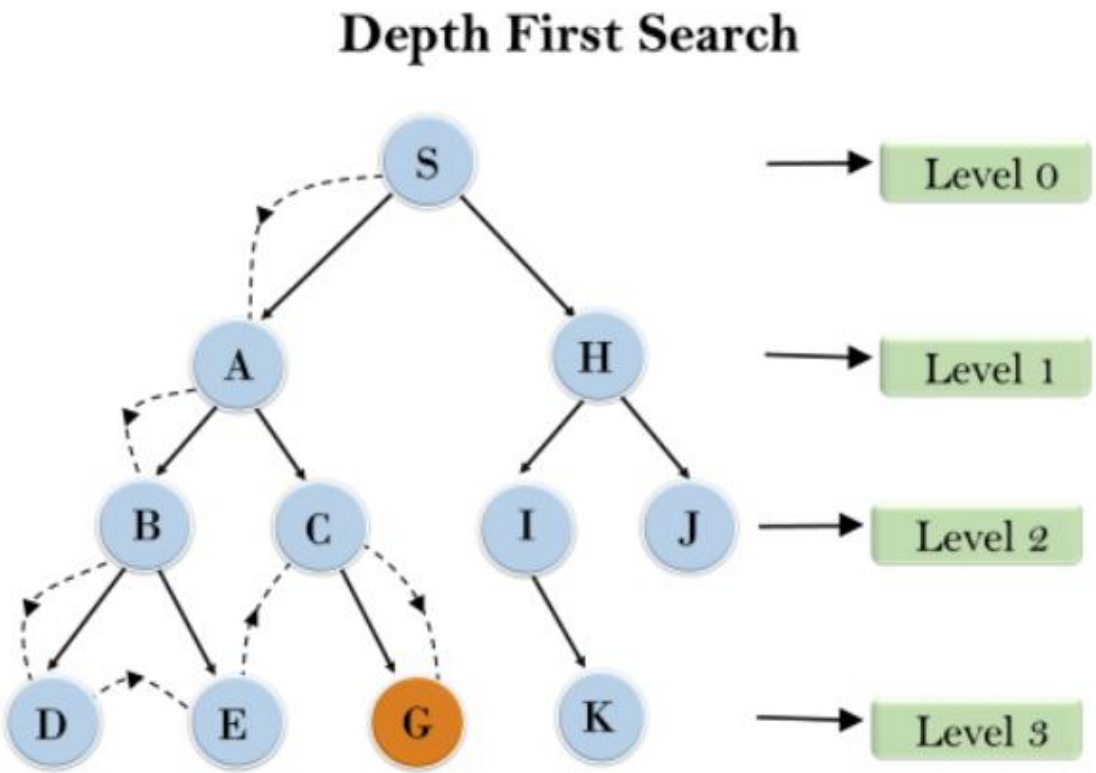
THE FUNDAMENTALS OF ARTIFICIAL INTELLIGENCE

BY VAISHNAVEE RATHOD

D F S

- In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:
- Root node--->Left node ----> right node.
- It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.
- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
- **$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$**
- **Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)**
- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

DFS


Depth First Search
S
A
H
B
C
I
J
D
E
G
K
Level 0
Level 1
Level 2
Level 3

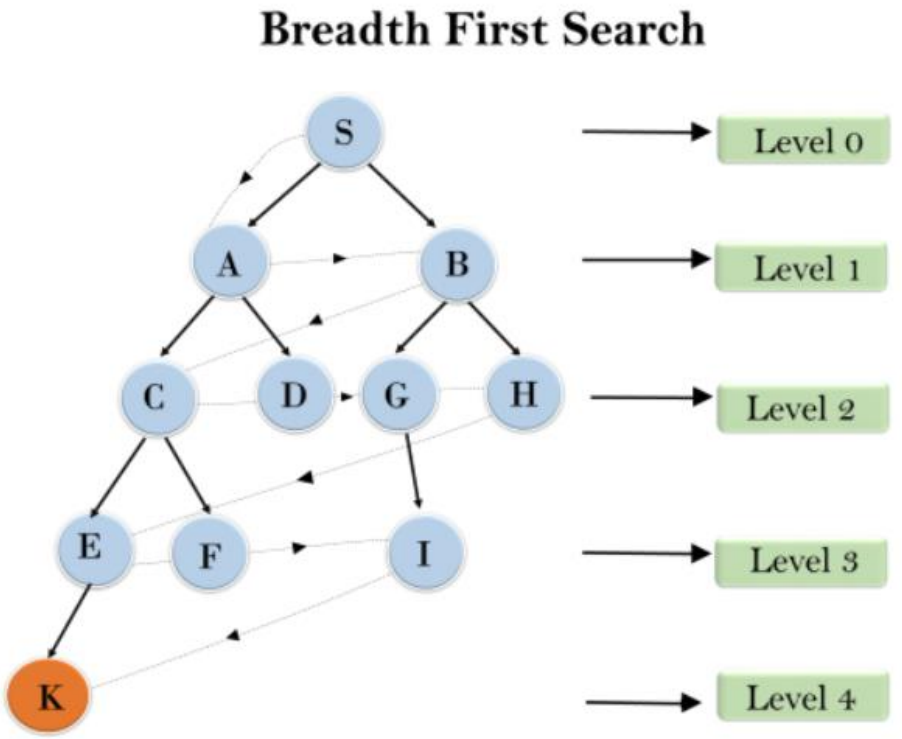
B F S

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.
- **$T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)$**
- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.
- **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

BFS

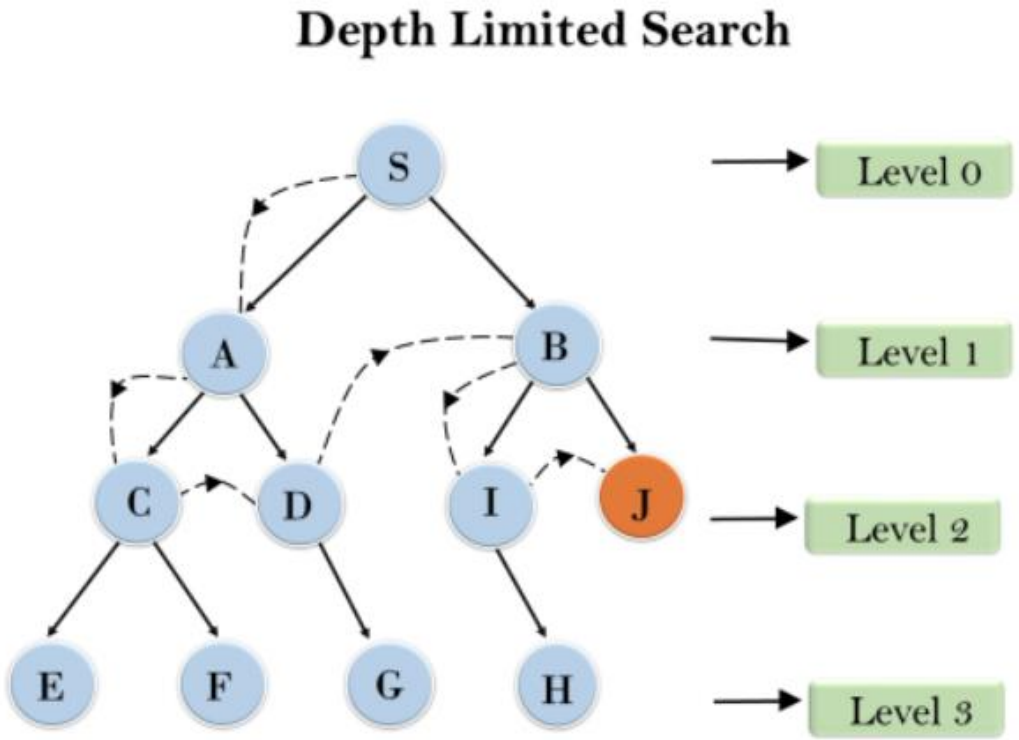

Breadth First Search
S
A
B
C
D
G
H
E
F
I
K
Level 0
Level 1
Level 2
Level 3
Level 4

DEPTH LIMITED SEARCH DLS

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

- **Depth-limited search can be terminated with two Conditions of failure:**
- **Standard failure value:** It indicates that problem does not have any solution.
- **Cutoff failure value:** It defines no solution for the problem within a given depth limit.
- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$** where b is the branching factor of the search tree, and l is the depth limit.
- **Space Complexity:** Space complexity of DLS algorithm is **$O(b \times \ell)$** where **b** is the branching factor of the search tree, and **l** is the depth limit.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

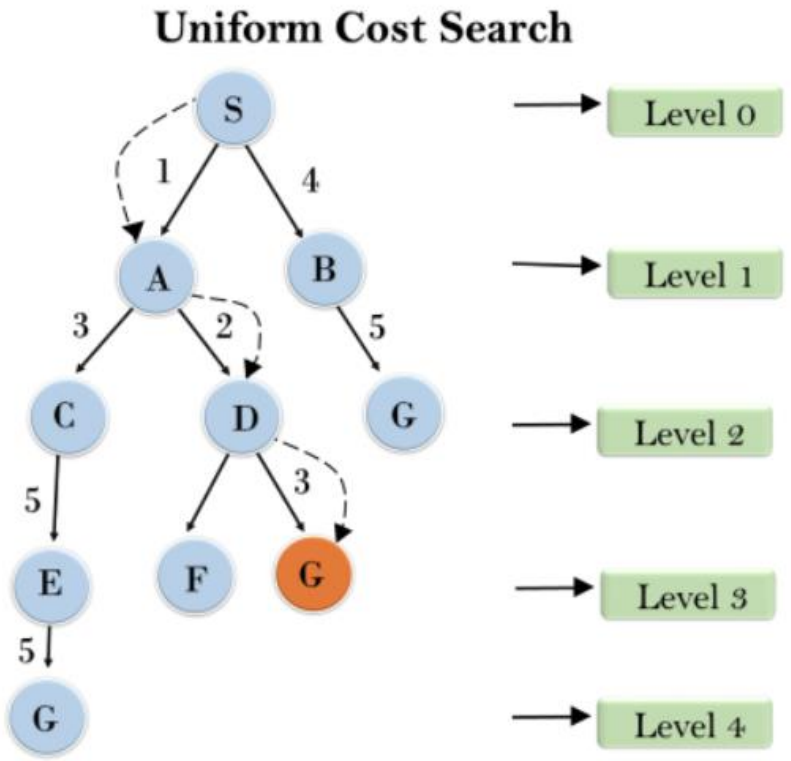
D L S


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

U C S

- **Completeness:**
- Uniform-cost search is complete, such as if there is a solution, UCS will find it.
- **Time Complexity:**
- Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.
- Hence, the worst-case time complexity of Uniform-cost search is$\mathbf{O(b^{1 + [C^*/\varepsilon]})/}$.
- **Space Complexity:**
- The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$.
- **Optimal:**
- Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

UCS

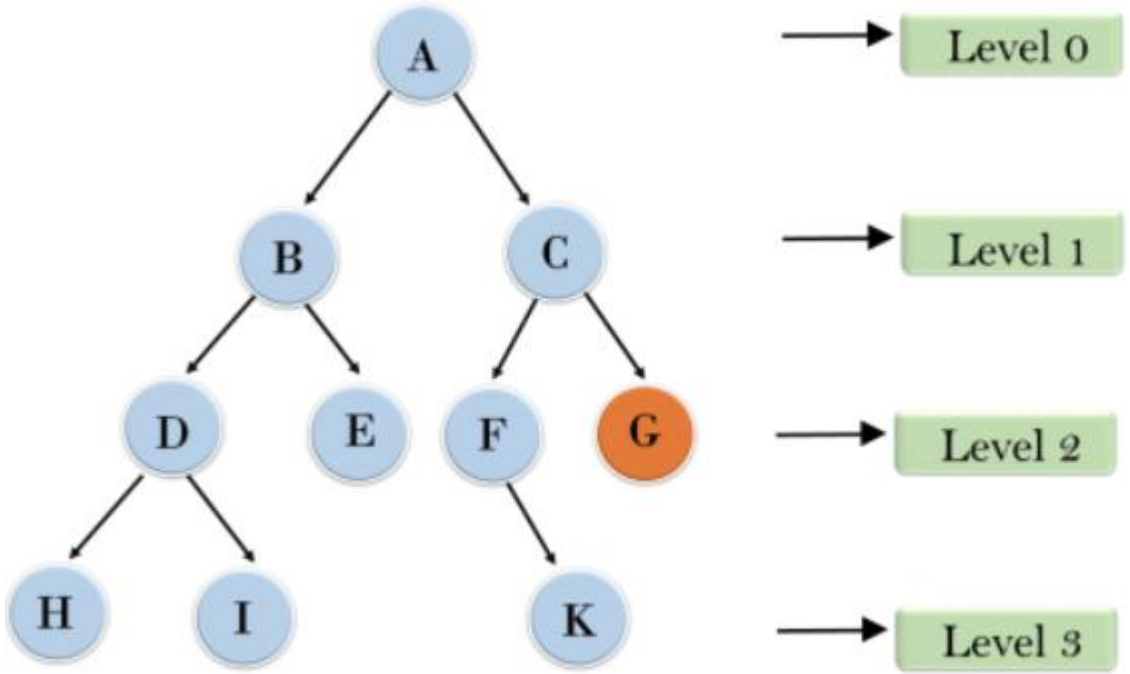

Uniform Cost Search
S
A
B
C
D
G
E
F
G
G
1
4
3
2
5
5
3
5
Level 0
Level 1
Level 2
Level 3
Level 4

I D D F S

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

- **Completeness:**
- This algorithm is complete is ifthe branching factor is finite.
- **Time Complexity:**
- Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(b^d)**.
- **Space Complexity:**
- The space complexity of IDDFS will be **O(bd)**.
- **Optimal:**
- IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

IDDFS

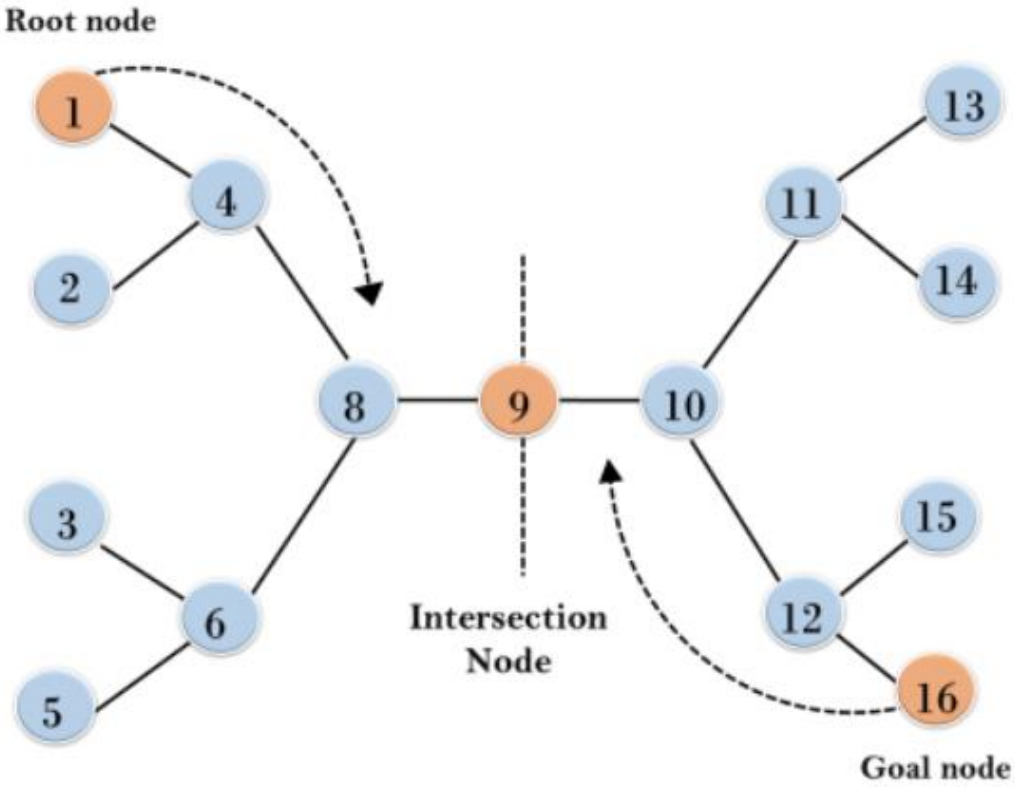


BIDIRECTIONAL SEARCH

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is **$O(b^d)$**.
- **Space Complexity:** Space complexity of bidirectional search is **$O(b^d)$**.
- **Optimal:** Bidirectional search is Optimal.

BIDIRECTIONAL SEARCH



HILL CLIMBING

In an 8-puzzle game, we need to rearrange some tiles to reach a predefined goal state. Consider the following 8-puzzle board.

- This is the goal state where each tile is in correct place. In this game, you will be given a board where the tiles aren’t in the correct places. You need to move the tiles using the gap to reach the goal state.
- Suppose f (n) can be defined as: the number of misplaced tiles.

HILL CLIMBING

In the above figure, tiles 6, 7 and 8 are misplaced. So f (n) = 3 for this case.

For solving this problem with hill climbing search, we need to set a value for the heuristic. Suppose the heuristic function h (n) is the lowest possible f (n) from a given state. First, we need to know all the possible moves from the current state. Then we have to calculate f(n) (number of misplaced tiles) for each possible move. Finally we need to choose the path with lowest possible f (n) (which is our h (n) or heuristic).

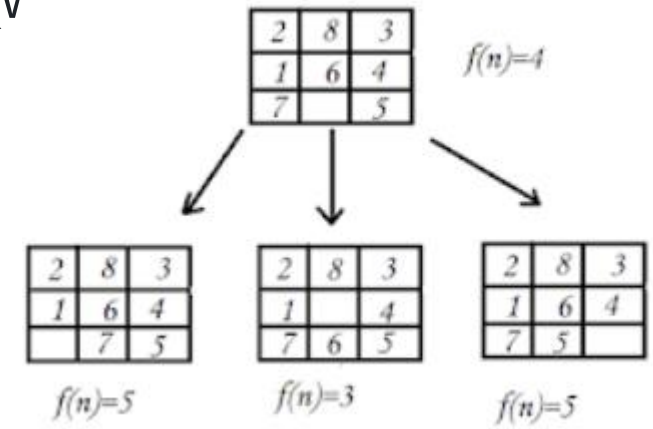
HILL CLIMBING

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

HILL CLIMBING

- In the above figure, tiles 6, 7 and 8 are misplaced. So f (n) = 3 for this case.
- For solving this problem with hill climbing search, we need to set a value for the heuristic. Suppose the heuristic function h (n) is the lowest possible f (n) from a given state. First, we need to know all the possible moves from the current state. Then we have to calculate f(n) (number of misplaced tiles) for each possible move. Finally we need to choose the path with lowest possible f (n) (v



HILL CLIMBING

- Consider the figure above. Here, 3 moves are possible from the current state. For each state we have calculated f (n). From the current state, it is optimal to move to the state with f (n) = 3 as it is closer to the goal state. So we have our h (n) = 3.
- However, do you really think we can guarantee that it will reach the goal state? What will you do if you reach on a state(Not the goal state) from which there are no better neighbour states! This condition can be called a local maxima and this is the problem of hill climbing search. Therefore, we may get stuck in local maxima. In this scenario, you need to backtrack to a previous state to perform the search again to get rid of the path having local maxima.
- What will happen if we reach to a state where all the f (n) values are equal? This condition is called a plateau. You need to select a state at random and perform the hill climbing search again!

Practice solving 8 puzzle problem, Travelling Salesman problem and other problems using uninformed and informed searches

ADVERSARIAL SEARCH

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

- In previous topics, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
- But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
- The environment with more than one agent is termed as **multi-agent environment**, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, **Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games**.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

FORMULATION OF A PROBLEM

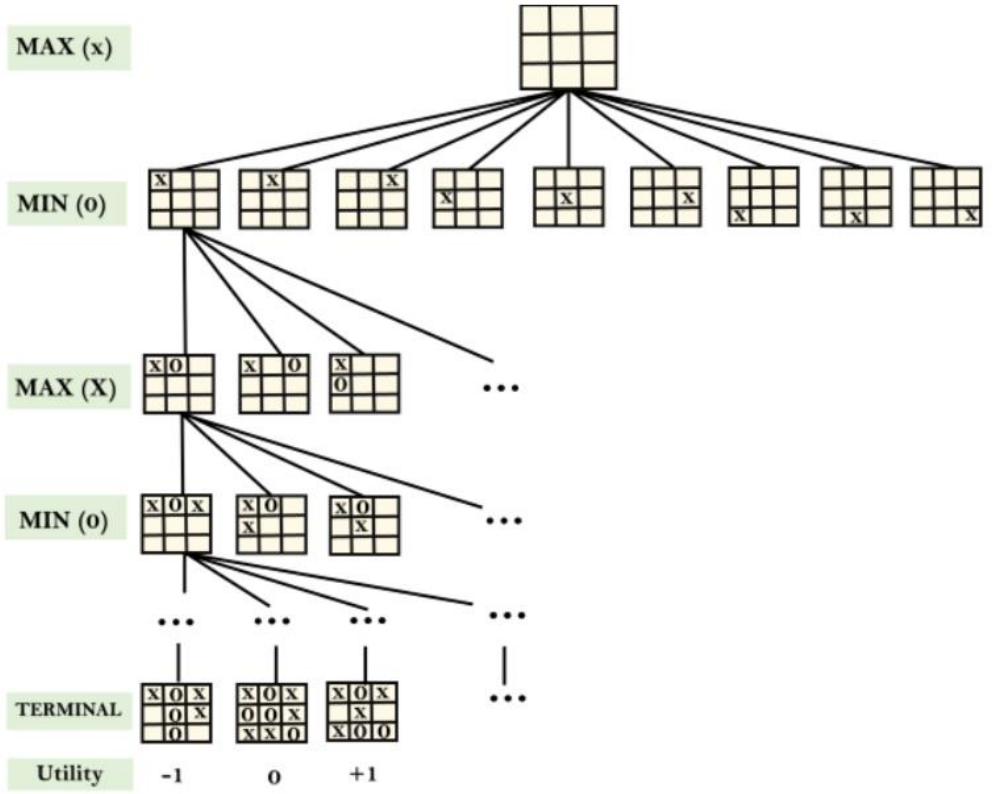
A game can be defined as a type of search in AI which can be formalized of the following elements:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

TIC-TAC-TOE GAME TREE

- The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:
- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.

TIC-TAC-TOE GAME TREE



TIC-TAC-TOE GAME TREE

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

TIC-TAC-TOE GAME TREE

In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

$$
\begin{cases}
\text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\
\max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\
\min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN.}
\end{cases}
$$

ALPHA-BETA PRUNING

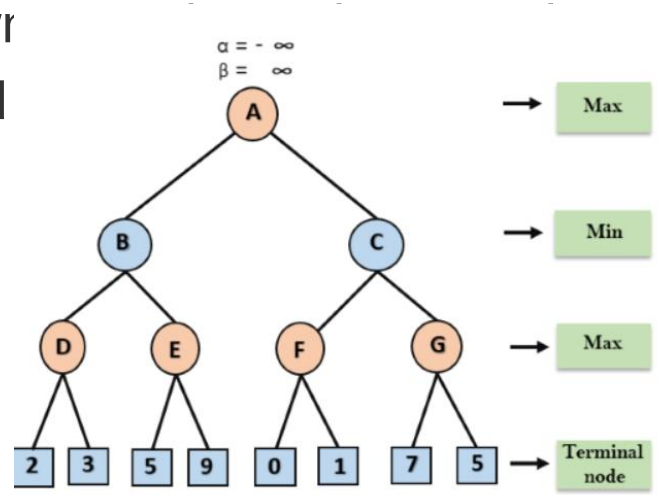
- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

ALPHA BETA PRUNING

- The two-parameter can be defined as:
  - **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  - **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.
- The main condition which required for alpha-beta pruning is: $\alpha >= \beta$

ALPHA BETA PRUNING

- Let's take an example of two-player search tree to understand the working of Alpha-beta pruning
- **Step 1:** At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed dowr -∞ and β= +∞, and Node B passes the same value to its child

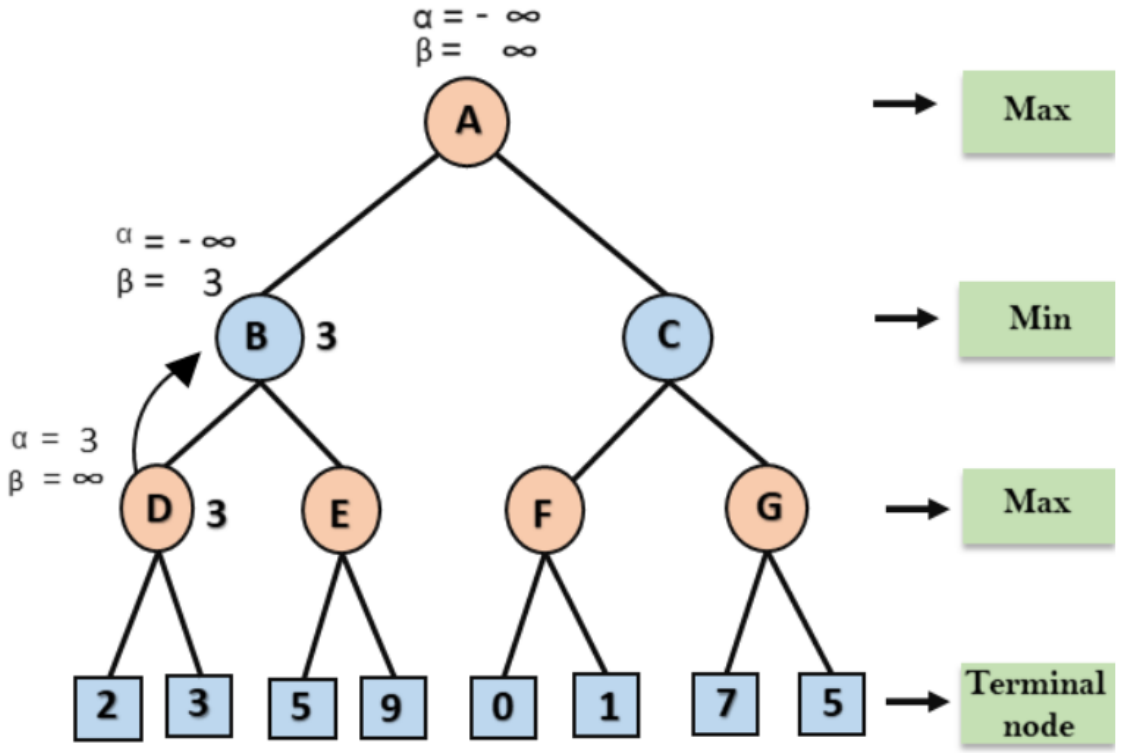


ALPHA BETA PRUNING

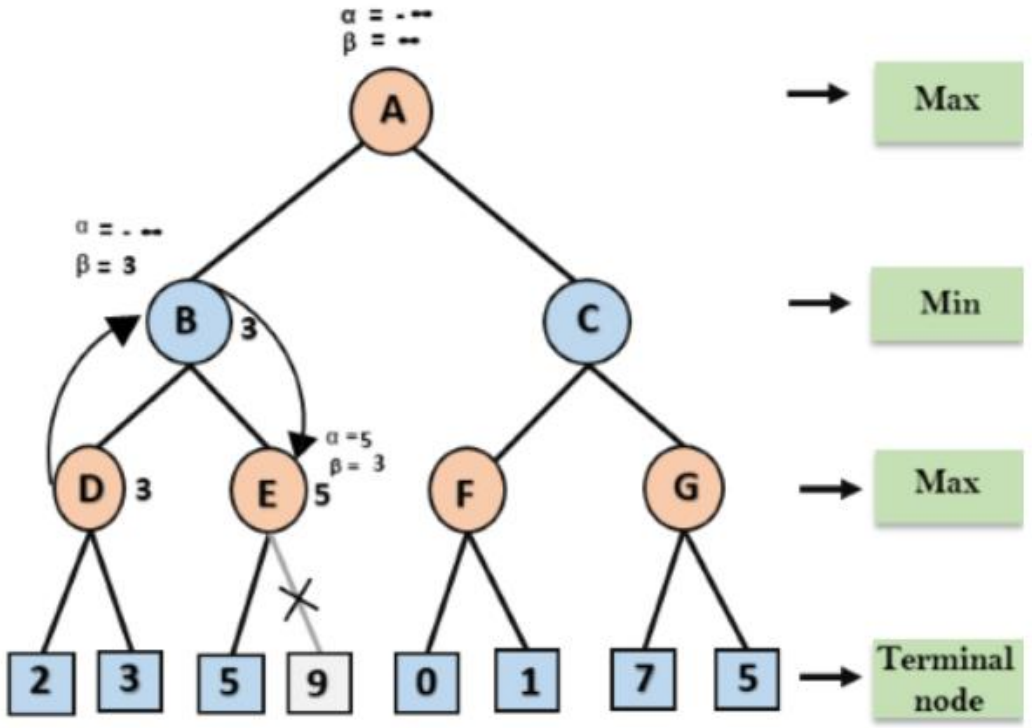
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

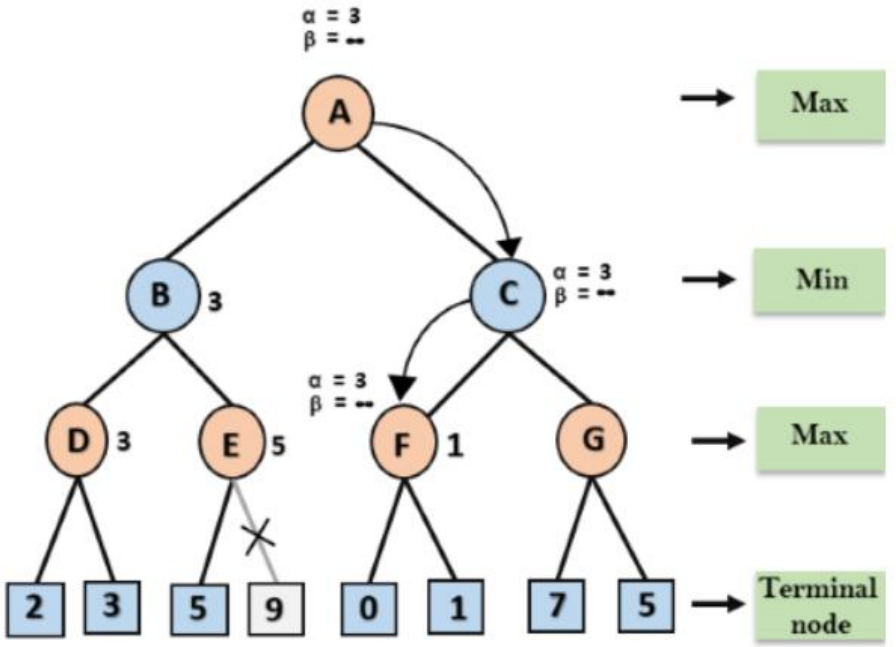
ALPHA BETA PRUNING



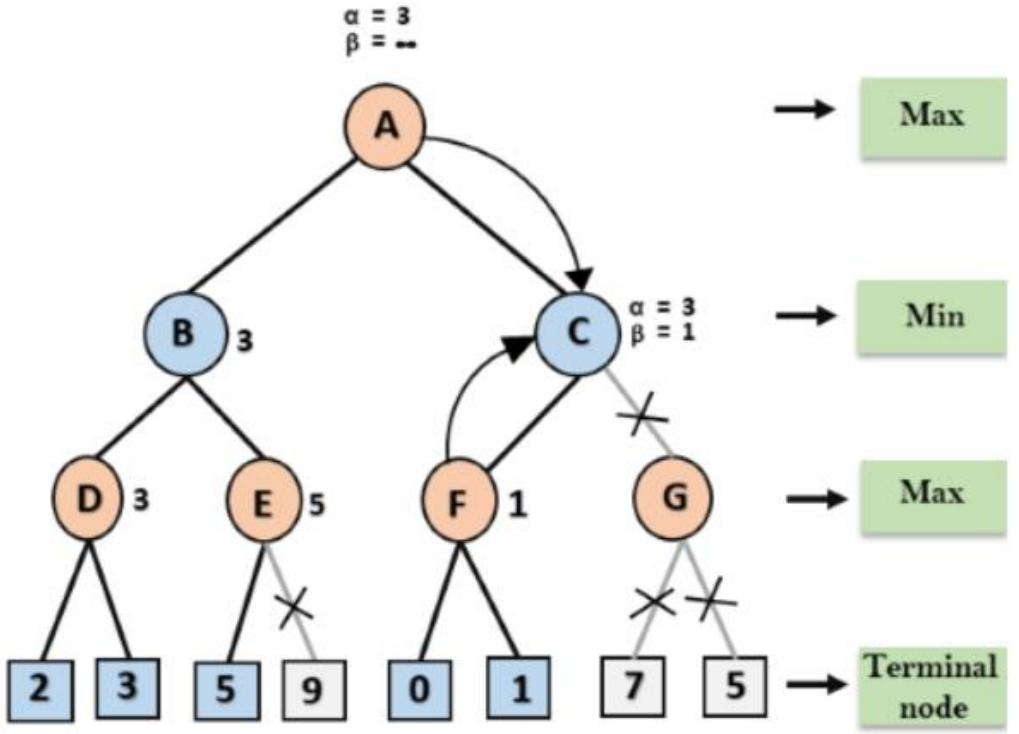
- In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.
- **Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node



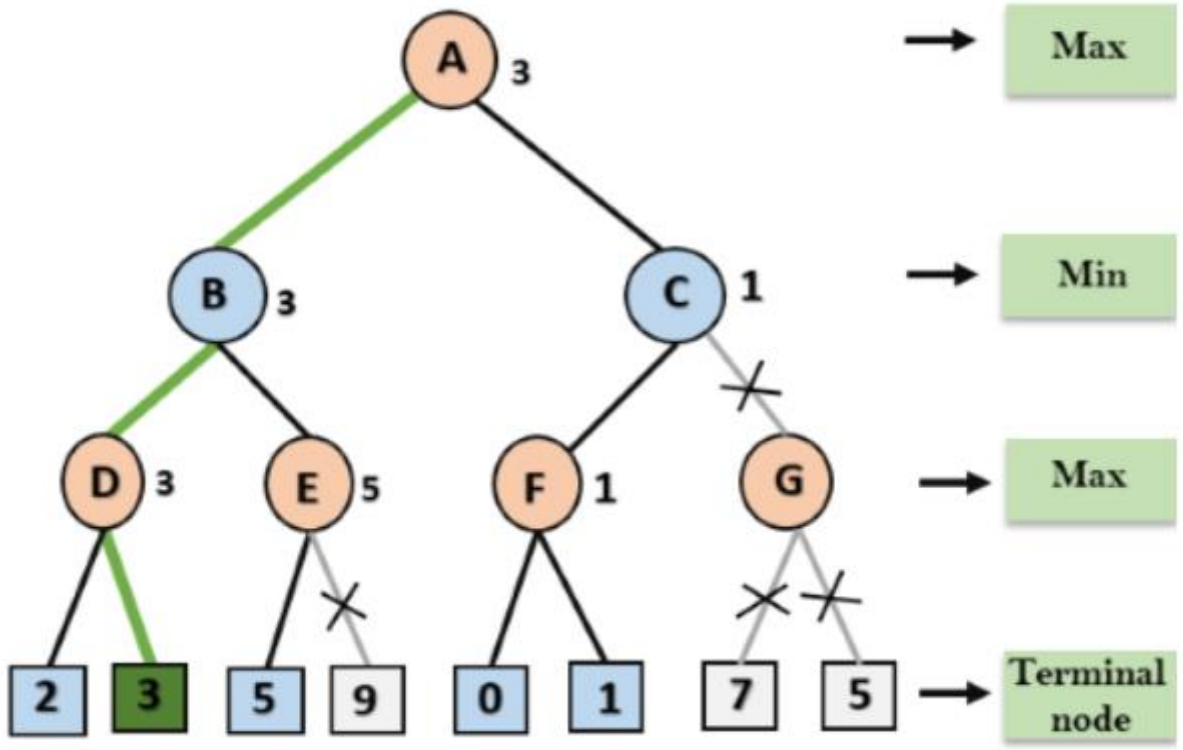
- **Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
- At node C, α=3 and β= +∞, and the same values will be passed on to node F.
- **Step 6:** At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F



- **Step 7:** Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



- **Step 8:** C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



WAITING FOR QUIESCENCE

"Waiting for quiescence" is a concept used in game playing and search algorithms within the field of artificial intelligence, especially when dealing with board games like chess or similar games with a high branching factor and complex decision trees. Quiescence refers to a state in which the position of the game is relatively stable and has limited tactical or forcing moves.

In the context of game playing algorithms, waiting for quiescence refers to temporarily suspending the search at certain nodes in the game tree until the position becomes quieter or more stable. The goal is to avoid the horizon effect, where the search algorithm might cut off its evaluation prematurely, leading to suboptimal decisions. The horizon effect occurs when the search reaches a certain depth, and the evaluation is based on positions where tactics (captures, threats, etc.) are still in flux.

HOW IT WORKS

Search Process: When traversing the game tree using a search algorithm (e.g., minimax, alpha-beta pruning), the search process explores different possible moves and positions.

Quiescence Nodes: At certain nodes in the search tree, the algorithm might pause the search and evaluate the position more deeply if it is not yet quiet.

Forcing Moves: During the quiescence search, the algorithm examines only "forcing" moves, which are moves that have a significant impact on the game state, such as captures or checks.

Evaluation: The algorithm evaluates the position's static evaluation (heuristic evaluation) at these quiescence nodes. This evaluation helps assess the position's overall strength and potential outcomes.

Quiescence Threshold: The search continues beyond quiescence nodes only if a certain threshold of stability is reached. If the position remains unstable or has a high tactical activity, the search might wait for quiescence before proceeding.

WAITING FOR QUIESCENCE

The main motivation behind waiting for quiescence is to ensure that the evaluation of a position is more accurate and takes into account important tactical considerations. By analyzing only forcing moves and avoiding positions where tactics are still in flux, the algorithm can make more informed decisions and avoid making premature cutoffs that could lead to missing critical moves or tactics.

Waiting for quiescence can be particularly important in positions where tactical interactions are complex and extend beyond the search horizon. It helps address the horizon effect and ensures that the search algorithm's decisions are based on a deeper and more stable evaluation of the game state.

WHAT IS A CONSTRAINT SATISFACTION PROBLEM

A **constraint satisfaction problem** (**CSP**) requires a value, selected from a given finite domain, to be assigned to each variable in the **problem**, so that all **constraints** relating the variables are **satisfied**. Many combinatorial **problems** in operational research, such as scheduling and timetabling, can be formulated as CSPs.

C S P

• Standard search problem: *state is a “black box”, *any old data structure that supports goal test, eval, successor, *Goal test and Successor can be any function over states, *Part of representation of atomic state

• Constraint satisfaction problems (CSPs):

–A special subset of search problems–state is defined by variables Xi with values from domain Di((sometimes Ddepends on i)

–goal test is a set of constraints specifying allowable combinations of values for subsets of variables

• Simple example of a formal representation language

• Allows useful general-purpose algorithms with more power than standard search algorithms

• AI as search →AI as Represenation→ AI as MLMindset

–Constraint Satisfaction Problem

–Preposition Logic

–Bayesian network

EXAMPLE : CLASS

Representation of atomic state

CSP

–Understand the problem

–Now we need to understand the structure of the state i.e class

–factored representation of each state

• set of variable and each variable has a value

C S P

- The goal in a CSP is to assign values to all variables from their respective domains such that all constraints are satisfied.

CSP is one of the standard search problem where instead of saying state is black box, we say state is defined by variables and values.

- CSP:
  - state is defined by variables X_i with values from domain D_i
  - goal test is a set of constraints specifying allowable combinations of values for subsets of variables

Allows useful general-purpose algorithms with more power than standard search algorithms

EXAMPLES OF CSPS

- **Sudoku**: Filling a 9×9 grid with digits so that each row, column, and 3×3 subgrid contains all digits from 1 to 9 without repetition.
- **Map Coloring**: Coloring a map with a limited number of colors so that no adjacent regions share the same color.
- **N-Queens**: Placing N queens on an N×N chessboard so that no two queens threaten each other.

BACKTRACKING SEARCH IN CSP

Backtracking is a depth-first search algorithm that incrementally builds candidates for the solutions, abandoning a candidate (backtracks) as soon as it determines that the candidate cannot possibly be completed to a valid solution.

Steps in Backtracking

- **Initialization**: Start with an empty assignment.
- **Selection**: Choose an unassigned variable.
- **Assignment**: Assign a value to the chosen variable.
- **Consistency Check**: Check if the current assignment is consistent with the constraints.
- **Recursion**: If the assignment is consistent, recursively try to assign values to the remaining variables.
- **Backtrack**: If the assignment is not consistent, or if further assignments do not lead to a solution, undo the last assignment (backtrack) and try the next possible value.

CRYPTARITHMATIC PROBLEM

- Cryptarithmatic problem is a type of CSP where each alphabet and symbol is associated with unique digit.
- Rules

- Each alphabet has unique digit
- Digits are from 0 to 9
- Only one carry should be found
- Can be solved from both the sides

EXAMPLE: CRYPTARITHMETIC

$$\begin{array}{r} \quad S\ E\ N\ D \\ +\ M\ O\ R\ E \\ \hline M\ O\ N\ E\ Y \end{array}$$

Constraints

1. Every letter must have a digit.
2. Each letter must have different digit

$Variables,\ X = \{S, E, N, D, M, O, R, Y\}$

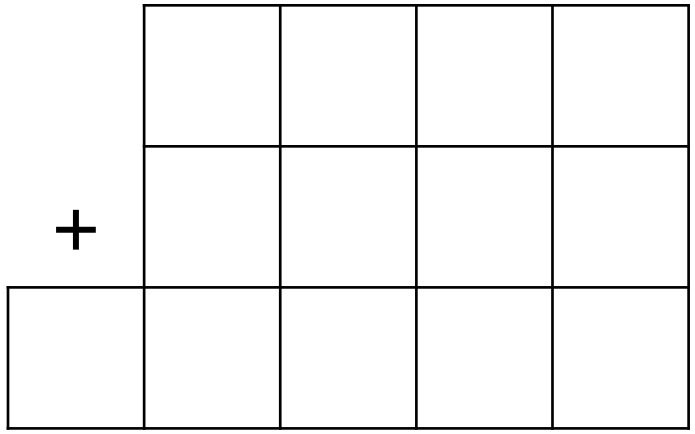
$Domains, D_{(except\ S\ \&\ M)} = \{0,1,2,3,4,5,6,7,8,9\}$

$Domains, D_{(S\ \&\ M)} = \{1,2,3,4,5,6,7,8,9\}$

$Constraints: Alldif(S, E, N, D, M, O, R, Y)$

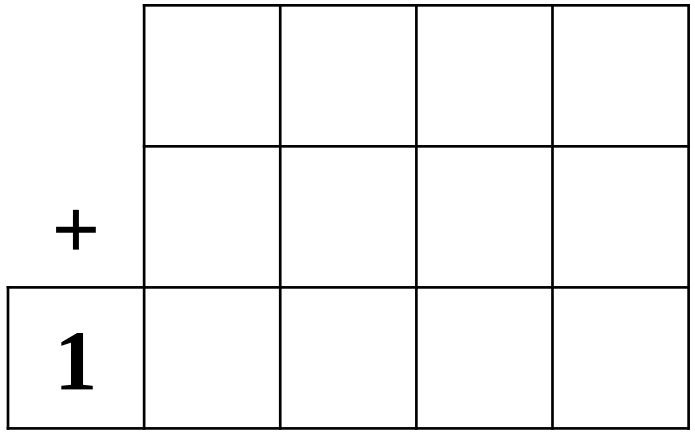
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

EXAMPLE: CRYPTARITHMETIC



Character	Code
S	
E	
N	
D	
M	
O	
R	
Y	

	S	**E**	**N**	**D**
+	**M**	**O**	**R**	**E**
M	**O**	**N**	**E**	**Y**



Character	Code
S	
E	
N	
D	
M	
O	
R	
Y	

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

	9			
+	**1**			
1	**0**			

Character	Code
S	9
E	
N	
D	
M	1
O	
R	
Y	

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

E + 0 = N

	9	?		
+	1	0		
1	0	N		

Character	Code
S	9
E	
N	
D	
M	1
O	0
R	
Y	

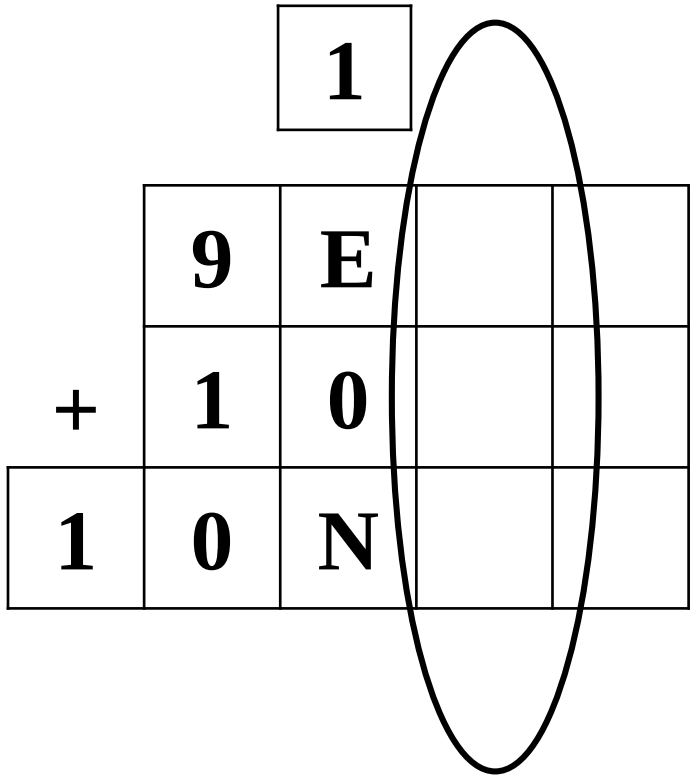
	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

	1	↰	CARRY FROM HERE	
	9	E		
+	1	0		
1	0	N		

Expression: E + 1 = N (N & E differ by 1)

Character	Code
S	9
E	
N	
D	
M	1
O	0
R	
Y	

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

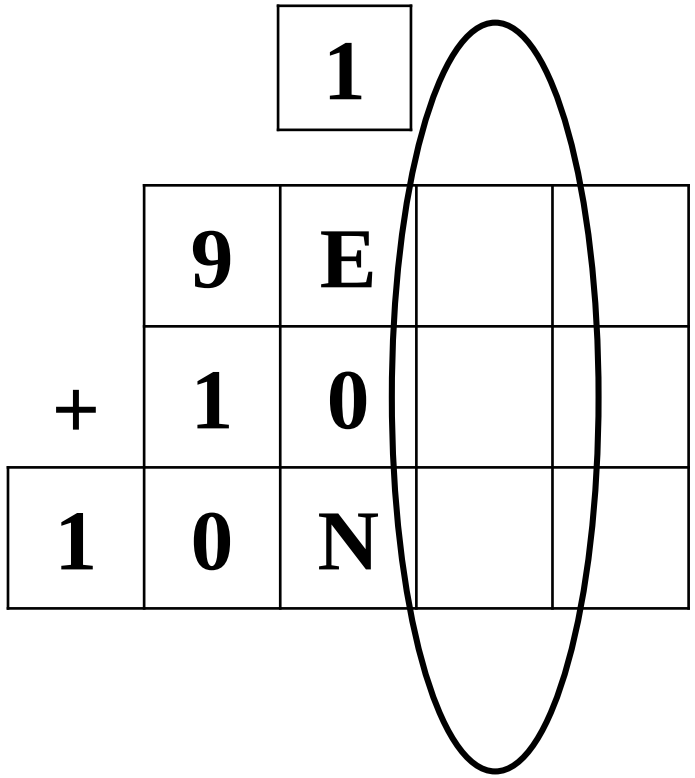


Character	Code
S	9
E	
N	
D	
M	1
O	0
R	
Y	

Expression:

1. E + 1 = N [N & E differ by 1]
2. N + R (+1) = E + 10 [(+1) will be considered only if needed]

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y



Substituting the values:

E + 1 + R (+1) = E + 10

Hence, R (+1) = 9

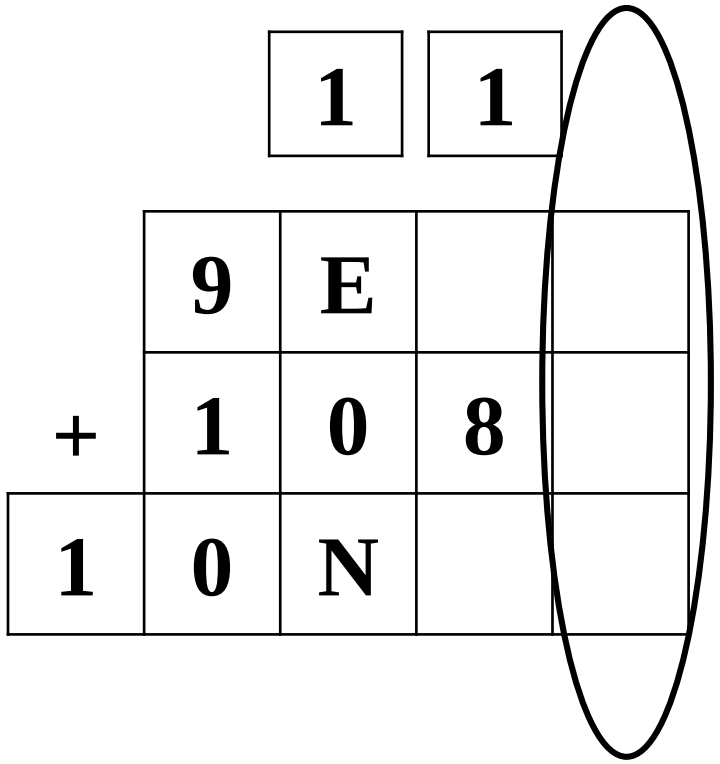
If we do not consider carry then R must be 9 but which is not possible because 9 has already taken, so R might be 8.

Character	Code
S	9
E	
N	
D	
M	1
O	0
R	8
Y	

Expression:

1. E + 1 = N [N & E differ by 1]
2. N + R (+1) = E + 10 [(+1) will be considered only if needed]

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y



Character	Code
S	9
E	
N	
D	
M	1
O	0
R	8
Y	

Now D+E= Y, has to be such that generates carry, D+E should be sum up to more than 11 because Y can not be 0 or 1 as they have already been taken, so to get that, the possibilities are 7+5 or 7+6 and so on.

So, if we take D = 7, E = 5, Hence Y = 2

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

		1	1	
	9	5		7
+	1	0	8	5
1	0	N		2

Character	Code
S	9
E	5
N	
D	7
M	1
O	0
R	8
Y	2

Expression:
1. E + 1 = N

	S	**E**	**N**	**D**
+	**M**	**O**	**R**	**E**
M	**O**	**N**	**E**	**Y**

		1	**1**	
	9	**5**	**6**	**7**
+	**1**	**0**	**8**	**5**
1	**0**	**6**	**4**	**2**

Hence N = 6

Character	Code
S	9
E	5
N	6
D	7
M	1
O	0
R	8
Y	2

```
  9 5 6 7
+ 1 0 8 5
---------
1 0 6 4 2
```

```
    S E N D
+   M O R E
-----------
M O N E Y
```

Character	Code
S	9
E	5
N	6
D	7
M	1
O	0
R	8
Y	2

Thank You