A* search

* It evaluates nodes by combining $g(n)$, the ~~cost~~ to reach the node, and $h(n)$, the cost to get from the node to the goal : $f(n) = g(n) + h(n)$.

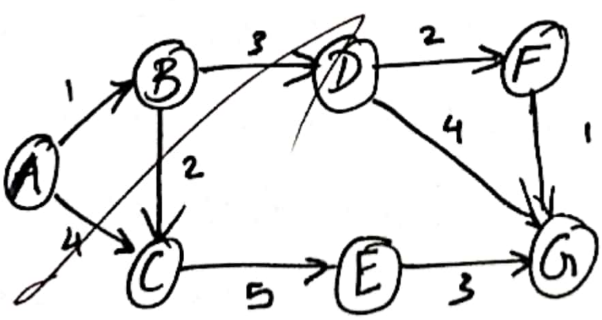
Algorithm of A* search:

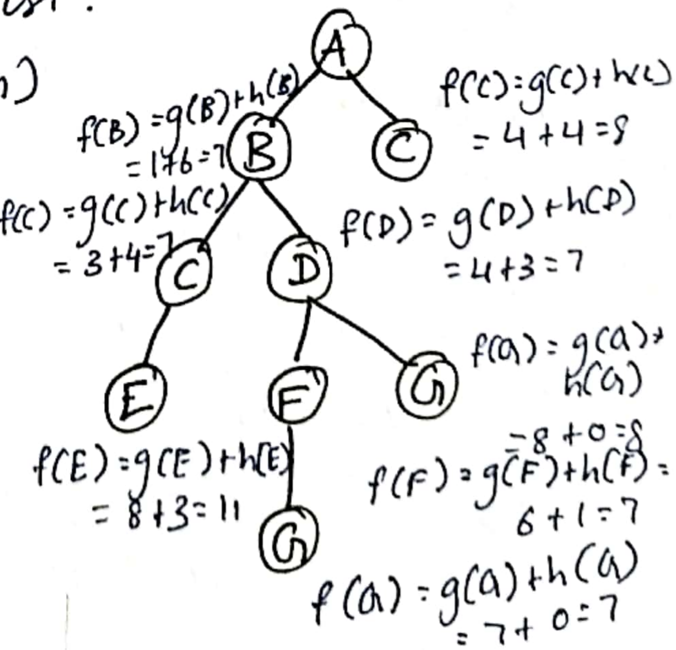
Step 1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function $(g+h)$, if node n is goal node then return success and stop, otherwise.

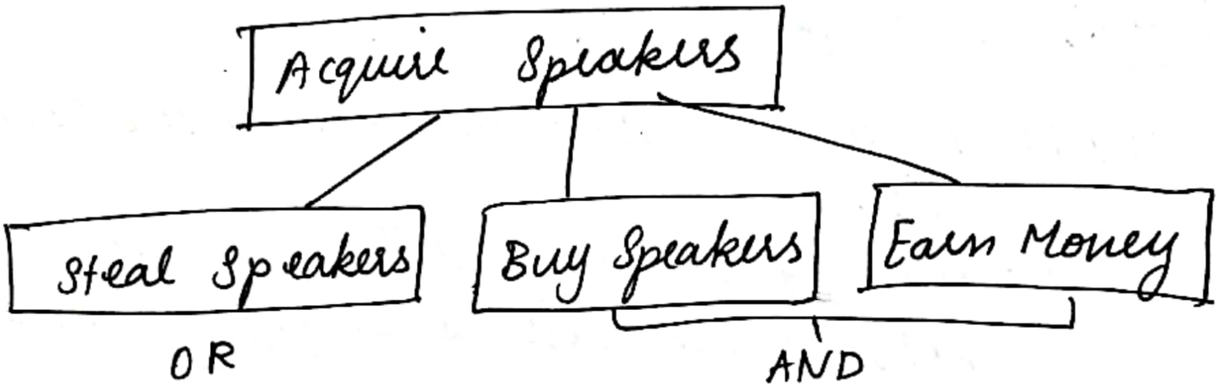
$$f(n) = g(n) + h(n)$$





AO* Algorithm:

* It is best search Algorithm.
* It uses the concept of AND-OR graphs to decompose any complex problem given into smaller set of problems which are further solved.



* Compared to the A* search algorithm, AO* algorithm is very efficient in searching the AND-OR trees very efficiently.

Algorithm :-

i) Create an initial graph with a single node.

ii) Transverse the graph following the current path. accumulating node that has not yet been expanded or solved.

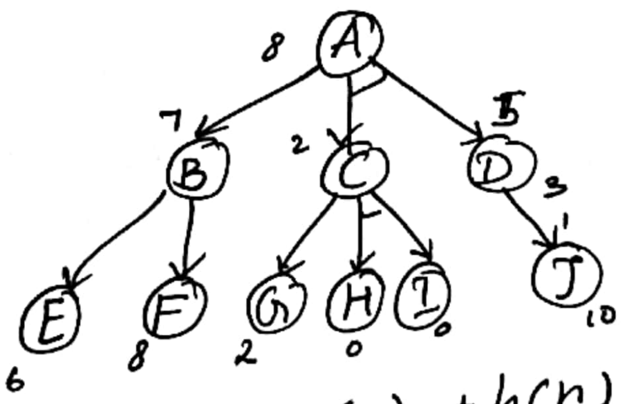
iii) Select any of these nodes and explore it. If it has no successors then call this value

(iv) If $f(n) = 0$, when mark the node as solved.

(v) Change the value of $f'(n)$ for the newly created node reflect its successors by back propagation.

(vi) whenever possible use the most promising routes, if a node is marked as solved then mark the parent node as solved.

(vii) If starting node is solved or value is greater then Futility then stop else repeat from step 2.



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

$$f(A-C-D) = 1 + 2 + 1 + 1 = 5$$

10/10

Assignment - 2

Constraint Satisfaction Problem :

A CSP is a problem defined by :

X : A set of variable

D - Domains for each variable

C : Constraints specifiying allowable combination of values.

A solution is a complete assignment of values to variable that satisfies all constraints.

Type of Assignment :-

i) Consistent / Legal : Doesn't violate any constraint.

ii) Complete : Every variable has a value. constraints are satisfied.

iii) Partial : Only some variables are assigned

Types of Domains :-

i) Discrete : Infinite possibilities (eg, any integers)

ii) Finite / continuous : Limited specific value

Types of Constraints :-

i) Unary : Apply to one variable

ii) Binary : Relate two variable

Cons

Scanned with CamScanner

Constraint Propogation :-

i) Node consistency :- unary constraints satisfied.

ii) Arc Consistency :- Every value has a consistent counterpart.

iii) Path consistency :- Binary constraints over variable triplets.

iv) K - consistency :- Higher- level consistency

Common CSP Examples :-

i) unique value

ii) 0-9 only

iii) Start - cannot be zero

Ex :

$$\begin{array}{r} T\ O \\ +\ G\ O \\ \hline O\ U\ T \end{array} \qquad \begin{array}{r} 2\ 1 \\ +\ 8\ 1 \\ \hline 1\ 0\ 2 \end{array}$$

T	2
O	1
G	8
U	0

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array} \qquad \begin{array}{r} 7\ 3\ 4 \\ +\ 7\ 3\ 4 \\ \hline 1\ 4\ 6\ 8 \end{array}$$

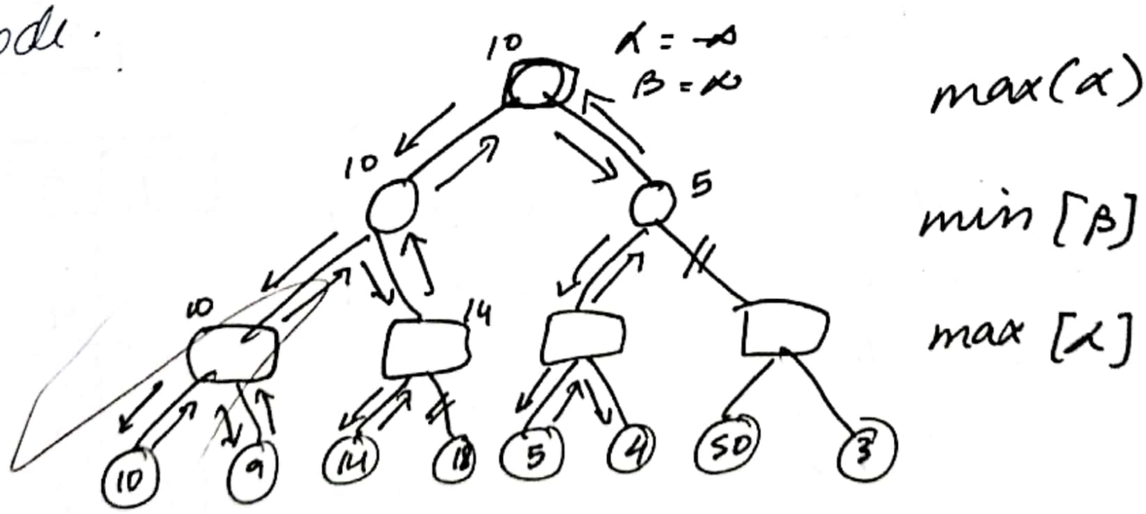
T	7
W	3
O	4
F	1
U	6
R	8

Scanned with CamScanner

Alpha-beta Pruning :-

The problem with minimax search is that the number of game states it has to examine is exponential in the number of moves.

* α-β process to find the optimal path without looking at every node in the game tree.
* Max - α , min - β
* We return α ≥ β which compares with its parent node only.
* Both minimax and α-β cutt-off given same path.
* α-β gives optimal solution as it takes less time to get the value for the root node.



Algorithm:-

```
function minimax (node , depth , alpha , beta,
                    maxplayer ):
    if depth ==0  or node is terminal:
        return evaluate (node)
    if maximizing player :
        maxEval = -∞
        for each child in node:
            eval = minimax (child, depth-1, alpha,
                             beta, False)
            maxEval = max(maxEval , eval)
            alpha = max (maxEval, eval)
            if beta <= alpha :
                break;
        return maxEval
    else :
        minEval = +∞
        for each child in node :
        eval = minimax (child , depth-1,
                  alpha , beta , True)
            minEval = min (minEval , eval)
            beta = min (beta , eval)
            if beta <= alpha :
                break;
        return minEval
```