AI-Module-3-Important-Topics

For more notes visit

https://rtpnotes.vercel.app

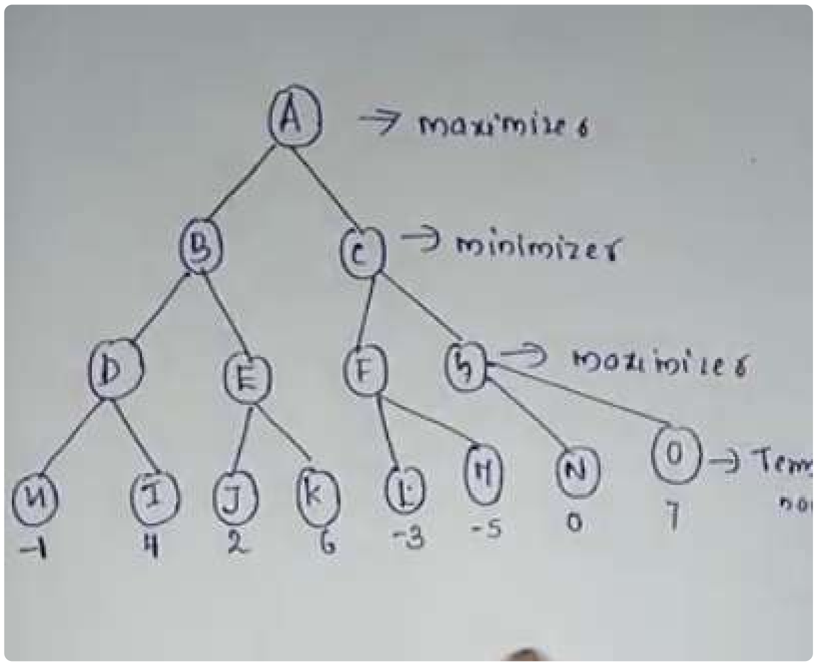
Playlist to refer

https://youtube.com/playlist?list=PLnzz0gSUYIN3mo-LB-l2Vfadz6HiPi8Po&feature=shared

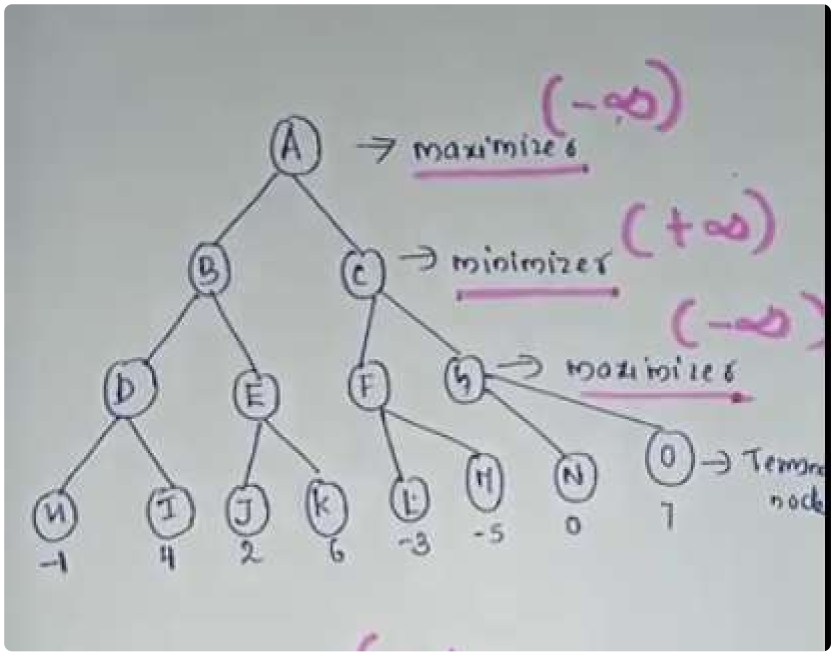
1. MiniMax Algorithm

- **Mini-Max Algorithm** is a backtracking algorithm which is used in decision making and to get game strategy.
- It provides optimal move to all the players
- The only condition is that success is only acheived by a single player
- Minimax algorithm is mostly used for game playing in AI, **such as chess and various 2 player games.**
- In this algorihtm two players play the game, one is called **MAX and other is called MIN**
- The player who got minimum value and player who got maximum value will get a optimised value
- This algorithm perform **depth-first search algorithm**

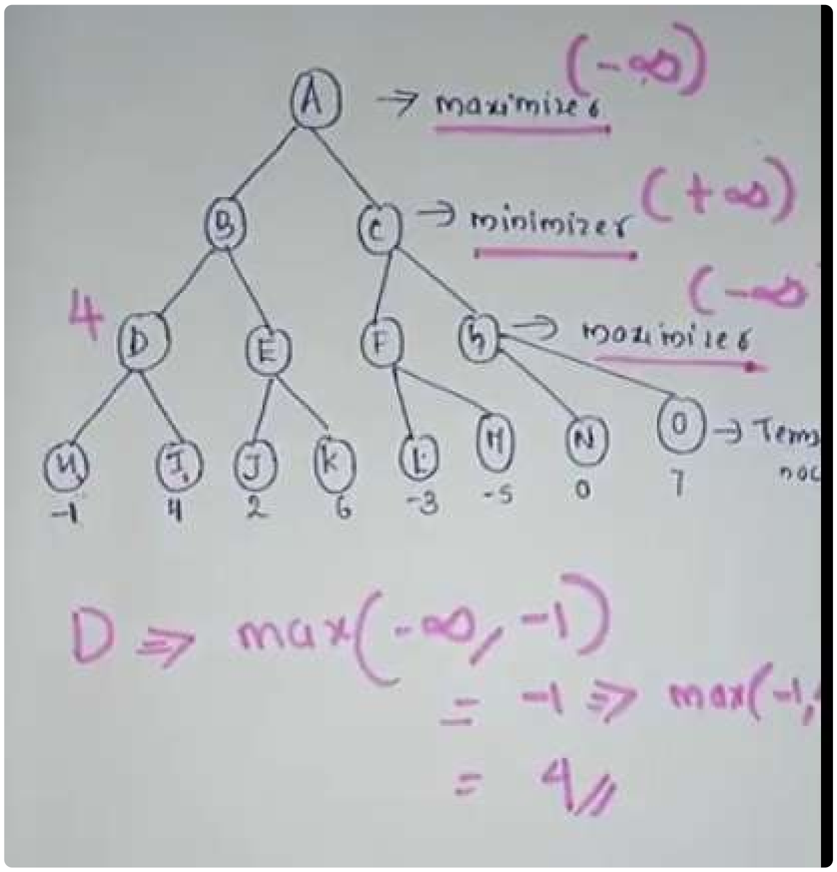
Implementation (Game Tree)



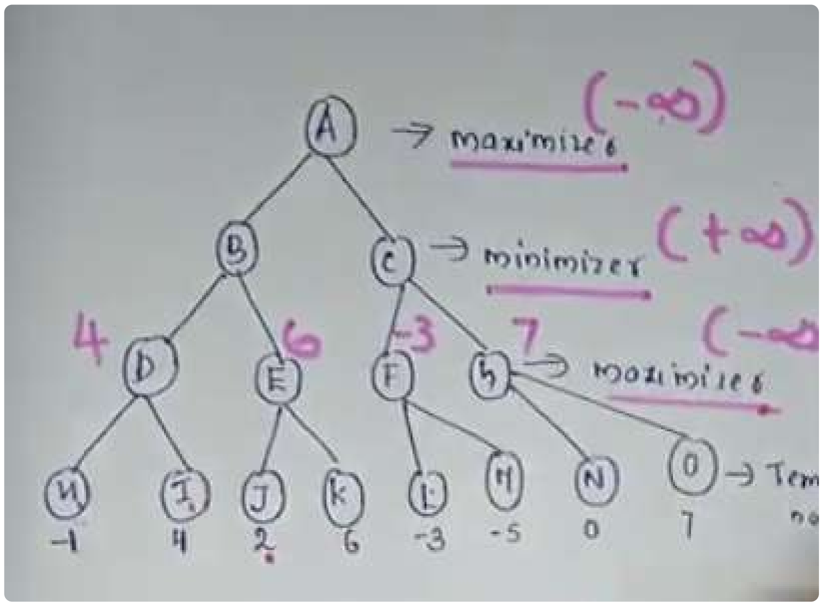
- The above is a game tree.
    - The first level node we can name as maximizer
    - 2nd level can be named minimizer
    - 3rd level is named maximizer
    - The last level are terminal nodes
- We can see that all terminal nodes have values
    - But the other nodes at level 1,2 and 3 doesnt have them
    - We need to assign the values to them



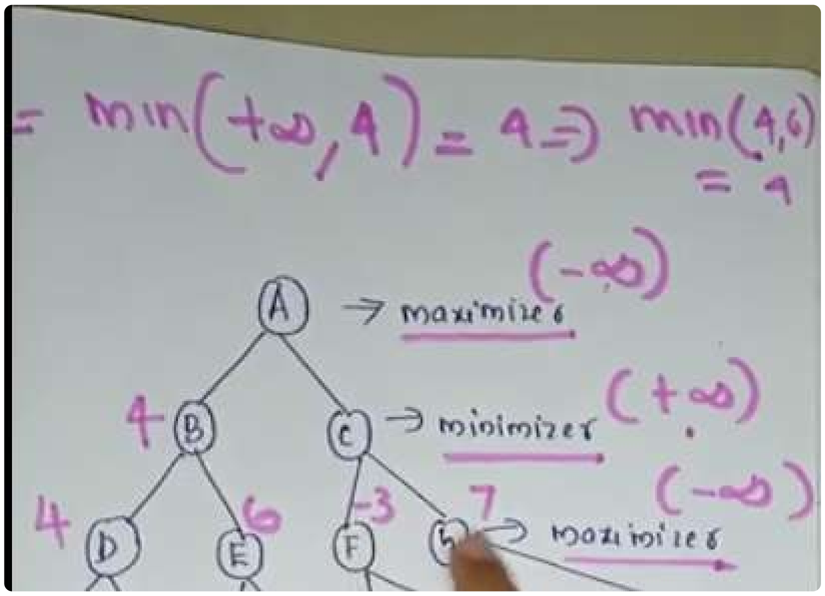
- We will be denoting - Infinity for maximizer
- And + infinity for minimizer



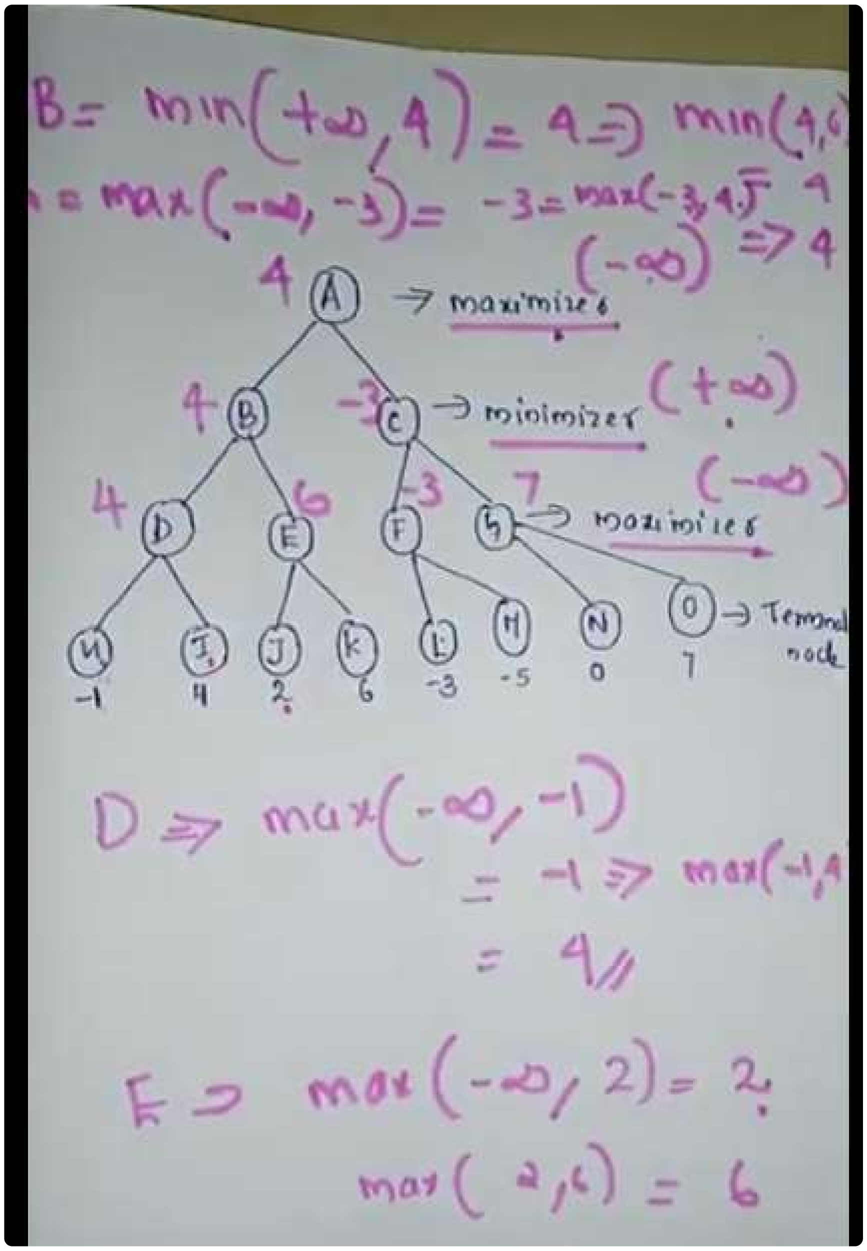
- Lets find the value of D first
  - We can see that D is in maximizer, which is - infinity
- We will first take the child of D with the least value and compare it with - infinity
  - -1 is the child of D with least value
    - Doing max comparison
    - D => max(-infinity,-1)
    - The larger number is -1
  - Now we need to take the other child of D which is larger
    - Doing max comparison
    - D => max(-1,4)
    - **The value is 4**



- Similarly, we can find the values of all the nodes in the third level



- Now lets find the values in the 2nd level, lets start with B
  - We can see that its a minimzer, with +infinity
- We have to take the smaller child of B and compare it with +infinite
  - The minimum of +infinite and 4 is 4.
  - Now compare this 4, with the other child of B
  - min(4,6) is 4



- Similarly, we will get remaining values on the 2nd level
- For the first level, its the same as the 3rd level

Properties of Mini-Max Algorithm

- Completeness
  - Mini Max algorithm is complete
  - It will definitely find a solution in the finite search tree
- Optimal
  - Mini Max algorithm is optimal if both opponents are playing optimally
- Time complexity
  - $O(b^m)$

    - $b = branchingfactor$
    - $m = depth$
- Space complexity
    - $O(bm)$

Limitations

- It is very slow during the complex games such as chess because it generates very large game tree

Algorithm of Minmax algorithm

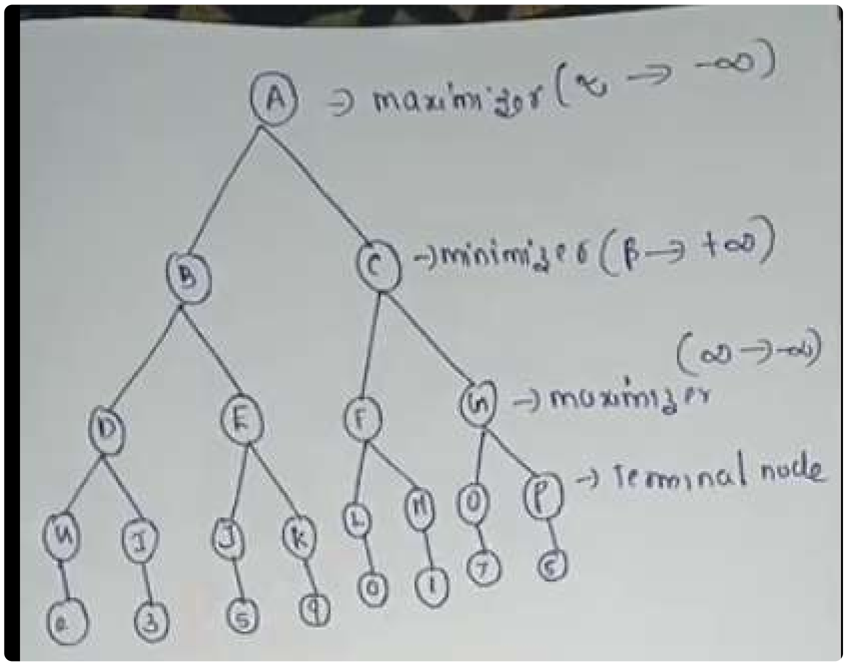
```
function MINIMAX_DECISION(state) returns an action
return arg max (a belongs to actions(s) MIN_VALUE (RESULT (state,action))

function MAX_VALUE(state) returns a utility value
        if TERMINAL_TEST(state) then return UTILITY(state)
                mu <- -infinity
                for each a in actions(state) do
                        mu <- MAX(mu, MIN_VALUE(RESULT(s,a)))
                        return mu

function MIN_VALUE(state) returns a utility value
        if TERMINAL_TEST(state) then return UTILITY(state)
                mu <- +infinity
                for each a in actions(state) do
                        mu <- MIN(mu, MAX_VALUE(RESULT(s,a)))
                        return mu
```

2. Alpha Beta Pruning Problem

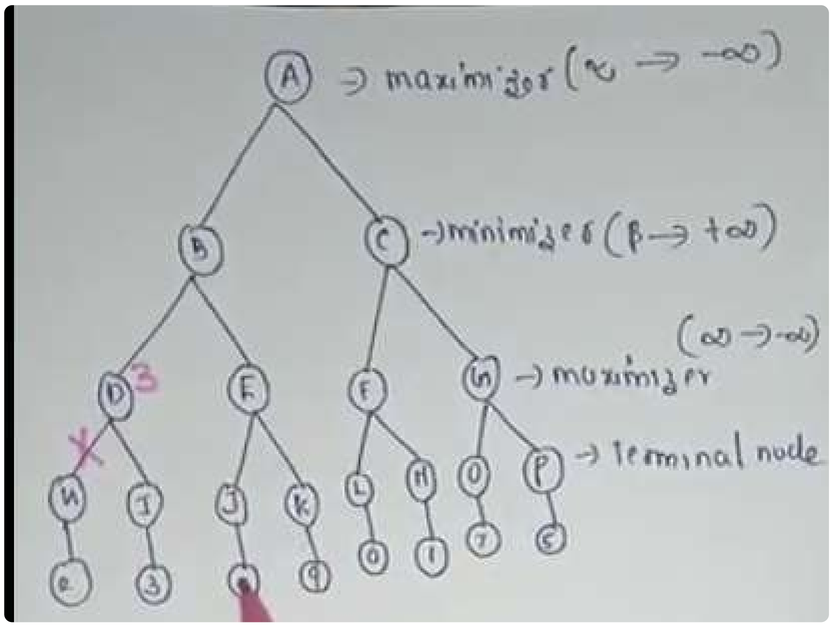
- Its a **modified version of mini-max algorithm**
- In this Alpha-Beta pruning algorithm, two threshold values are considered that is **alpha and beta**
- Backtracking is also included

- This algorithm does not consider all the nodes it just considered the nodes which satisifes certain conditions and the rest of the nodes are eliminated
- Alpha
    - The maximiser value (-infinity)
- Beta
    - The minimizer value (+infinity)
- Condition
    - Alpha >=Beta

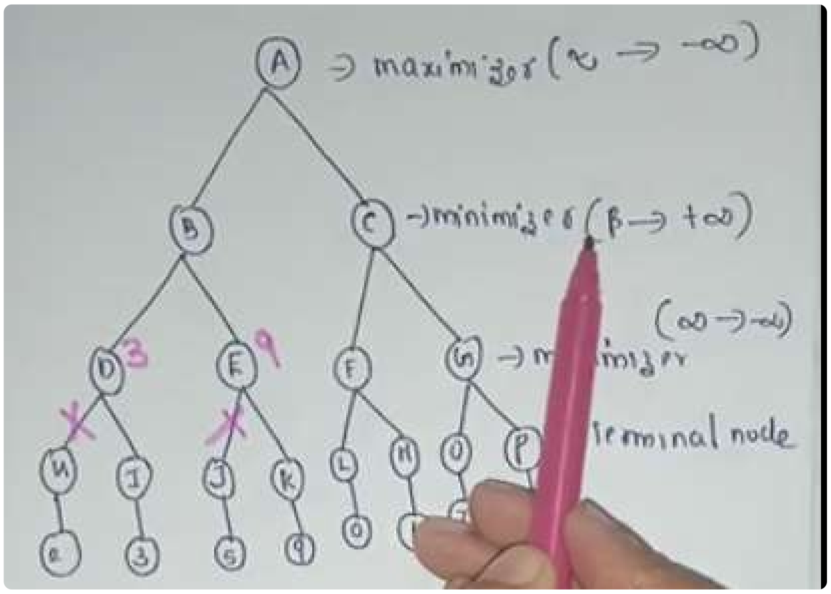
Example of Alpha-Beta Pruning



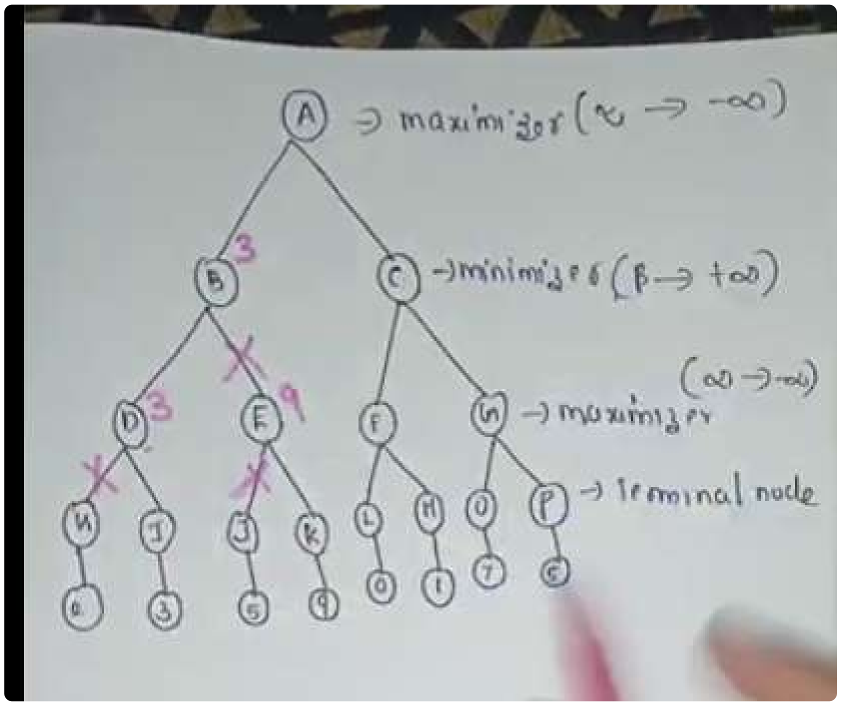
- Just like min-max algorithm, we need to assign the maximizer and minimizer, which are alpha and beta
- We will be moving from left to right



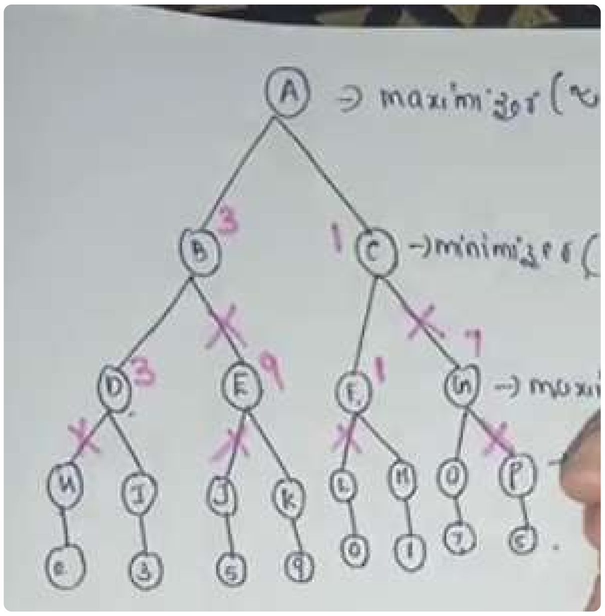
- Starting from the left, taking H and I
  - The values are 2 and 3, maximum is 3, taking that
  - Pruning H from the tree



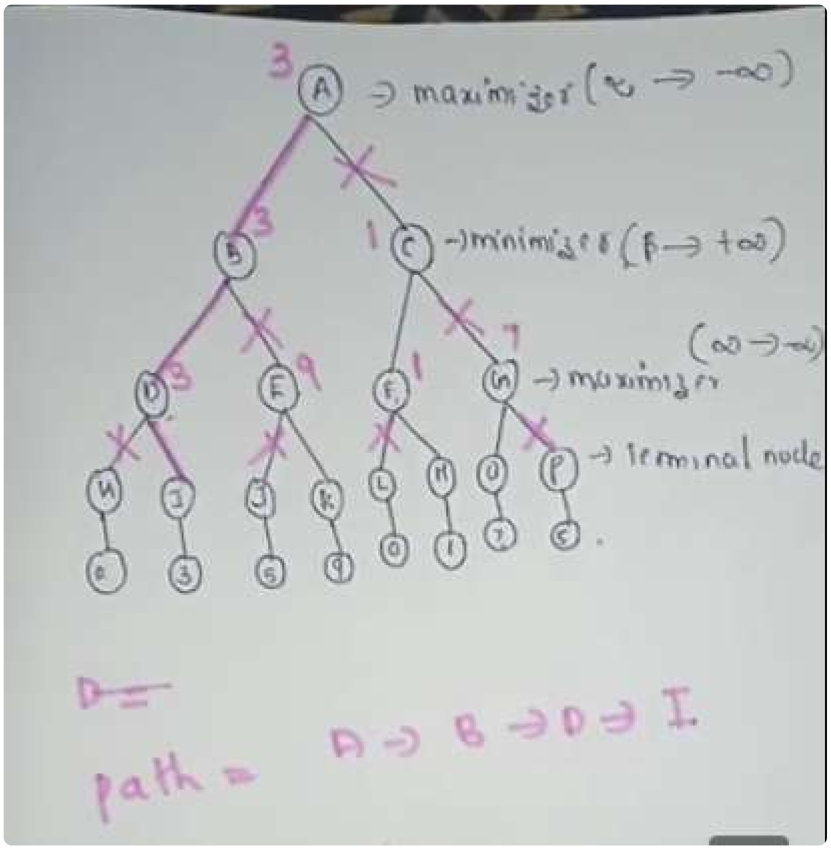
- Taking J and K, maximum value is 9
- Pruning J

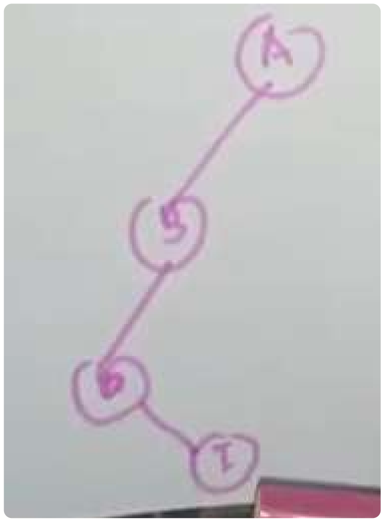


- Taking D and E
- Minimum value is 3
- Pruning E



- Similarly, pruning, we get the above values



- Similarly, Comparing 3 and 1, max value is 3, assigning 3 to A
- When we go through the unpruned areas in the tree we get the following path
- Path = A -> B -> D -> I
- Giving us the following tree
  - 

3. Define CSP with an example

Constraint Satisfaction Problems (CSP) are a class of problems in which an **agent** must find a solution that satisfies a set of constraints. These problems involve **variables**, **domains**, and **constraints**, and the agent must assign values to the variables such that all the constraints are satisfied.

Components of CSP

1. **Variables (X)**
   A set of variables $V = \{V_1, V_2, V_3, \ldots, V_n\}$.
   Example: In Sudoku, each empty cell is a variable.
2. **Domains (D)**
   Each variable has a domain, which is a set of possible values it can take.
   Example: In Sudoku, the domain of each cell is $\{1, 2, 3, \ldots, 9\}$.
3. **Constraints (C)**
   Constraints are rules that restrict the values the variables can take.
   Example: In Sudoku, all numbers in a row, column, or block must be unique.

Example of CSP: Sudoku

Problem: Solve a Sudoku puzzle.

1. **Variables (X):** Each empty cell in the Sudoku grid.
2. **Domains (D):** Possible values $\{1, 2, 3, \ldots, 9\}$.
3. **Constraints (C):**
   - Each row, column, and 3×3 block must have unique numbers from 1 to 9.
   - A cell can only take values from 1 to 9.

Solution: Assign numbers to each empty cell such that all constraints are satisfied.

Example CSP with Details

1. **Variables (X):**
   x_1, x_2, x_3.
2. **Domains (D):**
   - $D(x_1) = \{1, 2\}$
   - $D(x_2) = \{0, 1, 2, 3\}$
   - $D(x_3) = \{2, 3\}$
3. **Constraints (C):**

- $x_1 > x_2$
- $x_1 + x_2 = x_3$
- x_1, x_2, x_3 should not be equal.

Solution:

- Assign values to x_1, x_2, x_3 such that all constraints are satisfied.
    - $x_1 = 2, x_2 = 1, x_3 = 3$.

Types of Constraints in CSP

1. **Unary Constraints:** Apply to a single variable (e.g., $x_1 \leq 3$).
2. **Binary Constraints:** Apply to pairs of variables (e.g., $x_1 + x_2 < 6$).
3. **Ternary Constraints:** Apply to three variables (e.g., $x_1 + x_2 = x_3$).
4. **Linear Constraints:** Variables appear in linear form (e.g., $2x_1 + x_2 = 10$).
5. **Non-linear Constraints:** Variables appear in non-linear form (e.g., $x_1^2 + x_2^2 = 9$).
6. **Preference Constraints:** Prioritize certain assignments (e.g., $x_1 = 5$ preferred).

Solving CSP

1. **State Space:**
   Assign values to variables. The state must satisfy constraints.
    - **Legal Assignment:** All constraints are satisfied.
    - **Complete Assignment:** All variables have values assigned.
    - **Partial Assignment:** Only some variables are assigned.
2. **Constraint Propagation:**
   Use constraints to reduce the domain of variables and simplify the problem.
   Example:
    - Domain of $x = \{1, 2, 4\}$, $y = \{1, 2, 5\}$.
    - Constraint: $x = y$.
    - After filtering:
        - $x = \{1, 2\}$, $y = \{1, 2\}$.

Applications of CSP

1. **Sudoku**: Assign numbers to cells satisfying Sudoku rules.

2. **N-Queens Problem**: Place N queens on an $N \times N$ chessboard such that no two queens attack each other.
3. **Map Coloring**: Assign colors to regions on a map such that no two adjacent regions have the same color.
4. **Cryptarithmetic**: Solve puzzles by assigning numbers to letters under certain rules.

4. CSP - Backtracking search

Backtracking is a method for solving **Constraint Satisfaction Problems (CSP)** by trying to assign values to variables one at a time while ensuring all constraints are satisfied.

Key Features of Backtracking

1. **Partial Assignments:**
   Backtracking works by assigning values to some variables and leaving others unassigned.
2. **Depth-First Search (DFS):**
   It explores one variable at a time, creating a search tree. If a conflict arises (constraints are violated), it "backtracks" to try a different value.

How It Works

Branching Factor:
At each level of the tree, the number of choices depends on the number of variables left to assign and the size of the domain:

- At the **top level** (initial step):
  Each of the n variables can take one of d values, so there are $n \cdot d$ branches.
- At the **next level**:
  With one variable assigned, there are $(n-1) \cdot d$ branches.
- This process continues until all variables are assigned.
- The **total number of leaves** in the tree is approximately $n! \cdot d^n$ (factorial of n times d^n).

Sudoku Example:

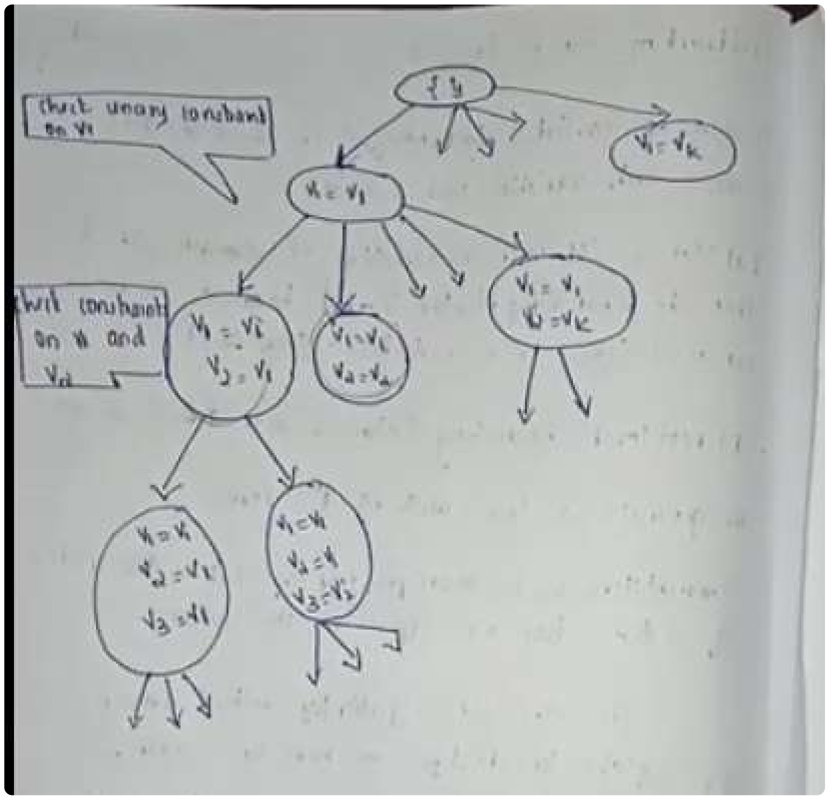
1. Start by assigning a number to the first empty cell.

2. If a constraint (e.g., no duplicate numbers in a row) is violated, backtrack and try the next number.
3. Continue assigning numbers to the next empty cell until the grid is complete or backtracking exhausts all options.

To Solve CSP Efficiently:

We need to address three key questions:

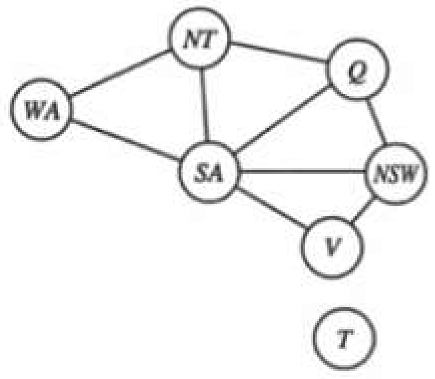
1. **Which Variable to Assign Next?**
   - Decide the order in which variables are assigned. For example, assign the most constrained variable first (variable with the fewest options).
2. **In What Order to Try Values?**
   - Choose the order to try the domain values. For example, pick the value least likely to cause conflicts.
3. **How to Handle Constraints (Interference)?**
   - Check constraints after each assignment. For example, remove invalid values from other variables' domains to simplify the problem (constraint propagation).



5. CSP-Minimum Remaining Values(MRV) Heuristic, Degree Heuristic, Least Constraining Value (LCV)

MRV (Minimum Remaining Values) Heuristic

MRV helps in solving problems more efficiently by choosing the variable that is the hardest to assign first. Here's how it works:

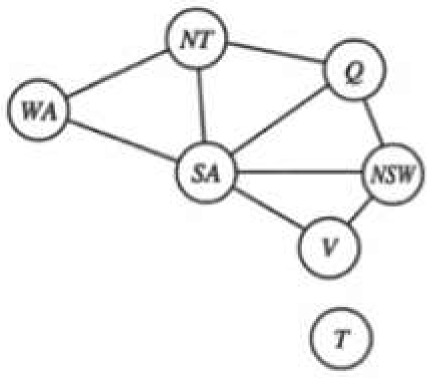
1. **What it does**: It picks the variable with the fewest legal values left (most constrained).
2. **Why it’s useful**: By choosing the variable most likely to fail, it detects issues earlier, reducing wasted effort on impossible solutions.
3. **Fail-first approach**: If a variable has no valid options, MRV selects it immediately, causing failure to be detected right away.
4. **Example**:
    1. 

    2. If WA = red and NT = green, SA has only one possible value left (blue).
    3. MRV suggests assigning SA = blue next instead of Q since it's the most constrained variable.
        1. Here our constraint is, the adjacent nodes should not be of the same color
        2. And our available colors are Red green and blue
        3. Neighbours of SA has already Red and Green assigned to them, So the only option for SA is setting it as Blue.

Degree Heuristic

The **Degree Heuristic** helps pick the best variable to work on by looking at how connected it is to others

1. **When MRV doesn’t help**: If all variables have the same number of options (like when starting a problem), MRV isn’t useful.

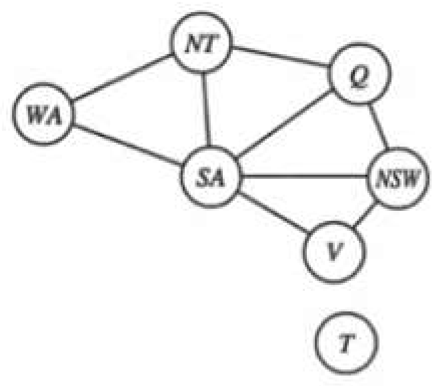
2. **What Degree Heuristic does**: It picks the variable connected to the most other unassigned variables. This reduces future complexity by addressing the most "influential" variable first.
3. **Why it's useful**: Focusing on a highly connected variable reduces the chances of conflicts later, minimizing backtracking.
4. **Example**:
    1. 

    2. In Australia's map-coloring problem, SA is the most connected region (linked to 5 others). Other regions are linked to only 2 or 3 regions, and T has no connections.
    3. Starting with SA and choosing consistent colors at each step solves the problem smoothly without needing to backtrack.

Least Constraining Value

The **Least Constraining Value** heuristic helps decide which value to assign to a variable by considering how it affects other variables. Here's how it works:

1. **What it does**: After selecting a variable, it picks the value that restricts the fewest options for the neighboring variables.
2. **Why it's useful**: It leaves more options open for future choices, increasing the chance of finding a solution without backtracking.
3. **Example**:



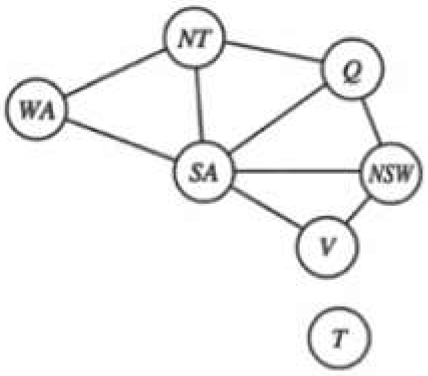
1.
2. If WA = red and NT = green, and now we need to assign a color to Q:
3. **Blue** would be a bad choice for Q because it would leave SA with no valid color left.
    1. Because The neighbours of SA are Red, green and blue.
    2. The only choices given were Red green and blue, so since there is no other color, then SA will have no choice
4. **Red** would be better because it doesn't block SA's options as much.

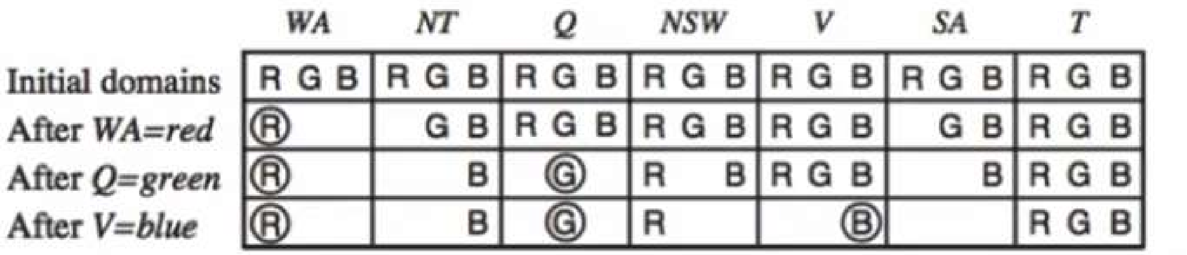
4. **Goal**: The idea is to avoid choices that limit future options too much, keeping the problem flexible as you go.
5. **When it doesn't matter**: If you're finding *all* possible solutions, or if no solution exists, the order doesn't matter because you'll have to check every possibility anyway.

6. CSP Forward Checking

Forward Checking is a simple method used to reduce the search space when solving constraint satisfaction problems (CSPs). Here's how it works:

1. **What it does**:
    - When you assign a value to a variable (e.g., X = red), forward checking looks at all the neighboring variables (like Y) connected to X by a constraint. It removes any values from Y's options that conflict with X's value. This helps narrow down the choices for Y right away.
2. **When not needed**: If you've already done arc consistency (a similar process) as a preprocessing step, there's no need to do forward checking again.
3. **Example**:

1. 

2. 

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

3. After assigning **WA = red** and **Q = green**, the possible choices for **NT** and **SA** are reduced to only one value each, eliminating the need for further checks on these variables.
4. However, when **V = blue** is assigned, it leaves **SA** with no valid values left. Forward checking immediately detects this and triggers **backtracking** to try a different path.

4. **Advantages**:
   - Combining forward checking with the **MRV heuristic** (which chooses the most constrained variable) can make the search process more efficient for many problems.
5. **Disadvantages**:
   - Forward checking only ensures that the current variable is consistent with others, but it doesn't look ahead to check all variables. It only works locally, not globally.

7.Constraint Propogation for inference in CSP (Local consistencies)

What is Constraint Propagation?

- **Constraint Propagation** is a method used in Constraint Satisfaction Problems (CSPs) to make the problem easier to solve.

- It involves using the constraints between variables to **reduce the number of possible values** a variable can take.
- Once one variable's domain (set of possible values) is reduced, it can help reduce the domains of related variables as well.

How Does It Work?

1. **Constraints** (like $X = Y + 1$ or $X \neq Y$) help **narrow down** the possible values for variables.
2. When you reduce the options for one variable, it might also **affect other variables** that are connected by constraints. For example:
    - If X can only be 1, and Y must be greater than X, then Y can only be 2 or higher.

When is Constraint Propagation Used?

- **During search**: It can be used as you search for a solution, making the search faster by reducing the possibilities.
- **Preprocessing step**: Sometimes, constraint propagation is used before the search starts, helping to solve parts of the problem right away.

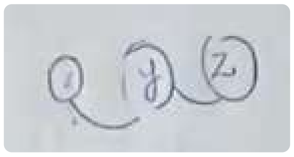
Can Constraint Propagation Solve the Whole Problem?

- Sometimes, if the constraints are strong enough, constraint propagation can solve the entire problem without the need for any further search. This means that **no backtracking or searching** is needed at all.

Local consistency

- If we treat each variables as a node in graph
    -

- If we treat each binary constraint as an arc
    - x = y^3
    - y^3 = z^4
    - Each of these rules we take it as an arc

- 

- Then the process of checking the consistency of each part rather than checking the whole unit together is called local consistency

Node consistency

- **Node consistency** checks if a variable's domain (the set of possible values) satisfies its **unary constraint** (a constraint that involves only that variable).
- If a value in the domain violates the unary constraint, it is removed.

Example:

Let's consider a map coloring problem where **South Indians dislike the color yellow**.

- The variable **South Indian** has a domain: {red, yellow, blue}.
- The constraint is that **yellow is not allowed** for South Indians.
  To make the **South Indian** variable **node consistent**, we remove **yellow** from the domain because it violates the constraint.
- **Updated domain**: {red, blue}

Arc Consistency

Arc consistency ensures that every value in a variable's domain satisfies the constraints when paired with values from another variable's domain.

- A variable X_i is **arc consistent** with another variable X_j if:
  - For every value of X_i, there is at least one value in X_j's domain that satisfies the constraint between them.

Example:

Suppose we have the constraint $y = x^2$, and the domains are:

- X: {0, 1, 2, 3}
- Y: {0, 1, 4, 9}

Step 1: Make X Consistent with Y

- Check each value in X's domain to ensure it satisfies the constraint $y = x^2$:
  - $x = 0$: $y = 0$, which is in Y's domain.
  - $x = 1$: $y = 1$, which is in Y's domain.
  - $x = 2$: $y = 4$, which is in Y's domain.
  - $x = 3$: $y = 9$, which is **not** in Y's domain.
- Remove $x = 3$ because it doesn't have a valid y.
  - **Updated X's domain**: {0, 1, 2}

Step 2: Make Y Consistent with X

- Check each value in Y's domain to ensure it satisfies the constraint $y = x^2$:
  - $y = 0$: Valid for $x = 0$.
  - $y = 1$: Valid for $x = 1$.
  - $y = 4$: Valid for $x = 2$.
  - $y = 9$: No x satisfies $x^2 = 9$.
- Remove $y = 9$.
  - **Updated Y's domain**: {0, 1, 4}

Why Use Arc Consistency?

- It helps **reduce domains early**, making the problem easier to solve.
- It can detect when a problem is **unsolvable** sooner than forward checking.

For example, if after making variables arc consistent, one domain becomes empty, we know the problem has no solution.

Arc consistency simplifies the CSP before attempting to assign values.

Path Consistency

Path consistency goes beyond **arc consistency** by checking constraints involving **three variables at a time**. It ensures that if two variables are consistent, there is a valid value for a third variable to maintain consistency.

Why Do We Need Path Consistency?

- **Arc consistency** only considers constraints between **two variables at a time**, which can help reduce domains or detect if a problem cannot be solved.
- However, arc consistency alone might miss cases where the problem requires checking **three variables** to detect inconsistencies.

Example: Map Coloring Problem

Imagine you're trying to color a map of Australia using **only two colors** (Red and Blue), and you must ensure that neighboring regions have different colors.

- Regions like **Western Australia (WA)**, **Northern Territory (NT)**, and **South Australia (SA)** all touch each other.
- **Arc consistency** checks the constraints between pairs of regions but won't detect that you need a third color for all three regions to be valid.
- **Path consistency** solves this by considering triples of regions and identifying that two colors are insufficient.

How Does Path Consistency Work?

Path consistency checks **triples of variables** (X_i, X_j, X_m):

1. If X_i is assigned a value a and X_j is assigned a value b,
2. There must be a value for X_m that satisfies both:
   - The constraint between X_i and X_m
   - The constraint between X_m and X_j
   This ensures that the "path" between X_i and X_j via X_m is consistent.

Why is it Called Path Consistency?

You can think of it as checking a "path" of constraints between two variables (X_i and X_j) with a third variable (X_m) acting as a middle link.

K-Consistency

K-consistency ensures that constraints involving **k variables** are satisfied.

- **1-Consistency (Node Consistency)**: Each variable in isolation satisfies its own constraints. For example, a variable with a domain of {1, 2} is consistent if 1 and 2 are valid values.
- **2-Consistency (Arc Consistency)**: For every pair of variables, their values satisfy the constraints between them. This is what we achieve with the **AC-3 algorithm**.
- **3-Consistency (Path Consistency)**: For any three variables, if the first two are consistent, then there exists a value for the third variable that satisfies the constraints.

Why K-Consistency is Useful

- If a CSP is **k-consistent**, we can assign consistent values to **any set of k variables**.
- For example:
  - In a 2-consistent graph (arc consistent), we can choose values for any pair of variables (e.g., X_1, X_2).
  - In a 3-consistent graph (path consistent), we can also include a third variable (e.g., X_3).

Using K-Consistency to Solve Problems

- In general, **2-consistency (arc consistency)** is sufficient and commonly used, as 3-consistency requires more computation and memory.
- To solve a CSP:
  - We ensure 2-consistency first.
  - Then assign values to variables step by step, checking consistency at each step.

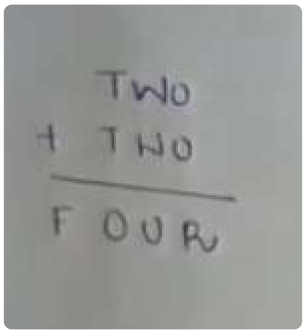
Complexity

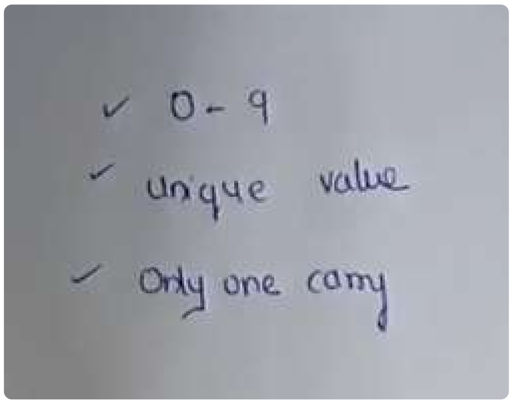
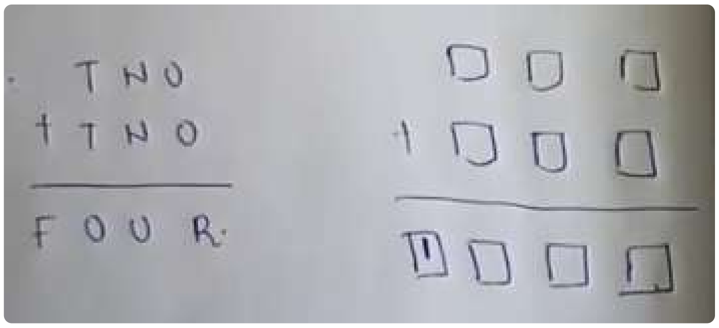
- Searching for consistent values involves checking the **d** possible values in the domain for each variable.
- Total time complexity is $O(n^2 \cdot d)$, where:
  - n is the number of variables.
  - d is the size of the domain for each variable.

Note: Higher consistency (e.g., 3-consistency) uses more memory and computation, so it's less commonly applied.

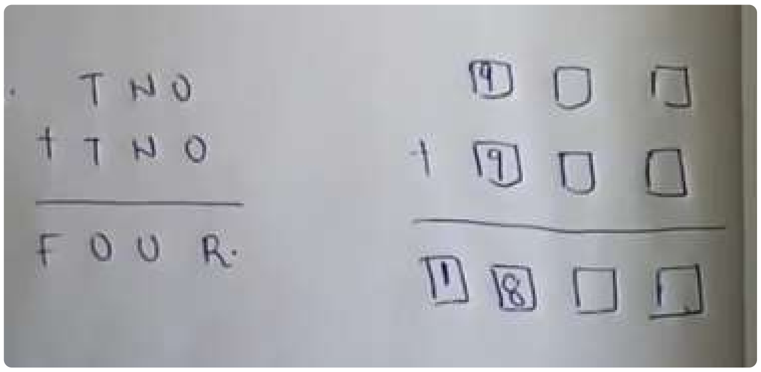
8. Crypt Arithmetic Problem

Example-1

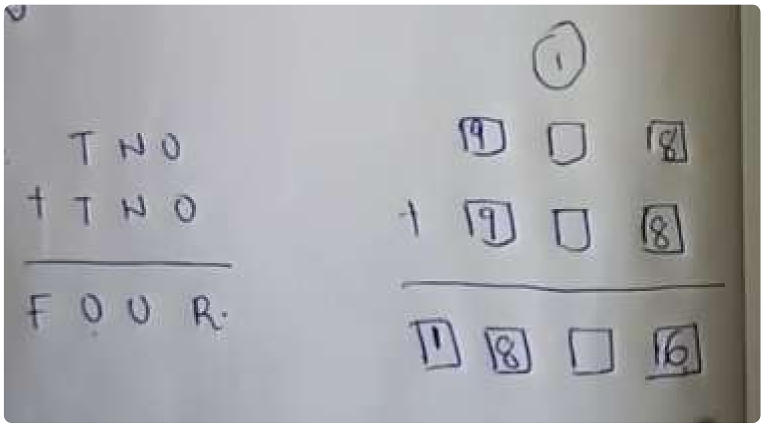
- Consider this example
    - 

    - We need to assign values to each of these letters
    - Value for T, Value for W, etc..
    - We are also given the following conditions for assigning values to these letters
        - 
          ✓ 0 - 9
          ✓ unique value
          ✓ Only one carry
- **Step 1**
    - 

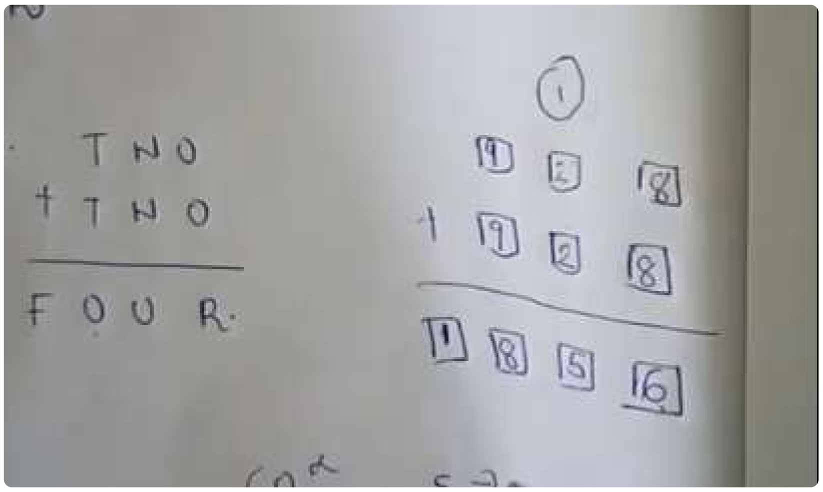
    - From above we can see that
        - O+O = R

        - W+W = U
        - T+T gives O with F as carry
    - From the conditions its said that there can be only one carry
    - Our carry here is F
    - The only value the carry can have is 1
        - Because, 9+9 = 18
        - 1+9 = 10
        - These two are the maximum and minimum values we will get those have carry, and the carry is only having 1
    - So we can assign 1 to F
- **Step 2**
    - 

    - Lets see T + T
    - T+T will give O with carry F
    - We know that F is 1
    - Which number combinations will give 1 as carry?
        - 5+5 = 10
        - 6+6 = 12
        - 7+7 = 14
        - 8+8 = 16
        - 9+9 = 18
        - Numbers from 5 to 9 gives 1 as carry.
        - You can choose any one of these.
        - Im choosing 9
    - So we got Values of W as 9
    - Value of O as 8
- **Step 3**

- 

- Since we got value of O
- O+O = R
- 8+8 = 16
- We can assign R as 6 and we need to put the carry 1.

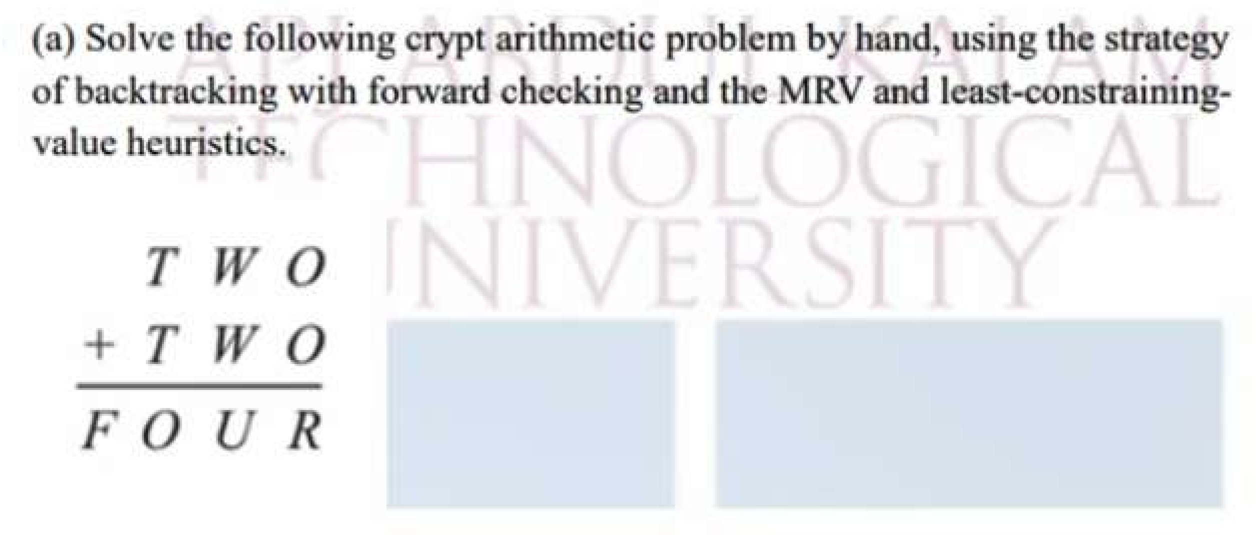
- **Step 4**
  - 

  - W+W = U
  - Since there is a carry, we need to include that as well.
  - W+W+1 = U
  - Our only condition here is that, W+W+1 should not have a carry
  - Possible combinations
    - 0 ❌
      - W = 0 (unique)
      - U = W + W + 1
      - U = 0+0+1 = 1, which is not unique
      - This number is not possible
    - 1 ❌
      - W = 1 (not unique)

        - Cant assign 1, its already given to F
    - 2 ✅
        - W = 2 (Unique)
        - U = 2+2+1 = 5 (unique)
        - 2 and 5 are unique values, so we can assign
- **Solution**
    - T = 9
    - O = 8
    - U = 5
    - W = 2
    - F = 1

Example-2

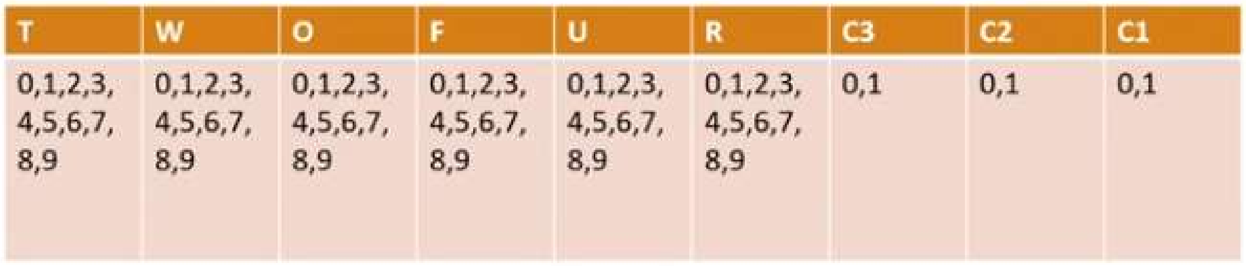
Solve the following crypt arithmetic problem by hand, using strategy of backtracking with **forward checking** and **MRV** and **least constraint value heuristic**



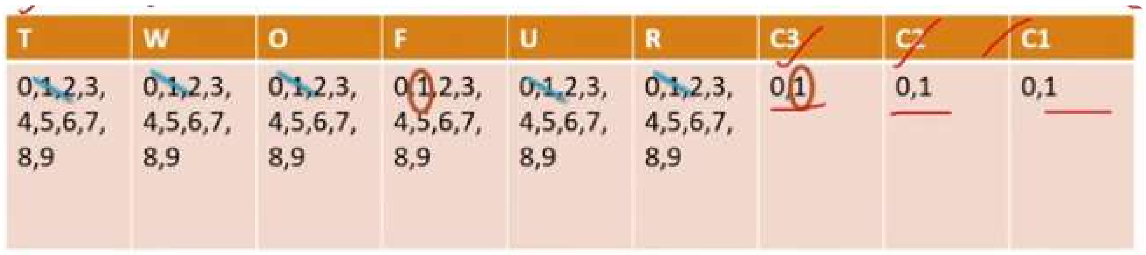
(a) Solve the following crypt arithmetic problem by hand, using the strategy of backtracking with forward checking and the MRV and least-constraining-value heuristics.

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

- Lets define it as a CSP
    - Variables = {T,W,O,F,U,R,C1,C2,C3}
    - Domain of {T,W,O,F,U,R} = {0,1,2,3,4,5,6,7,8,9}
    - Domain of {C1,C2,C3}= {0,1}

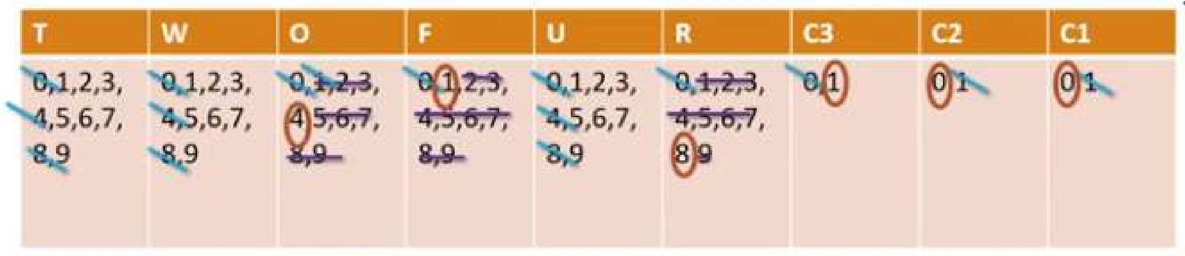
- 

T	W	O	F	U	R	C3	C2	C1
0,1,2,3, 4,5,6,7, 8,9	0,1,2,3, 4,5,6,7, 8,9	0,1,2,3, 4,5,6,7, 8,9	0,1,2,3, 4,5,6,7, 8,9	0,1,2,3, 4,5,6,7, 8,9	0,1,2,3, 4,5,6,7, 8,9	0,1	0,1	0,1

  - This is a table, which shows all the available values for each variable
- **Step 1:** Lets use Minimum Remaining Value Heuristic to choose the variable with fewest legal values
  - Here we take C3 whose domain={0,1}
  - If we set C3 as 0
    - By forward checking we can see that F will be 0, which is not possible.
    - So we choose C3 as 1
  - This Makes F = 1
  - Cutting 1 from others domain, since its used by F
    - 

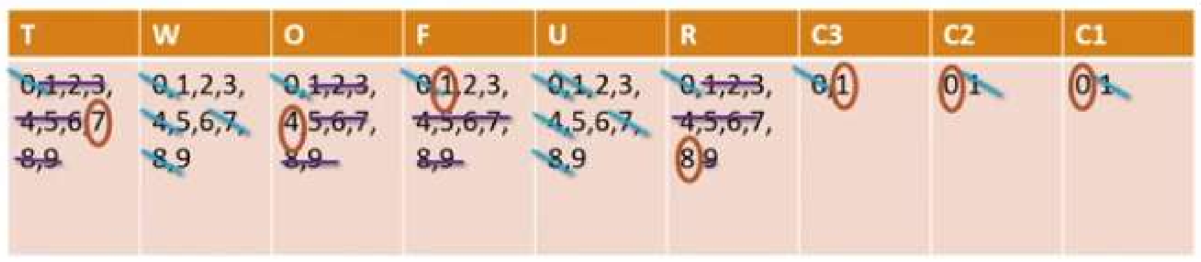
T	W	O	F	U	R	C3	C2	C1
0,~~1~~,2,3, 4,5,6,7, 8,9	0,~~1~~,2,3, 4,5,6,7, 8,9	0,~~1~~,2,3, 4,5,6,7, 8,9	0,1,2,3, 4,5,6,7, 8,9	0,~~1~~,2,3, 4,5,6,7, 8,9	0,~~1~~,2,3, 4,5,6,7, 8,9	0,1	0,1	0,1

- **Step 2**: Using MRV again
  - C2 has domain as {0,1}, so choosing that since it has only 2 legal values
  - Putting C2 as 0
  - Not affecting any other variable, so proceeding
- **Step 3** Using MRV
  - C1 has domain as {0,1}
  - Putting C1 as 0
- **Step 4**
  - Choosing O
  - O has few constraints
    - T+T = O with carry F
      - Since T+T = 2T = O
      - O will be divisible by 2, making it an even number
      - Removing odd numbers from the domain of O

- O+O = R
  - O+O = R should be less than 10
  - This is because if R becomes 10 or more, it will give Carry.
  - But we have set C1 = 0. Which means there shouldnt be a carry
- From these constraints, we can arbitrarily choose 4 as the value of O
- O = 4
- R = 8
- Removing 4 and 8 from others domains
- 

T	W	O	F	U	R	C3	C2	C1
~~0~~,1,2,3, ~~4~~,5,6,7, ~~8~~,9	~~0~~,1,2,3, ~~4~~,5,6,7, ~~8~~,9	~~0,1,2,3~~, 4,~~5,6,7~~, ~~8,9~~	~~0~~,1,~~2,3~~, ~~4,5,6,7~~, ~~8,9~~	~~0~~,1,2,3, ~~4~~,5,6,7, ~~8~~,9	~~0,1,2,3~~, ~~4,5,6,7~~, 8,~~9~~	~~0~~,1	0,~~1~~	0,~~1~~

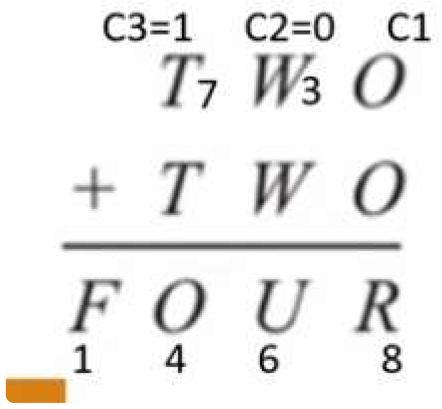
-

- **Step 5**
  - Choosing T
  - The constraints are
    - T+T = O
    - O = 4
    - T+T has a carry
  - Combining there constraints, T+T will be 14
  - T = 7
  - Removing 7 from others domains
  - 

T	W	O	F	U	R	C3	C2	C1
~~0,1,2,3~~, ~~4,5,6~~,7, ~~8,9~~	~~0~~,1,2,3, ~~4~~,5,6,~~7~~, ~~8~~,9	~~0,1,2,3~~, 4,~~5,6,7~~, ~~8,9~~	~~0~~,1,2,3, ~~4,5,6,7~~, ~~8,9~~	~~0~~,1,2,3, ~~4~~,5,6,~~7~~, ~~8~~,9	~~0,1,2,3~~, ~~4,5,6,7~~, 8,~~9~~	~~0~~,1	0,~~1~~	0,~~1~~

  -

- **Step 6**
  - Choosing U
  - Constraints
    - W+W = U
    - 2W = U

        - U is an even number
        - When removing odd numbers, the only 2 values are 2 and 6
        - It cant be 2, since W has to be 1, which is already taken
        - Only number that can satisfy is U=6, which makes W = 3. Which is also unique
    - U = 6
    - W = 3
- **End Result**
    - 

9. Arc Consistency Algorithm

Scenario

Imagine a puzzle where you assign values to variables under constraints. For example:

- **Variables**: X, Y, Z
- **Domains**:
    - $D(X) = \{1, 2, 3\}$
    - $D(Y) = \{1, 2\}$
    - $D(Z) = \{2, 3\}$
- **Constraints**:
    - $X > Y$
    - $Z = Y + 1$

Steps of AC-3 Algorithm

1. **Initialize the queue**:
   Add all pairs of variables with constraints between them to a queue.
   Queue: $\{(X,Y),(Y,X),(Y,Z),(Z,Y)\}$
2. **Process each arc**:
   AC-3 pops an arc from the queue, checks consistency, and revises domains if necessary.

Example Walkthrough

Step 1: Process (X, Y)

- **Constraint**: $X > Y$
- Check each value of X and remove values that cannot satisfy $X > Y$ for any $Y \in D(Y)$:
  - $X = 1$: $1 > Y$ is false for all Y (1,2). Remove 1 from $D(X)$.
  - $X = 2$: Satisfies $X > Y$ for $Y = 1$.
  - $X = 3$: Satisfies $X > Y$ for $Y = 1, 2$.

  Revised Domain:
- $D(X) = \{2, 3\}$
- Since $D(X)$ changed, add neighbors of X back to the queue.
- Neighbors are variables that share a constraint with X.
- These are the variables that might be affected by the change in $D(X)$. For this example, the neighbors of X are Y. Thus, the arcs (Y, X) are added back to the queue.

Step 2: Process (Y, Z)

- **Constraint**: $Z = Y + 1$
- Check each value of Y and revise $D(Z)$:
  - $Y = 1$: $Z = 2$ satisfies the constraint.
  - $Y = 2$: $Z = 3$ satisfies the constraint.

  Domain remains unchanged:
  $D(Y) = \{1, 2\}, \quad D(Z) = \{2, 3\}$

Step 3: Process (Y, X)

- **Constraint**: $X > Y$
- Re-check $D(Y)$ based on $D(X) = \{2, 3\}$:

- $Y = 1$: Satisfies $X > Y$ for all X.
- $Y = 2$: $X > Y$ is false for $X = 2$. Remove 2 from $D(Y)$.

Revised Domain:
$D(Y) = \{1\}$

Step 4: Process (Z, Y)

- **Constraint**: $Z = Y + 1$
- Re-check $D(Z)$ based on $D(Y) = \{1\}$:
  - $Z = 2$: Satisfies $Z = Y + 1$.
  - $Z = 3$: Does not satisfy the constraint. Remove 3 from $D(Z)$.

Revised Domain:
$D(Z) = \{2\}$

Final Domains

- $D(X) = \{2, 3\}$
- $D(Y) = \{1\}$
- $D(Z) = \{2\}$

At this point, the CSP is arc-consistent: all constraints are satisfied by the current domains.

AC-3 algorithm for arc consistency

1. Here we are using queue data structure
   1. Intially queue contains all the arc in the CSP
2. AC-3 pops off an arbitary arc (xi,xj) and make xi arc consistent with xj
3. If Di is unchanged (Domain is unchanged) the algorithm moves to next arc
4. Else If the domain is smaller
   5. 4.1 Add the neighbours of the arc xi
   1. 1. Example: x = y^2
   2. The domain of X = {0,1,2,3,4,5...}
   3. But the domain of y is smaller, since its the square of x
   4. Domain of Y = {0,1,4,9...}
   5. xk is the neighbour of xi
5. else if Di is revised down to nothing (revised means domain size reducing) return failure

6. Otherwise we kept checking to make it consistent
7. At a point we get a CSP equivalent to original CSP

Function based algorithm

```
function Revise(CSP, xi,xj) returns True if we revise the domain of xi
        revised <- False
        for each x in di do
                if no value in y in D allow (x,y) to satisfy the constraint
between xi and xj then delete x from Domain D
                        revised <- true
                        return Revised
```

Complexity of AC-3

- Best Case -> O(d^3)
- Worst Case -> O(d^3)