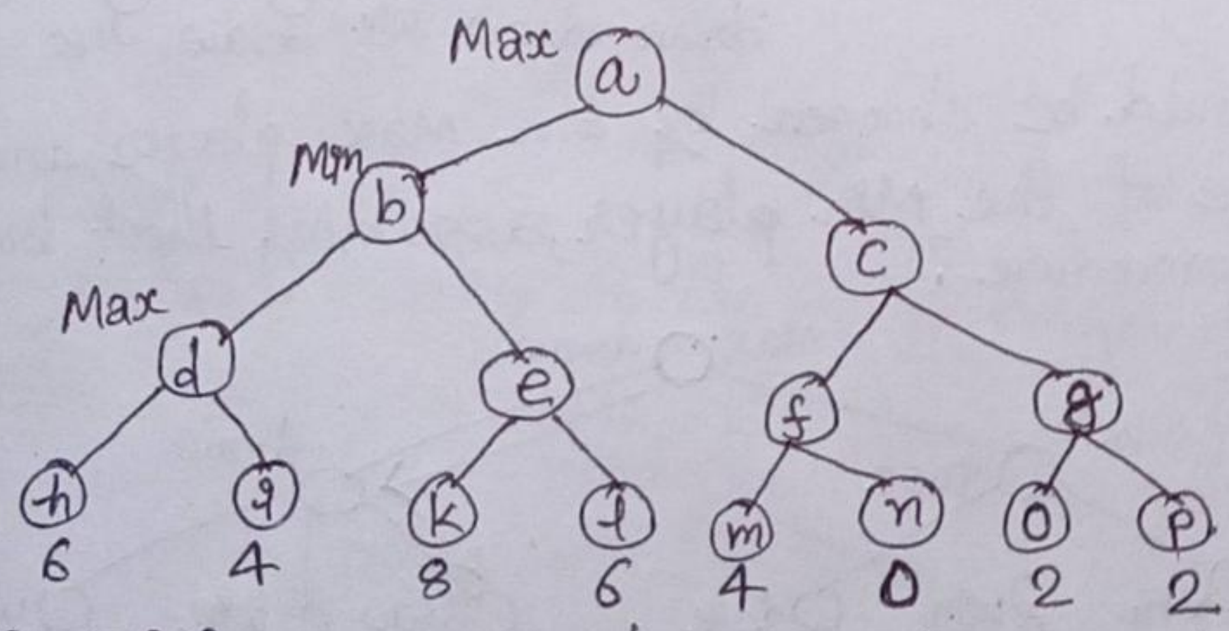
Use the alpha-beta ~~pruning~~ pruning procedure to determine the moves which should be choosen by two players, assuming a depth-first search of the tree. Which nodes need not to be examined?

Solution:



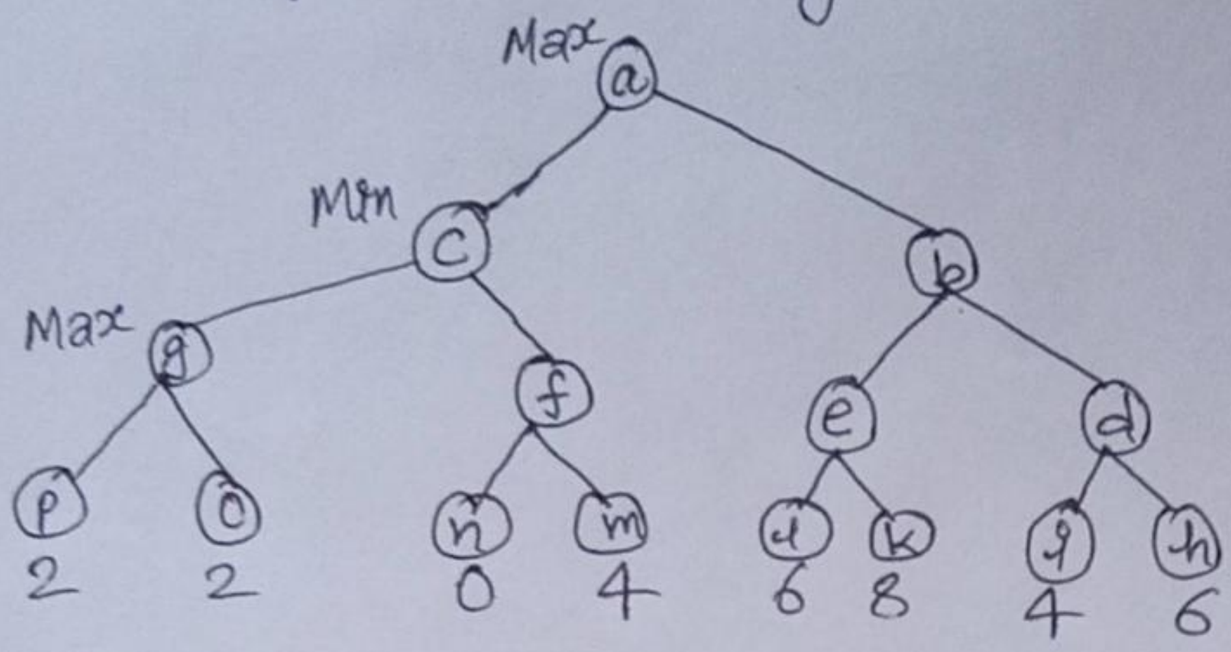
Max moves to C and Min moves to H and Max moves to S or R. The game ends in draw. The nodes ~~draw~~ T, P. and D does not need to be examined.

Example 4:- Consider the following game tree (drawn from the point of view of the Maximizing player):



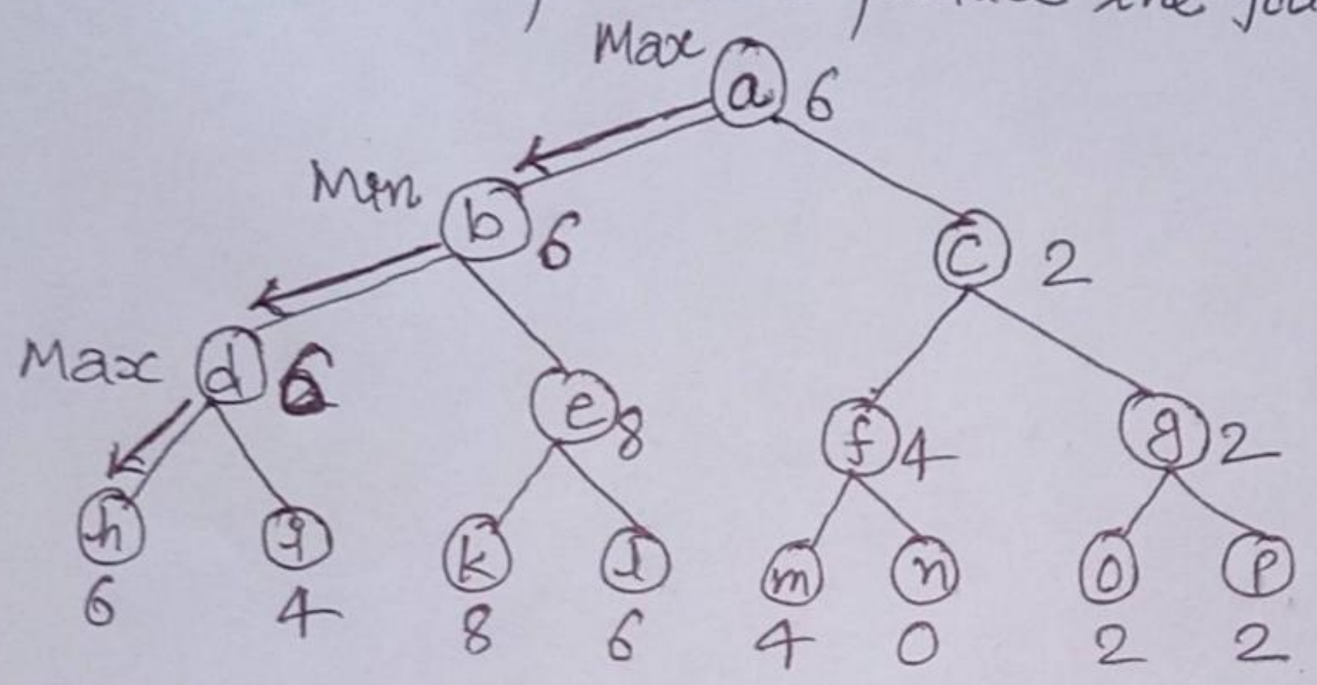
a) Use the mini-max procedure and show what moves should be choosen by the two players.
b) Use the alpha-beta pruning procedure and show what nodes would not need to be examined.
c) Consider the mirror image of above tree and apply again the alpha-beta pruning procedure. What do you notice?

The mirror image is the following tree:

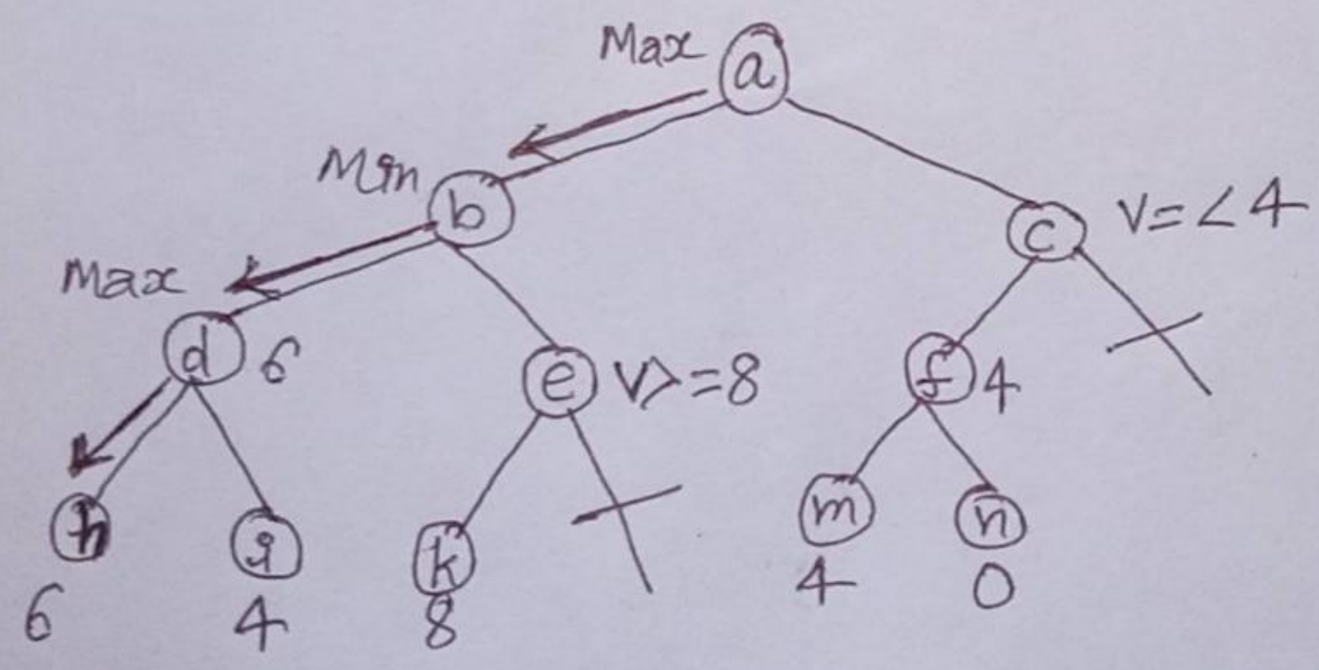


Solution:

a) The mini-max procedure produce the following tree.



b) The cut branches need not be examined.



c) Using alpha-beta pruning on the mirror image of the tree does not produce any savings: