

### UNIT III

**Constraint Satisfaction Problems: Backtracking search for CSP's, Local search for CSP**

**Game Playing: Adversarial search, Games, Minimax algorithm, optimal decisions in multiplayer games, Alpha-Beta pruning, Evaluation functions, cutting of search.**

**Constraint Satisfaction Problems (CSPs)** are mathematical problems defined by a set of variables, each with a domain of possible values, and a set of constraints specifying allowable combinations of values. CSPs are widely used in artificial intelligence (AI) to model and solve combinatorial problems in areas like scheduling, resource allocation, and puzzle solving.

#### Key Components of CSPs

1. **Variables:**  
Represent the entities involved in the problem (e.g.,  $X_1, X_2, \dots, X_n$ ).
2. **Domains:**  
Each variable has a domain of possible values it can take (e.g.,  $D_1 = \{1, 2, 3\}$ ).
3. **Constraints:**  
Rules or relationships that define which combinations of variable assignments are valid. Constraints can be:
  - **Unary:** Apply to a single variable.
  - **Binary:** Relate two variables.
  - **Global:** Involve multiple variables.
4. **State:**  
A partial or complete assignment of values to variables.

#### Example of a CSP

**Problem:** Assign colors to regions on a map so that no two adjacent regions have the same color.

- **Variables:** Regions A, B, C, D.
- **Domains:** Possible colors {Red, Blue, Green}.
- **Constraints:** Adjacent regions must have different colors (e.g.,  $A \neq B, B \neq C$ ).

#### A CSP consists of:

- **Variables:**  $X_1, X_2, \dots, X_n$ .
- **Domains:** Each variable  $X_i$  has a non-empty domain  $D_i$  of possible values.
- **Constraints:**  $C_1, C_2, \dots, C_m$  which specify allowable combinations of values for subsets of variables.
- **State:** An assignment of values to variables (e.g.,  $X_i = v_i, X_j = v_j$ ).

## Key Concepts:

1. **Domains:**
  - **Discrete Domain:** Infinite possibilities for variables.
  - **Finite Domain:** Continuous or limited values for variables (e.g., integers, real numbers).
2. **Constraint Types:**
  - **Unary:** Restricts a single variable.
  - **Binary:** Relates two variables (e.g.,  $X_2$  lies between  $X_1$  and  $X_3$ ).
  - **Global:** Involves multiple variables simultaneously.
3. **Solution Requirements:**
  - **State-Space:** Defined by variable assignments.
  - **Assignments:**
    - **Consistent/Legal Assignment:** Satisfies all constraints.
    - **Complete Assignment:** Assigns values to all variables consistently.
    - **Partial Assignment:** Assigns values to some variables only.
4. **Constraint Types in Solving CSP:**
  - **Linear Constraints:** Used in linear programming; variables in linear forms.
  - **Non-linear Constraints:** Used in non-linear programming; variables in non-linear forms.
  - **Preference Constraints:** Constraints reflecting real-world preferences.

## To solve a CSP:

- Define a state-space.
- Ensure consistency with constraints.

## Algorithms and Approaches:

1. **Search-Based Approaches:** Explore the state-space to find a solution.
2. **Constraint Propagation:**
  - Reduces the legal number of values for variables.
  - Achieves local consistency through inference.

CSPs are solved using techniques like **backtracking**, **local search**, and **constraint propagation**, making them fundamental in AI problem-solving.

## Common Examples of CSPs:

1. **Graph Coloring Problem**
2. **Sudoku**
3. **N-Queen Problem**
4. **Latin Square Problem**

## Graph Coloring Problem

### Objective:

Color the vertices of a graph using  $M$  colors such that no two adjacent vertices have the same color.

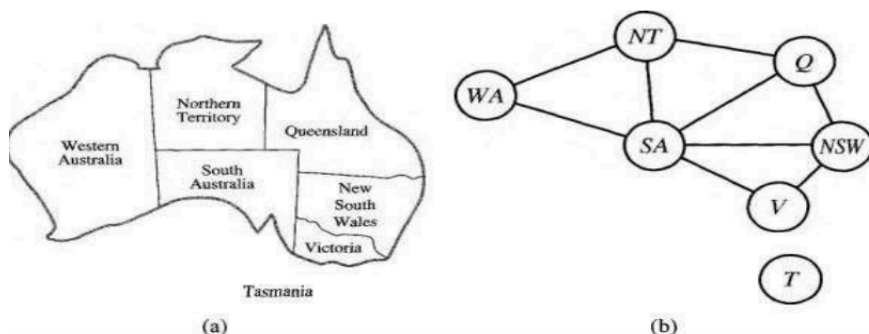
### Details:

- **Input:** A graph with vertices and edges, and  $M$  colors.
- **Output:**
  - If a solution exists: The colored graph.
  - If no solution exists: Output "No solution possible."
- **Constraints:**
  - Each vertex must be assigned one of the  $M$  colors.
  - No two adjacent vertices can share the same color.

### Example:

- **Graph:**  $A \setminus - B \setminus - C$ .
- **Colors:** {Red, Blue}.
- **Solution:**  $A = \text{Red}, B = \text{Blue}, C = \text{Red}$

This problem is a classic example of a CSP, requiring algorithms like **backtracking** or **constraint propagation** to find solutions efficiently.



**Figure 5.1** (a) The principal states and territories of Australia. Coloring this map can be viewed as a **constraint satisfaction** problem. The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

## Example: Map Coloring



- **Variables:**  $X = \{WA, NT, Q, NSW, V, SA, T\}$
- **Domains:**  $D_i = \{\text{red, green, blue}\}$
- **Constraints:** adjacent regions must have different colors
- **Solution?**

## Solution: Complete and Consistent Assignment



- **Variables:**  $X = \{WA, NT, Q, NSW, V, SA, T\}$
- **Domains:**  $D_i = \{\text{red, green, blue}\}$
- **Constraints:** adjacent regions must have different colors
- **Solution?**  $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{red}\}$ .

## Sudoku

### Objective:

Fill a 9x9 grid with digits (1–9) such that:

1. Each row contains all digits exactly once.
2. Each column contains all digits exactly once.
3. Each 3x3 subgrid contains all digits exactly once.

### Details:

- **Input:** A partially filled 9x9 grid.
- **Output:** A complete grid satisfying the above constraints.
- **Constraints:** Numbers must not repeat in any row, column, or subgrid.

### Solution Technique:

Sudoku is solved using techniques like backtracking, constraint propagation, and local search.

## N-Queen Problem

### Objective:

Place N queens on an N×N chessboard such that no two queens attack each other.

### Details:

- **Input:** Size of the chessboard (N).
- **Output:** A valid arrangement of queens.
- **Constraints:**
  - No two queens can share the same row, column, or diagonal.

### Example (N=4):

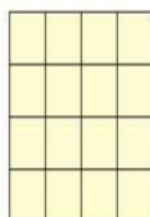
Q	—	—	—
—	—	Q	—
—	—	—	Q
—	Q	—	—

Example:  $n$ -queens

Example: 4-queens

Example: 4-queens

Place  $n$ -queens on an  $n \times n$  board so that no pair of queens attacks each other.



Variables:

$x_1, x_2, x_3, x_4$

Domains:

$\{1, 2, 3, 4\}$

Constraints:

$x_i \neq x_j$  and  $|x_i - x_j| \neq |i - j|$

	$x_1$	$x_2$	$x_3$	$x_4$
1				
2				
3				
4				

STEP-1

One solution:

$x_1 \leftarrow 2$

$x_2 \leftarrow 4$

$x_3 \leftarrow 1$

$x_4 \leftarrow 3$

	$x_1$	$x_2$	$x_3$	$x_4$
1			Q	
2	Q			
3				Q
4		Q		

STEP-2

**Latin Square Problem** Fill an  $N \times N$  grid with  $N$  symbols such that:

1. Each row contains all symbols exactly once.
2. Each column contains all symbols exactly once.

**Details:**

- **Input:** Grid size ( $N$ ) and symbols (e.g.,  $1, 2, \dots, N$ ).
- **Output:** A completed Latin square.
- **Constraints:** Rows and columns must have unique symbols.

**Example ( $N=3$ ):**

1	2	3
3	1	2
2	3	1

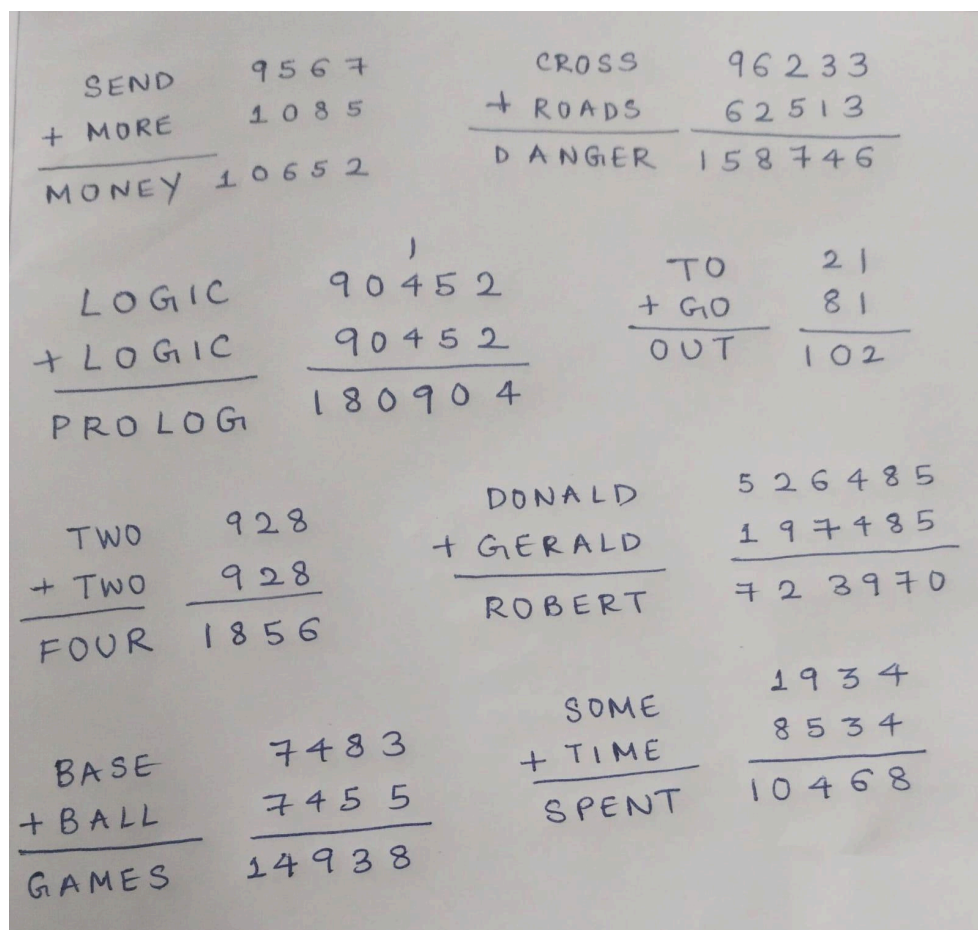
### Cryptarithmic Problems

**Definition:**

Cryptarithmic problems are a type of mathematical puzzle where the digits in a numerical equation are replaced by letters or symbols. The goal is to assign each letter a unique digit (0–9) to satisfy the equation.

**Rules of Cryptarithmic Problems**

1. Each letter represents a unique digit (0–9).
2. Leading digits of any number cannot be zero.
3. The numerical equation must be valid when letters are replaced with their corresponding digits.



The image shows several handwritten cryptarithm solutions. Each problem consists of words representing numbers, with their numerical values written to the right. The solutions are as follows:

SEND + MORE ----- MONEY	9567 1085 ----- 10652
LOGIC + LOGIC ----- PROLOG	90452 90452 ----- 180904
TWO + TWO ----- FOUR	928 928 ----- 1856
BASE + BALL ----- GAMES	7483 7455 ----- 14938
CROSS + ROADS ----- DANGER	96233 62513 ----- 158746
TO + GO ----- OUT	21 81 ----- 102
DONALD + GERALD ----- ROBERT	526485 197485 ----- 723970
SOME + TIME ----- SPENT	1934 8534 ----- 10468

## Backtracking Search for CSPs

Backtracking search is a depth-first search (DFS) algorithm used to solve CSPs by assigning values to one variable at a time and reverting (backtracking) when no legal value can be assigned to a variable.

### How It Works:

1. **Variable Selection:** Choose an unassigned variable.
2. **Value Assignment:** Assign a value from the variable's domain that satisfies all constraints.
3. **Constraint Checking:** Verify the assignment does not violate constraints with already assigned variables.
4. **Backtracking:**
  - If no legal values remain for a variable, revert to the previous variable and try a different value.
  - Continue until a solution is found or all possibilities are exhausted.

### Enhancements for Backtracking:

1. **Forward Checking:** Eliminate values from the domains of unassigned variables that violate constraints with the current assignment.
2. **Arc Consistency:** Use algorithms like AC-3 to maintain consistency in constraints.
3. **Heuristics:**
  - **Minimum Remaining Values (MRV):** Select the variable with the fewest legal values.
  - **Degree Heuristic:** Choose the variable involved in the largest number of constraints.
  - **Least Constraining Value:** Assign the value that leaves the most options for other variables.

**Propagating Information through Constraints:** Propagation of information in CSPs refers to the process of narrowing down the possible values of variables based on the constraints. It involves using the constraints to eliminate inconsistent values or domains, ensuring that the solution space is progressively reduced, and more specific solutions can be found.

1. **Forward Checking:** Forward checking is a technique used in CSPs to reduce the search space during the solving process. It works by examining the variables that are yet to be assigned and checks the constraints for these variables in advance. After assigning a value to a variable, forward checking eliminates values from the domains of unassigned variables that are inconsistent with the current assignment, thus avoiding paths that will not lead to a solution.
2. **Constraint Propagation:** Constraint propagation is a general technique used in CSPs to reduce the domains of variables by enforcing the constraints. It iteratively revises the domains of the variables based on the constraints, propagating information about which values can no longer be possible for certain variables. This reduces the search space and makes it easier to find a solution.
3. **Arc Consistency:** Arc consistency is a form of constraint propagation used to ensure that for each variable in a binary constraint, every value in the domain of that variable

has a valid corresponding value in the domain of the other variable involved in the constraint. A CSP is considered arc-consistent if for every pair of variables (X, Y) with a constraint between them, for every value in the domain of X, there is some value in the domain of Y that satisfies the constraint. Arc consistency is typically enforced using algorithms like AC-3 (Arc-Consistency 3).

The **Graph Coloring Problem** involves coloring the vertices of a graph using **M** colors such that no two adjacent vertices have the same color. The goal is to find the minimum number of colors (chromatic number) required to color the graph, or to determine if it's possible to color the graph with the given **M** colors.

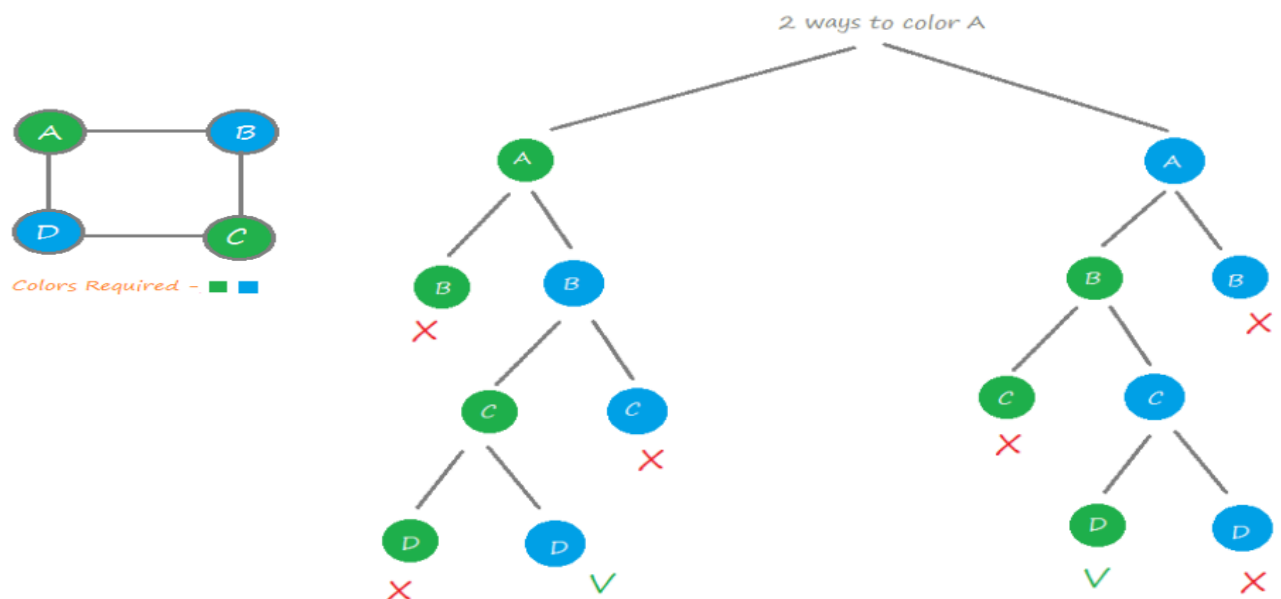
### Backtracking Algorithm for Graph Coloring:

1. **Assign a color** to the current vertex if it's valid (i.e., no adjacent vertex has the same color).
2. **Recursively color** the next vertex. If all vertices are colored, the algorithm ends successfully.
3. If no color can be assigned, **backtrack** by removing the color from the previous vertex and trying a different color.
4. If all possible colorings fail, return **"no solution possible."**

The algorithm explores different coloring options, backtracks when necessary, and ensures that adjacent vertices never share the same color.

### Key Points:

- **Valid Color:** A color is valid if no adjacent vertex has the same color.
- **Backtracking:** Undo color assignments and try different options.
- **Recursive Approach:** Move to the next vertex after coloring the current one.



## .Local Search for CSPs

- Local search explores the **space of complete assignments** of values to variables, where each assignment satisfies some or all constraints.
- **Neighbors** of a current node are defined as similar assignments that differ in the value of one or more variables.
- The algorithm moves from one node to another, guided by a **scoring function** that evaluates how good each assignment is (e.g., how many constraints are satisfied).

### Example :

- **Initial State:** A Sudoku grid with some violations of constraints (e.g., repeated numbers in rows, columns, or boxes).
- **Transition to Neighbor:** Change the value of one or more cells to reduce the constraint violations.
- **Final State:** A valid Sudoku solution where all constraints are satisfied.

### Definition of a Local Search Problem:

A local search problem for CSP consists of:

1. **CSP:** A set of variables, their domains, and the constraints on their joint values.
  - Each node in the search space corresponds to a **complete assignment** of values to all variables.
2. **Neighbor Relation:**
  - Defines how nodes (assignments) are connected.
  - A neighbor is an assignment that differs slightly from the current assignment (e.g., changing the value of a single variable).
3. **Scoring Function  $h(n)$ :**
  - Evaluates the cost of a node  $n$ , which the search algorithm aims to minimize.
  - Examples:
    - **Constraint Violations:** The number of constraints not satisfied by the assignment in  $n$ .
    - **Cost:** In optimization problems, the cost associated with the solution.

### Applications of Local Search in CSP:

- Solving **Sudoku** by incrementally improving assignments.
- Finding solutions for **N-Queen problems** by reducing queen conflicts.
- **Timetabling and Scheduling** by optimizing assignments to minimize conflicts.

## Game Playing

**Adversarial search** is used in AI when two or more players (or agents) compete against each other, each trying to win or maximize their gain while minimizing the opponent's. It's commonly used to model games where opponents take turns, such as Chess or Tic-Tac-Toe.

### Types of Games in AI

1. **Deterministic Games (Also Known As: Perfect Predictability Games)**
  - **Description:** No randomness; outcomes are entirely determined by player actions.
  - **Example:** Chess, Tic-Tac-Toe.
2. **Stochastic Games (Also Known As: Games of Chance)**
  - **Description:** Involve random elements like dice rolls or card draws.
  - **Example:** Backgammon, Poker.
3. **Perfect Information Games (Also Known As: Fully Observable Games)**
  - **Description:** All players have complete information about the game state at any time.
  - **Example:** Chess, Checkers.
4. **Imperfect Information Games (Also Known As: Partially Observable Games)**
  - **Description:** Players lack full information; some details are hidden.
  - **Example:** Poker, Battleship.
5. **Zero-Sum Games (Also Known As: Competitive Games)**
  - **Description:** One player's gain is exactly balanced by the other player's loss.
  - **Example:** Chess, Tic-Tac-Toe.
6. **Non-Zero-Sum Games (Also Known As: Cooperative or Collaborative Games)**
  - **Description:** Players can win or lose independently, and cooperation may be possible.
  - **Example:** Negotiation games, cooperative board games like Pandemic

### Minimax Algorithm

The **Minimax algorithm** is used in two-player games to determine the optimal move for a player, assuming both players play optimally.

Key Points:

1. **Players:**
  - **MAX** tries to maximize the score.
  - **MIN** tries to minimize the score for MAX.
2. **Game Tree:**
  - Represents all possible game states. Nodes are game states, and edges represent moves.
  - The algorithm explores this tree recursively, evaluating terminal states (win, loss, or draw).
3. **Evaluation:**
  - **Terminal nodes** are assigned a score (e.g., +1 for a win, -1 for a loss).

- The algorithm backtracks from these nodes to select the best move for MAX, assuming MIN plays optimally.

Algorithm Steps:

1. **Generate the game tree** with all possible moves.
2. **Evaluate terminal nodes** with win/loss/draw values.
3. **Backtrack** to determine the best move by choosing MAX or MIN nodes.
4. **Optimal Decision:** The root node gives the best move for the current player.

Properties of the Minimax Algorithm

1. **Complete:**
  - The algorithm will always find a solution if one exists in a finite game tree.
2. **Optimal:**
  - Minimax provides the optimal move if both players play optimally.
3. **Time Complexity:**
  - $O(b^m)$ , where **b** is the branching factor (average number of moves per player) and **m** is the maximum depth of the game tree.
4. **Space Complexity:**
  - $O(b^m)$ , similar to the time complexity due to the need to store all game states in the tree.

Limitations of the Minimax Algorithm:

1. **Slow for Complex Games:**
  - In games like Chess or Go, with large branching factors, the algorithm becomes very slow due to the vast number of possible game states.
2. **Branching Factor Problem:**
  - The large number of possible moves at each state in complex games makes it computationally expensive. This issue can be mitigated with **Alpha-Beta pruning**.

Example: [https://www.youtube.com/watch?v=tDv7lrklaQE&list=PL4gu8xQu0\\_5JrWjrWNMmXNx4zFwRrpqCR&index=26](https://www.youtube.com/watch?v=tDv7lrklaQE&list=PL4gu8xQu0_5JrWjrWNMmXNx4zFwRrpqCR&index=26)

## Optimal Decisions in Multiplayer Games

In **multiplayer games** (games with more than two players), the **Minimax algorithm** can be extended by considering the **multiple players' strategies**. Here's how it's done:

1. **Multiple Values for Nodes:**
  - Instead of a single value for each game state, a **vector of values** is used. Each element in the vector represents the outcome for a specific player.
  - For example, in a three-player game with players A, B, and C, each node in the game tree would have a vector  $(v_A, v_B, v_C)$ , where:
    - **vA:** Value for Player A's outcome.
    - **vB:** Value for Player B's outcome.
    - **vC:** Value for Player C's outcome.

## 2. Strategy for Each Player:

- Each player evaluates the game state based on their position and the potential outcomes for themselves and their opponents.
- Players aim to **maximize their own value** while considering how the moves will affect the others.

**Alpha-beta pruning** is an optimization technique used with the **Minimax algorithm** to make decision-making in game trees more efficient. It reduces the number of nodes evaluated by pruning branches that will not affect the final decision.

Key Concepts:

### 1. Pruning:

- Involves **eliminating** branches of the game tree that won't affect the final decision, thus speeding up the process.

### 2. Alpha and Beta Parameters:

- **Alpha ( $\alpha$ )**: The best value found so far for the **Max** player (initially  $-\infty$ ).
- **Beta ( $\beta$ )**: The best value found so far for the **Min** player (initially  $+\infty$ ).

### 3. Working:

- As the algorithm traverses the tree:
  - **Max player** updates **Alpha** (trying to maximize the value).
  - **Min player** updates **Beta** (trying to minimize the value).
- If **Alpha  $\geq$  Beta**, the current branch is pruned (i.e., no need to evaluate further because it won't change the decision).

### 4. Pruning Criteria:

- Prune the branch when further exploration won't affect the final decision due to previously encountered values of **Alpha** and **Beta**.

### 5. Efficiency:

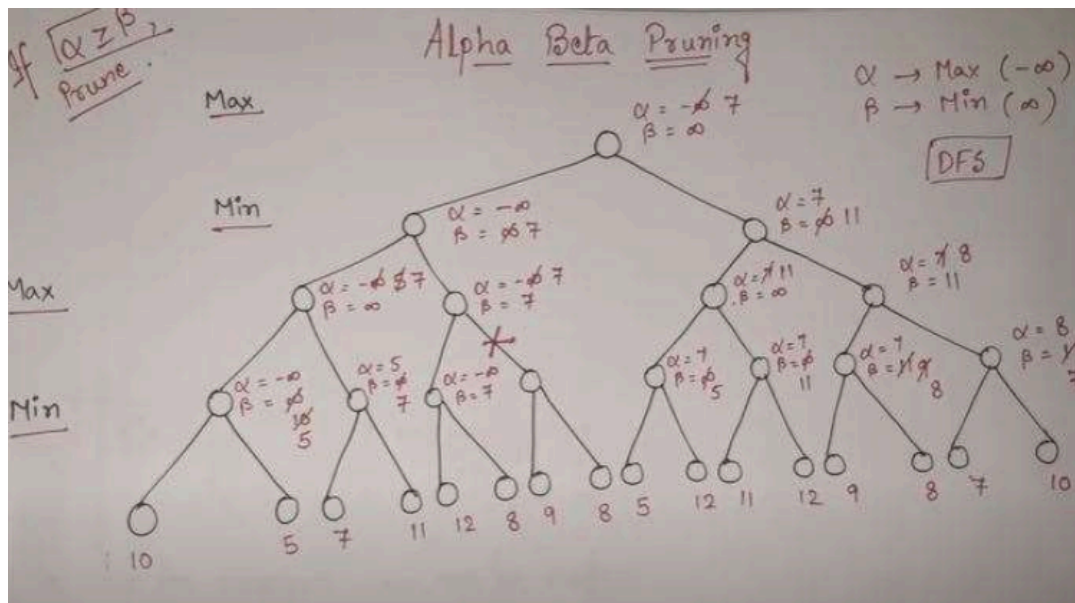
- This technique allows **significant reduction** in the number of nodes evaluated, making the algorithm much faster, especially in deep game trees.

Key Points:

- **Max player**: Only updates **Alpha**.
- **Min player**: Only updates **Beta**.
- During backtracking, the node values are passed up the tree.
- The **Alpha** and **Beta** values are propagated to child nodes, not the full evaluation values.

Example: [https://www.youtube.com/watch?v=9D1hVGumxCo&list=PL4gu8xQu0\\_5JrWjrWNMmXNx4zFwRrpqCR&index=28](https://www.youtube.com/watch?v=9D1hVGumxCo&list=PL4gu8xQu0_5JrWjrWNMmXNx4zFwRrpqCR&index=28)

[https://www.youtube.com/watch?v=DCNP5L9\\_t4E&list=PL4gu8xQu0\\_5JrWjrWNMmXNx4zFwRrpqCR&index=29](https://www.youtube.com/watch?v=DCNP5L9_t4E&list=PL4gu8xQu0_5JrWjrWNMmXNx4zFwRrpqCR&index=29)



**Evaluation Functions** are used in game-playing AI to help make faster decisions by estimating the value of a game state without fully exploring all possible moves.

Key Points:

1. **Purpose:**

- They estimate how good a non-terminal (not game-over) position is for the player.
- This helps the AI decide on the best move faster.

2. **Improvement to Algorithms:**

- They speed up the **Minimax** and **Alpha-Beta** algorithms by allowing the AI to cut off the search earlier and use estimates instead of exploring all possible game states.

3. **Learning Evaluation Functions:**

- Evaluation functions can be **learned** using methods like **neural networks**.
- Example: The **TD-GAMMON** program used a neural network to learn how to play **Backgammon** by estimating possible outcomes of the game.

**Cutting Off Search:**

1. **When to Stop:**

- Instead of searching the entire game tree, we can stop the search early by using an evaluation function to estimate a state's value when the depth limit is reached.

2. **Fixed Depth Limit:**

- Set a limit on how deep the AI will search, and use the evaluation function to estimate states beyond that depth.

3. **Horizon Effect:**

- This happens when the AI doesn't see upcoming important moves because it stopped searching too early. It's harder to fix but is something to watch for.