```
value(node n)
{ int evalue; /* evalue is current value */
  if(n is terminal) return e(n); /* defined by depth bound */
  evalue = (n is MIN node)? ∞ : -∞;
  for(each successor ni of n) {
      generate ni;
      evalue = (n is MAX node)? max(evalue, value(ni)):
               min(evalue, value(ni));
  }
  return evalue;

}
```

Remarks: 1) In the above algo, the scan is from left to right and evaluate all theoriminal nodes

2) It examines and evaluate all terminal node at depth D

3) B^D is the no of terminal node at depth D if B is the branching factor.

4) Memory req. is of the order B^D.

5) A move giving the root value must also be outputted.

6) The game tree search algo. attempts to find the value of the root by examining minimum no of terminal nodes also intermediate nodes.

7) The no of terminal node examined serves as performance measure of the algorithm since calculation of state evaluation score of a terminal leaf is most time consuming and costly.

Compute the maximum of two numbers.

```
max(X,Y,X):- X>=Y,!
max(X,Y,Y).
```

```
member(X,[X|_]):-!.
member(X,[Y|L]):-member(X,L).
```

Add an element to a list without duplication.

write a relation efface(X,Y,Z) to remove the first occurance of element X from list Y giving a new list Z.

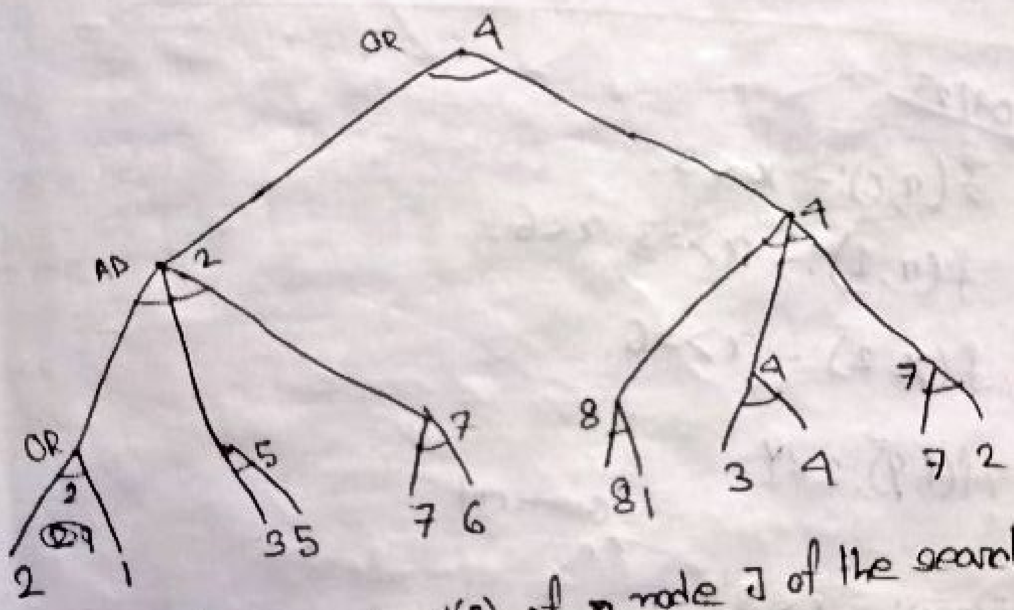
Max of a list.

```
maxlist([X],X).
maxlist([X|[Y|Rest]]
```

b) If C is to move the problem is solved if ~~any~~ wining strategy can be found by any of its successor, hence OR-node.

c) If H is to move the problem is solved with respect to C if C can force a win from each of the H's nodes hence And-node.

Since easy to write programes to win, as executive evaluation of all plays will guide the computer to find its best move at any point of time.



Mini-Max Rule: i) The valo V(J) of a node J of the search frontier is its static evaluation score otherwise

(ii) If J is a max node V(J) is the maximum value of any of its sucessors.

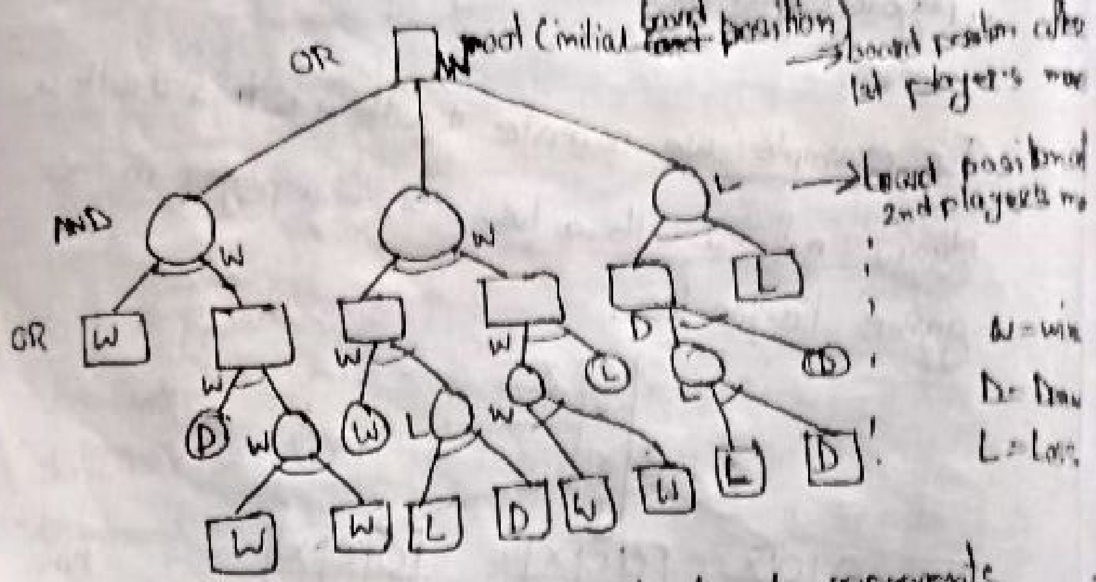
(iii) If J is a min node V(J) " " minimum value of any of its sucessors.

Game: MAX make the first move
MIN makes the second move

Component: Initial, successor function, terminal state.

Utility function give numeric value to all the terminal state.
High value, good for max & bad for min & vice versa

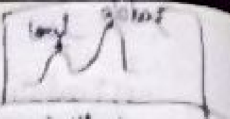
A game begins from an initial state & ends in a position that using a simple criteria can be declared as win/loss/draw
The game tree is an explicit representation of all possible ways of the game for a given position.



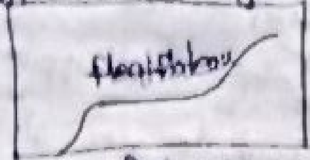
Each path from the root to a terminal node represents a different complete play of a game. Given a game tree it is easy to find a wining play for a player. Tree is nothing but a and-or tree since

a) For a game position J playing the game means to find a winning strategy from J for computer C.

Limitations of hill climbing approach:



i) Local maximum: Once the top of a hill is reached, the algo will halt since every possible steps leads down.



ii) If the landscape is flat it means many states have the same goodness then the algo degenerates to a random walk.

iii) If a landscape contains ~~riges~~ Ridges local improvements may follow a zigzac up the ridges slowing down the search.

Game Tree Search

We consider only two persons perfect info games i.e. chess, tic-tac-toe etc. Played by two players to move in turn. They respectively try to maximize OR minimize a scoring function also called utility function. They each know completely what both the player done and can do. That is al informations about the board position & moves are ~~avali avaibleable~~ available to the
There is no chance function like ~~dise~~ dice throwing game.
Game playing is an intelectual activity. Our attempt is to increase the intelectual ability of the computers which in turn will increase the intelectual ability of humans.

i) Evaluate the initial state.

ii) Loop until a soln is found or there are no operators left.

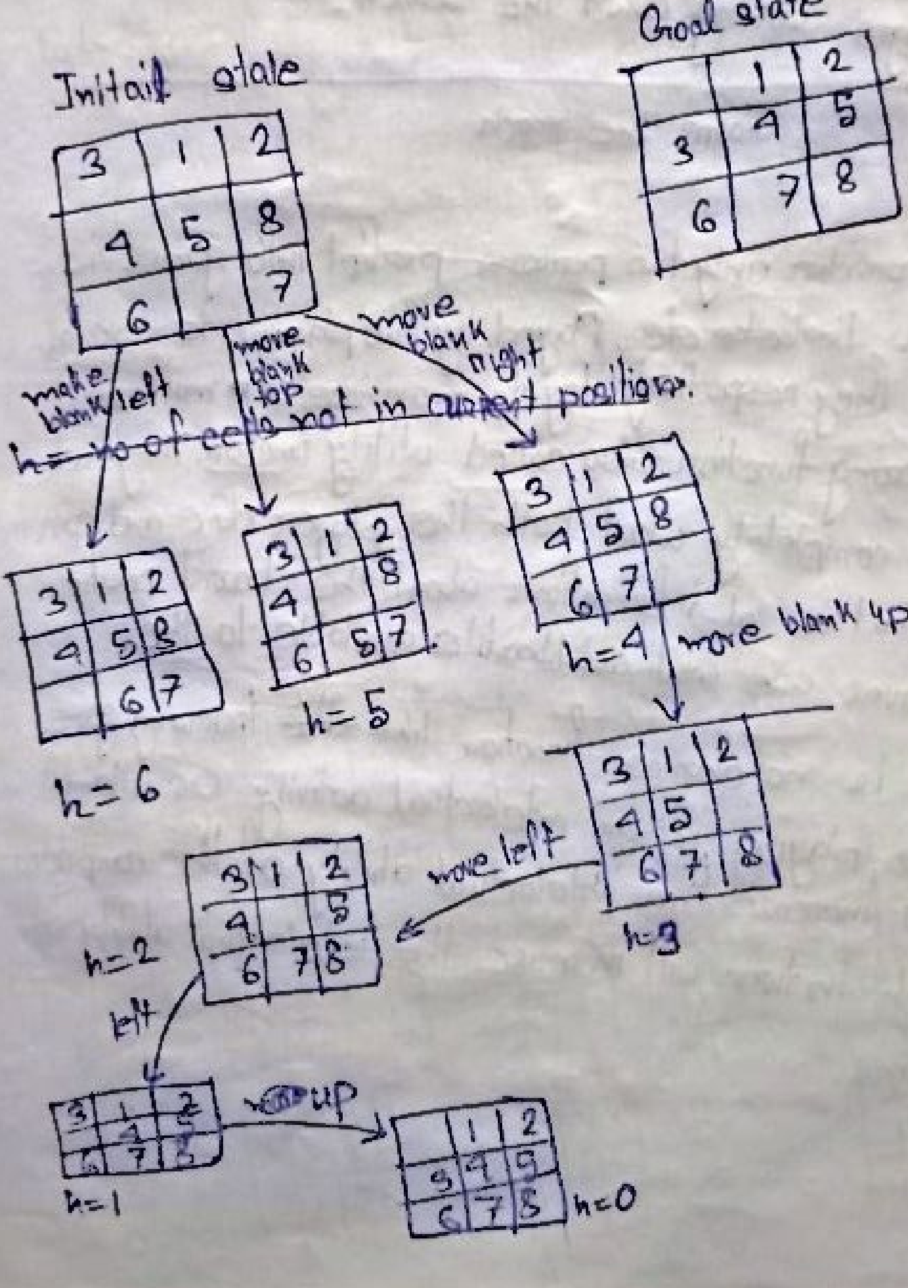
new iii) Select and apply new operator.

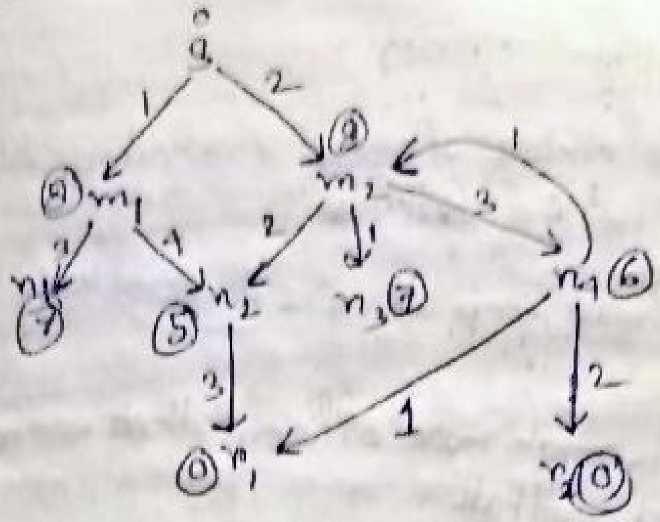
iv) Evaluate new state:-

a) If it is a goal state then quite.

b) If it is better than the current state then make it the new

c) If it is not better than the current state then goto step (ii).





Instant	S	m_1	m_2	n_1	n_2	n_3	n_4	r_1	r_2
0	0+0 $\boxed{0+0}$	1+8 $\boxed{1+8}$	2+8 2+8 $\boxed{2+8}$	4+7 4+7 4+7 4+7 $\boxed{4+7}$	5+5 4+5 $\boxed{4+8}$	3+7 3+7 3+7 $\boxed{3+7}$	5+6 5+6 5+6 5+6 $\boxed{5+6}$	7+0 $\boxed{7+0}$ 6+0 $\boxed{6+0}$	7+0 7+0

24/03/23

A heuristic h is admisible if for each node n belonging to g, $0 \leq h(n) \leq h^*(n)$ otherwise it is called admisible.

A* search properties:

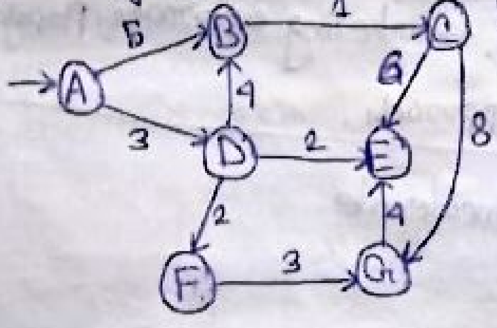
i) The algorithm A* is admisible ~~h~~ i.e provided a soln exists The first soln found by A* is an optimal soln i.e. A* finds optimal soln path if heuristic estimates are ~~op~~ admisible.

ii) A* is also complete.

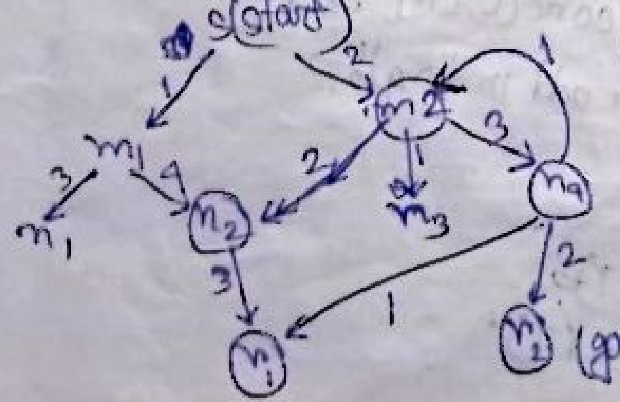
iii) A* is optimally efficient for a given heuristic of the optimal search algorithm. that expands such path from a root node, it can be shown that no other

```
begin
   g(ni)=g; put ni in OPEN;
   direct backward pointers from ni to n
end
else if ni in OPEN and g(ni)>g then
begin
      g(ni)=g;
      redirect backward pointer from
                                ni to n;
   end;
  end;
 end;
end;
    if found then output g(n) and solution path
          by tracking back through pointers
     else output failure message;
end;
```

Trace the optimal solution path from A to G in the following graph:



Instant	A	B	C	D	E	F	G
0	0						
	$\boxed{0}$	5		3			
				$\boxed{3}$	5	5	
		5			[illegible]	$\boxed{5}$	8
		~~5~~					
		$\boxed{5}$	6		5	5	8
			6		$\boxed{5}$	5	8
			6			$\boxed{5}$	8
			$\boxed{6}$				8



Instant	S	m_1	m_2	n_1	n_2	n_3	n_4	$n_1 n_2$
0	0							
	$\boxed{0}$		2					
		1						
		$\boxed{1}$	2	4	5			
			$\boxed{2}$	4	4	3	5	
				4	4	$\boxed{3}$	5	
				$\boxed{4}$	4		5	
					$\boxed{4}$		5	
							$\boxed{5}$	

2) CLOSED is the list of expanded nodes in the graph and is checking for duplicate nodes.

3) The idea of clean up is not applied here, because the are not considering the paths.

4) There are B^D nodes in a tree, having depth D and Branching factor B. In the worst case we have to store all the B^D NODES so the overhead is high compared to DFS.

5) DFS is recursive at the time of expanding it basically keeps the path in implicit stack but in case of BFS, it is relatively difficult to implement in recursive manner because it switches from one path to another.

6) Solution path is always optimal in terms of depth of a node but may not be optimal in terms of cost.

7) Duplicate nodes are not expanded but copies of the same node may appear in the OPEN.

8) Advantages are -

i) Shallowed goal guaranteed. May not be optimal in terms of cost in case of graphs.

ii) Faster solution in certain graphs.

9) Disadvantages are -

i) Higher memory required.

ii) More housekeeping work, more overhead and more difficult to impliment.

iii) Not suited to human problem solving approach.

Best First Search (1st page)

06/03/23

Q) Find the last member of a list.

~~member member last(_X,_~~

last (X, [X]).

last (X, [_|L]):- last(X,L).

Q) Find the reverse of a list.

rev([X],[X]).

rev([],[]).

rev([X|L],L1):- rev(L,L2), conc(L2,[X],L1).

Q) Check if a given list is palindrome.

~~Palindrome(X, L) :- rev(X, L)~~

Checkpal(X):- rev(X,X).

Q) Insert a element at all possible position.

iresrt(X,[],[X]).

~~insert(X,L,[X|L]):-~~

insert(X,[Y|T],[X,Y|T1]):- insert(X,T,T1).

Note:

① Clean up CLOSED ~~is~~ purges from closed. All ancestors of the node that do not have sons in OPEN. The effects are -

(i) CLOSED will only contain the expanded nodes on the current path

(ii) Reduces the size of ~~closed~~ CLOSED, saves memory.

② The list CLOSED ~~forms~~ a single path from the start node to the currently expanded node. This restricts the maximum storage, which is depth form and branching factor, in case of a graph OPEN holds the node generated in the explicit graph not yet expanded.

③ For a search graph (not a tree) the depth of the node, is the depth of its shallowest parents the then leads to maintaining depth first in true sense then leads to complication. In practise DFS takes depth of its current parents.

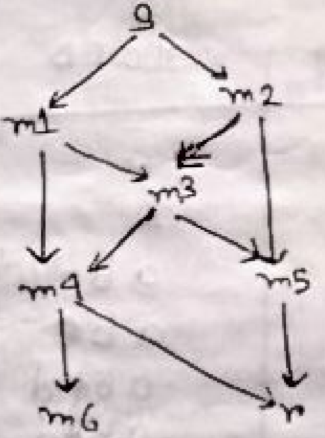
④ No duplications are not strictly checked in practise, only checked against the current path to avoid looping.

⑤ Effective recursive formulation are possible with lesser amount of storage requirements.

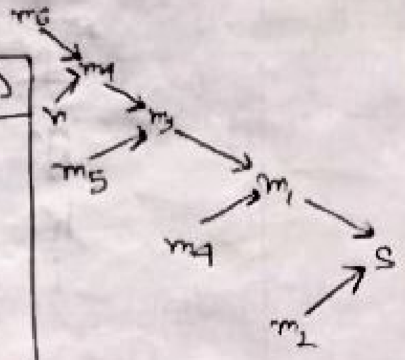
if OPEN is empty then Output "Solution path not found";

else output "The solution path is obtained by tracking back through pointers";

end



OPEN	CLOSED
s	
~~s~~	s
~~m1m2~~	s
m1, m2	s, m1
~~m1~~, m2	s, m1
m3, m4, m2	s m_1 m_3
m_4 m_2	s m_1 m_3
m_4 m_5 m_4 m_2	s m_1 m_3 m_4
~~m_4~~ m_5 m_4 m_2	s m_1 m_3 m_4
m_6 r m_5 m_4 m_2	s m_1 m_3 m_4



$s \to m_1 \to m_3 \to m_4 \to r$

Time complx: b^d

Space complexity: bN

open	closed
s	
~~s~~	s
~~m_2~~ m_1	s
m_3 m_5 m_1	s m_2
~~m_3~~ m_5 m_1	s m_2 m_3
~~m_4~~ m_5 m_5 m_1	s m_2 m_3
m_6 ~~m_4~~ r m_5 m_5 m_1	s m_2 m_3 m_4

b: Branching factor.
d: Depth of the tree.
N: Maximum depth that can be reach.

Depth First Search:-

begin
put the start node & OPEN;
found = False;
while (OPEN! Empty and not found do
{begin
remove the top most node n from
OPEN and put it in CLOSED;
if depth of n = depth-bound
then cleand up CLOSED;

begin
Expand n generating all its succe
put these successors (in no particula
order) on top of OPEN except the
successors already appeared in CLOSED;

direct backward pointer to n for
each successors n_i of n;
if any of these successors is goal
then found = True;
else if any of the successor is
a dead end then remove it
from OPEN and clean up CLOSED
end;
end;

01/03/23 Effectiveness of search algo:-

State vs node

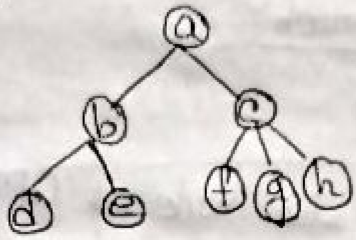
state ⊂ no

i) Completeness.

ii) Time complexity.

iii) Space complexity.

iv) Optimality.



bfs abcdefgh

dfs a ~~b~~ ~~d~~ ~~e~~ c ~~f~~ ~~g~~ h

Factors to evaluate

Problem solving performance of various search strat

① completeness: The stratig garantees to find a soln if one exists.

② Optimality: The soln has an optimal cost.

③ Time Complexity: Time taken (No of nodes expanded to find a soln. in worst case or avg case)

④ Space Complexity: Space used by the algorithm, measured in terms of maximum size of the fringe

[Fringe is a list of nodes, that have been generated but not yet explored. It represents the frontier of the search tree generated. Initially it contains a single node corresponding to the start state.]

Q) Adding an item X to the ~~beggi~~- begining of the list L.

```
add(X,L, [X|L]).
```

using concatenation:

```
add(X, L1, L) :- conc([X], L1, ).
```

Q) Delete the first ~~elem~~ occurance of element X

Assuming that the element exists.

```
del(X, [X|L], L).
del(X, [Y|Rest], [Y|L]) :- del(X, Rest, L).
```

a, a, b, a ;

a a b a ;

a b a ;

a a b

Q) Delete all occurance of X.

```
delall(X, [ ], [ ]).
delall(X, [X|L], L1) :- delall(X, L, L1).
delall(X, [Y|L], L1) :- delall(X, L, L2),
                        conc([Y], L2, L1).
```

23/01/23

List [ ]

[x], [X], [_]

[X|Rest]   [X, Y|Rest]

[X | [Y | Rest1]]

Q) Given a list L we want to find whether an item X is member of that list or not.

```
member(X, [X|_]).   member(X, [Y|Rest]):-
member(X, Rest).
```

Q) Concatinate list L1 & L2 to form list L3.

```
conc([], L, L).
conc([X|L1], L2, [X|L3]):- conc(L1, L2, L3).
```

Q) Define Member function using concatenate.

```
member1(X, L):- conc(L1, [X|L2], L).

conc([], L, L).
conc([X|L1], L2, [X|L3]):- conc(L1, L2, L3).
```

wolf goat cabbage problem

A farmer has a wolf, a goat, and a cabbage on the left bank of a river. He also has a boat that can carry atmost one of the 3 and must transport this trio to the right bank. The problem is that he cannot keep this pairs at one side - (i) goat-cabbage.

(ii) wolf-goat.

(l, [w,g,c], [ ]) $\xrightarrow{g}$ (r, [w,c], [g])

Initial state.

(r, [ ], [w,g,c])

Goal state.

Sol^n

Initial

(l, [w,g,c], [ ])

↓

(r, [w,c], [g])

↓

(l, [w,c], [g])

↓

(r, [w], [g,c])

↓

(l, [w,g], [c])

↓

(r, [g], [w,c])

↓

(l, [g], [w,c])

↓

(r, [ ], [w,g,c])

Q) Water jug problem

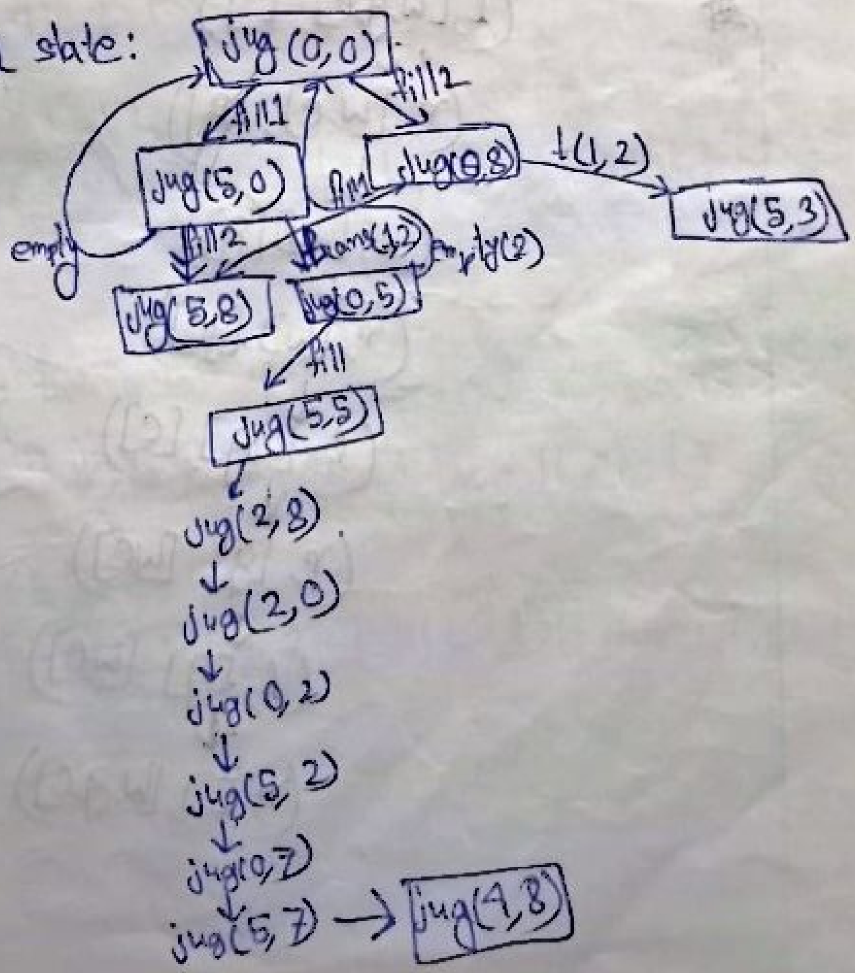
Two jugs of capacities 8 liters and five liters of no markings are given. The problem is to measure out exactly 4 liters from a vat containing enough water. The possible operations are-

(1) Filling up the jug from the vat.

(2) Empting the jug into the vat.

(3) Transfering the contents of one jug into other until the pouring jug is completely empty or the other jug is completely fill to the capacity.

Initial state:



Example:

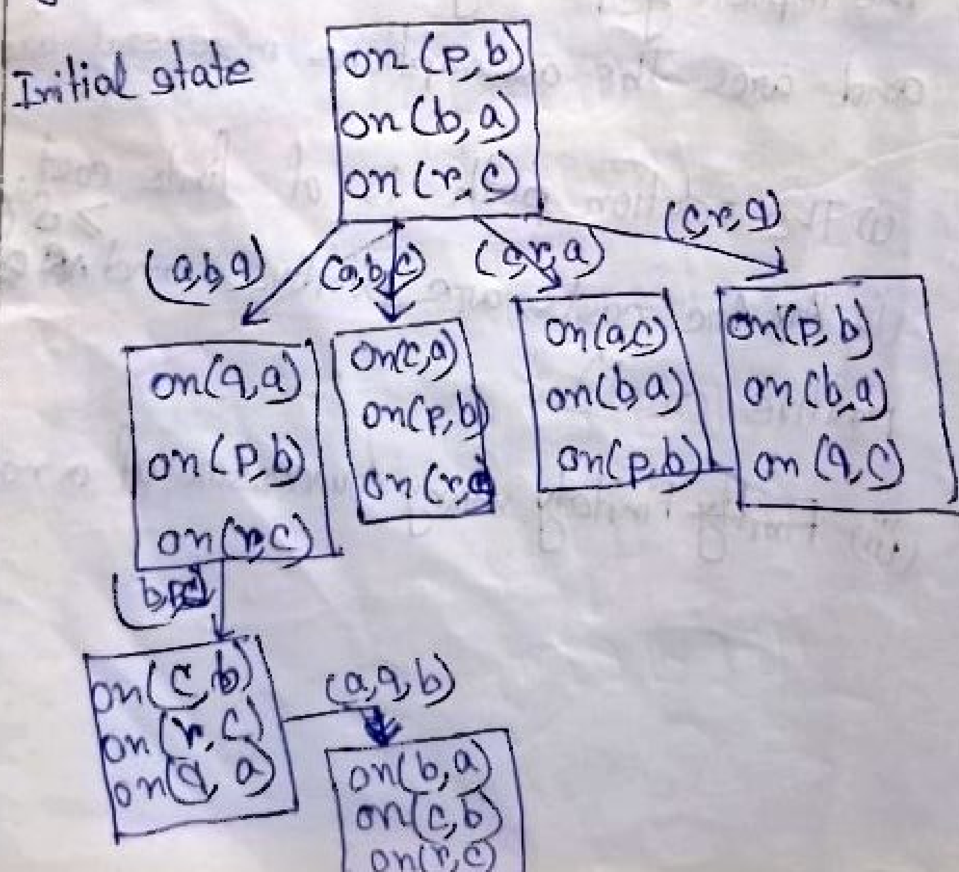
initial state Final state

The types of moves are -

i) The topmost block in one position to another position.

ii) The topmost block in one position to an empty position.

2) Assuming Arc costs are unity then draw search graph.



In general, the search is supplied implicitly to any algorithm. The start node, goal nodes and the operators are given as inputs. Any search algorithm starts by expanding the state start node i.e by generating the successor of the start node with the help of operators and adding its successors ~~to the~~ to the the implicit graph.

At any state in general a node in explicit graph is chosen for expansion and the algorithm terminates when the goal node is chosen. This rule will be followed in every algorithm.

The implicit graph may have infinitely many nodes and arcs. The assumptions in general are

(i) The solution path is of finite cost.

(ii) ~~Arc~~ Arc costs are positive and ~~(~~ $\geq \delta$ (some positive const).

(iii) ~~Finitly~~ Finitely many successor of a node.

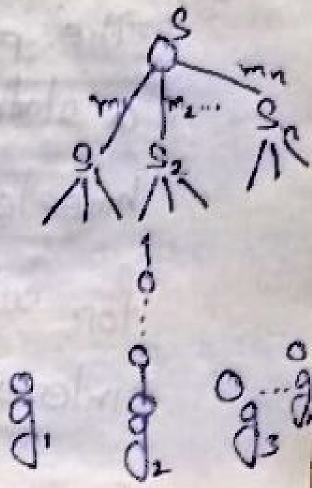
The problem may not come naturally indicate a state space representation. The problem may have to reformulated in order to be suited for solving by search methods. This itself is an interesting exercise for various problems.

The state space is basically a directed graph known as space graph with a special node called the start node called 'S' and a set of goal nodes called g. If an arc is directed from n_i to n_j, then node n_j said to be the successor of node n_i. Given a node (state), the problem defines the operators which can be applied to the nodes -

to generate a successor node. There is a cause associated with the generation of each successor node and hence each arc is labled with the operator and the cost appling the operator. Nodes of the graphs are labled by description. Solving the problem reduces the to finding the sequence of ~~states~~ steps of move required from the start node to a goal node.

Artificial Intelegence

State search space



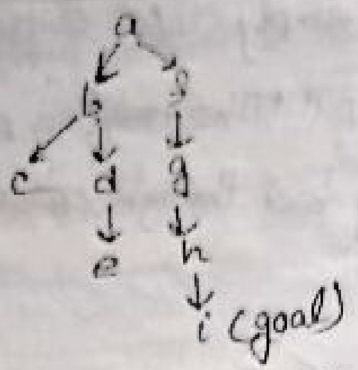
Input: Starting S
set of operators/moves
set of goal states

To find a solution path (optimal) from S to any of the goal nodes.

A state is defined by the specification of the values of all attributes of interest in the world.
An operator changes one state into another.
The initial state is where we start. The goal state is where we finish; it is partial description of solution. A plan is a sequence of actions.

Search space graph / search graphs

Search methods are used to solve problems by searching a state space to find a path from a given initial state to a final state.



OPEN	CLOSED
a	a
~~a~~	
~~b~~f	a
~~c~~df	ab (clean up)
~~d~~f	abc
~~e~~f	abcd
~~f~~	abcd
~~g~~	abcde (clean up)
~~h~~	abcdef
i	abcdefg
	abcdefgh

OPEN	CLOSED
a	
~~a~~	a
~~b~~f	a
cdf	ab
~~e~~df	ab (clean up)
~~d~~f	ab
ef	abd
~~e~~f	a~~bd~~ (clean up)
~~f~~	a
~~g~~	af
~~h~~	afg
i	afgh

(6) Advantages are -
(i) low memory and overhead are easy to implem
(ii) Quick solution in certain graphs.
(iii) Human problem solving methods.

(7) Disadvantages are.
(i) Sometimes exploring non promising path deeper
(ii) Unnecessary node expansion.
(iii) solution paths not optimal interms of depth.
(iv) A node may be expanded more than once along various paths. Infact it may be exponential in certain cases.

(8) The algorithm takes exponential time in the worst case the algo will take $O(b^d)$ time, where b is the branching factor & d is the depth where soln can be found.

(9) The space taken is linear in the depth of the search tree is $O(B*N)$ where n is the length of the longest path or maximum depth.

(10) If the search tree has infinite depth the algo may not terminate. This can happen if the search space is infinite or search contains cycles.

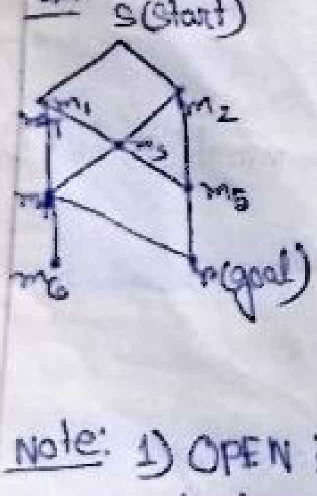
Thus DFS is not complete.

15/02/23

BFS

Begin
put start node S in OPEN;
found = false;
while OPEN is not empty and not(found) do
begin
remove the topmost node n from OPEN and put it in CLOSED;
expand n generating all its successors. put these successors (in particular order) at the end of OPEN except the successors already appearing in CLOSED.
Direct backward pointers to n from each successor n;
if any of these successors is a goal node then found = true
else if one of these successors is a deadend then remove it from OPEN.
end
if OPEN is empty then
Output failure message.
else
Output solution by tracking back through pointers
end

Ex:



OPEN	CLOSED
s	—
$\cancel{s}$	s
$m_1 m_2$	s
$\cancel{m_1} m_2$	$s m_1$
$\cancel{m_2} m_3 m_4$	$s m_1 m_2$
$m_3 m_4 m_3 m_5$	$s m_1 m_2$
$\cancel{m_3} m_4 m_3 m_5$	$s m_1 m_2 m_3$
$\cancel{m_4} m_3 m_5 m_4 m_5$	$s m_1 m_2 m_3$
$m_3 m_5 m_4 m_5 m_6 n$	$s m_1 m_2 m_3 m_4$
	$s m_1 m_2 m_3 m_4$

Note: 1) OPEN is the list of nodes generated in the explicit graph but not yet expanded (just like DFS)

Algorithm: Uniform Cost

Input: Root/Start node, goal node

Output: Optimal solution path.

OPEN and CLOSED are initially empty.

begin
U1: g(s)=0; put s in OPEN; found = false;
while OPEN not empty and not(found) do
begin
U2: select a node n from OPEN with minimum
~~g-value~~ g-value;
/* resolve ties arbitrarily but in favour of a goal node */
remove n from OPEN and put it in CLOSED;
if n is a goal node then found = true
else
begin
expand n generating all its immediate successors
for each immediate successor n_i of n
begin
$g = g(n) + c(n, n_i)$;
if n_i is not already in OPEN or ~~close~~ CLOSED then
begin

2	8	3
1	6	4
	7	5

h_1 = no of tiles not in proper position.
= 7

h_2 = Manhatten-distance
= 1+2+0+1+1+2+2+1
= 10

1	2	3
4	5	6
7	8	

Heuristic estimates

i) Non negative heuristic estimates $h(n)$ associated with each node in a graph G, here $h(n)$ is an estimate of the cost of a minimum cost path from n to a goal node; $h(n)$ is estimated from the problem domain.

ii) $f(n) = g(n) + h(n)$. $f(n)$ gives an estimate of the cost of the minimal from a start node s to a goal node which is a constraints to pass through n.

iii) $h^*(n)$ is the actual cost of a minimal cost path from n to a node. (infinite no path exists) $g^*(n)$ is the actual cost of the minimal cost path from s to n; $f^*(n) = g^*(n) + h^*(n)$ is the actual cost of a minimal cost path from s to a goal node which is constraints to pass through n.

(iv)

Find all prefixes of a list.

Find the permutation of a list.

22/03/23

Heuristic means rule of thumb

Heuristic are criteria methods or principle for deciding which among several alternative courses of promises to be more efficient in order to achieve some goal. A heuristic function at node n denoted by h(n) is an estimate of the optimum cost from the current node to a goal node.

example 1: The heuristic for the path from bolpur to kolkata may be the straight line distance from between Bolpur & Kolkata

h(Bolpur) = Euclidean-distance(Bolpur-Shantiniketan)

Note 1 ~~Step~~: - Uniform cost search lacks problem specific problem. Such methods are probabilitably inefficient in many cases. Using problem specific knowledge can dramatically improve the search speed.

Note 2: Uniform cost works with $g(n)$ values of nodes. It does not use any info about the cost of the remaining path from a node to a goal node. So in many problems this method tends to expand to many nodes.

Note 3: It is possible to cut down the number of node expansion by using the estimate of the cost of the remaining path from a node to a goal node. Heuristic search approach attempts to direct the search by using heuristic estimates of the cost of the path from a node to a goal node.

Note 4: Several approaches, to use heuristic search are there such as A* approach, marketing approach, Propagation of values, Recursive approach.

20/03/23

Check palindrome using concatenate

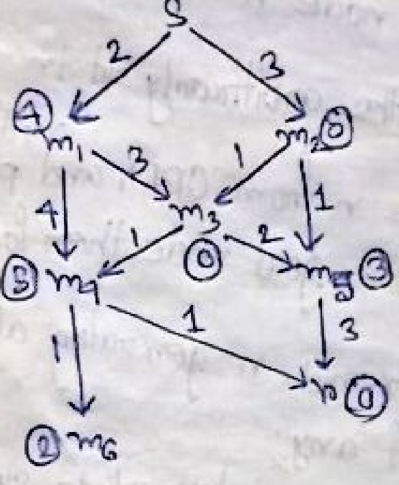
```
palindrom([]).
palindrom([_]).
palindrom([A|T]):- conc(L,[H],T), palindrom(L).
```

Check whether even or odd in length.

```
even([])
odd([_])
odd([X,Y|T]):- odd(T).
odd([X,Y|T]):- even(T).
even([X,Y|T]):-
```

direct backward pointer from n_i to n;
end
else if $g(n_i) > g$ then
begin
$g(n_i) = g$; $f(n_i) = g + h(n_i)$
redirect backward pointer from n_i to n
if n_i is in CLOSED then remove
n_i from CLOSED and put it in OPEN
end
end
end
end

if found then output $f(n)$ and solution path by tracing
back through pointers;
else output failure message;
end



Instant	S	m_1	m_2	m_3	m_4	m_5	m_6	n
0	0+0							
	$\boxed{0+0}$	2+4	3+5					
		$\boxed{2+4}$	3+5	5+0	6+5			
			3+5	$\boxed{5+0}$	6+5	7+3		
			$\boxed{3+5}$	4+0	6+5	4+3		
				$\boxed{4+0}$	5+6	4+3		7+0
					5+5	$\boxed{4+3}$		$\boxed{7+0}$
					5+5			

(iv) Assumption about $h(n)$: a) $h(m) > 0$ for every node m.

b) If $h(s) = 0$ and $h(n) \geq 0$

c) If m lies on the soln path then $h(m)$ is finite otherwise $h(m)$ may be infinite.

Algorithm A*:

```
begin
A1: g(s)=0; f(s)=0; found=false;
    put s in OPEN;
    while OPEN is not empty and not (found) do
    begin
    A2: Select a node n from OPEN with minimum f-value;
        /* Resolve ties arbitrarily but in favour of a goal node */
        Remove n from OPEN and put it in CLOSED;
        if n is a goal node then found= true else
        begin
            expand n generating all its successors
            if any;
            for each intermediate successors
            ni of n do
            begin
                g=g(n) + c(n,ni);
                if ni is not in OPEN or CLOSED
                then
                begin
                    g(ni)=g; f(ni)=g + h(ni)
                    put ni in OPEN;
```

optimal algorithm
However the number of nodes searched is still exponential in the worst case.

iv) A* must keep all nodes it's considering in memory.

v) A* is much more efficient than ~~are~~ uninformed methods.

Limitations of A*:

i) Worst case performance is poor.

ii) Size of memory is appreciable is compared to DFA.

iii) Repeated node expansion.

iv) In case of ~~adm~~ inadmissible heuristic, we have no Idea about the quality of soln.

v) Overhead of open is just as BFS.

vi) OPEN must be implemented as priority queue.

Advantages of A*:

i) A* is quite fast on the average

ii) A* never ~~use~~ stores the entire explicit graphs and hence makes repeated expansion.

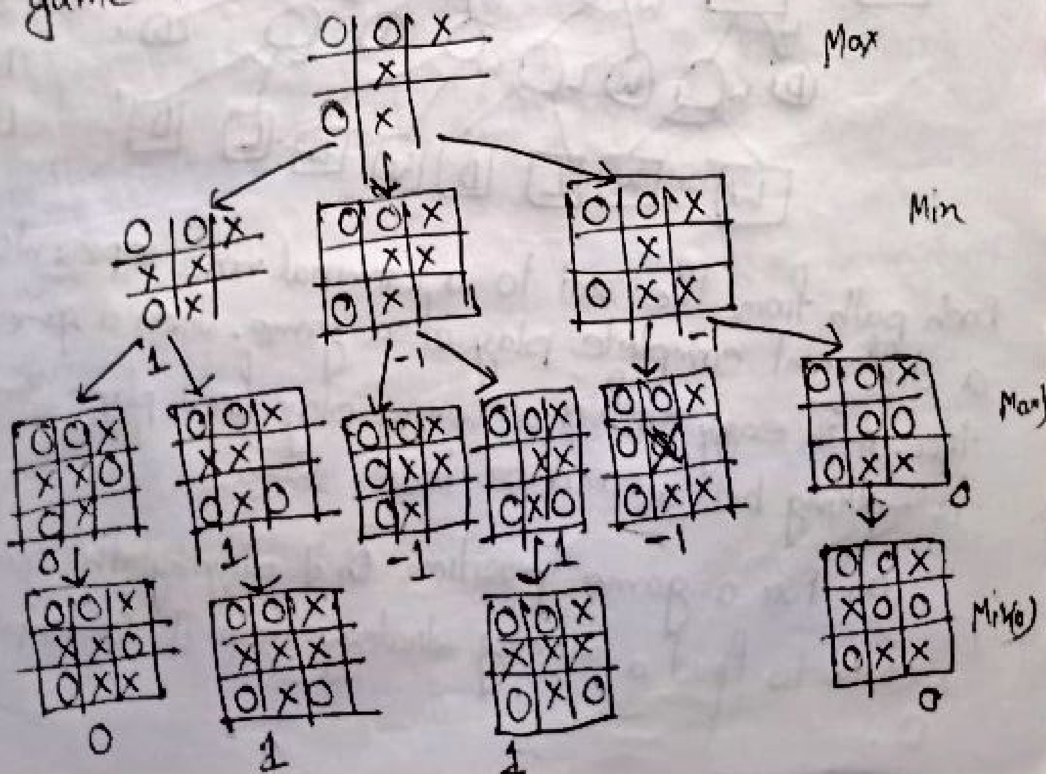
iii)

This is a local search algorithm with greedy approach and with no backtracking.

Game tree: Adversarial goal search can be represent as a tree where the nodes represent the current state of and the arcs represents the moves. The game tree consists of all possible moves for the current player starting at the root and all possible move for the next player as the children of these nodes and so forth as far to the future of the goal as desired. Each individual move by one player is called ply.

The leaves of a game tree represent terminal position where the outcome of a game is clear (win, loss, draw). Each terminal position has a score. Fli are good for one of the player called the max player. The other player called the mean player tries to minimize the score.

For example: We associate 1 with a win & 0 with a draw & -1 with a loss for max player in the game tic-tac-toe.



Min-Max Algo: To determine V(J) do the following.

(i) If J is a terminal value return V(J) = e(J)

otherwise,

(ii) Generate J's successors say $J_1 J_2 \ldots J_b$ where b is the branching factor.

(iii) Evaluate $V(J_1)\, V(J_2) \ldots V(J_b)$ from left to right

(iv) If J is a max node return, $V(J) = \max_{1 \le i \le b} [V(J_i)]$

v) " " " " min " " $V(J) = \min_{1 \le i \le b} [V(J_i)]$

03/04/23

f(x,0):- x<3.

f(x,1):- x>=3, x<6.

f(x,2):- x>=6.

?f(1,Y), 2<Y

f(x,0):- x<3,!.

f(x,1):- x>=3, x<6!.

f(x,2):- x>=6

Green cut

f(x,0):- x<3,!

f(x,1):- x<6,!

f(x,2).

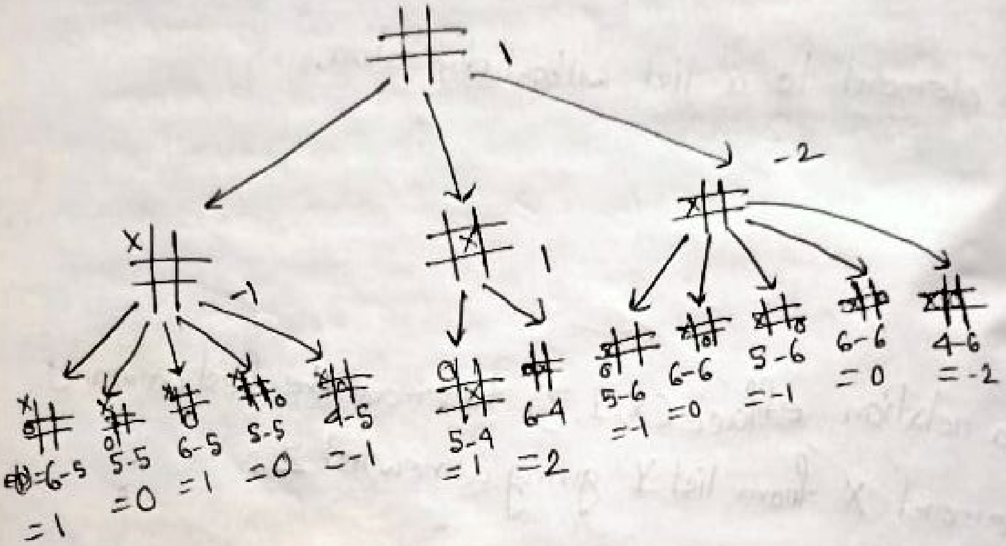
Red cut.

05/04/23

Lets assume that max (the computer) marks process (X) and MIN marks (O) and MAX is to play first in the game of Tic-Tac-Toe.

Let the static evaluation score e(P) = (no of complete rows, columns or diagonals that are still OPEN for MAX)

-(no of com. . . . . . OPEN for MIN)



Disadvantage: To use this method we have to -

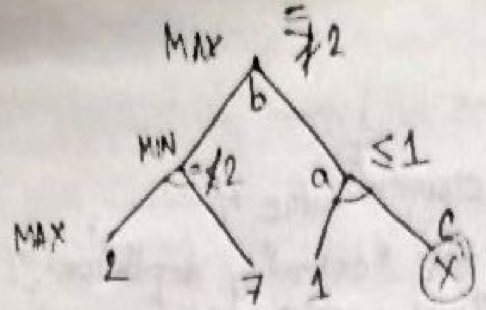
i) Generate the full game tree first.

ii) Evaluate all the terminal nodes.

iii) Examine the full game tree.

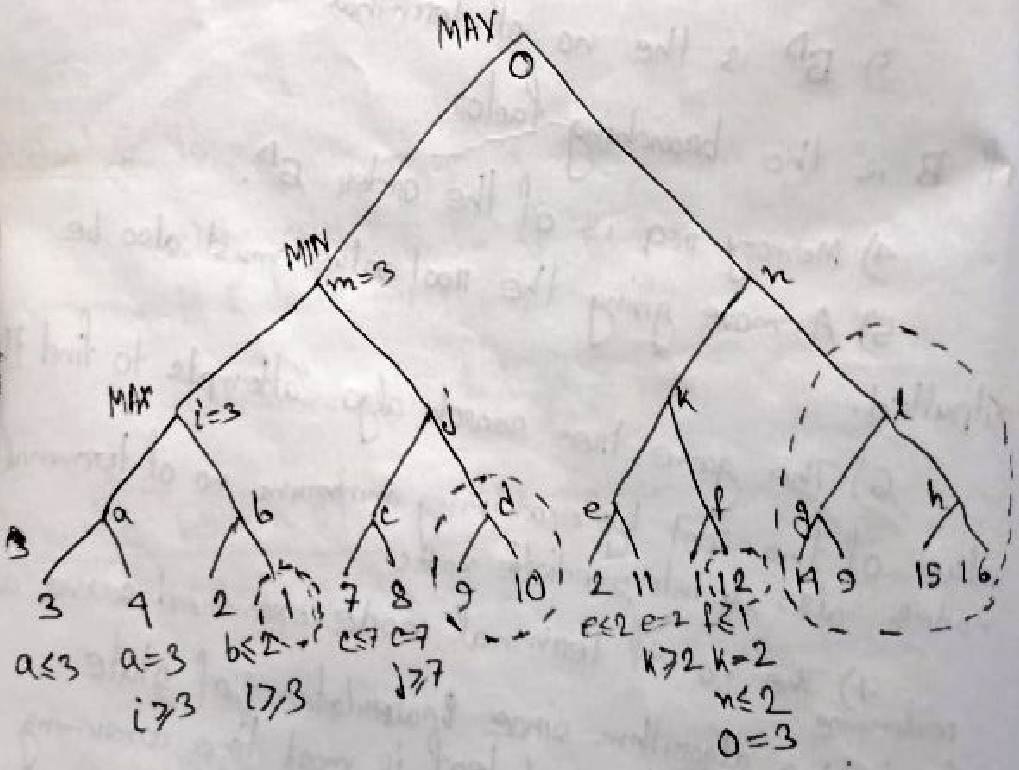
Recursive version of MIN-MAX algorithm:

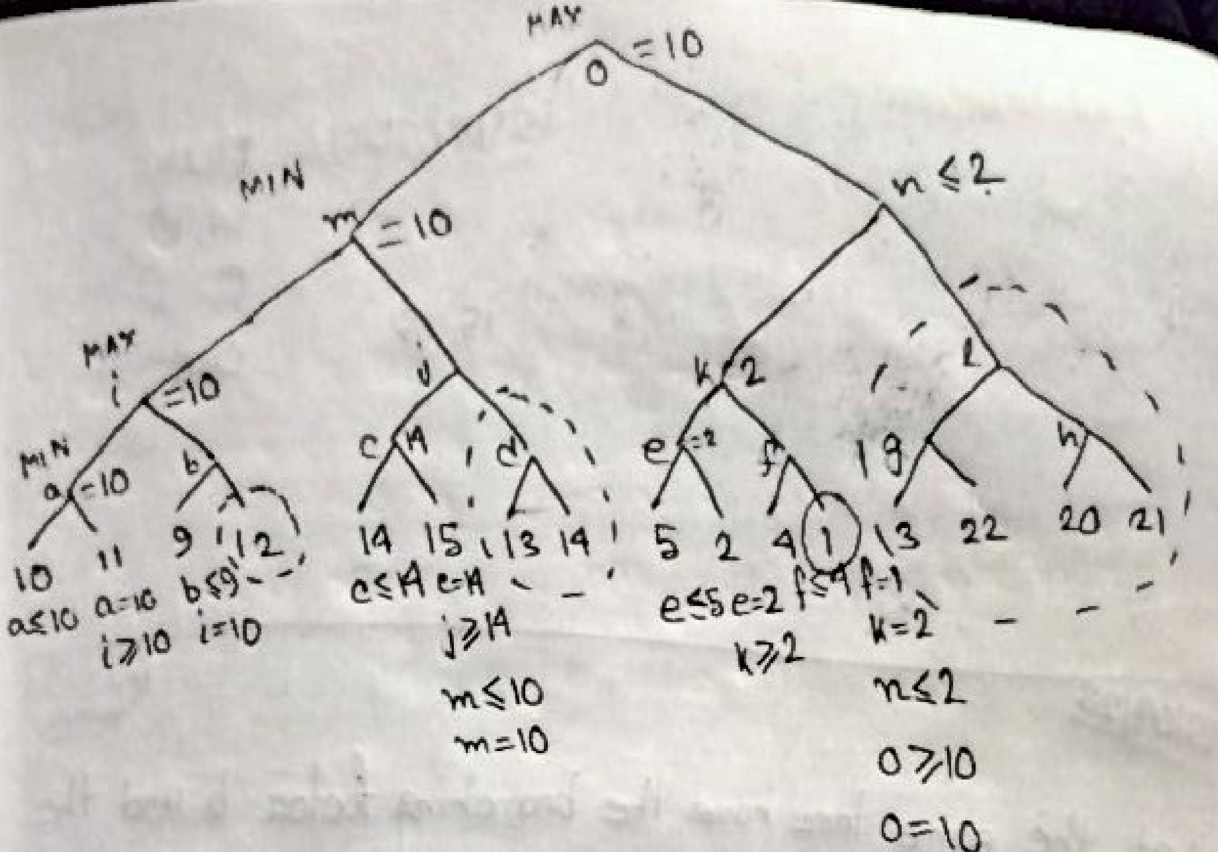
```
main()
{
   v=value(root);
   output v;
}
```



α-β prunning: It is a backtracking strat

This is to reduce the no of terminal nodes examined. By the ~~min~~ MIN-MAX procedure we evaluate all ~~for~~ four terminal node but we need not even look at the last node c, since at a the score is ≤ 1 and at b the score is ≥ 2. Thus at b the score must be 2 irrespective of the quality no at c.





L=[a,b]

- ([_|[_|Rest]], Rest)

10/04/23

~~doubletone~~

whether a list is not double tone ~~or not~~

notdoubletone([]).~~[X|Y]~~

~~([X|[Y|~~

- - . . ([_]).

- - . ([X|[Y|[Z|Rest]]]).

Whether two list are same length or not.

even-odd length.

X is less

$(s(15), s(20)) = 15, 19$

14, 13

0,

15, 15

0, 0

13/09/23

Let the game tree have the branching factor b and the depth d, then the minimum no. of static evaluation required to solve the game tree is -

$$E = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$$

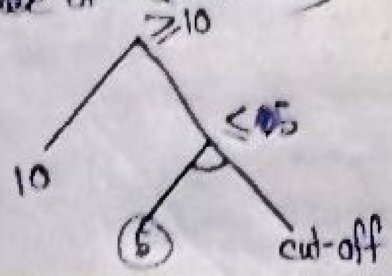
$$= \begin{cases} 2b^{\lceil d/2 \rceil} - 1 & \text{if } d \text{ is even} \\ b^{\frac{d-1}{2}} + b^{\frac{d+1}{2}} - 1 & \text{if } d \text{ is odd} \end{cases}$$

We can say the bound on a node as a current value of a node. Actual value of a node is set to its current value -

(i) If all the sons are yet to be exa

(ii) If some of the sons are yet to be examined but their examinations can in no way affe the value of the root.

Case-I (case of α- cutoff)



The max ancestor of a MIN node cuts-off the sons of the MIN node if the current value of MIN node is ≤ the current value of MAX ancestor (Reason since current value of a MIN node in such a case can never increase the value of a MAX ancestor).

The MIN ancestor of a MAX node cuts-off the sons of the MAX node if the current value of a MAX node is ≥ the current value of MIN ancestor (Reason since current value of a MAX node in such a case can never decrease ~~increase~~ the value of a MIN ancestor).

The current value of a MAX node is known as α-value or α bound and the current value of the MIN node is known as β-value or β bound.

Modification of MIN-MAX algorithm

main()

{

va

α-bound is the cut off bound and is equal to the current ~~value~~ value of all MAX ancestor of J i.e. highest current lower bound of all " " " J When the current value of the ~~…~~ MIN node J $\leq$ α-bound for J, the sons of

β-bound

α-β pruning algorithm

```
main()
{ v=αβvalue(root,-∞,∞)      /* α=-∞, β=∞ */
  output v;
}
αβvalue(node n, int α, int β)
{ int cvalue;
  if (n is a terminal)
      return e(n);
  if (n is a MAX node)
   { cvalue=α;
      for(each successor ni of n)
       {
         generate ni;
         cvalue = max(cvalue, αβvalue(ni,cvalue,β));
         if (cvalue≥β)
            return cvalue;
       }
      return cvalue;
   }
```