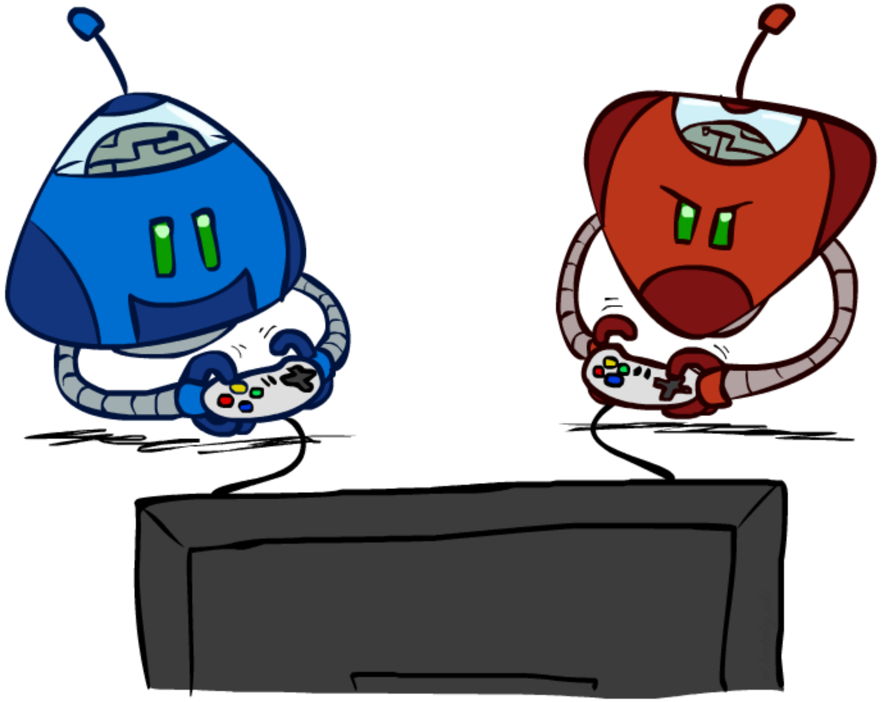
Adversarial Search and (Games) (Chapter 5)

Dr. Nasima Begum
Assistant Professor
Dept. of CSE, UAP

Adversarial Search



[Slides adapted from Dan Klein and Pieter Abbeel (U of California, Berkeley)]

- In this search, we examine the problem that arises when we try to plan ahead of the world and other agents are planning against us.

Adversarial Search (Game)

- ❑ **Multi-Agent Environment:** The environment with **more than one agent**, 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.

- Cooperative vs. competitive
  - Competitive environment is where the agents' goals are in conflict

❑ Adversarial Search Problems

Search problems 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.

❑ Game Theory

- Study of mathematical models of strategic interaction among rational decision-makers. It has applications in all fields of social science, as well as in logic, systems science and computer science.
- A branch of economics
- Views the impact of agents on others as significant rather than competitive (or cooperative).

General Games

- ❑ What makes something a game?

- There are two (or more) agents making changes to the **world** (the state).
- Each agent has their own interests and goals.
- Each agent assigns **different costs** to different paths/states.
- Each agent independently tries to alter the world so as to best benefit itself.
- **Co-operation** can occur but only if it benefits both parties.

❑ What makes games hard?

- How you should play depends on how you **think** the other person will play;
- How the other person plays depends on how they **think** you will play.

Hence, a joint-dependency.

Properties of Games

- Game Theorists
  - Deterministic, turn-taking, two-player, zero-sum games of perfect information

- In AI:
  - Deterministic
  - Fully-observable
  - Two agents whose actions must alternate
  - Utility values at the end of the game are equal and opposite
    - In chess, one player wins (+1), one player loses (-1)
    - It is this opposition between the agents' utility functions that makes the situation adversarial

Types of Games

Perfect information: A game in which **agents can look into the complete board**. Agents have **all** the **information** about the game, and they can see each other moves also. Examples: Chess, Checkers, Go, etc.

Imperfect information: A game where **agents do not have all information** about the game and not aware with what's going on, such type of games are called the game with imperfect information. Examples: Tic-tac-toe, Battleship, Blind, Bridge, etc.

	Deterministic	Nondeterministic
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

Types of Games

Deterministic games: Games which follow **a strict pattern** and **set of rules** for the games, and there is **no randomness** associated with them. Examples: Chess, Checkers, Go, Tic-tac-toe, etc.

Non-deterministic games: Games which have various **unpredictable events** and has **a factor of chance or luck**. This factor of chance or luck is introduced by either dice or cards. These are **random**, and each action **response is not fixed**. Such games are also called as **stochastic** games. Example: Backgammon, Monopoly, Poker, etc.

	Deterministic	Nondeterministic
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

Games as Search Problems

- Games have a state space search:
  - Each potential board or game position is a state
  - Each possible move is an operation to another state
  - The state space can be HUGE!!!!!!!
    - **Large branching factor (b)** (about **35 for chess**)
    - **Terminal state** could be deep (about **50 for chess**)
    - So, $b^m = 35^{50+50}$ ....its huge

Games vs. Search Problems

- Unpredictable opponent
- Solution is a strategy
  - Specifying a move for every possible opponent reply
- Time limits
  - Unlikely to find the goal…agent must approximate

Example Computer Games

- Chess – Deep Blue (World Champion 1997)
- Checkers – Chinook (World Champion 1994)
- Othello – Logistello
  - Beginning, middle, and ending strategy
  - Generally accepted that humans are no match for computers at Othello
- Backgammon – TD-Gammon (Top Three)
- Go – Goemate and Go4++ (Weak Amateur)
- Bridge (Bridge Barron 1997, GIB 2000)
  - Imperfect information
  - multiplayer with two teams of two

Optimal Decisions in Games

- Consider games with **two players** (MAX, MIN)
- Initial State
  - **Board position** and identifies the **player** to move
- Successor Function
  - Returns a list of (move, state) pairs; each a legal move and resulting state
- Terminal Test
  - Determines if the game is over (at terminal states)
- Utility Function
  - Objective function, payoff function, a numeric value for the terminal states (+1, -1) or (+192, -192)

AI and Games

- In AI, “games” have special format:
  - deterministic, turn-taking, 2- player, zero-sum games of perfect information
  - Zero-sum describes a situation in which a participant’s gain or loss is exactly balanced by the losses or gains of the other participant(s)
  - Or, the total payoff to all players is the same for every instance of the game (constant sum)
  - In game theory and economic theory, a **zero-sum game** is a mathematical representation of a situation in which **each participant's gain or loss** of utility is exactly balanced by **the losses or gains** of the utility of the other participants.
  - If the total gains of the participants are added up and the total losses are subtracted, they will sum to zero.

Go! 围棋

Zero Sum Game: Example

Example: ***Red chooses action 2 and Blue chooses action B****. When the payoff is allocated, Red gains 20 points and Blue loses 20 points.*

In this example game, both players know the payoff matrix and attempt to maximize the number of their points. Red could reason as follows: "With **action 2**, it could **lose up to 20 points** and can **win only 20**, and with **action 1**, it can **lose only 10** but can **win up to 30**, so action 1 looks a lot better."

Blue Red	*A*	*B*	*C*
1	−30 30	10 −10	−20 20
2	10 −10	−20 20	20 −20

A zero-sum game

With similar reasoning, Blue would choose action C. If both players take these actions, Red will win 20 points. If Blue anticipates Red's reasoning and choice of action 1, Blue may choose action B, so as to win 10 points. If Red, in turn, anticipates this trick and goes for action 2, this wins Red 20 points.

Game 1: Rock, Paper, Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a **matrix (known as payoff matrix)**:

Player I chooses a row, Player II chooses a column.

- 1: win

0: tie
-1: loss

	R	P	S
R	0/0	-1/1	1/-1
P	1/-1	0/0	-1/1
S	-1/1	1/-1	0/0

Game 2: Prisoner’s Dillema

- Two prisoners in separate cells.

The sheriff doesn’t have enough evidence to convict them.
They agree ahead of time to both **deny** the crime (they will **cooperate**).

- If one confesses and the other doesn’t:

– Confessor goes free;
– Other sentenced to 4 years.

- If both confess:

– both sentenced to 3 years.

- If both cooperate (neither confesses):

– both sentenced to 1 year.

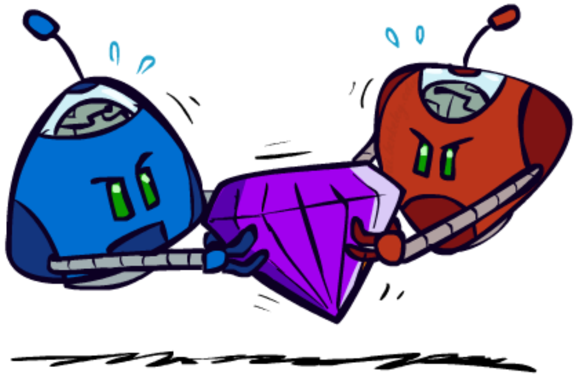
Payoff (possible outcomes)**:** 4

	Coop	Confess
Coop	1/1	4/0
Confess	0/4	3/3

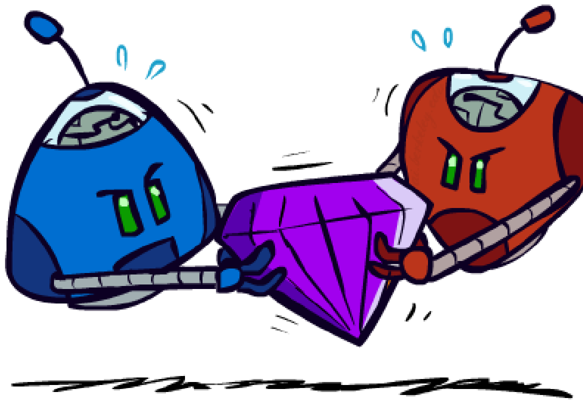
Zero-Sum Games

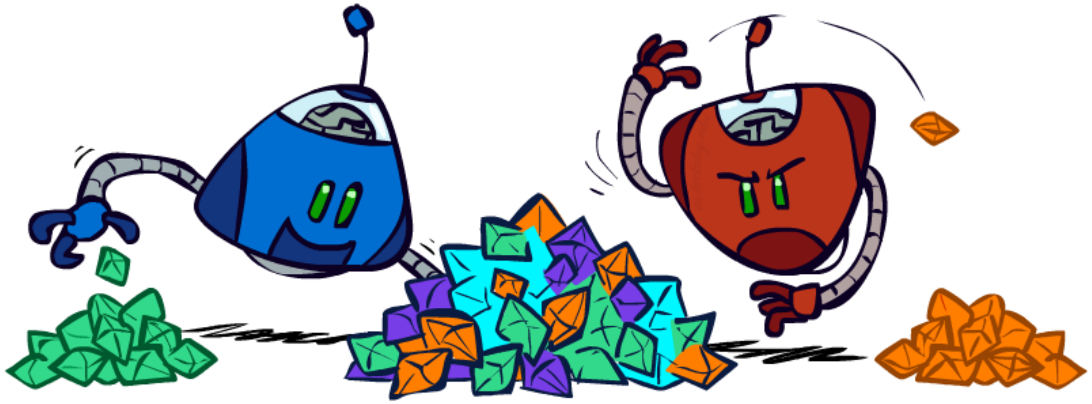
- Zero-sum games are **adversarial search** which involves **pure competition**.
- In Zero-sum game **each agent's gain or loss** of utility is **exactly balanced** by the losses or gains of utility of another agent.
- One player of the game try to **maximize** one single value, while other player tries to **minimize** it.
- Fully competitive, **total payoff** to all players is constant.
- If one player gets a higher payoff, the other player gets a lower payoff.
- Each move by one player in the game is called as **ply**.
- Examples: Chess, tic-tac-toe etc.

Poker – one win what the other player lose



Zero-Sum Games





❑Zero-Sum Games

- Agents have opposite utilities (values on outcomes)
- Lets us think of **a single value** that one maximizes and the other minimizes
- Adversarial, pure competition

❑General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- A non zero sum game is a situation where there is a net benefit or net loss to the system based on the outcome of the game.

Two Player Zero-Sum Games

A **Two-Player Zero-Sum** game consists of the following components:

- Two **players** Max and Min.
- A set of **positions** P (states of the game).
- A **starting position** $p \in P$ (where game begins).
- A set of **Terminal positions** $T \subseteq P$ (where game can end).
- A set of directed **edges** E_{Max} between some positions, representing Max's moves.
- A set of directed **edges** E_{Min} between some positions, representing Min's moves.
- A **utility (or payoff) function** $U : T \to \mathbb{R}$, representing how good each **terminal state** is for player **Max**.

Formalization of the problem (Deterministic Games):

➢ Many possible formalizations, one is:

- **Initial state:** It specifies how the game is set up at the start.
  - States: S (start at s_0)

- **Player(s):** It specifies which player has moved in the state space.

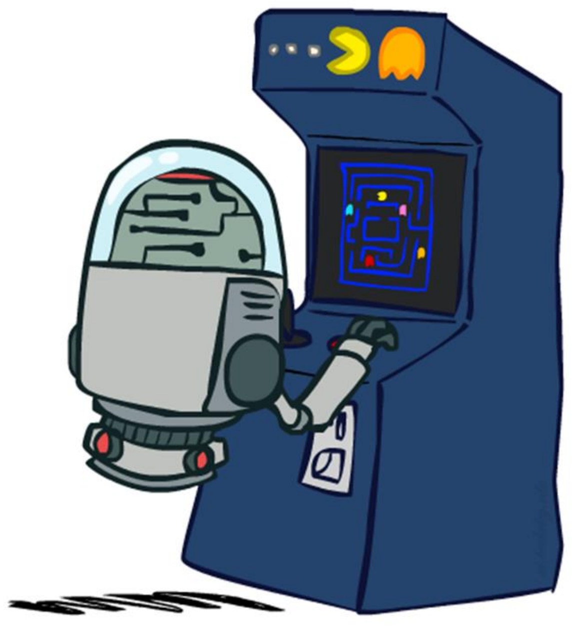
  Players: P={1...N} (usually take turns)

- **Action(s):** It returns the set of legal moves in state space.

  A (may depend on player / state)

- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.

  $SxA \rightarrow S$



Formalization of the problem (Deterministic Games):

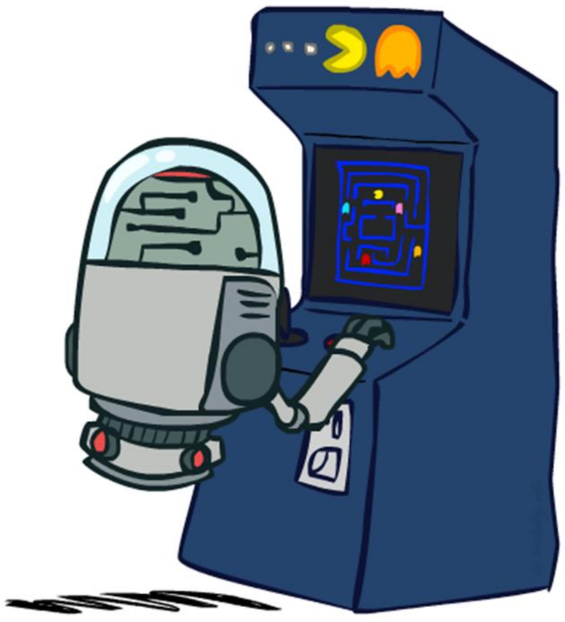
- **Terminal-Test(s):** The state where the game ends is called terminal states. Terminal test is true if the game is over, else it is false at any case.

  $S \rightarrow \{t,f\}$

- **Terminal Utilities(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.

  $SxP \rightarrow R$

  - Solution for a player is a policy: $S \rightarrow A$



Game Trees

- A tree where nodes of the tree are the game states and edges of the tree are the moves by players.
- Game tree involves ***initial state***, ***actions function***, and ***transition/result function***.

❑ **Example: Tic-Tac-Toe game tree:**

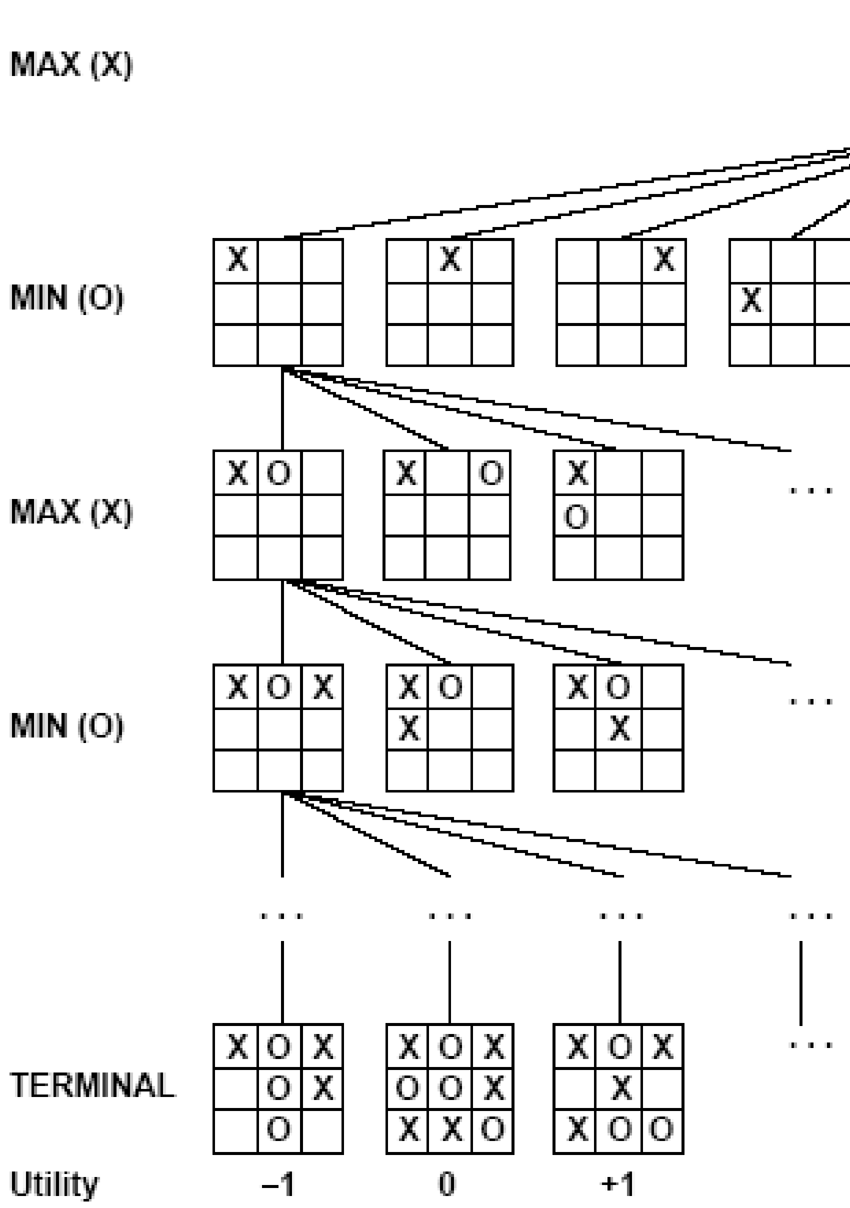
➢ Some key points of the game:

- 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.

Game Trees

- The **root** of the tree is the **initial state**
  - Next level is all of MAX’s moves
  - Next level is all of MIN’s moves
  - …
- Example: Tic-Tac-Toe
  - Root has 9 blank squares (MAX)
  - Level 1 has 8 blank squares (MIN)
  - Level 2 has 7 blank squares (MAX)
  - …
- Utility function:
  - win for X is +1
  - win for O is -1

Game Tree (2-player, deterministic)



- 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.

Game Tree Explanation

- 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.

Minimax Strategy

- Adversarial Search for the minimax procedure :

- It aims to find the optimal strategy for MAX to win the game.
- It follows the approach of *Depth-first search*.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.
- 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.

Minimax Strategy

- Basic Idea:
  - Choose the **move** with the **highest minimax value**
    - best achievable payoff against best play
  - Choose **move**s that will **lead to a win**, even though min is trying to block

- Max's goal: get to 1 (Max wants to maximize the terminal payoff.)

- Min's goal: get to -1 (Min wants to minimize the terminal payoff.)

- Minimax value of a node (backed up value):
  - If node N is terminal, use the utility value
  - If node N is a Max move, take max of successors
  - If node N is a Min move, take min of successors

Minimax Strategy

- We can compute a **utility** (**MinMAx value**) for the **non-terminal** states by assuming both players always play their best move.

  - Back the **utility values** up the **tree**:

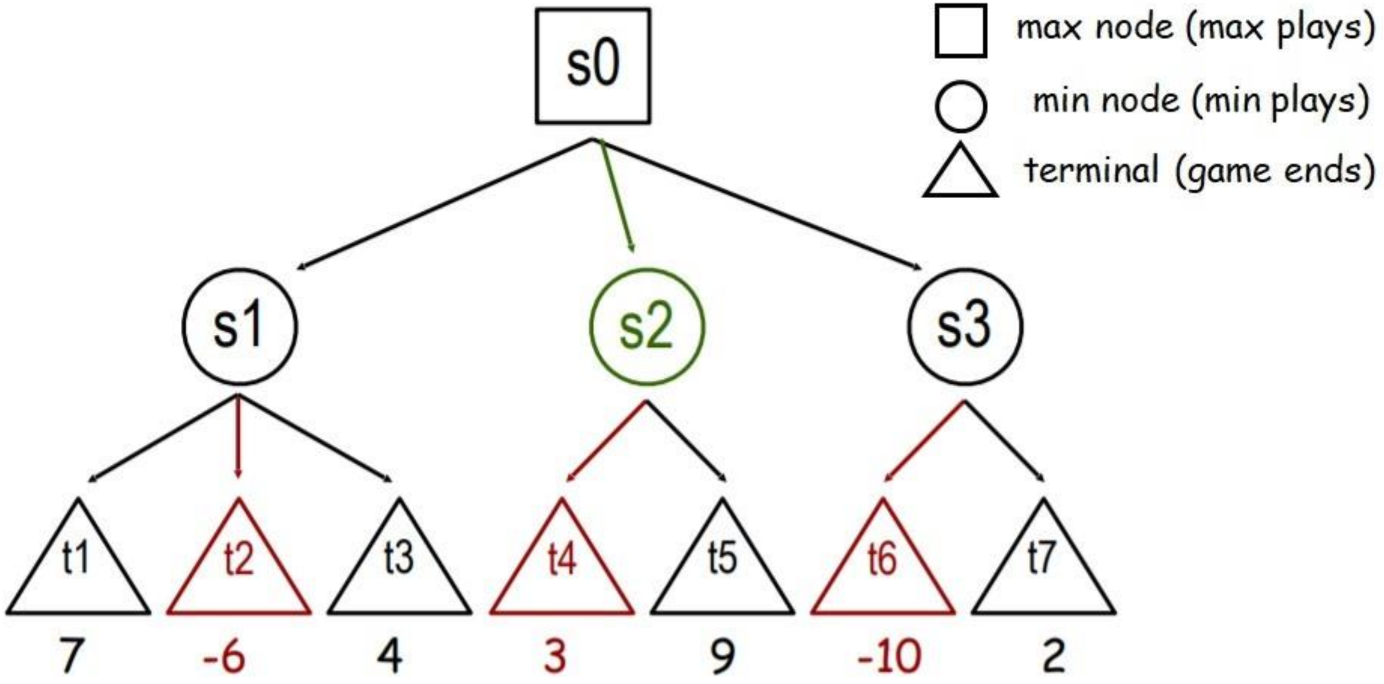
$$U(s) = \begin{cases} U(s) & \text{if } s \text{ is a terminal } (U \text{ is defined} \\ & (U \text{ is defined for all terminals} \\ & \text{as part of input)} \\ min\{U(c) : c \text{ is a child of } s\} & \text{if } s \text{ is a Min node.} \\ max\{U(c) : c \text{ is a child of } s\} & \text{if } s \text{ is a Max node.} \end{cases}$$

- The values *U*(*s*) labeling each state ***s*** are the values that Max will achieve in that state if both **Max** and **Min** play their best move

Minimax Algorithm

- A recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally (best).
- It uses recursion to search through the game-tree.
- Mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- Two players play the game, one is called MAX and other is called MIN. Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- It performs a depth-first search algorithm for the exploration of the complete game tree
- It proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Minimax Strategy

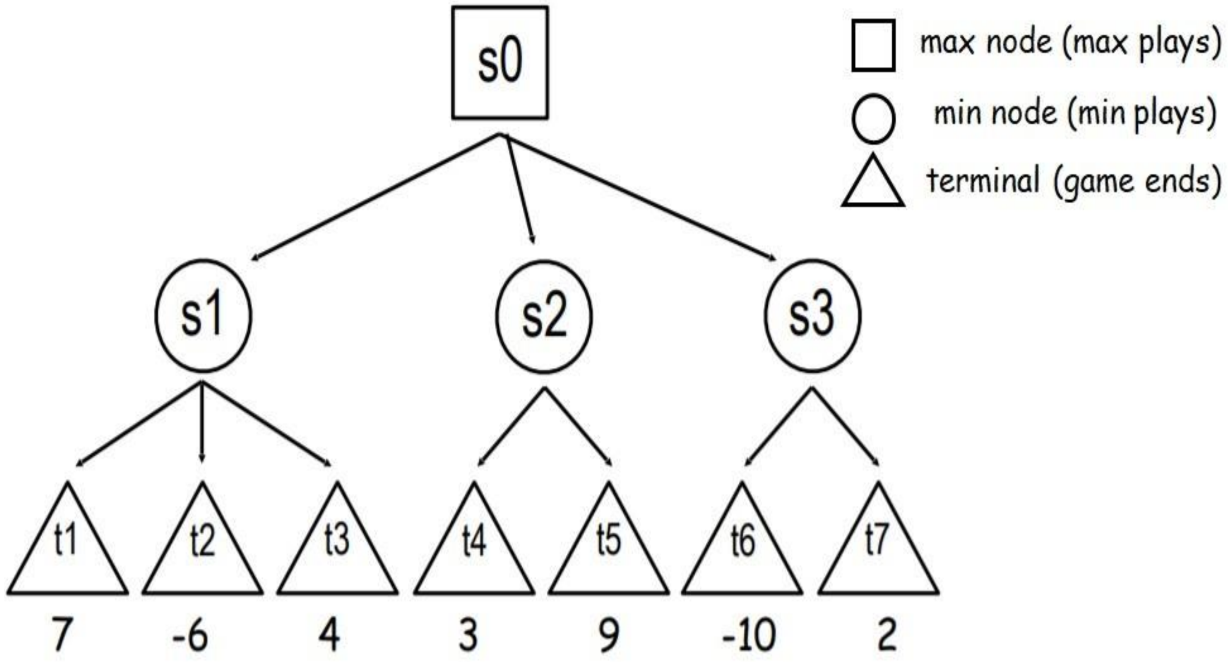


Minimax Strategy:

- Max always plays a move to change the state to the highest valued child.
- Min always plays a move to change the state to the lowest valued child.

If **Min plays poorly** (does not always move to lowest value child), Max could do better, but **never worse**.

Minimax Strategy



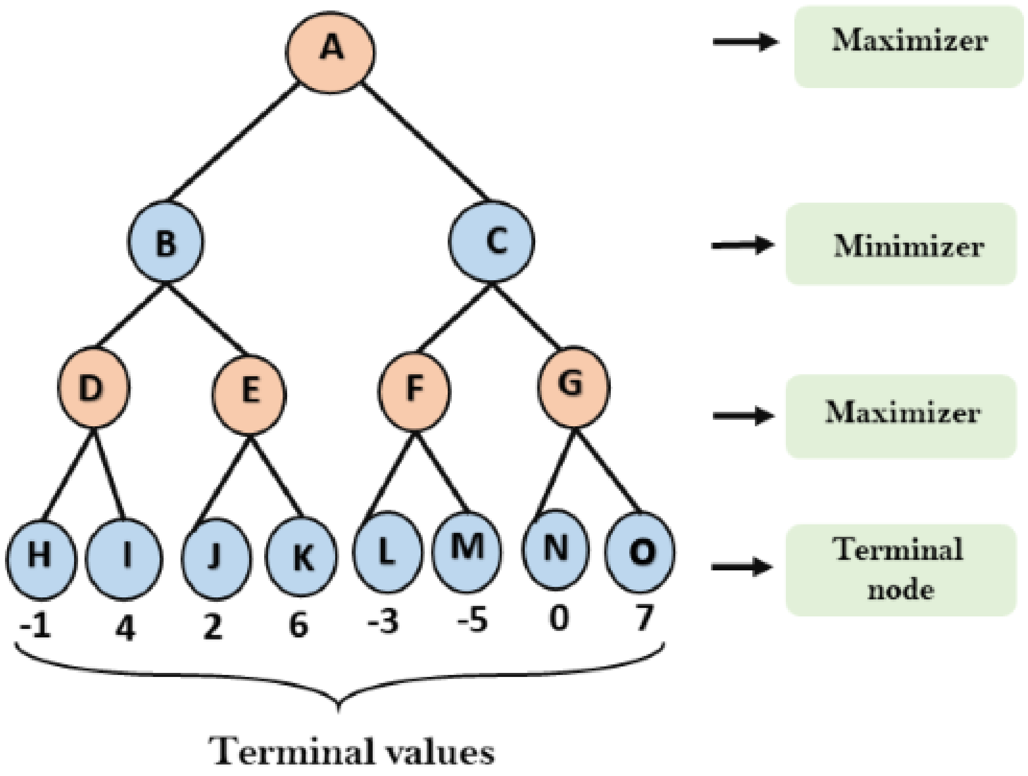
- If MAX goes to s1, MIN goes to t2, MIN {U(t1), U(t2), U(t3)} = MIN {7, -6, 4} = -6 → t2
- If MAX goes to s2, MIN goes to t4, MIN {U(t4), U(t5)} = MIN {3, 9} = 3 → t4
- If MAX goes to s3, MIN goes to t6, MIN {U(t6), U(t7)} = MIN {-10, 2} = -10 → t6
- MAX: MAX {U(s1), U(s2), U(s3)} = MAX {-6, 3, -10} = 3 → s0

Depth-First Implementation of Minimax

```
def DFMiniMax(s, Player):
//Return Utility of state s given that Player is MIN or MAX
1.   If s is TERMINAL
2.        Return U(s)    # Return terminal states utility,
                         # specified as part of game
//Apply Player's moves to get successor states.
3.   ChildList = s.Successors(Player)
4.   If Player == MIN
5.        return minimum of DFMiniMax(c, MAX) over c ∈ ChildList
6.   Else # Player is MAX
7.        return maximum of DFMiniMax(c, MIN) over c ∈ ChildList
```

Example of Minimax

Step-1: Generates the entire game-tree and apply the utility function to get the utility values for the terminal states. Let us consider A is the initial state of the tree. Suppose, maximizer has the **worst-case initial value = - ∞** and minimizer has **worst-case initial value = + ∞** .



Example of Minimax

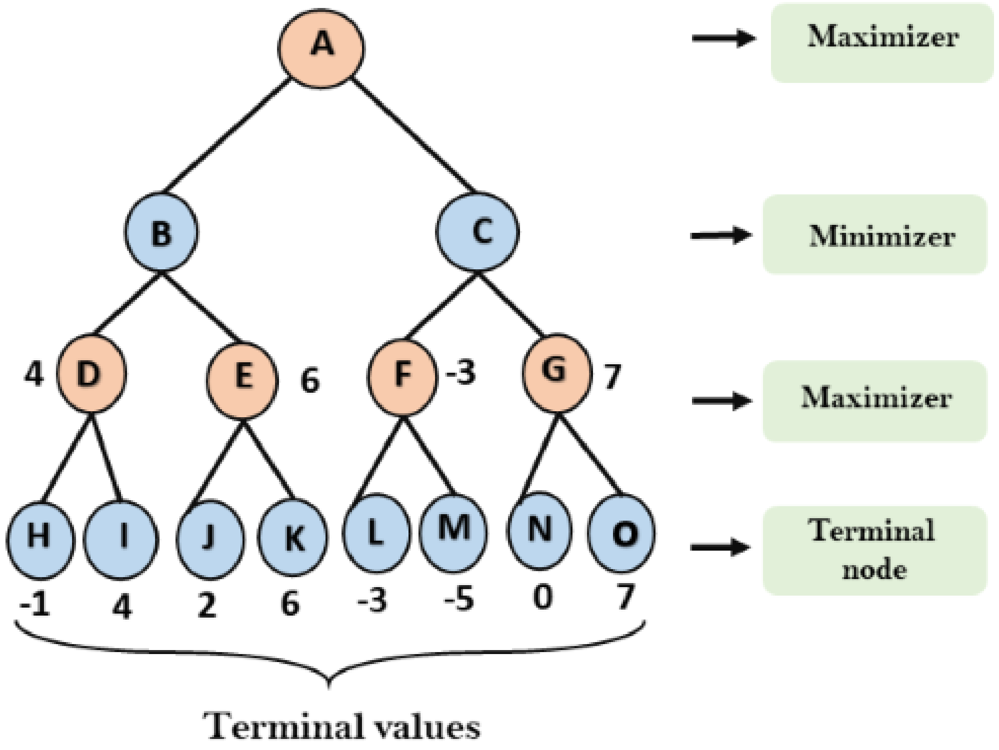
Step-2: Let's compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among all.

For node D max{(-1, -∞), (4, -∞)} => max(-1,4)= 4

For Node E max(2, -∞) => max(2, 6)= 6

For Node F max(-3, -∞) => max(-3,-5) = -3

For node G max(0, -∞) = max(0, 7) = 7

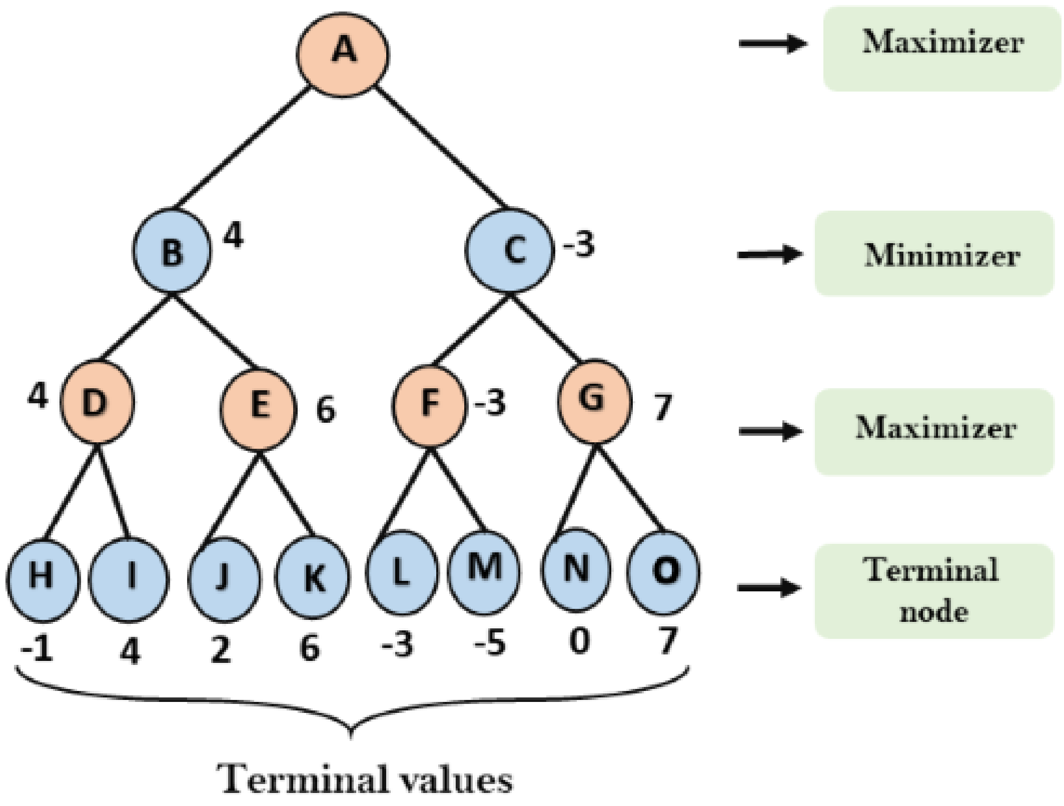


Example of Minimax

Step-3: It's turn for **minimizer** now, so it will compare all nodes value with **+∞**, and will find the 3rd layer node values.

For node B= min(4, ∞) => min(4,6) = 4

For node C= min(-3, ∞) => min (-3, 7) = -3

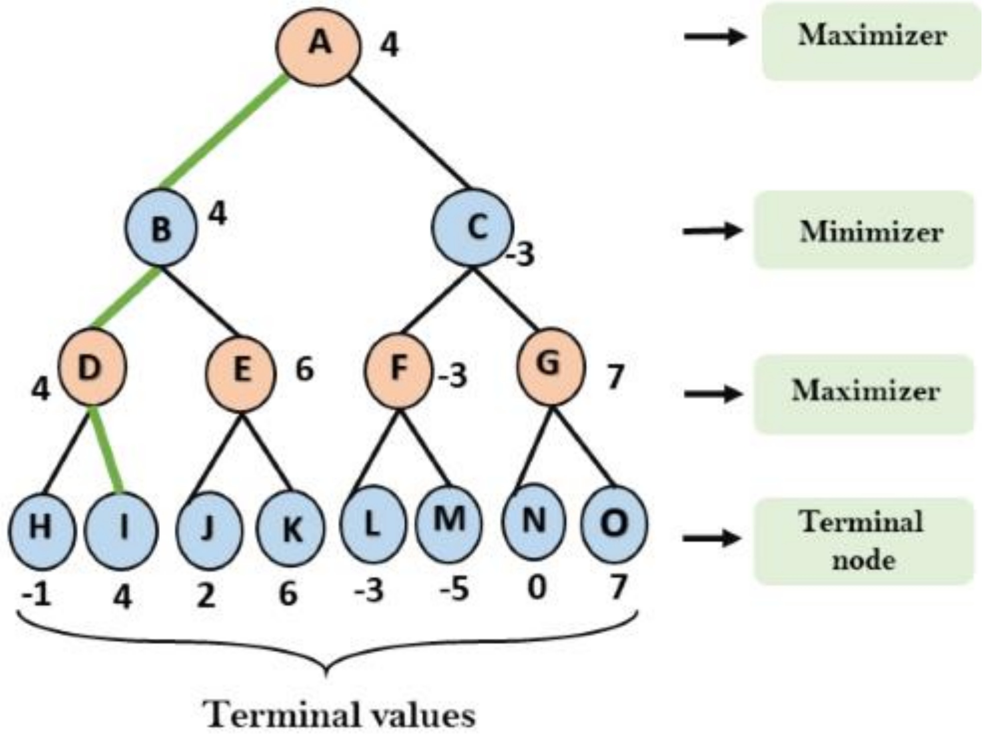


Example of Minimax

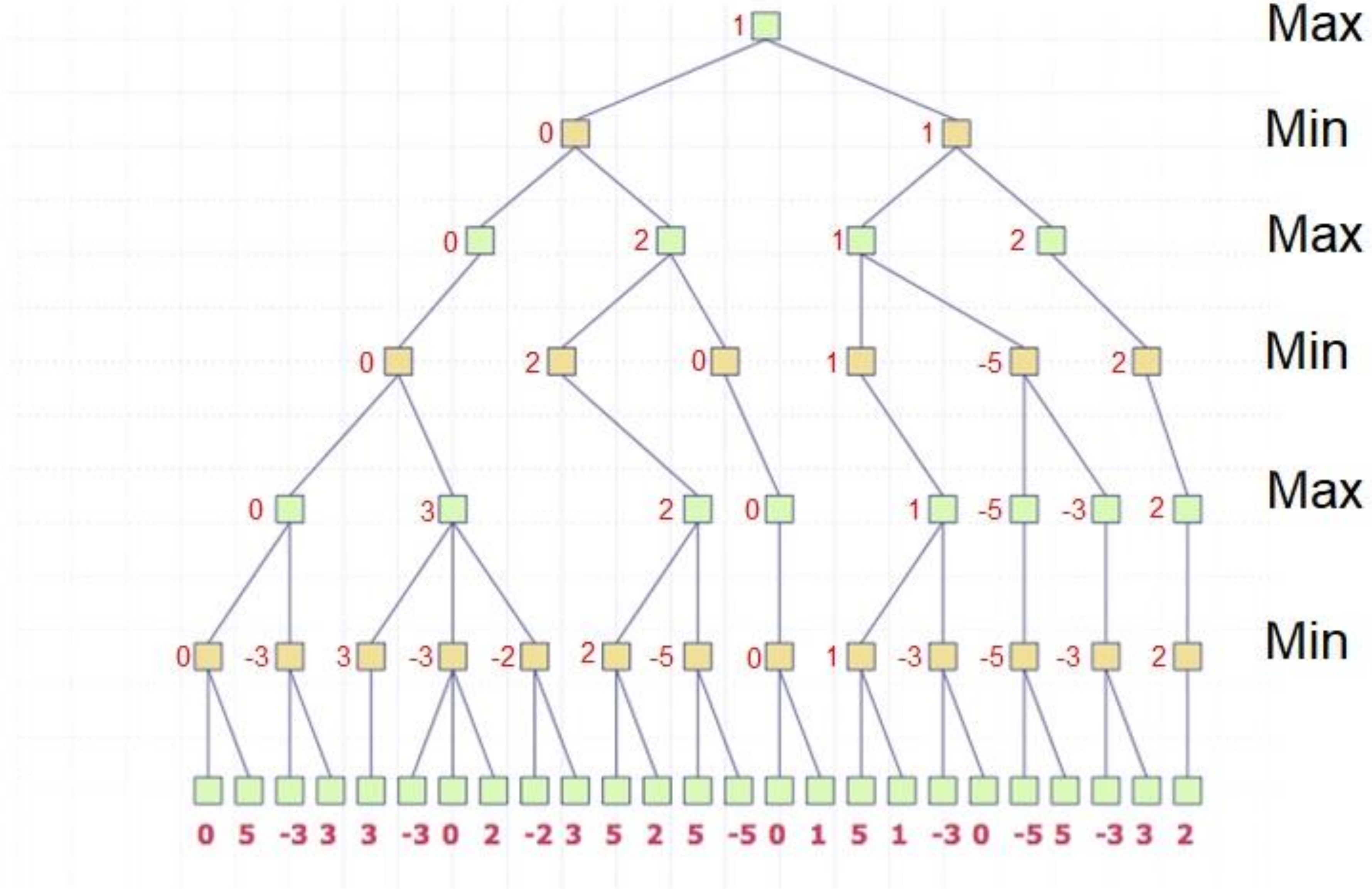
Step-4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node.

In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

For node A = max(4, -3)= 4



Example of Minimax



Properties of Minimax

- Complete
  - Yes, if the tree is finite (e.g. chess has specific rules for this)
- Optimal
  - Yes, if both opponents are playing optimally.
- Time Complexity

  Performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **O(b^m)**, where ***b*** is branching factor of the game-tree, and ***m*** is the maximum depth of the tree.
- Space Complexity
  - Similar to DFS which is **O(bm)** (depth first exploration of the state space)

Problems of Minimax

❑ The main drawback of the minimax algorithm is:

- It gets very slow for complex games such as Chess, go, etc.
- Such games has a huge branching factor, and the player has lots of choices to decide.
- It can be improved by pruning some part of the game space tree from **alpha-beta pruning** algorithm.

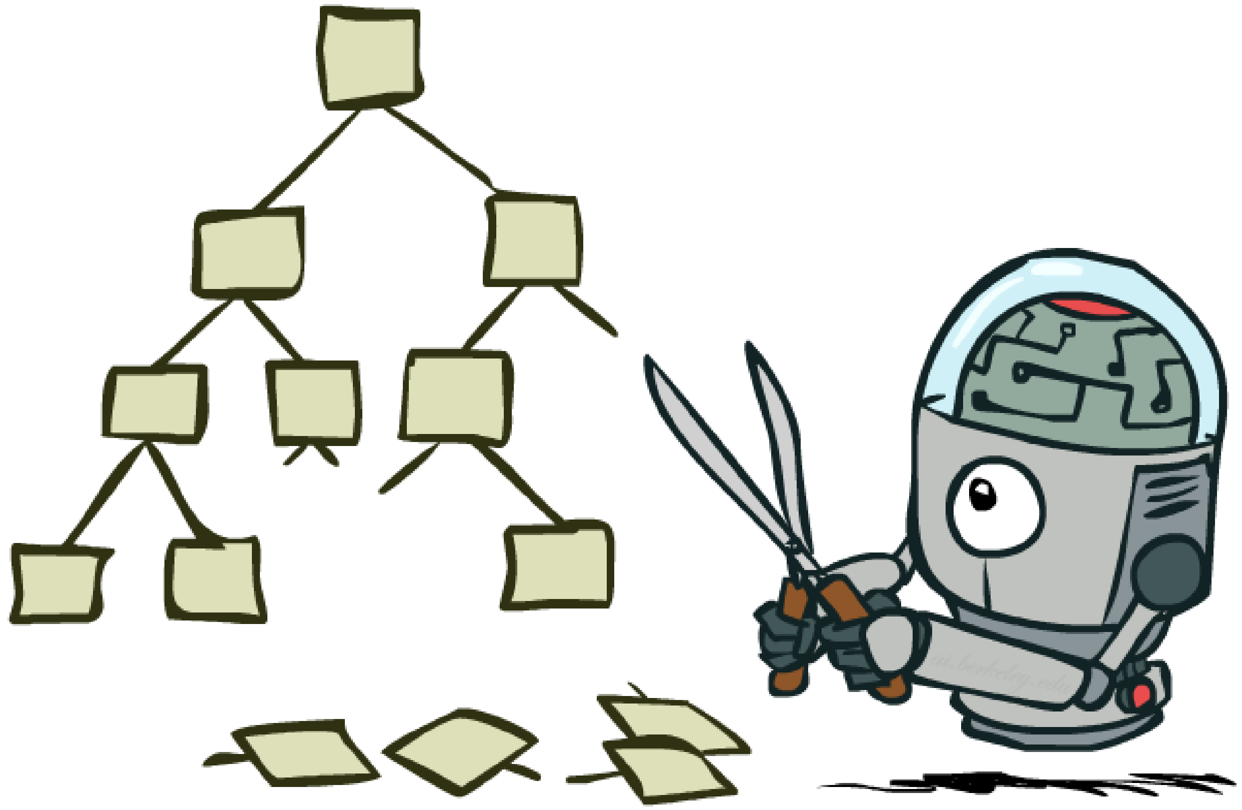
- Building the **entire game tree** and **backing up values** gives each player their **strategy**.
- However, the game tree is **exponential** in size and might be too **large to store** in memory.
- We can save space by computing the minimax values with a **depth-first** implementation of minimax.

Although **run-time** complexity is still **exponential**.

- We run the depth-first search **after each move** to compute what is the next move for the MAX player.
- This avoids explicitly representing the exponentially sized game tree: we just compute

each move as it is needed.

Game Tree Pruning



Alpha-Beta Pruning

- There are ways to **avoid** examining the **entire** tree to make correct Minimax decision.
- When using **depth-first search** of a game tree:

  – After generating value for only **some of *s*'s children** we can prove that we **never reach** *s* in a Minimax strategy.

  – So we need NOT generate or evaluate any further children of *s*.

These other children can be **pruned**.

Alpha-Beta Pruning

- An optimization technique for the minimax algorithm. A modified version of the minimax algorithm.
- 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.
- The 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 technique 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**.
- It 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 (α)**: 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 (β):** 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.

Alpha-Beta Pruning Condition

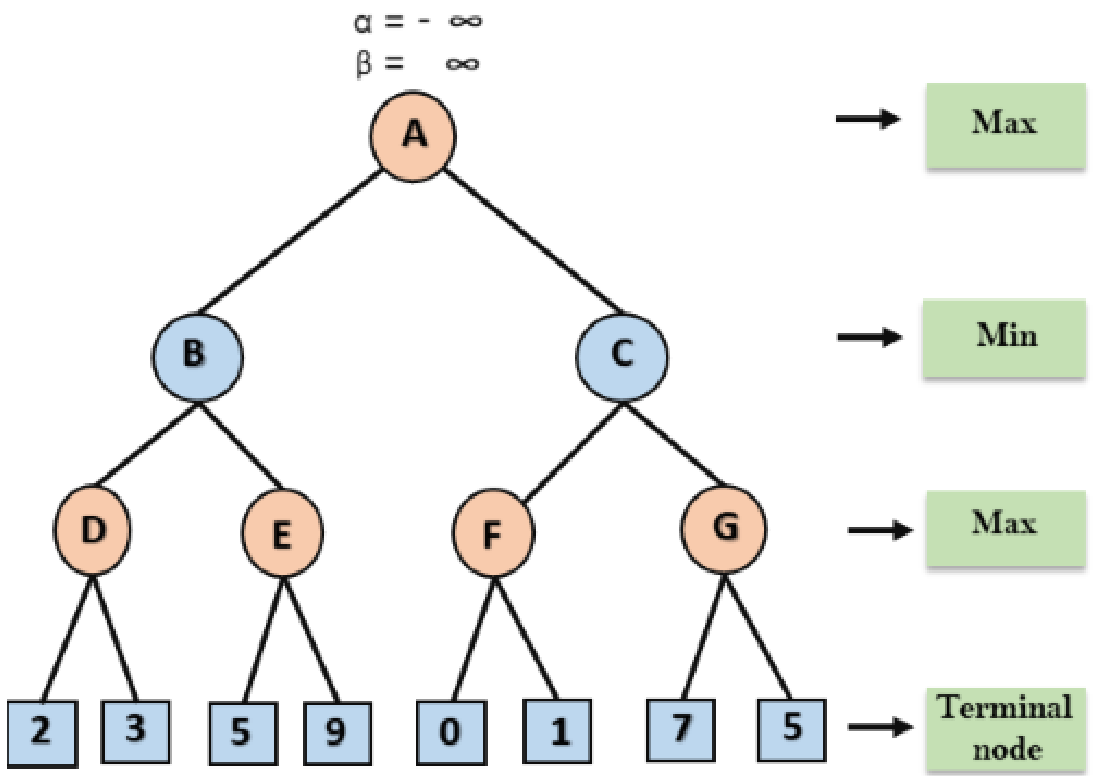
- The main condition: $\boxed{\alpha >= \beta}$

❑ <u>Key points about alpha-beta pruning:</u>

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

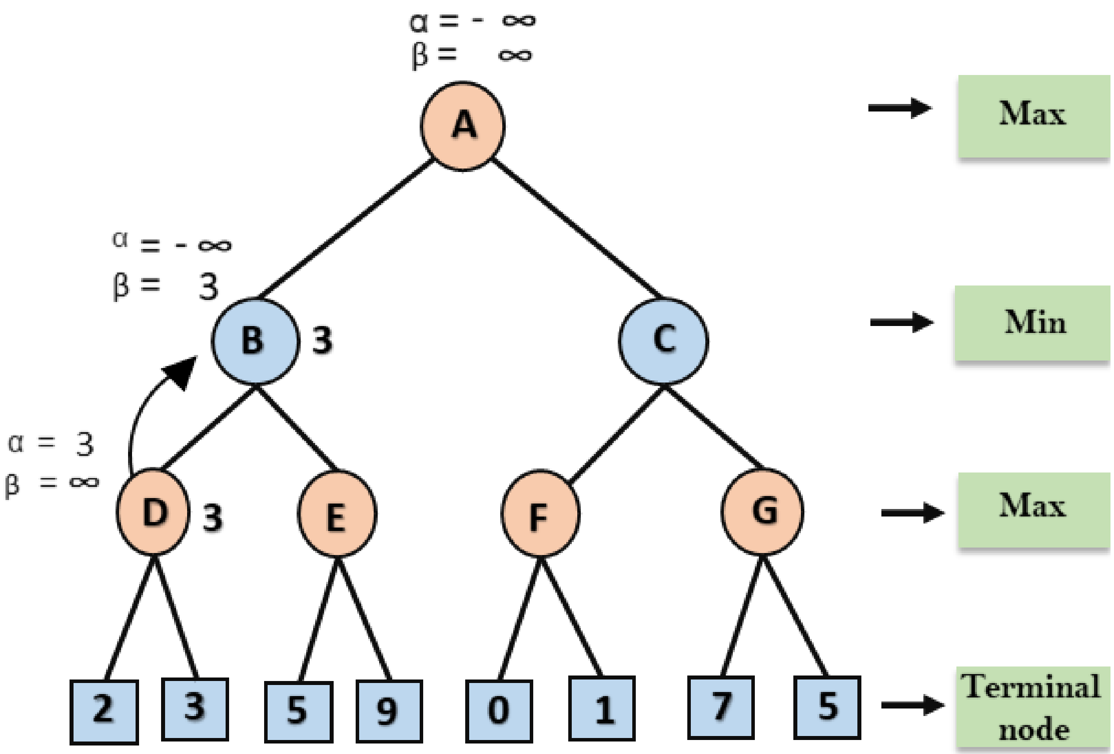
Example of Alpha-Beta Pruning

Step 1: 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.



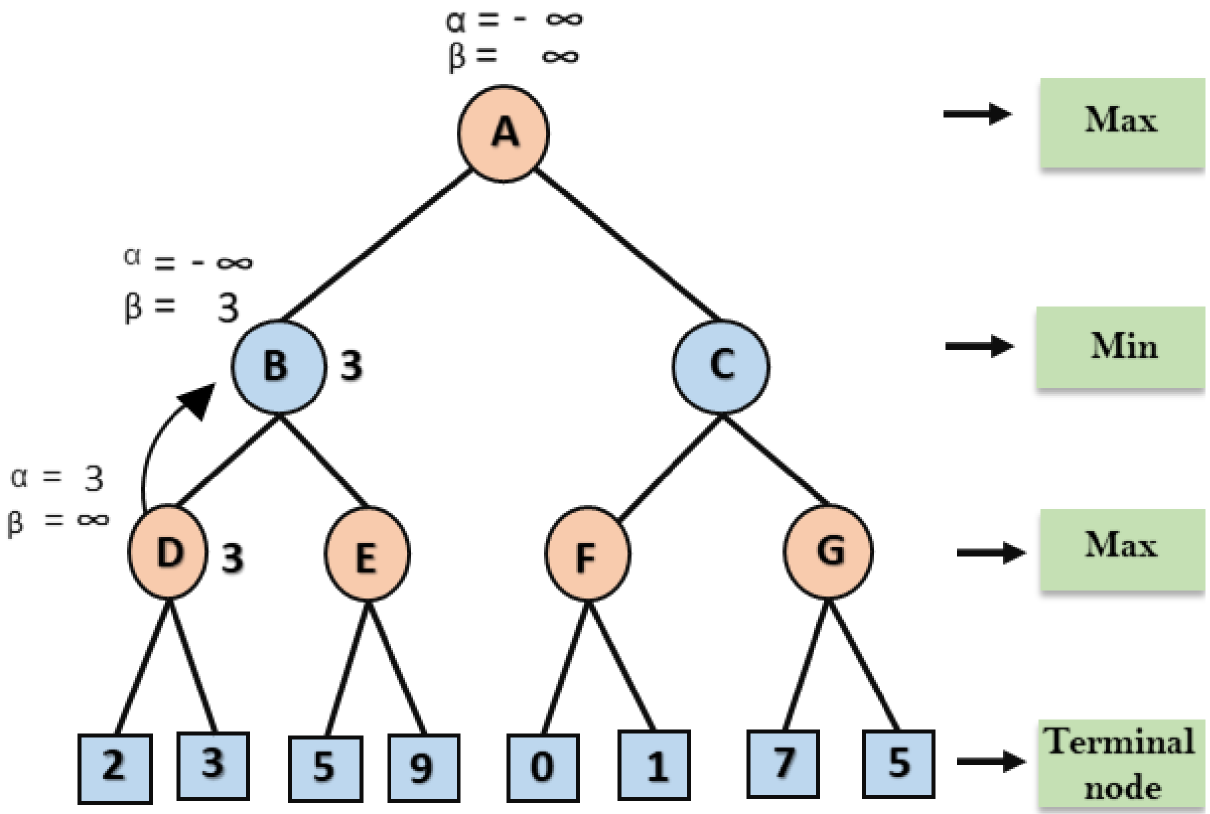
Example of Alpha-Beta Pruning

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



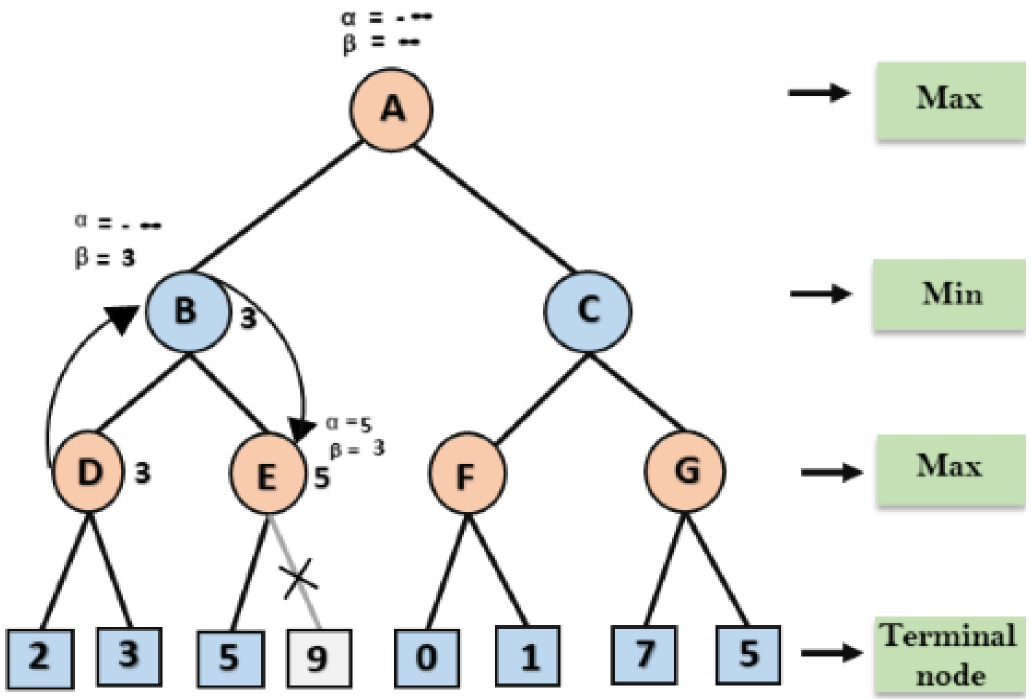
Example of Alpha-Beta Pruning

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



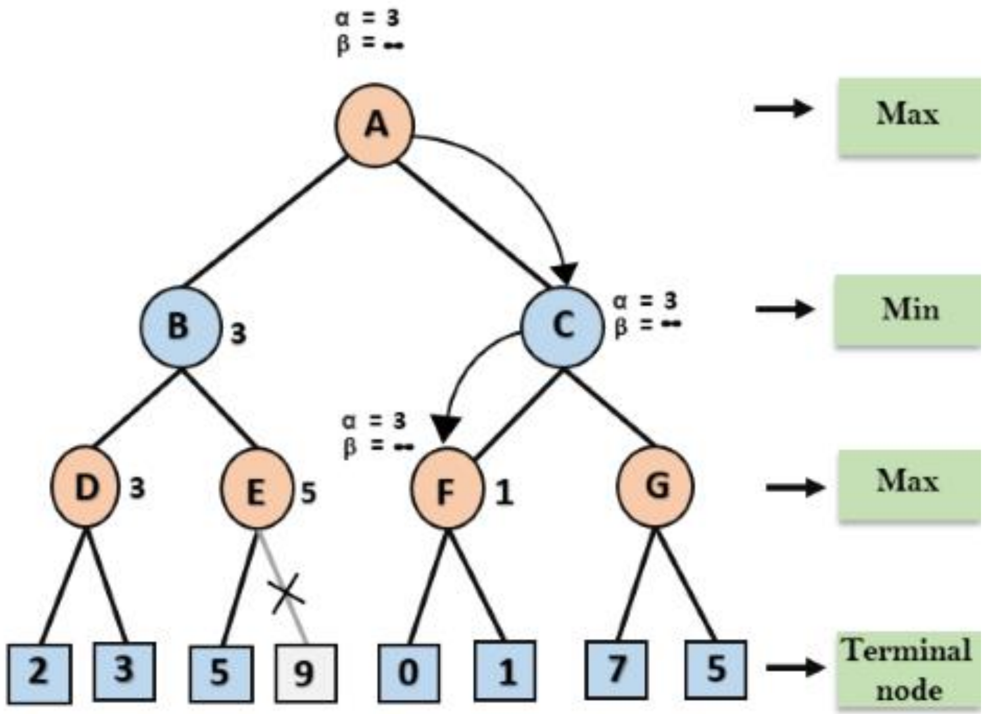
Example of Alpha-Beta Pruning

- Now, the next successor of node B that is node E is traverse and the values of α= -∞, and β= 3 will also be passed to node E.
- **Step 4:** At node E, its Max turn, and the value of alpha will change. The current value of alpha will be compared with left child (5), so max (-∞, 5) = 5, so 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.



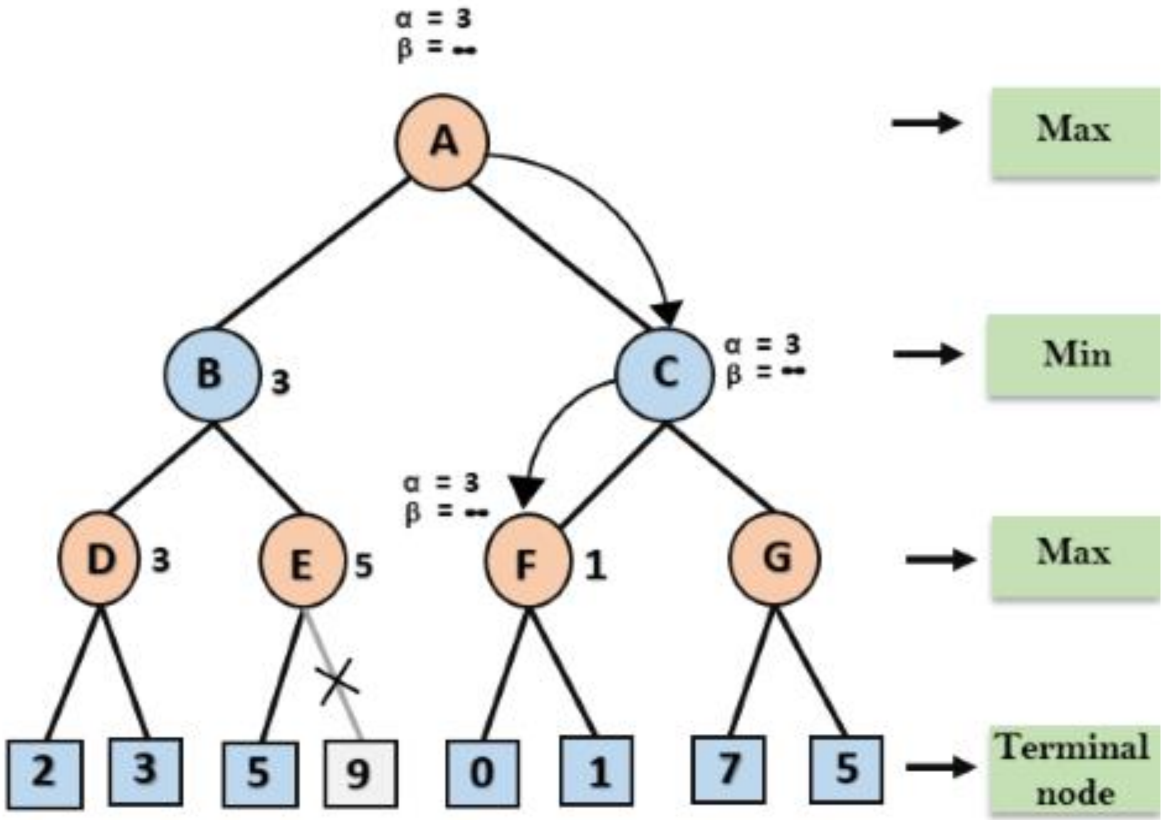
Example of Alpha-Beta Pruning

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



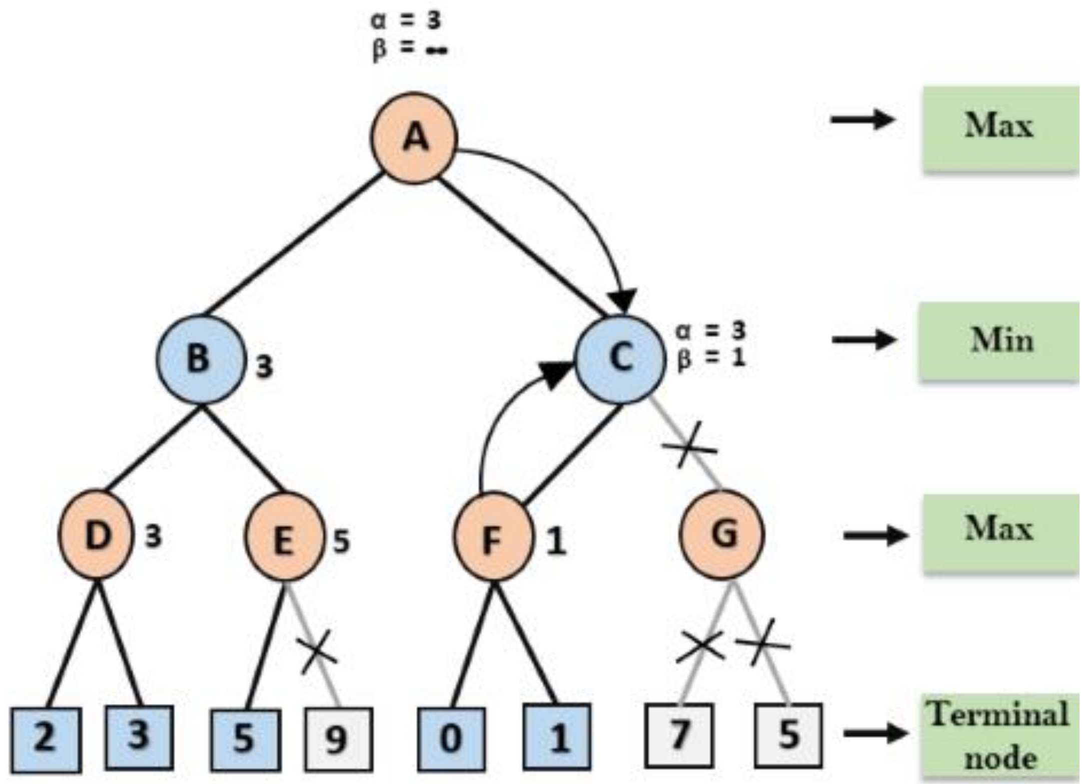
Example of Alpha-Beta Pruning

- **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 will become 1 at node F.



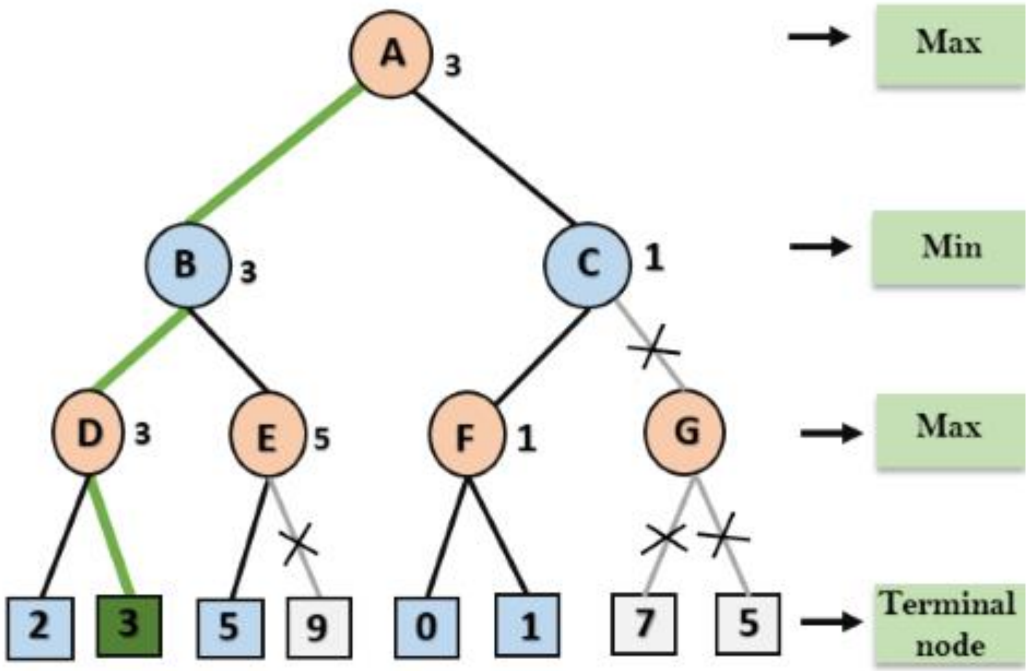
Example of Alpha-Beta Pruning

- **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 right child of C which is G will be pruned. So, the algorithm will not compute the entire sub-tree G.

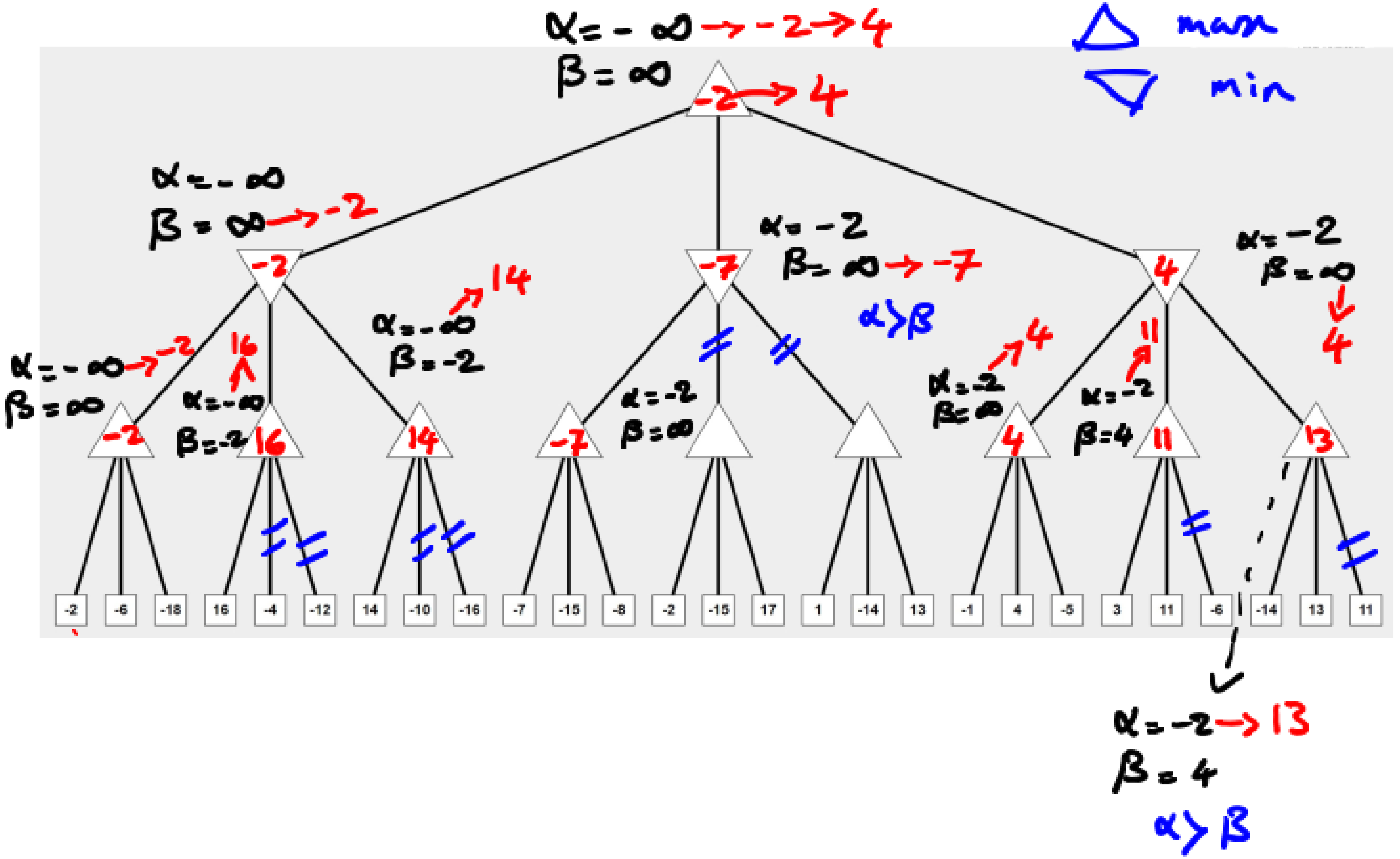


Example of Alpha-Beta Pruning

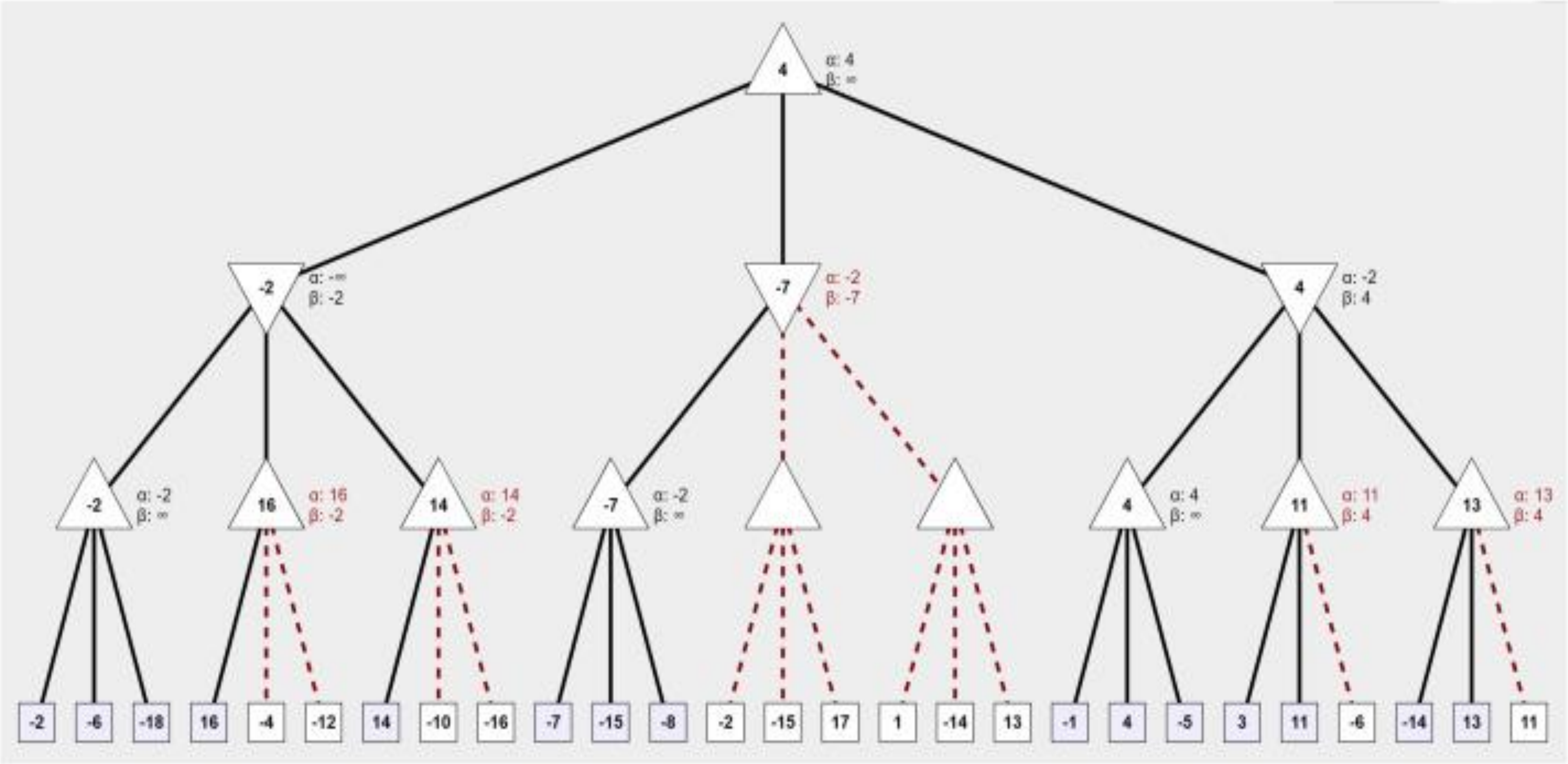
- **Step 8:** Now, C returns the value of 1 to A. Here the best value for A is max (3, 1) = 3.
- The final game tree is showing the nodes which are computed and nodes which has never computed. Thus, the optimal value for the maximizer is 3 for this example.



Alpha-Beta Pruning Example



Alpha-Beta Pruning Example



Alpha-Beta Pruning Implementation

```
def AlphaBeta(s,Player,alpha,beta):
// Return Utility of state s given that Player is MIN or MAX
If s is TERMINAL
    Return U(s)   # Return terminal states utility
ChildList = s.Successors(Player)
If Player == MAX
    ut_val = -infinity
    for c in ChildList
        ut_val = max(ut_val, AlphaBeta(c,MIN,alpha,beta))
        If alpha < ut_val
           alpha = ut_val
           If beta <= alpha: break
    return ut_val
 Else # Player is MIN
    ut_val = infinity
    for c in ChildList
        ut_val = min(ut_val, AlphaBeta(c,MAX,alpha,beta))
        If beta > ut_val
           beta = ut_val
           If beta <= alpha: break
    return ut_val
```

Ordering of Move

- For MIN nodes the best pruning occurs if the best move for MIN (child yielding lowest value) is **explored first**.

- For MAX nodes the best pruning occurs if the best move for MAX (child yielding highest value) is **explored first**.

- We don't know which child has highest or lowest value without doing all of the work! But we can use **heuristics** to estimate the value, and then choose the child with highest (lowest) heuristic value.

Effectiveness of Alpha-Beta Pruning

- With no pruning, $\mathcal{O}(b^d)$ nodes are explored, which makes the run time of a search with pruning the same as plain Minimax.

  If, however, the move **ordering** for the search is **optimal** (meaning the best moves are searched first), the number of nodes we need to search using alpha beta pruning is $\mathcal{O}(b^{d/2})$.

- In Deep Blue, they found that **alpha-beta pruning** meant the average branching factor at each node was about **6 instead of 35**.

Acknowledgement

- AIMA = Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norving (3rd edition)
- UC Berkeley (Some slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley)
- U of toronto
- Other online resources

Thank You