MODULE - 4
Constrain satisfaction problems

What is a Constraint Satisfaction Problem (CSP)?

A **Constraint Satisfaction Problem** is a mathematical problem where the solution must meet a number of constraints. In a CSP, the objective is to assign values to variables such that all the constraints are satisfied. CSPs are used extensively in artificial intelligence for decision-making problems where resources must be managed or arranged within strict guidelines.

Common applications of CSPs include:

• Assigning resources like employees or equipment while respecting time and availability constraints.

• Organizing tasks with specific deadlines or sequences.

• Distributing resources efficiently without overuse.

Components of Constraint Satisfaction Problems

CSPs are composed of three key elements:

The representation of Constraint Satisfaction Problems (CSPs) is crucial for effectively solving these problems. Let's explore how to represent CSPs using variables, domains, and constraints:

1. Variables as Placeholders:

Variables in CSPs act **as placeholders for problem components that need to be assigned values.** They represent the entities or attributes of the problem under consideration. For example:

•In a Sudoku puzzle, variables represent the empty cells that need numbers.

•In job scheduling, variables might represent tasks to be scheduled.

2 Domains: Each variable in a CSP is associated with a domain, which defines the set of values that the variable can take. Domains are a critical part of the CSP representation, as they restrict the possible assignments of values to variables. For instance:

- In Sudoku, the domain for each empty cell is the numbers from 1 to 9.
- In scheduling, the domain for a task might be the available time slots.
- In map coloring, the domain could be a list of available colors.

3. Constraints:

Constraints in CSPs specify the relationships or conditions that must be satisfied by the variables. Constraints restrict the combinations of values that variables can take. Constraints can be unary (involving a single variable), binary (involving two variables), or n-ary (involving more than two variables). Constraints are typically represented in the form of logical expressions, equations, or functions. For example:

•In Sudoku, constraints ensure that no two numbers are repeated in the same row, column, or subgrid.

•In scheduling, constraints might involve ensuring that two tasks are not scheduled at the same time.

Significance of Constraint Satisfaction Problem in AI

CSPs are highly significant in artificial intelligence for several reasons:

•They model a wide range of **real-world problems where decision-making** is subject to certain conditions and limitations.

•CSPs offer a **structured and general framework for representing and solving problems, making them versatile in problem-solving applications.**

•Many AI applications, such **as scheduling, planning, and configuration**, can be mapped to CSPs, allowing AI systems to find optimal solutions efficiently.

Types of Constraint Satisfaction Problems

CSPs can be classified into different types based on their constraints and problem characteristics:

1.Binary CSPs: In these problems, each constraint involves only two variables. For example, in a scheduling problem, the constraint could specify that task A must be completed before task B.

2.Non-Binary CSPs: These problems have constraints that involve more than two variables. For instance, in a seating arrangement problem, a constraint could state that three people cannot sit next to each other.

3. **Hard and Soft Constraints**: Hard constraints must be strictly satisfied, while soft constraints can be violated, but at a certain cost. This distinction is often used in real-world applications where not all constraints are equally important.

Representation of Constraint Satisfaction Problems (CSP)

In **Constraint Satisfaction Problems (CSP)**, the solution process involves the interaction of variables, domains, and constraints. Below is a structured representation of how CSP is formulated:

Finite Set of Variables (V1, V2, ……Vn) :

The problem consists of a set of variables, each of which needs to be assigned a value that satisfies the given constraints.

Non-Empty Domain for Each Variable (D1, D2,…Dn) :

Each variable has a domain—a set of possible values that it can take. For example, in a Sudoku puzzle, the domain could be the numbers 1 to 9 for each cell.

Finite Set of Constraints (C1, C2, …. Cn):

Constraints restrict the possible values that variables can take. Each constraint defines a rule or relationship between variables.

Constraint Representation:

Each constraint Ci, is represented as a pair <scope, relation>, where:

Scope: The set of **variables involved in the constraint**.

Relation: A list of valid combinations of variable values that satisfy the constraint.

CSP Algorithms: Solving Constraint Satisfaction Problems Efficiently

Constraint Satisfaction Problems (CSPs) rely on various algorithms to explore and optimize the search space, ensuring that solutions meet the specified constraints. Here's a breakdown of the most commonly used CSP algorithms:

1.Backtracking Algorithm

The backtracking algorithm is a depth-first search method used to systematically explore possible solutions in CSPs. It operates by assigning values to variables and backtracks if any assignment violates a constraint.

How it works:

The algorithm selects a variable and assigns it a value.

It recursively assigns values to subsequent variables.

If a conflict arises (i.e., a variable cannot be assigned a valid value), the algorithm backtracks to the previous variable and tries a different value.

The process continues until either a valid solution is found or all possibilities have been exhausted.

This method is widely used due to its simplicity but can be inefficient for large problems with many variables.

2. Forward-Checking Algorithm

The **forward-checking algorithm** is an enhancement of the backtracking algorithm that aims to reduce the search space by applying **local consistency** checks.

How it works:

- For each unassigned variable, the algorithm keeps track of remaining valid values.
- Once a variable is assigned a value, local constraints are applied to neighboring variables, eliminating inconsistent values from their domains.
- If a neighbor has no valid values left after forward-checking, the algorithm backtracks.

This method is more efficient than pure backtracking because it prevents some conflicts before they happen, reducing unnecessary computations.

3. Constraint Propagation Algorithms

Constraint propagation algorithms further reduce the search space by enforcing **local consistency** across all variables.

How it works:

•Constraints are propagated between related variables.

•Inconsistent values are eliminated from variable domains by leveraging information gained from other variables.

•These algorithms refine the search space by making **inferences**, removing values that would lead to conflicts.

Constraint propagation is commonly used in conjunction with other CSP algorithms, such as **backtracking**, to increase efficiency by narrowing down the solution space early in the search process.

Real-World Examples of CSPs

To illustrate CSPs, consider the following examples:

- **Sudoku Puzzles:** In Sudoku, the variables are the empty cells, the domains are numbers from 1 to 9, and the constraints ensure that no number is repeated in a row, column, or 3x3 subgrid.
- **Scheduling Problems:** In university course scheduling, variables might represent classes, domains represent time slots, and constraints ensure that classes with overlapping students or instructors cannot be scheduled simultaneously.
- **Map Coloring:** In the map coloring problem, variables represent regions or countries, domains represent available colors, and constraints ensure that adjacent regions must have different colors.

Backtracking Search for CSPs

Backtracking is a systematic search algorithm commonly used for solving CSPs. It explores the possible assignments to variables incrementally, checking constraints at each step to prune invalid paths.

Key Features

1. **Depth-First Search:** Backtracking operates in a depth-first manner, where variables are assigned values one by one.
2. **Constraint Checking:** At each step, the algorithm checks whether the current assignment satisfies the constraints.
3. **Backtracking:** If no valid assignment is found for a variable, the algorithm backtracks to the previous variable and tries a different value.

Backtracking Search

Overview:

- A systematic, depth-first search algorithm.
- Builds variable assignments incrementally, backtracking when constraints are violated.

Algorithm Steps:

Steps:

1.Start with the first unassigned variable.

2.Assign a value from its domain.

3.Check if the assignment satisfies all constraints.

4.If no violations occur, move to the next variable; otherwise, backtrack and try another value.

5.Repeat until all variables are assigned or no solution exists.

Advantages:

•Simple and systematic.

•Guaranteed to find a solution if one exists.

Limitations:

•Can be inefficient for large CSPs due to exponential time complexity.

Example:

Solving a Sudoku puzzle by trying numbers in empty cells, backtracking if a number violates the row, column, or subgrid constraints.

Example of Backtracking

Problem: Map Coloring (e.g., coloring 4 regions such that no two adjacent regions have the same color).

•**Visualization:** Show a step-by-step tree of assignments and backtracking.

Optimizations in Backtracking

1. **Forward Checking:**
   1. Removes invalid values from the domains of unassigned variables after an assignment.
   2. Reduces future conflicts.
2. **Constraint Propagation:**
   1. Algorithms like **AC-3** maintain arc consistency by pruning domain values.
3. **Variable and Value Ordering:**
   1. **Most Constrained Variable (MCV):** Choose the variable with the **fewest valid values.**
   2. **Least Constraining Value (LCV):** Choose the value that **minimizes restrictions on other variables.**

Strengths and Weaknesses of Backtracking

- **Strengths:**
  - Complete: Always finds a solution if one exists.
  - Straightforward and easy to implement.
- **Weaknesses:**
  - Inefficient for large or dense CSPs due to exponential search space.
  - Prone to thrashing without proper heuristics.

Local Search

Overview:

- Starts with a complete but possibly invalid assignment.
- Iteratively modifies the assignment to reduce constraint violations.
- Suitable for large-scale CSPs.

Key Techniques:

- **Hill Climbing:**
  - Improves the solution by making small changes to reduce violations.
  - May get stuck in local optima.
- **Simulated Annealing:**
  - Occasionally accepts worse solutions to escape local optima.
- **Min-Conflicts Heuristic:**
  - Chooses values that minimize the number of conflicts with other variables.

Example of Local Search

•**Problem**: **N-Queens Problem** (place N queens on an NxN chessboard such that no two queens attack each other).

•**Visualization:**

- Show a chessboard with conflicts highlighted.
- Demonstrate iterative improvement using Min-Conflicts heuristic.

Comparison of Backtracking and Local Search

Feature	**Backtracking**	**Local Search**
Completeness	Complete (finds all solutions)	Incomplete (may miss solutions)
Scalability	Poor for large problems	Scales well for large CSPs
Best Use Case	Highly constrained problems	Approximate or large-scale CSPs

Strengths and Weaknesses of Local Search

•**Strengths:**

- Handles large CSPs efficiently.
- Can adapt to dynamic problems.

Weaknesses:

- No guarantee of finding a solution.
- Sensitive to parameter settings (e.g., temperature in simulated annealing).

- **Backtracking Applications:**
- Sudoku solvers.
- Cryptarithm puzzles.
- Timetabling.
- **Local Search Applications:**
- N-Queens problem.
- Resource scheduling.
- AI planning tasks.

What is Game Playing in Artificial Intelligence?

Game Playing is an important domain of artificial intelligence. Games don’t require much knowledge; the only knowledge we need to provide is the **rules, legal moves and the conditions of winning or losing the game.** Both players try to win the game. So, both of them try to make the best move possible at each turn

Searching techniques like BFS(Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time.

Game playing in AI is an active area of research and has many practical applications, including game development, education, and military training.

By simulating game playing scenarios, AI algorithms can be used to develop more effective decision-making systems for real-world applications.

The most common search technique in game playing is **Min max search procedure.**

It is depth-first depth-limited search procedure. It is used for games like **chess and tic-tac-toe.**

The Minimax Search Algorithm

One of the most common search techniques in game playing is the **Minimax algorithm**,
which is a **depth-first, depth-limited search procedure**. Minimax is commonly used for games like chess and tic-tac-toe.

Key Functions in Minimax:

MOVEGEN: Generates all possible moves from the current position.

STATICEVALUATION: Returns a value based on the quality of a game state from the perspective of two players.

In a two-player game, one player is referred to as PLAYER1 and the other as PLAYER2. The Minimax algorithm operates by **backing up values from child nodes** to their parent nodes. **PLAYER1 tries to maximize the value of its moves**, while **PLAYER2 tries to minimize the value of its moves**. The algorithm recursively performs this procedure at each level of the game tree.

Example of Minimax:

Figure 1: Before backing up values

(The diagram illustrates the game tree before Minimax values are propagated upward.)

Figure 2: After backing up values

The game starts with **PLAYER1**. The algorithm generates four levels of the game tree. **The values for nodes H, I, J, K, L, M, N, and O** are provided by the **STATICEVALUATION function**.

Level 3 is a **maximizing level,** so each node at this level takes the maximum value of its children.

Level 2 is a minimizing level, where each node takes the minimum value of its children.

After this process, the value of node A is calculated as 23, meaning that PLAYER1 should choose move C to maximize the chances of winning.

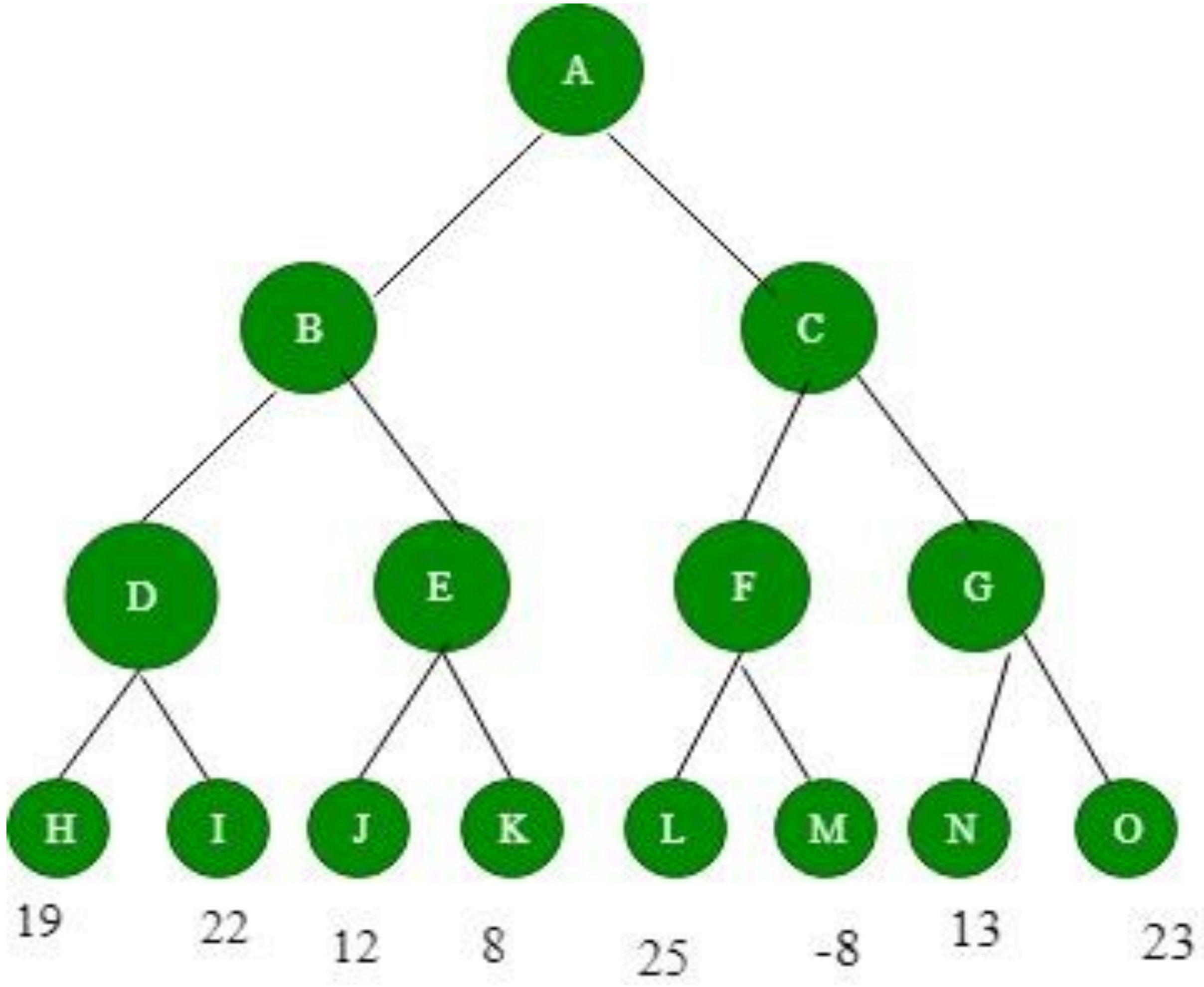


Figure 1: Before backing-up of values

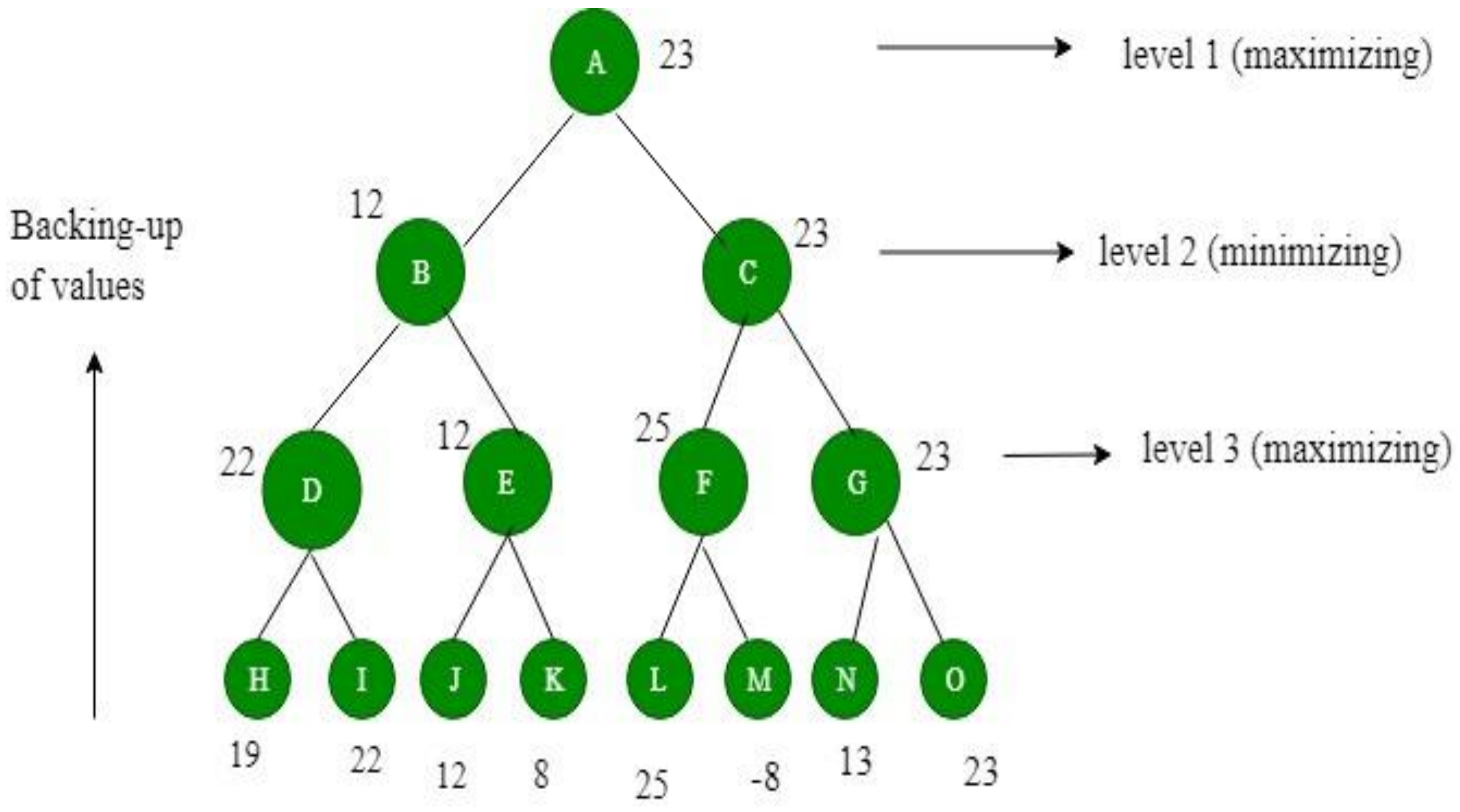


Figure 2: After backing-up of values We assume that PLAYER1 will start the game.

Advantages of Game Playing in Artificial Intelligence

1. Game playing has been a driving force behind the development of artificial intelligence and has led to the creation of new algorithms and techniques that can be applied to other areas of AI.

2. Game playing can be used to teach AI techniques and algorithms to students and professionals, as well as to provide training for military and emergency response personnel.

3. Game playing is an active area of research in AI and provides an opportunity to study and develop new techniques for decision-making and problem-solving.

4. The techniques and algorithms developed for game playing can be applied to real-world applications, such as robotics, autonomous systems, and decision support systems.

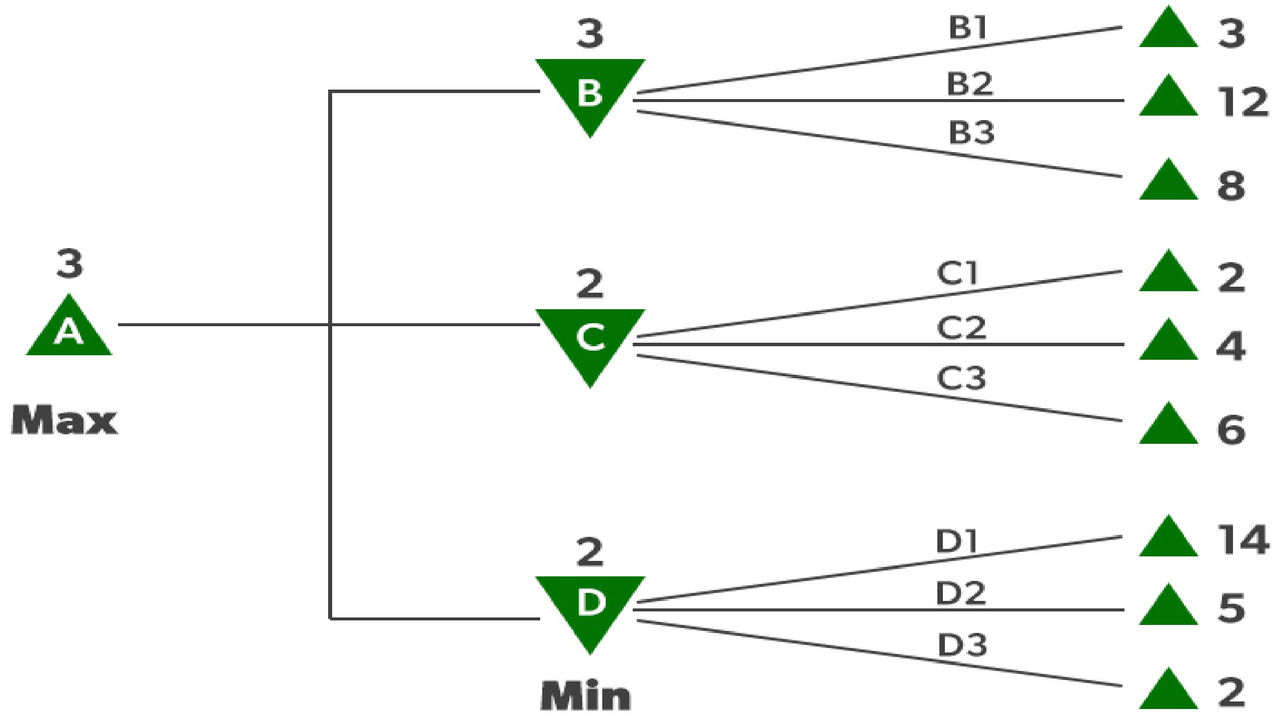
Disadvantages of Game Playing in Artificial Intelligence

1. The techniques and algorithms developed for game playing may not be well-suited for other types of applications and may need to be adapted or modified for different domains.

2. Game playing can be computationally expensive, especially for complex games such as chess or Go, and may require powerful computers to achieve real-time performance.

Optimal Decision Making in Multiplayer Games

The optimal solution becomes a contingent strategy when specifies MAX(the player on our side)'s move in the initial state, then Max move to the states resulting for every possible response by MIN. Then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on.



From the current state, the minimax method in the *Figure above* computes the minimax choice. It implements the defining equations directly using a simple recursive computation of the minimax values of each successor state. As the recursion unwinds, it progresses all the way down to the tree's leaves, where the minimax values are then backed up through the tree. In the figure below, for example, the algorithm recurses down to the three bottom-left nodes and uses the UTILITY function to determine that their values are 3, 12, and 8, respectively. Then it takes the smallest of these values, 3 in this case, and returns it as node B's backed-up value. The backed-up values of 2 for C and 2 for D are obtained using a similar approach. Finally, we add the maximum of 3, 2, and 2 to get the root node's backed-up value of 3.

The minimax algorithm explores the game tree from top to bottom in depth-first. The temporal complexity of the minimax method is O if the maximum depth of the tree is m and there are b legal moves at each point (bm). For an algorithm that creates all actions at once, the space complexity is O(bm), while for an algorithm that generates actions one at a time, the space complexity is O(m) The time cost is obviously impractical for real games, but this technique serves as a foundation for game mathematics analysis and more practical algorithms.

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.

- 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.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is: **$\alpha >= \beta$**

Key points about alpha-beta pruning:

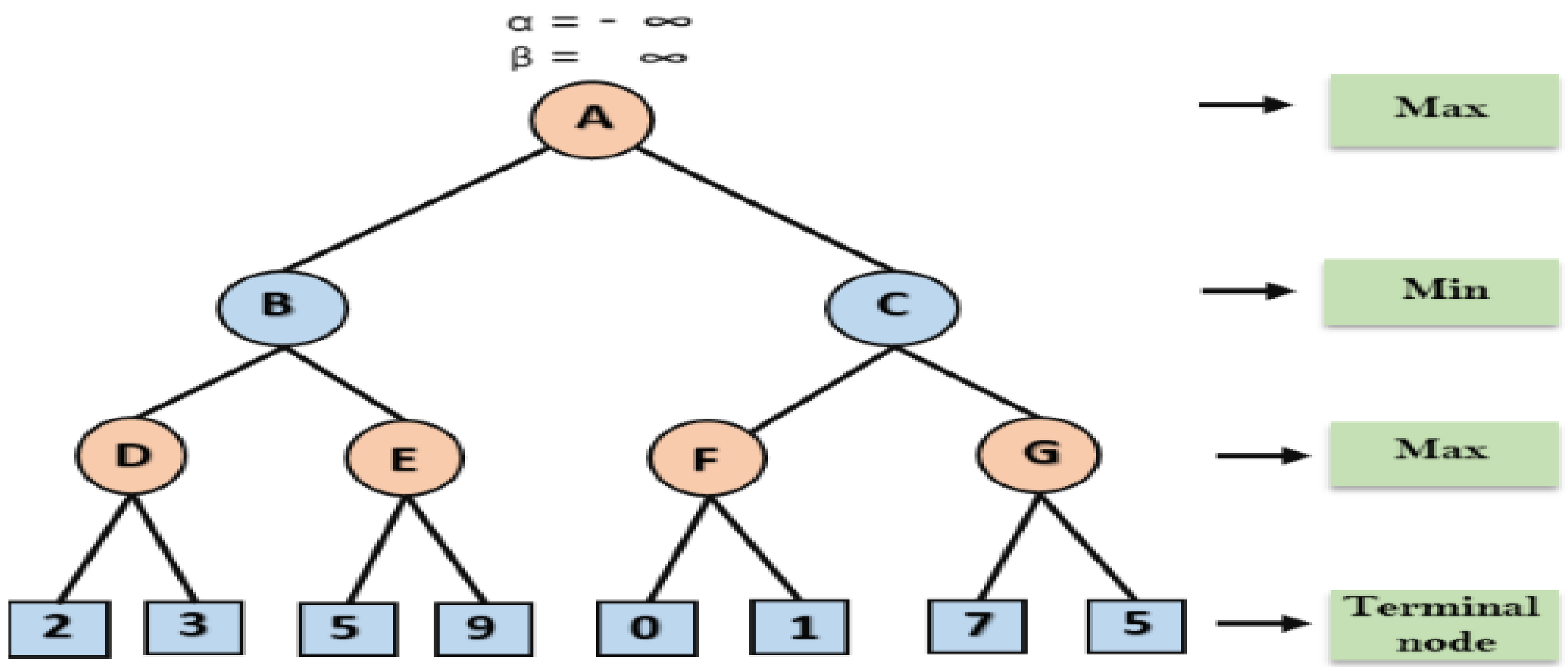
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Working of Alpha-Beta Pruning:

Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

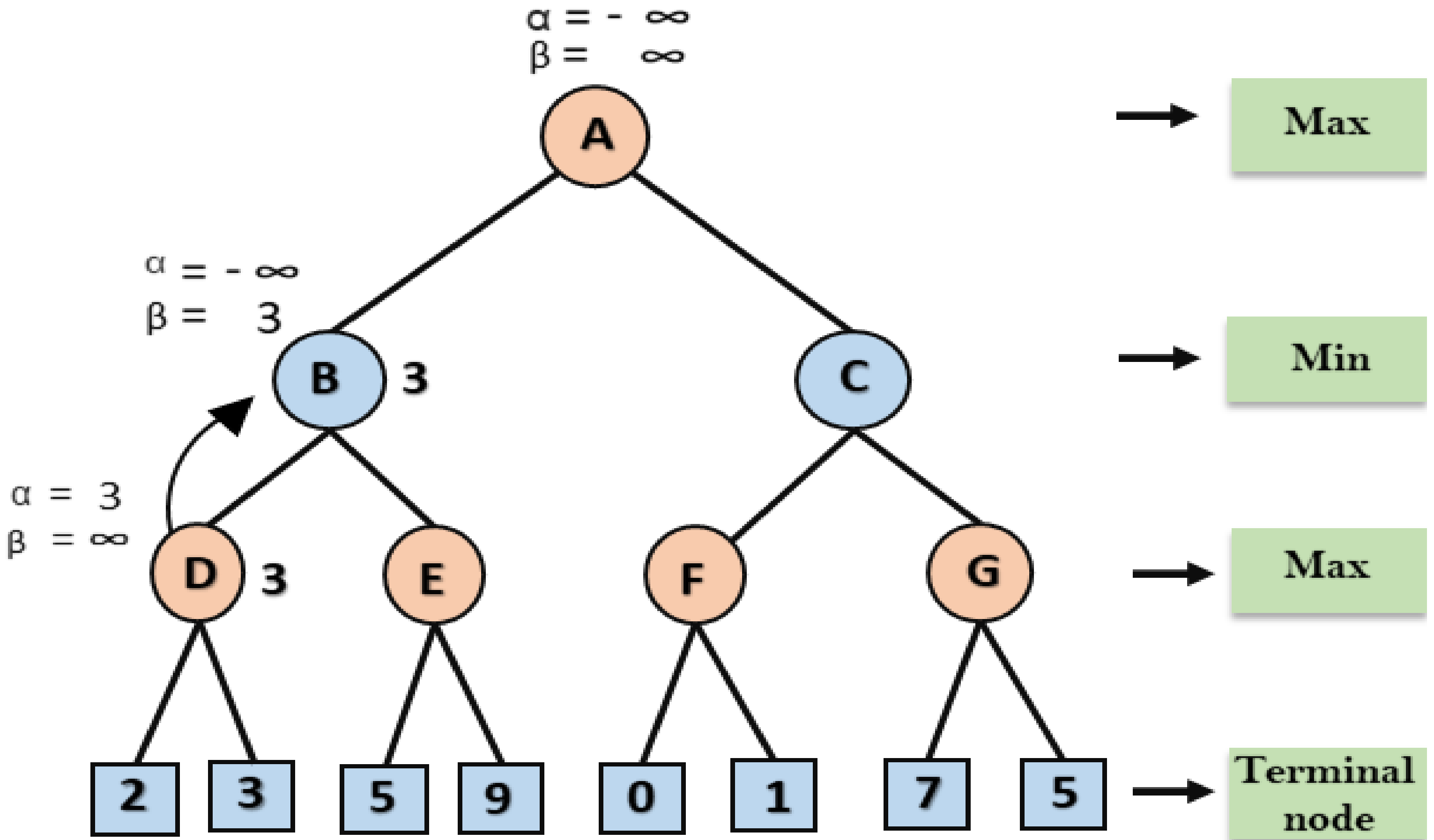
Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.

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.



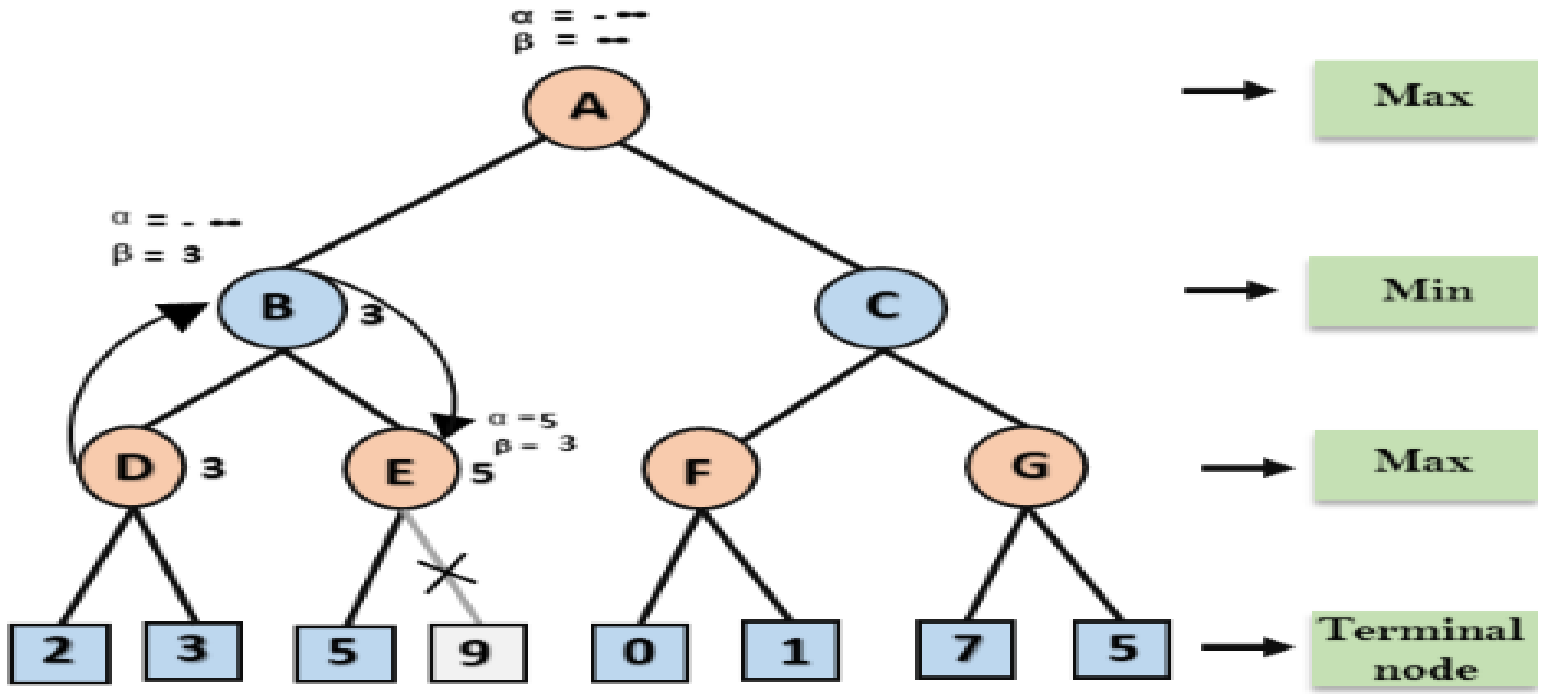
Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

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.



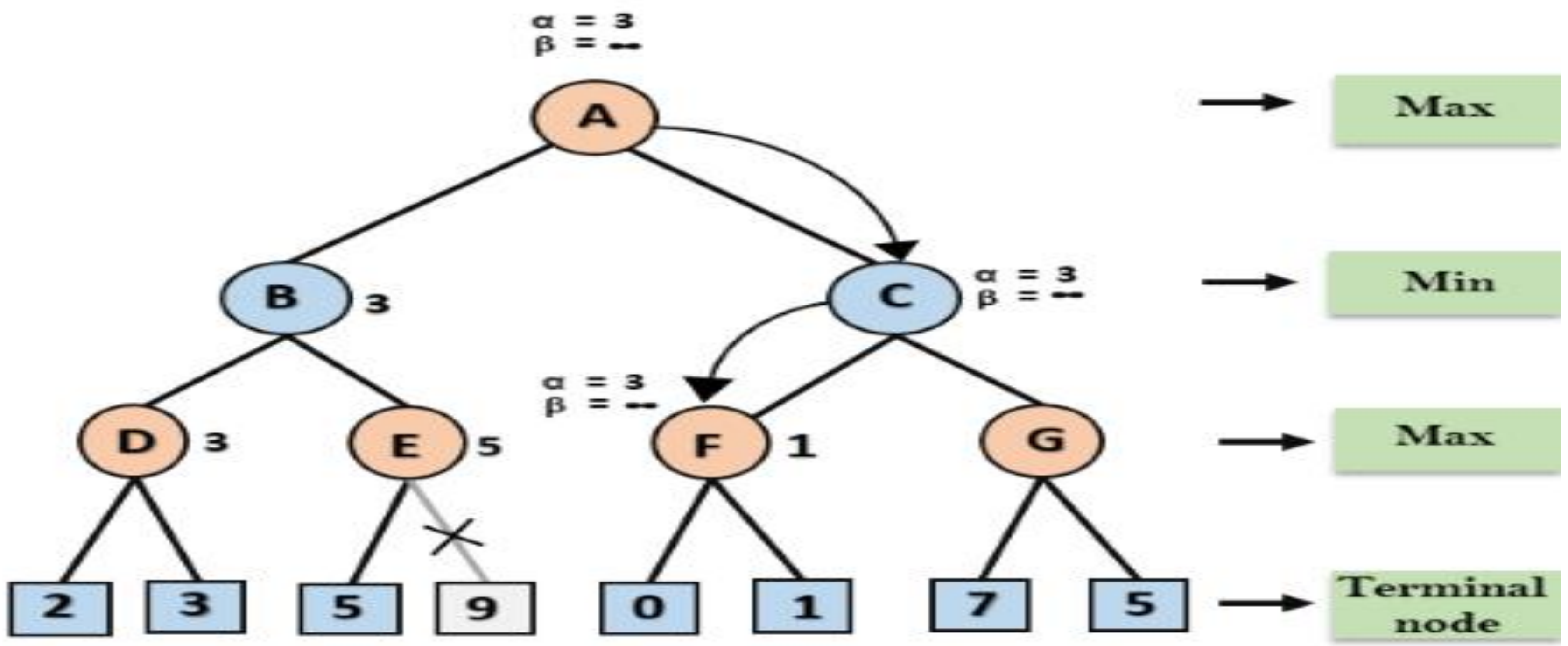
Step 4:

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 E will be 5.

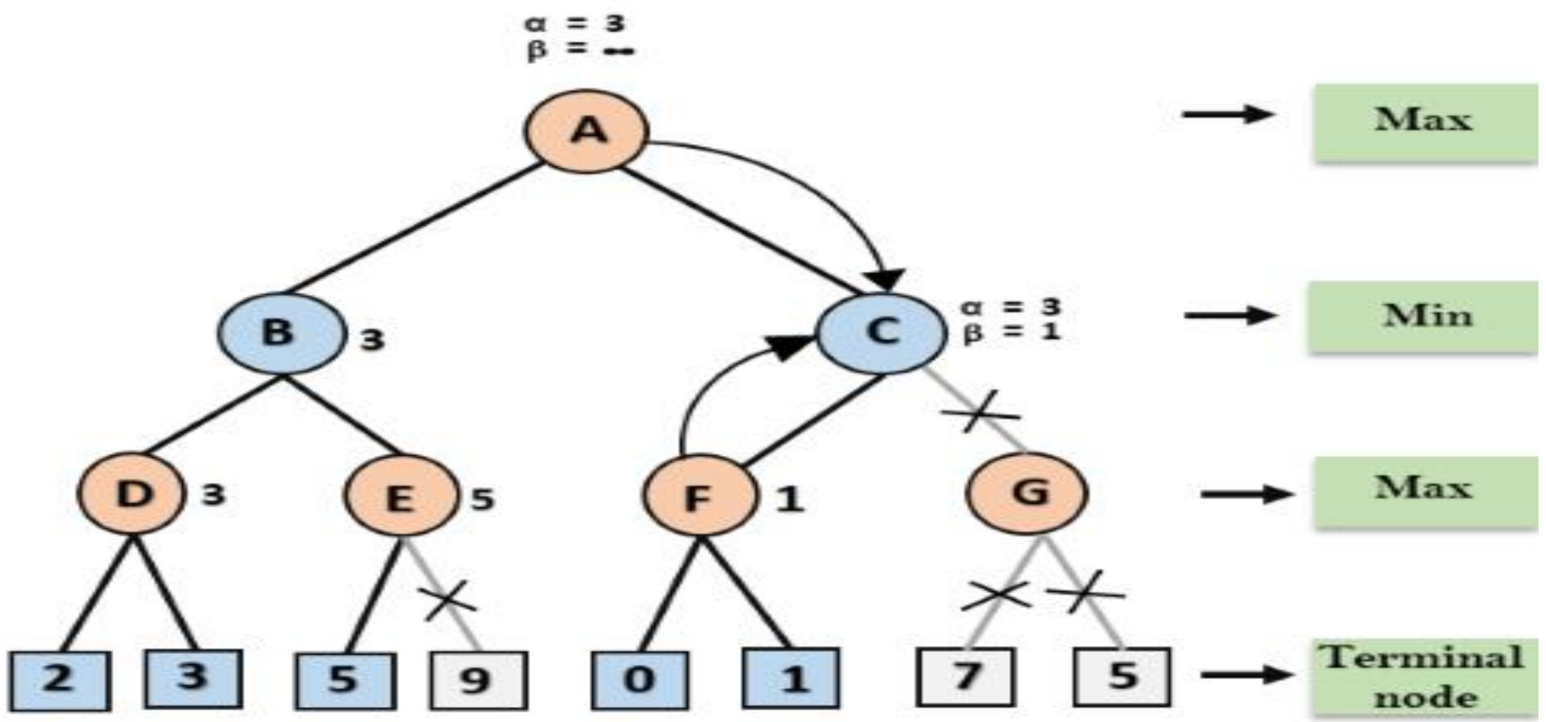


Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
At node C, α=3 and β= +∞, and the same values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

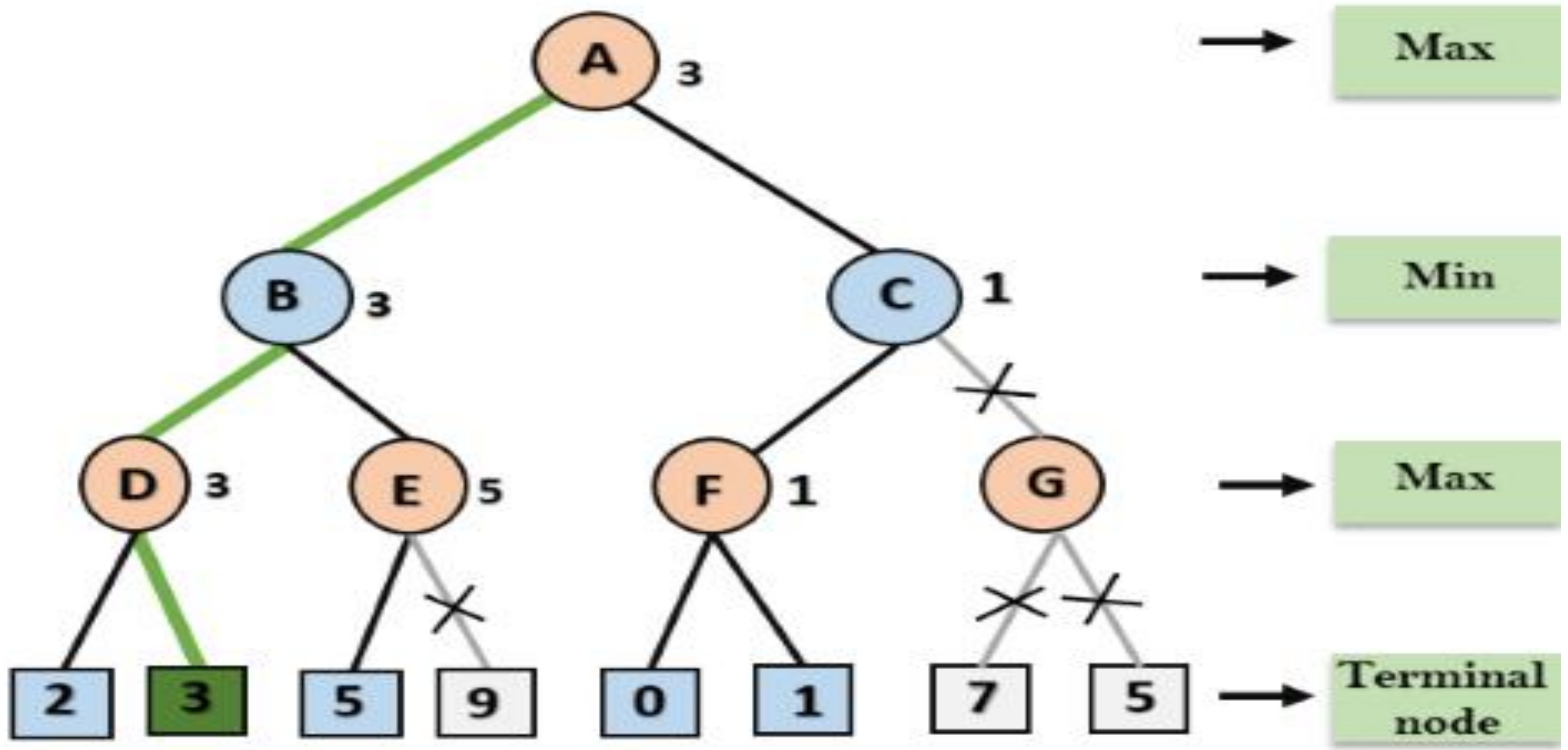
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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.


A 3
B 3
C 1
D 3
E 5
F 1
G
2
3
5
9
0
1
7
5
Max
Min
Max
Terminal node

Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

•Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is O(bm).

•Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is O(bm/2).

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

•Occur the best move from the shallowest node.

•Order the nodes in the tree such that the best nodes are checked first.

•Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.

•We can bookkeep the states, as there is a possibility that states may repeat.

Introduction to Evaluation Function of Minimax Algorithm in Game Theory

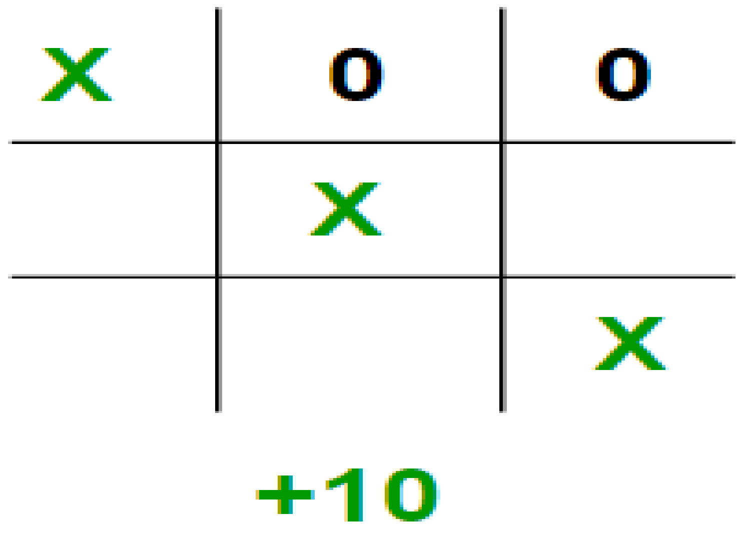
We had stored this value in an array. But in the real world when we are creating a program to play Tic-Tac-Toe, Chess, Backgammon, etc. we need to implement a function that calculates the value of the board depending on the placement of pieces on the board. This function is often known as Evaluation Function. It is sometimes also called a Heuristic Function.

The evaluation function is unique for every type of game. In this post, the evaluation function for the game Tic-Tac-Toe is discussed. The basic idea behind the evaluation function is to give a high value for a board if the maximizer turn or a low value for the board if the minimizer turn.

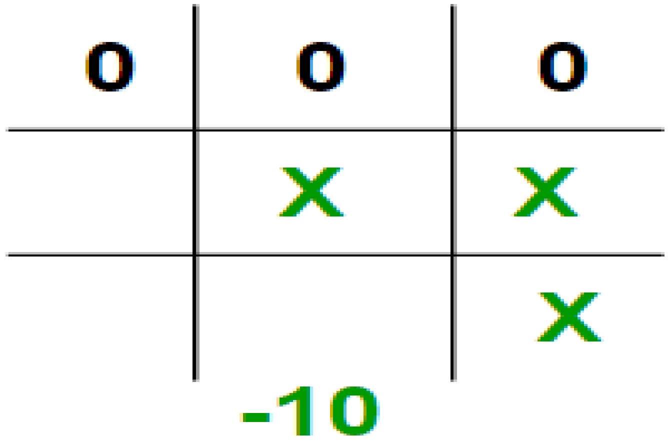
For this scenario let us consider X as the maximizer and O as the minimizer.

Let us build our evaluation function :

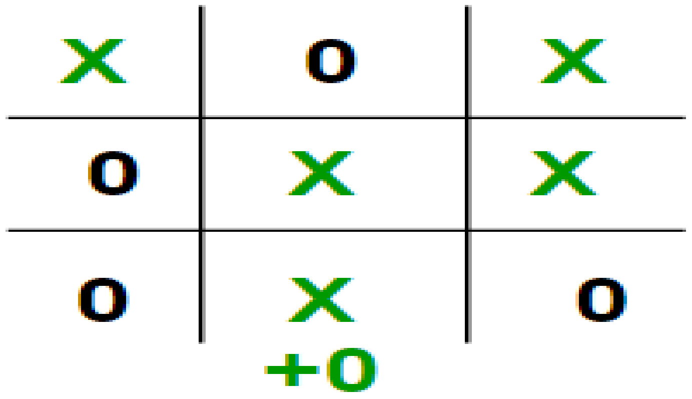
- If X wins on the board we give it a positive value of +10.



If O wins on the board we give it a negative value of -10.



If no one has won or the game results in a draw then we give a value of +0.



We could have chosen any positive/negative. For the sake of simplicity, we chose 10 and shall use lowercase ‘x’ and lowercase ‘o’ to represent the players and an underscore ‘_’ to represent a blank space on the board.
If we represent our board as a 3×3 2D character matrix, like char board[3][3]; then we have to check each row, each column, and the diagonals to check if either of the players has gotten 3 in a row.

Time Complexity: O(max(row,col))

Auxiliary Space: O(1)

Cut-off search

By using a minimax search, all we have to do is program, in a game playing situation our agent to look at the whole search tree from the current state of the game, and select the minimax solution before making a move. Unluckily, only in very trivial games like the one above is it possible to calculate the minimax answer all the way from the end states in a game. So, for games of higher complexity, we are forced to estimate the minimax option for world states using an evaluation function. Of course, this is a heuristic function .

In a normal minimax search, we write down the whole search space and then propogate the scores from the goal states to the top of the tree so that we can choose the best move for a player. In a cut off search, however, we write down the entire search space up to a specific depth and then note down the evaluation function for each of the states at the bottom of the tree. We then propagate   these values from the bottom to top the same way in exactly ,as minimax.

In advance, the depth is decided to ensure that the agent does not take too long to select a move.if it has longer, then we permit it to go deeper.

If our agent has a given time restriction for each move, then it makes sense to enable it to continue searching until the time runs out. There are several ways to do the search in such a way that a game playing agent searches as much as possible in the time available.

For an exercise, what possible ways can you find out of to perform this search? It is essential to bear in mind that the point of the search is not to find a node in the above graph but it is to determine which move the agent should make.