MODULE 2

Search Methods – Best First Search – Implementation in Python – OR Graphs, A* Algorithm, Problem Reduction – AND–OR Graphs, AO* algorithm, Constraint Satisfaction, Games as Search Problem, MINIMAX Search procedure, Alpha-Beta pruning.

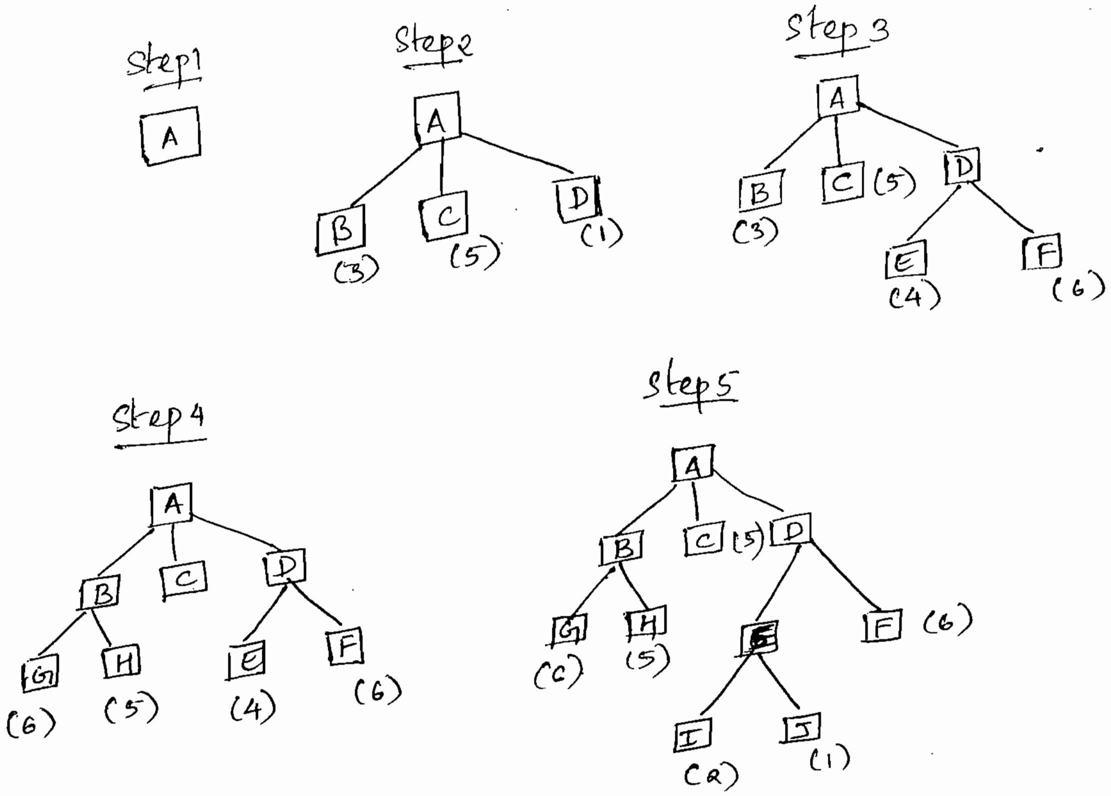
$\boxed{\text{Best First Search}}$ ~~or~~ Greedy Best First Search ($h(n) \rightarrow$ estimate to goal)
A* ($f'(n) = g(n) + h'(n)$)

- Best First Search is a way of combining the advantages of both DFS and BFS into a single method.
- DFS is good because it allows a solution to be found without all competing branches having to be expanded.
- BFS is good because it does not get trapped on dead end path.

→ One way to combine the two, is to follow a single path at a time, but switch paths whenever some ~~competing~~ path look more promising than the current one does.

→ At each step of the Best First Search process, we select the most promising of the nodes we have generated so far. This is done by applying an appropriate heuristic function to each of them. We then expand the chosen node by using the rules to generate its successors. If one of them is a solution, then we can quit. If not, all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process continues.

fig: A Best-First Search Procedure in a tree format.



The above fig shows the best first Search procedure.

Fig explaination

Initially there is only one node, so it will be expanded. 3 nodes are generated. In this case, heuristic fn is an estimat of the cost of getting to a solution from a given node. In step 2, node D appears as the most promising one. Hence it is expanded producing 2 successors E & F. Heuristic fn is applied on these nodes. Now another path going through B appears more promising, hence it is expanded next to generate G & H. The process continues until a solution is found.

Comparison with hill climbing

- This ~~alg~~ procedure is similar to the procedure of steepest-ascent hill climbing with 2 exceptions.
  - In hill climbing, one move is selected, and all the other are rejected, never to be reconsider. In best first search one move is selected, but others are kept around so that they can be revisited if selected path b...

- Further, the best available state is selected in best first search, even if that state has a value that is lower than the value of the state just explored. This contrasts hill climbing, Here the alg stops if there are no successors with better values than current state.

→ OR graph.

- The above eg illustrates a best first search of a tree. It is sometimes necessary to search a graph instead so that duplicate paths will not be pursued.
- An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space.
- Each node contains a description of the problem space it represents, an indication of how promising it is, a parent link that points back to the best node from which it came, and a list of nodes that were generated from it.
- The parent link will help to find the path to the goal once the goal is found. The list of successors/nodes will help to propogate the improvement down to its successors if a better path is found to an already existing node.
- The graph of this sort is called an OR graph, since each of ~~its~~ branches represents an alternative problem-solving path.

- To implement such a graph search procedure, 2 list of nodes are required.
  - **OPEN**: nodes that have been generated, and have had the heuristic fn applied to them but which have not yet been examined (i.e. their successors have not been generated). This is actually a priority queue in which the elements with the highest priority are those with the most promising values of the heuristic fn.
  - **CLOSED**: nodes that have already been examined. These nodes have to be kept in memory since whenever a new node is generated, we need to check whether it has been generated before.
- In Best-First search, the heuristic fn, $h(n)$ = estimated cost of the cheapest path from node n to goal state.

Algorithm: Best First Search

1. Start with OPEN containing just the initial state
2. Until a goal is found or there are no nodes left on OPEN do
   a) Pick the best node on OPEN.
   b) Generate ~~the~~ its successors.
   c) For each successor do:
      i) If it has not been generated before, evaluate it, add it to OPEN, and record its parent.
      ii) If it has been generated before, change the parent if this new path is better than the ~~parent~~ previous one. In that case, update the cost of getting to this node & to any successors that this node may already have.

→ Time complexity = Space Complexity = $O(b^d)$, where b is the branching factor & d is the depth

A* Algorithm — A search technique that finds minimal cost solutions

- A* Algorithm is a typical heuristic search algorithm, in which the heuristic function is an estimated shortest distance from the initial state to the closest goal state, and it equals distance travelled plus distance ahead.

ie heuristic fn f' is the sum of 2 components say g & h'

$$f' = g + h'$$

here, g is the measure of cost of getting from initial to current state

and h' is the estimate of the additional cost of getting from the current node to a goal state.

Thus, f' represents an estimate of the cost of getting from an initial state to a goal state along the path that generated current node.

Algorithm : A*

1. Start with OPEN containing only the initial node. Set the node's g value to 0, its h' value to whatever it is, and its f' value to $h'+0$, or h'. Set CLOSED to the empty list.

2. Until a goal node with min f' is found, repeat the following procedure:

If there are no nodes on OPEN, report failure. Otherwise, pick the node on OPEN with the lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED. See if BESTNODE is a goal node. If so, exit and report a solution. Otherwise generate the successors of BESTNODE.

For each successor do the following:

a) set SUCCESSOR to point back to BESTNODE. (These backward link will help to recover the path once solution is found)

b) Compute g(SUCCESSOR) = g(BESTNODE) + cost of getting from BESTNODE to SUCCESSOR.

c) check whether SUCCESSOR is the same as any node on OPEN If so, then call that node OLD. Add OLD to the list of BESTNODE's successors. See whether it is cheaper to get to OLD via its current parent or ~~to~~ via BESTNODE by comparing g values. If OLD is cheaper, then do nothing. Otherwise reset OLD's parent link to point to BESTNODE, record the new cheaper path in g(OLD), and update f'(OLD)

d) If SUCCESSOR was not on OPEN, see to it if it is on CLOSE If so, call the node on CLOSED as OLD and add it to the list of BESTNODE's successors. Check to see if the new path or the old path is better just as in step 2(c) & set the parent link - and g & f' values appropriately. If a better path is found to OLD, then we must propogate the improvement to OLD's successors.

e) If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. ~~Compute f'(SUCCESSOR) = g(succes~~
Compute f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)

- Several observations can be made from the algorithm:

① Role of the g function

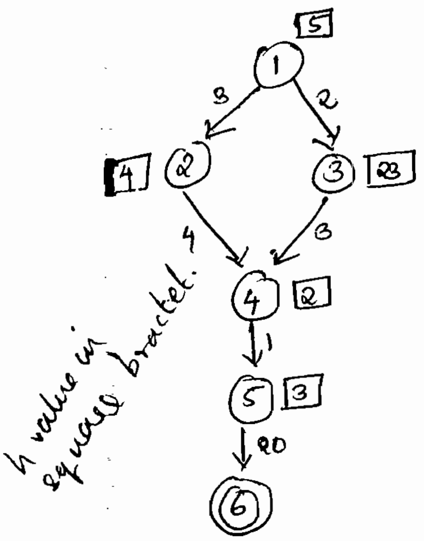
- It lets us choose which node to expand next not only on the basis of how good the node ~~is~~ itself looks (based on h'), but also on the basis of how good the path to the node was.
- This will help us to find the cheapest path to the goal state

② Role of h' function

- h' gives the distance of a node to a goal.
- If h' is a perfect estimator, then A* will converge immediately to the goal with no search.

③ A* is guaranteed to find an optimal path to a goal.

Eg: Showing Best First Search & A* Algorithm



Best First Search.

Open

$1^{(5)}$

$2^{(4)}\ 3^{(23)}$

$4^{(2)}\ 3^{(23)}$

$5^{(3)}\ 3^{(23)}$

$6^{(0)}\ 3^{(23)}$

goal.
terminate.

path: ①→②→④→⑤→⑥

Closed

$1^{(5)}\ 2^{(4)}\ 4^{(2)}$

A* Alg

Open

$1^{(5)}$

$2^{(7)}\ 3^{(25)}$

$3^{(25)}\ 4^{(9)}$

$3^{(25)}\ 5^{(11)}$

$3^{(25)}\ 6^{(28)}$

goal is found
but can't select as there are better path.

$6^{(28)}\ 4^{(7)}$

$6^{(28)}\ 5^{(9)}$

$6^{(28)}\ 6^{(26)}$

better.

So path: ①→③→④→⑤→⑥ (shorter compared to Best First search)

Closed

$1^{(5)}\ 2^{(7)}\ 4^{(9)}\ 5^{(11)}\ 4^{(7)}\ 5^{(9)}$

updated.

Problem Reduction

- Search strategies for OR graphs tries to find a single path to a goal. Out of the many branches available, it follows any one branch to reach a goal.

AND-OR Graphs

- It is useful for representing solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must be solved.

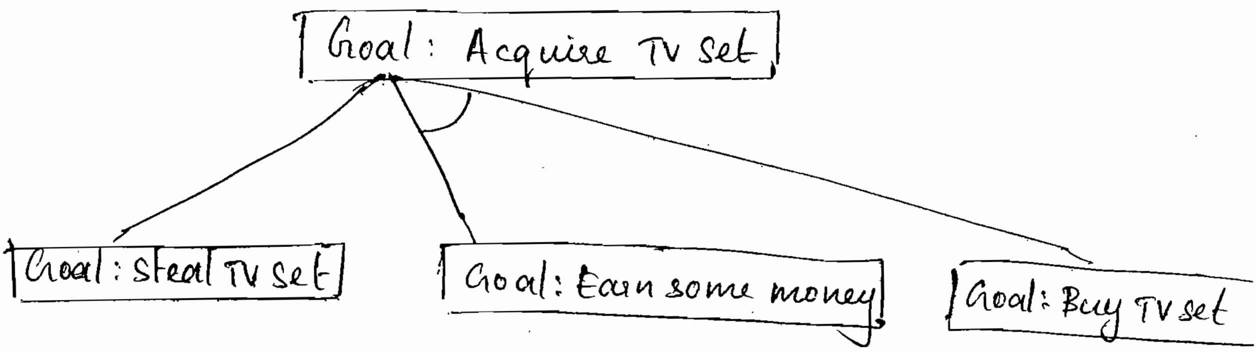
- Arcs generated by this decomposition or reduction is known as AND arcs.

- One AND arc may point to any number of successor nodes, all of which must be solved inorder for the arc to point to a solution.

- The graph is known as AND-OR graph because several arcs ma emerge from a single node, thereby indicating a variety of wa in which the original problem might be solved.

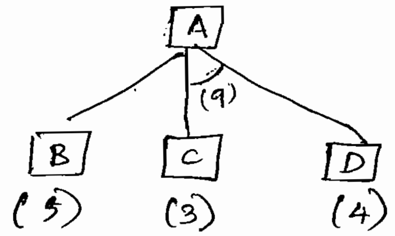
- AND arcs are indicated by a line connecting all the compone

eg: for AND-OR graph is given below:



- Best first Search algorithm is not appropriate to solve AND-OR graph.

eg: Consider AND-OR graph given below.



The top node A has been expanded, producing 2 arcs; one leading to B and one leading to C and D. The numbers at each node represent f' at that node. For simplicity, we assume a uniform cost of 1 along any branch.

If we follow best first search & choose the unexpanded node with the lowest f', then C has to be chosen.
But ~~we~~ looking at the nodes, we can find that C is along a path formed by AND arc. So to solve C, we must also use D, for a total cost of 9 (C+D+2). ~~so~~ So we can say that C is not along the current best path (as path B can be solved with a cost 6).

Hence B has to be expanded before expanding C. So it is not appropriate to use Best First Search for solving AND-OR graph.

→ Inorder to describe an algorithm for searching AND-OR graph, we use a value called FUTILITY. Futility should be chosen to correspond to a threshold such that any solution with a cost above it is too expensive to be practical

... the search is abandoned.

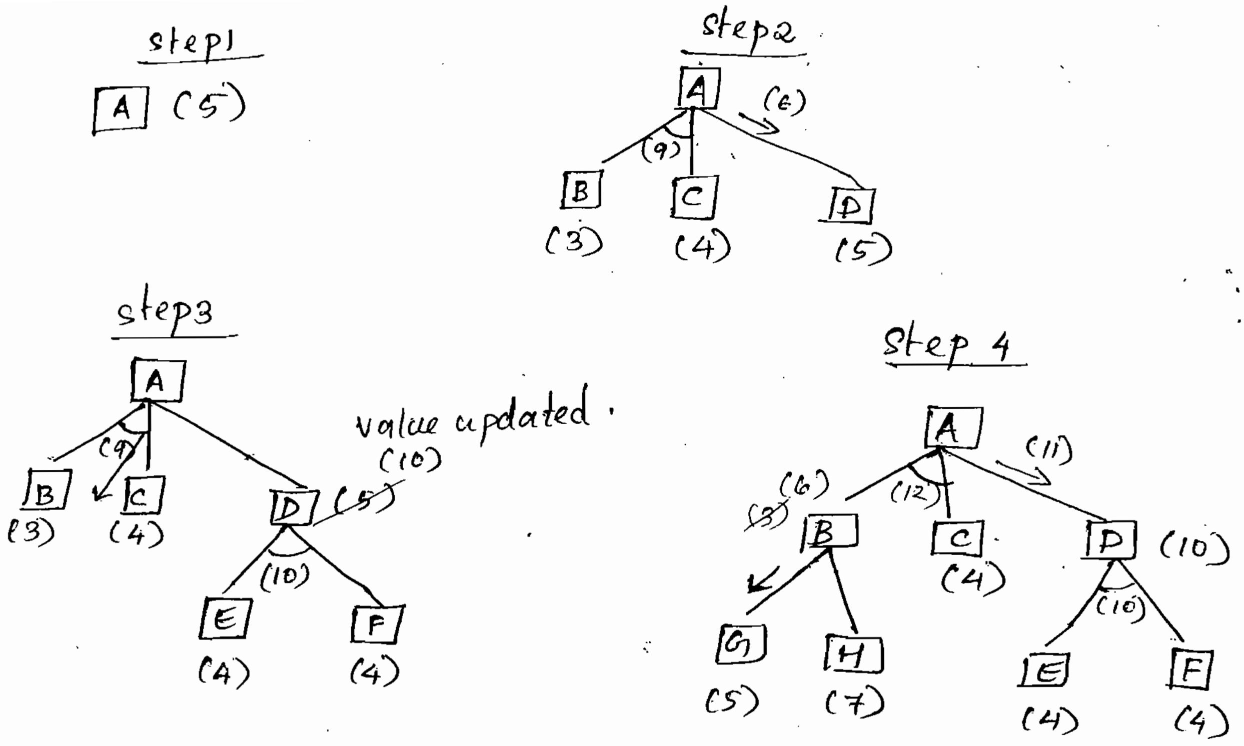
Algorithm: Problem Reduction

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY:

   a) Traverse the graph starting at the initial node & following the current best path, and accumulate the set of nodes that are on the path and have not ye[t] been expanded or labeled as SOLVED.

   b) Pick one of these unexpanded nodes & expand it. If there are no successors, assign Futility as the value of this node. Otherwise, add its successors to the graph & for each of them compute f' (cost of remaini[ng] distance) for each of them.

   c) Change the f' estimate of the newly expanded node to reflect the new information produced by its successo[rs]. Propogate this change backward through the graph. Decide which is the current best path. (If any node contains a successor arc whose descendants are all solved, label the node itself as SOLVED)



AO* Algorithm

→ An algorithm used to find a solution in an AND-OR graph.

- AO* algorithm will use a single structure GRAPH that represents a part of the search space that has been explicitly generated so far.
- Each node in graph will point both down to its immediate successors and upto its immediate predecessors.
- Each node has h' value associated with it (ie estimated cost from node to solution nodes). There is no storing of g values.

Algorithm : AO*

1. let GRAPH consists only of the node representing the initial state and call this node INIT. Compute h'(INIT).

2. Until INIT is labeled SOLVED or until h'(INIT) becomes greater than FUTILITY, repeat the following procedure.

a) Trace the marked arcs from INIT and select an unexpanded node (say NODE) on this path.

b) Generate the successors of NODE. If there are no successors then assign FUTILITY as h'(NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not an ancestor of NODE do the following:

i) Add SUCCESSOR to GRAPH.

ii) If SUCCESSOR is a terminal node, mark it SOLVE and assign it an h' value of 0.

iii) If SUCCESSOR is not a terminal node, compute its h' value.

c) Propogate the newly discovered information up the graph by doing the following. let S be a set of nodes that have been marked SOLVED or whose h' value has been changed & so need to have values propogated back to their parents. Initialize S to NODE. Until S is empty, repeat the following procedure.

i) Select a node from S call it CURRENT and remove it from S.

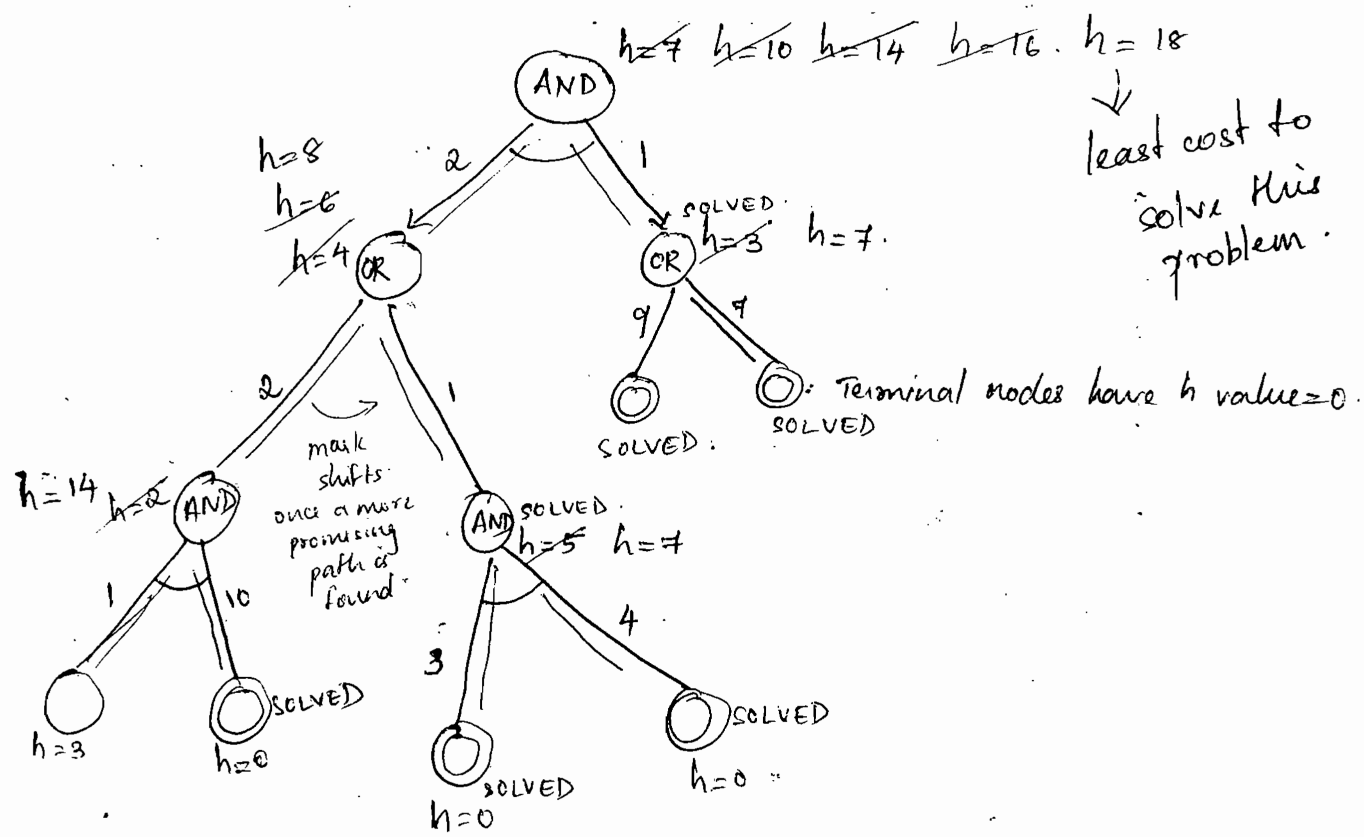
ii) Compute cost of each of the arcs emerging from CURRENT. Cost of each arc is equal to sum of the h' values of each of the nodes at the end of arc plus the cost of arc. Assign CURRENT's new h' value to the minimum of costs computed for the arcs emerging from it.

iii) Mark the best path out of CURRENT by marking the arc that had minimum cost as computed in previous step.

iv) Mark CURRENT SOLVED if all of nodes connected to it through the new marked arc have been labeled SOLVED.

v) If CURRENT has been marked SOLVED or if its h' value has just changed, then its new status must be propogated back up the graph. So add all ancestors of CURRENT to S.

Eg:



shortest distance or least cost of solving this problem is 18.

Constraint Satisfaction

- Many problems in AI can be viewed as problems of constraint satisfaction in which the goal is to discover some problem states that satisfies a given set of constra

- Egs include cryptarithmetic problems or puzzles and many real world perceptual labeling problems.

- Design tasks can also be viewed as constraint satisfactio problem in which a design must be created within fi limits of time, cost and materials.

- In a Constraint Satisfaction Problem (CSP), the states are defined by the values of a set of variables and the goal test specifies a set of constraints that the values ha to obey.

ie CSP is defined by a set of variables, $x_1, x_2 \cdots x_n$ and set of constraints, $c_1, c_2 \cdots c_m$.

Each variable x_i has a non empty domain D_i of possi values that the variables can take. It should be finite.

Problem characterization

State components:
- Variables
- Domains (possible values for variables)
- Constraints b/w variables.

Goal : to find a state (a complete assignment of values to variables, $\{x_i = v_i, x_j = v_j \cdots\}$, which satisfies the constraints.

- An assignment that does not violate any constraints is called consistent or legal assignment.

Examples:

- Map coloring
- Crossword puzzles
- n-queens
- resource assignment/distribution/location

Types of Constraints

1. Unary Constraints

   constraints involving a single variable.

   eg: x_i can take only integers, $x \neq$ Red.

2. Binary constraints

   Constraints involving 2 variables or pairs of variables.

   eg ~~xi~~ 2 variable x_i & x_j can take value such that

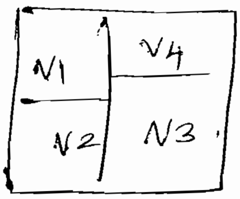
   $x_i < x_j$ or $x_i \neq x_j$ ~~if~~

   eg (1,3) (1,4) (2,3) (2,4) ... if D = (1,2,3,4)

3. Higher order constraints

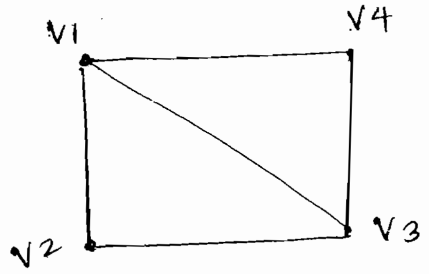
   It involves 3 or more variables.

eg: Consider Map Coloring Problem. (Binary Constraint)



Given: 3 Colors: Red, green and Blue. and there are regions in map.
Assign Colors ~~in~~ such a way that no two neighboring regions have the same color.

- This can be modeled as a graph where we assign color to the vertices which represent each region.



- Variable for each node.
- Domain, D = {Red, Green, Blue}
- Constraint for each edge
  ie Color on one side of edge should be different from color on other side of edge. ie $x_i \neq x_j$.
- Solution gives coloring of vertices which does not violate constraint.
- It is an example for binary constraint.
  As we consider pairs of variables for assigning colors.
  eg (V1, V4), (V1, V3), (V1, V2), (V2, V3) etc.

eg for higher order constraints - Cryptarithmetic puzzles.

- Each letter stands for a distinct digit. Aim is to find a substitution of digits for letter such that the resulting s ie arithmetically correct. with the added restriction that no leading zeroes are allowed.

```
   T W O
 + T W O
 -------
 F O U R
```

Variables : T, W, O, F, U, R

Domain, D = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

2 sets of Constraints

1. AllDiff (T, W, O, F, U, R)

2. Other set of Constraints. (Column Addition)

$$\begin{array}{cccc} & \scriptstyle C_3 & \scriptstyle C_2 & \scriptstyle C_1 \\ & T & W & O \\ & T & W & O + \\ \hline F & O & U & R \end{array}$$

$O + O = R$

$W + W + C1 = U$

$T + T + C2 = O$

$C3 = F$

(Max value = 9
9+9 = 18 cas

Step 1: $C3 = 1$, hence $F = 1$

step 2: $2T + C2 > 9$

Assume $C2 = 1$

then T can take values from 5 to 9.

let us start with $T = 5$

$2T + C2 = 11$

$\therefore O = 1$ but O cannot be 1 as $F = 1$. ×

Take $T = 6$.

$2T + C2 = 13$

$O = 3$

if $T = 6$, $O = 3$ $\therefore O + O = 6$ ie $R = 6$

But $R \neq 6$ since $T = 6$ ×

Take $T = 7$

$2T + C2 = 15$

$\therefore \underline{O = 5}$

Step 3: Since $O = 5$, $O + O = 10$ $\therefore \underline{R = 0}$

~~Step 4: $2W + C1 = U$~~
~~Since F = 1~~

step 4: $2W + C1 = U$
Now since $F = 1$ & $R = 0$, $W \neq 1$ or 0.
Since we have already assumed $C2 = 1$,

since $O + O = 10$ (O=
$C1 = 1$
For $2W + 1 > 9$,
we have values
from 5 to 9
But $W \neq 5$
$\because O = 5$
let us take
$2W + C1 = 13$
$\therefore U = 3$
Thus,
$F = 1$, $R = 0$
$T = 7$, $O =$
$U = 3$, U

CSP as a standard search Problem : Searching is assigning values to a set of variables.

- Incremental formulation.

  Initial state : empty assignment { }.

  Successor function : Assign value to unassigned variable provided that there is no conflict.

  Goal test : Current assignment is correct ie there is no constraint violation.

eg: Consider the 4 Queen Problem.

CSP Algorithm : DFS.

  Is → no queen on 4x4 board. Empty configuration of board

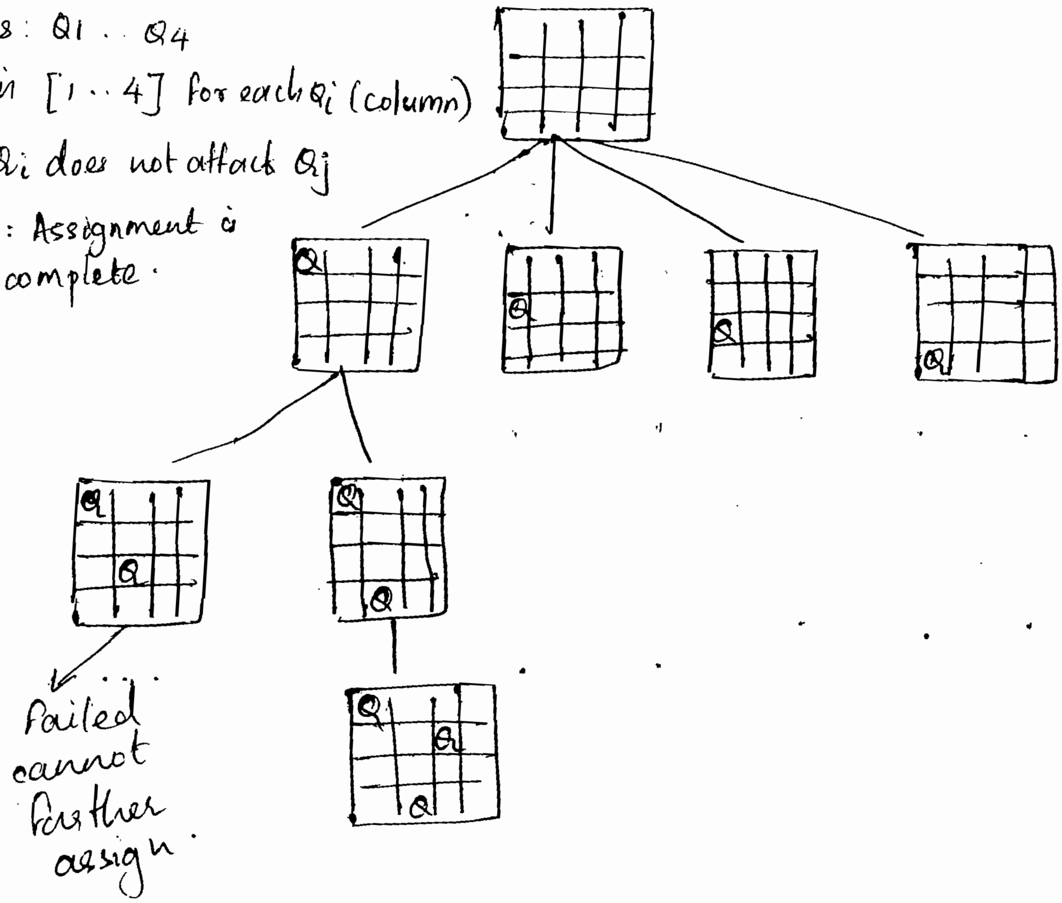
  Successor fn – assign a value to any variable provided no conflict occurs.

variables : Q1 .. Q4

Domain [1 .. 4] for each Q_i (column)

Constr : Q_i does not attack Q_j

Goal Test : Assignment is complete.



Backtracking Search for CSP (Recursive one)

- The term backtracking search is used for depth first search that chooses values for one variable at a time and backtracks when a variable has no legal values to assign.

Algorithm: Backtrack_rec

returns solution ie a Complete assignment or failure.

~~Recu~~

Backtrack_rec (assgn, csp)

- if assgn complete, return assgn (all variable are assigned correct values then return assign as solution)
- else select unassigned variable next one in order
- for each value in Domain of variable in order
  - if consistent (ie value is consistent acc. to constraints)
    - add variable = value to assignment. (call [illegible] assgn.)
    - if recurse succeeds, return assignment.
  - else remove (variable = value)
- return failure

Efficiency of CSP Problem

- Which variable has to be assigned next among unassigned variables. In what order the values should be tried?
- How does the assignment to the current variable influence the assignment for other unassigned variables?
- When a path fails, can the search avoid repeating suc

failures in subsequent paths?

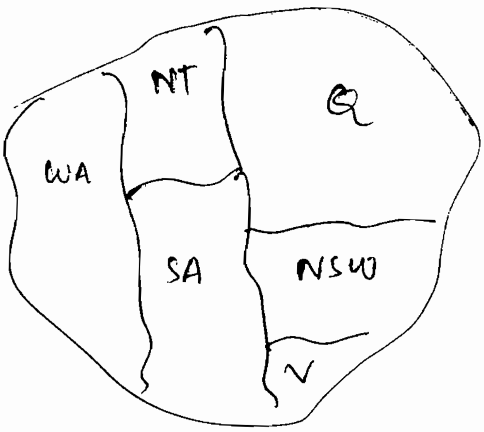
Variable ~~Or~~ and Value Ordering

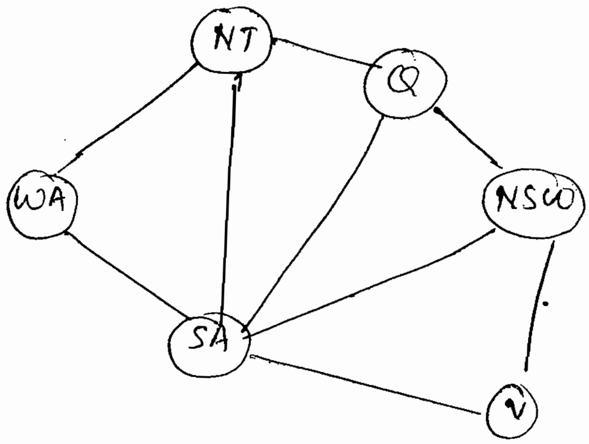
Select the next unassigned variable in such a way that.

1. Minimum Remaining Value heuristic or the Most Constraint Variable heuristic : Choose the variable with the fewest legal values.

2. Degree heuristics : Select the variable involved in the largest numbers of constraints on other unassigned variables.

eg:





case1

D = { Red, Green, Blue }

If ~~R.~~ WA = Red.

then NT can take values Green, Blue.

SA can take values Green, Blue.

Q, NSW and V can take any ~~1 of~~ among 3 values in Domain set.

The next variable chosen ~~is the~~ for assignment is the one with the fewest values which is NT and SA ~~to~~.

If NT = Green, then SA is next assigned with Blue as it is the only value it can take. Variables are selected in this manner

This is known as Minimum Remaining Value heuristic.

case 2 Degree heuristics

If we are not sure how to start assignment of values to variables → degree heuristics can be followed.

Find out degree of each node.

WA = 2, NT = 3, SA = 5, Q = 3, NSW = 3, V = 2

choose the node with highest degree and assign a value for the Domain set.

eg: SA = Red.

then choose the next node with highest degree & go on assigning values satisfying constraints.

It is likely to avoid failures in this manner. ie Success rate is high in the case of degree heuristics.

Propagating Information through Constraints

simplest propagation is Forward Checking.

Forward Checking

whenever a variable X is assigned, the forward checking proc looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X.

Fig shows progress of map-coloring search with forward checki

	WA	NT	Q	NSW	V	SA	T
Initial Domain	RGB	RGB	RGB	RGB	RGB	RGB	RGB
After WA = R	R	GB	RGB	RGB	RGB	GB	RGB
After Q = G	R	B	G	RB	RGB	B	RGB
After V = B	R	B	G	R	B		RGB

WA = Red is assigned first; then forward checking deletes red from the domains of the neighboring variables NT & SA. After Q = Green, green is deleted from the domains of NT, SA and NSW. After V = blue, blue is deleted from the domains of NSW and SA, leaving SA with no legal values.

- Forward checking detects the partial assignment as inconsistent when the assignment leaves a variable with no legal value. Hence the algorithm will backtrack immediately.

Data structure used.

→ ~~Var~~ For every x_i maintain a Current Domain, CD_i

- Initially set $CD_i = D_i$
- When we set variable $x_j = v$
  - remove $x_i = u$ from CD_i (ie domain of x_i) if some constraint is not consistent with both $x_j = v$ and $x_i = u$
- Stop search when CD_i = null for some variable x_i.

eg: Constraint Propagation in Graph Coloring Problem

After a node is instantiated with a color, propagate the color.

PropagateColor (node, color)

1. Remove color from all available ~~CD~~ current Domains of uninstantiated neighbours.
2. If any of these neighbours get the empty set, backtrack
3. Foreach n in these neighbours: if n previously had two or more available colors but now has only one color, c run PropagateColor(n, c).

Games as Search Problems

- Games provide a Structured task in which it is very easy to measure success or failure.
- They donot require large amounts of knowledge
- Games can be good models of competitive situations, so princi discovered in game playing programs can be applied to pract problems.
- To improve the effectiveness of a search based problem solving program two things can be done
    - Improve the generate procedure so that only good moves (or paths) are generated.
    - Improve the test procedure so that the best moves (or p will be recognized & explored first.
- As the number of legal moves available increases, it becomes increasingly important to apply heuristics to select only those that have some kind of promise.
- In Games as Search Problems:

  Initial State : Current board/position

  Operators : legal moves & their resulting states

  Terminal test : to decide whether game has ended.

  A utility function (Payoff function) : produces a numerical va for each of the node on the tree.

Game Trees

- The sequence of states formed by possible moves of a game is called a game tree.
- Each level of the tree is called a ply.
- In a 2 player game, the 2 players are called max and min players. ie we have 2 sets of nodes Max nodes & Min nodes
- Max node tries to maximize the value whereas min nodes tries to minimize it.
- At each ply the turn switches to the other player. Max is the first player and Min is the second player.
- Each player needs a strategy. eg: the strategy for MAX player is to reach a winning terminal state regardless of what MIN does.

Minimax Search Procedure

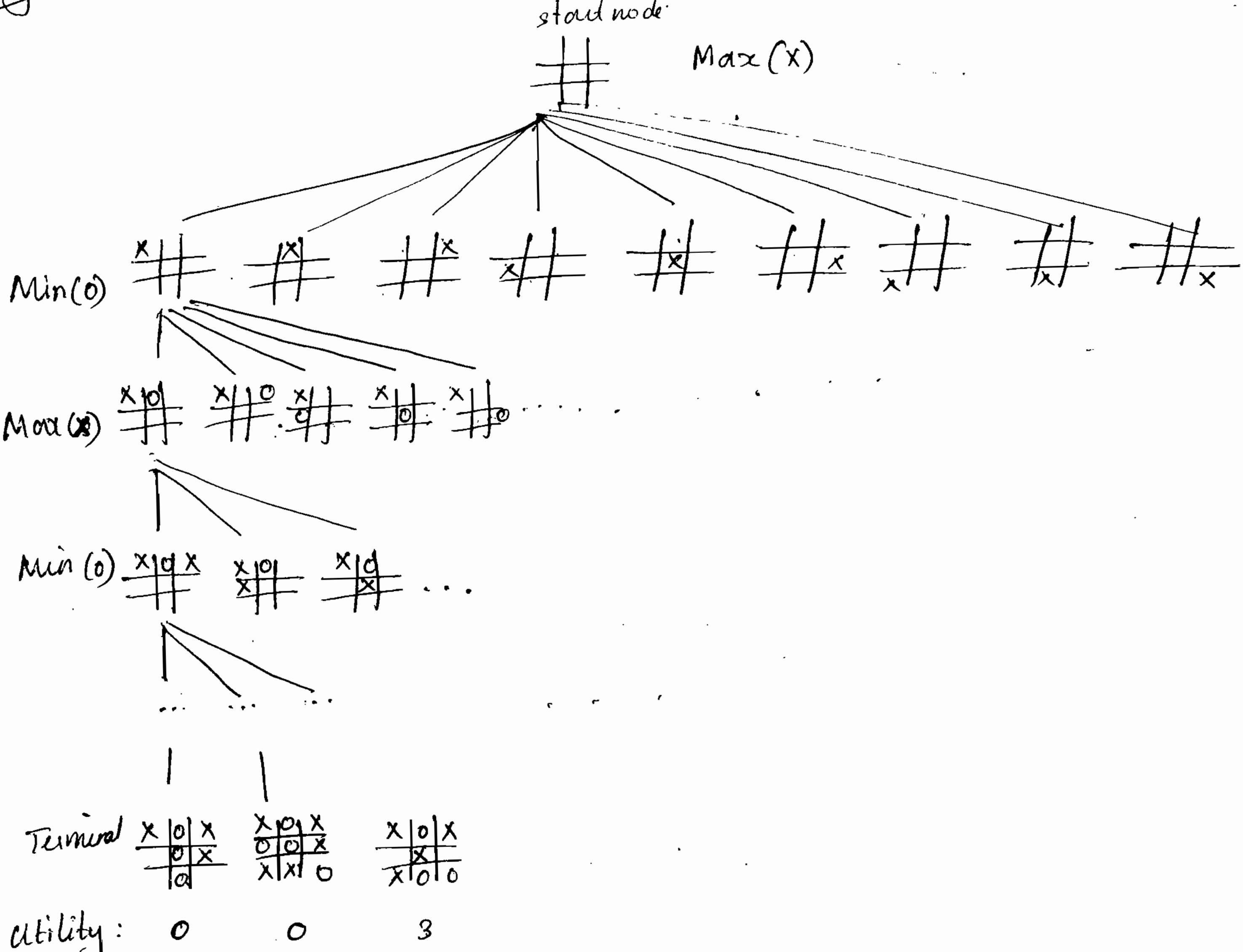
(usually used in 2-player games like Tic Tac Toe, Chess etc)

- Minimax Game trees is used for programming computers to play games in which there are two players taking turns to play moves
- It is an OR graph with two types of OR nodes, namely MAX and MIN.
- In MIN node select the minimum cost successor.
  In MAX node select the maximum cost successor.
  Terminal nodes are winning or loosing states
- The kind of games considered here are called Zero sum games. (Sharing the cost b/w 2 players).

W (cost) → player A
W (cost) → player B

If a win for one player, then it is a loss of another.

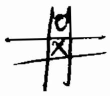
e.g. Example, let us look at a partial search tree for Tic Tac Toe.



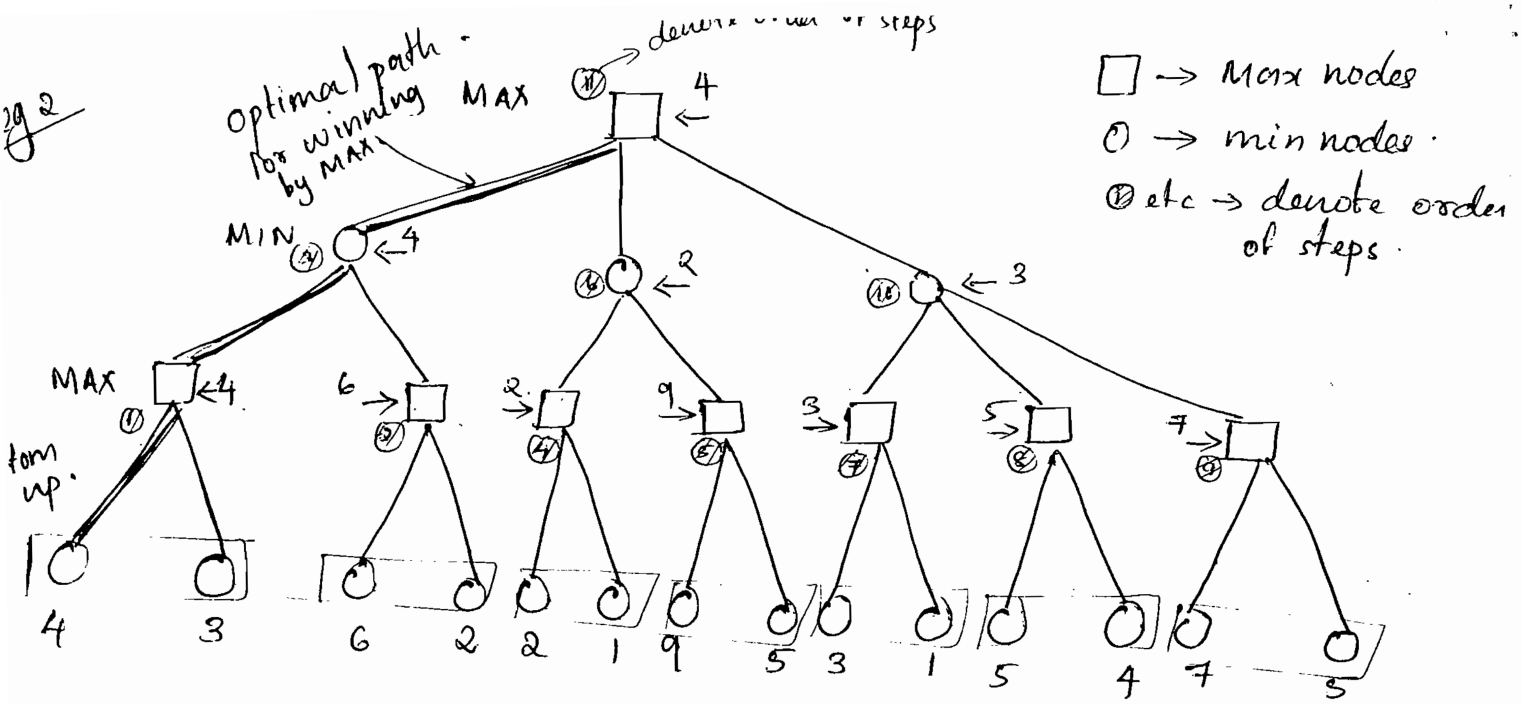
Utility function value at terminal node in the case of tic-tac-toe is

f(n) = (number of complete rows, columns or diagonal that are still open for Max) − (Number of complete rows, columns and diagonals open for MIN)

(where n is terminal node)

eg:  will give f(n) = 6 − 4 = 2.

Pg 2



~~Thus~~

- By looking at the order of steps, it is clear that Minimax algorithm follows a Depth First Search to get an optimal path.

Steps of Minimax Search or Algorithm : Minimax

1. A search tree is generated, depth first starting & with the current game position upto the end game position.

2. Compute the values (through the utility function) for all the terminal states.

3. Afterwards, compute the utility of the nodes one level higher up in the search tree (up from the terminal states). The nodes that belong to the MAX player receive the maximum value of its children. The nodes for the MIN player will select minimum value of its children.

4. Continue backing up values from the leaf nodes towards the root.

5. When the root is reached, Max chooses the move that le[ads] to the highest value (optimal move).

→ Optimal move or strategy is determined by examining the minimax value of each node.

$$\text{MINIMAX\_VALUE}(n) = \begin{cases} \text{UTILITY}(n) & \text{if } n \text{ is a terminal state} \\ \max_{x \in \text{SUCCESSORS}(n)} \text{MINIMAX\_VALUE}(x) & \text{if } n \text{ is a max node} \\ \min_{x \in \text{SUCCESSORS}(n)} \text{MINIMAX\_VALUE}(x) & \text{if } n \text{ is a min node.} \end{cases}$$

→ The minimax algorithm performs a complete depth first exploration of the game tree.
If the maximum depth of tree is d, and there are b legal moves at each point, the following are the measurements:

Performance Analysis

Complete : Yes if tree is finite.

Optimal : Yes, against an optimal opponent

Time Complexity : $O(b^d)$

Space Complexity : $O(bd)$

[→ Note: Minimax Algorithm is at the end of the module]

Alpha-Beta Pruning

- Alpha-Beta pruning is a procedure to reduce the amount of computation and searching during MINIMAX.
- The process of eliminating a branch of the search tree from consideration without examining it is called pruning the search

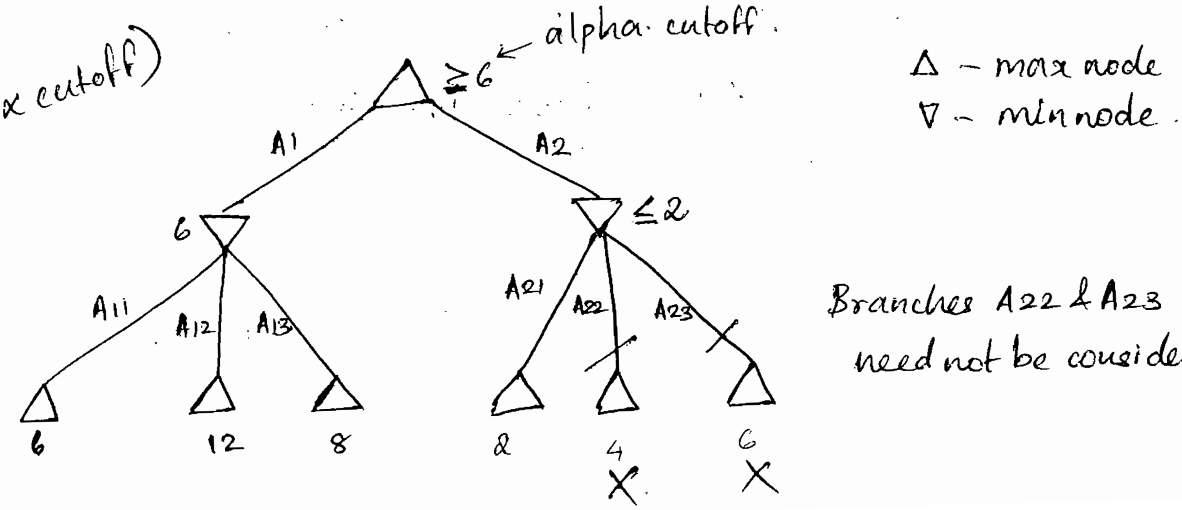
- An alpha value (or α cutoff) is an initial or temporary value associated with a MAX node. Because MAX nodes are given the maximum value among its children, alpha value can never decrease, it can only go up.

- A beta value (or β cutoff) is an initial or temporary value associated with a MIN node. Because MIN nodes are given minimum value among their children, a beta value can never increase, it can only go down.

- For example, suppose a MAX node's alpha = 6. Then the search needn't consider any branches starting from a MIN descendant that has a beta value that is less than or equal to 6. So if a MAX node has an alpha of 6, and if we know that one of its MIN descendants has a beta value less than 6, then no need to search further below the MIN node. This is known as alpha pruning.

  The reason is that no matter what happens below that MIN node, it cannot take on a value that is >6. So its value cannot be propagated upto its MAX (alpha) parent.

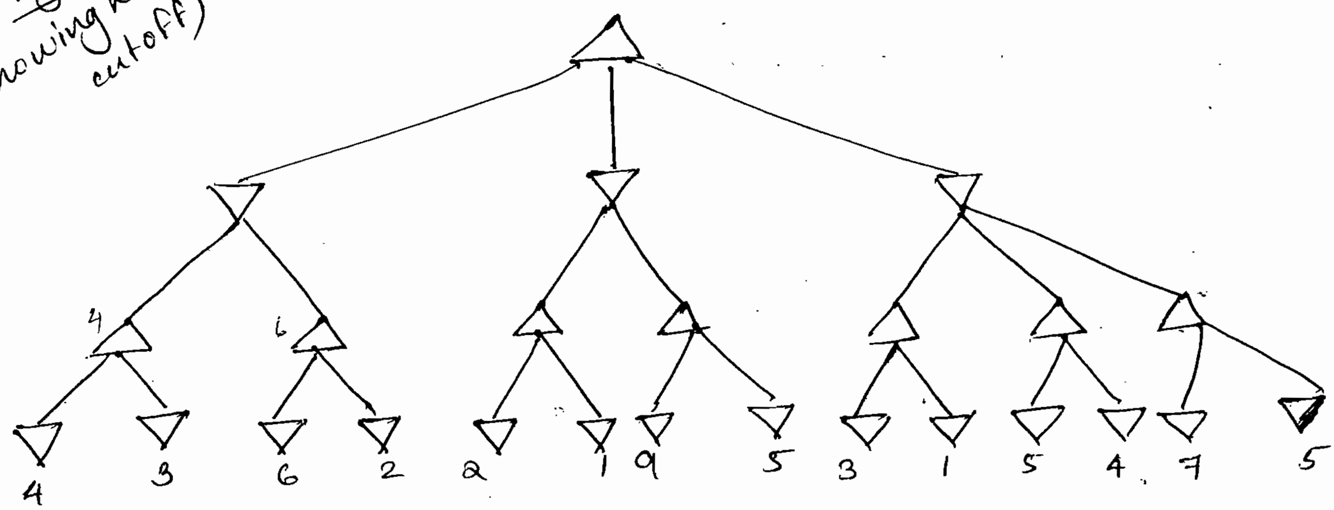
Fig 1 (showing α cutoff)



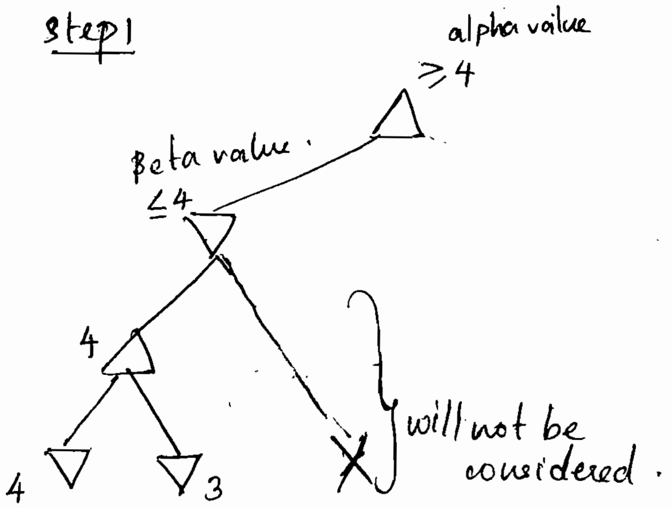
Branches A22 & A23 need not be considered.

- Similarly if a MIN node's beta value = 6, there is no need to search any further below a descendant MAX that has acquired an alpha value of 6 or more.

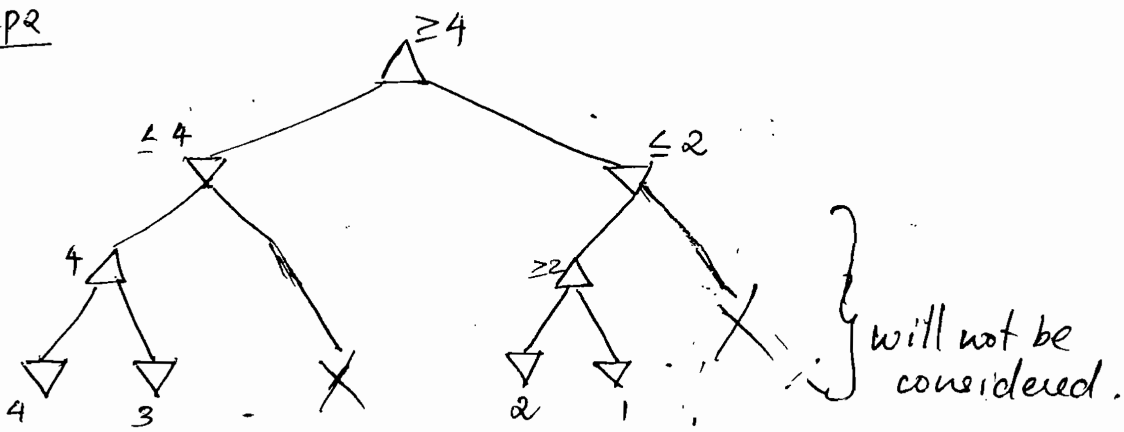
fig e (showing α & β cutoff)



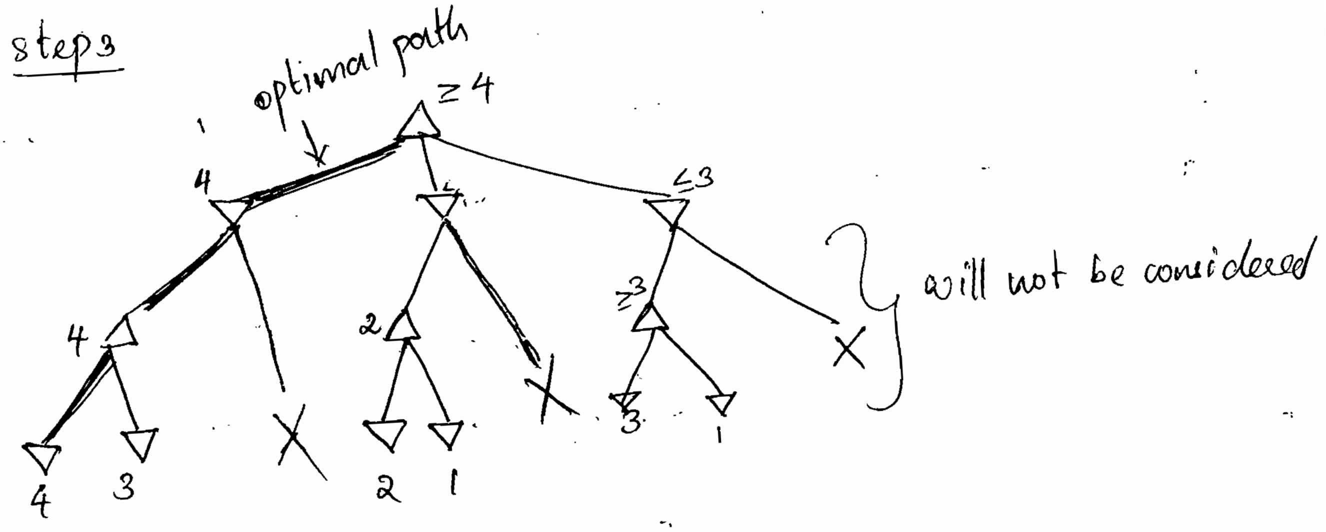
step 1



step 2



step3



Alpha-beta pruning algorithm.

Initially alpha = $-\infty$, beta = ∞

Algorithm F' (position p, value alpha, value beta) // max node

- determine the successor positions $p_1 \ldots p_b$
- if b=0, then return f(p) else begin

    m := alpha

    for i := 1 to b do

        t := G' (p_i, m, beta)

        if t > m then m := t

        if m ≥ beta then return(m) // beta cut off.

- end
- return m

Algorithm G' (position p, value alpha, value beta) // min node.

- determine the successor positions $p_1 \ldots p_b$.
- if b = 0, then return f(p) else begin

    m := beta

    for

for i := 1 to b do

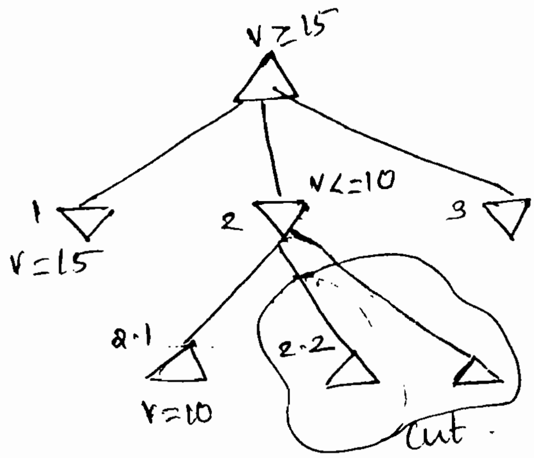
    t := F'(p_i, alpha, m)

    if t < m then m := t

    if m ≤ alpha then return(m) // alpha cut off

- end
- return m

Example

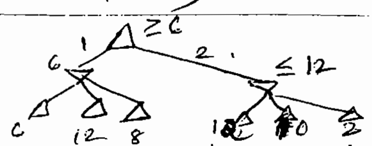


Initial call : F'(root, −∞, ∞)

- m = −∞
- call G'(node1 , −∞, ∞)
  - it is a terminal node
  - return value 15
- t = 15
  - since t > m , m = 15
- call G' (node 2, 15, ∞)
  - call F'(node 2.1 , 15, ∞)
  - ~~t is a~~ terminal node, return 10
  - t = 10
  - alpha = 15 , m = 10 so we have an alpha cut off.
  - so no need to F'(node 2.2 , 15, 10)

...

eg:



Disadv of alg

- Dependent on Search Order (ie the order of values in terminal nodes)
  (in the eg shown, no branches would have been eliminated)

Minimax Algorithm

a node or a state

Alg F' (position P) // max node

- determine the successor positions $p_1, \cdots p_b$
- if $b=0$, then return $f(p)$ else begin

    $m := -\infty$

    for $i=1$ to b do

        $t = G'(p_i)$ // call alg for min node

        if $t > m$ then $m = t$ // find max value

- end
- return m

Alg G' (position P) // min node

- determine the successor positions $p_1 \cdots p_b$
- if $b=0$, then return $f(p)$ else begin

    $m = \infty$

    for $i=1$ to b do

        $t = F'(p_i)$ // call alg for max node

        if $t < m$ then $m = t$ // find min value

- end
- return m