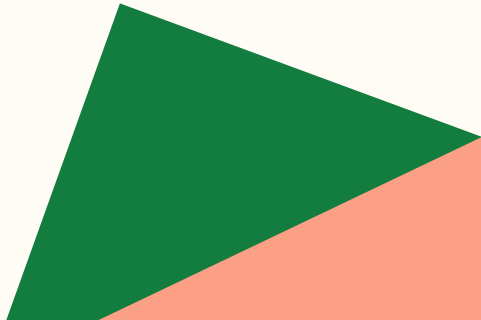




ADVERSARIAL SEARCH AND GAMES

GROUP A5 - FALCONS

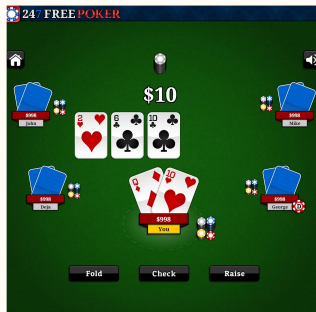
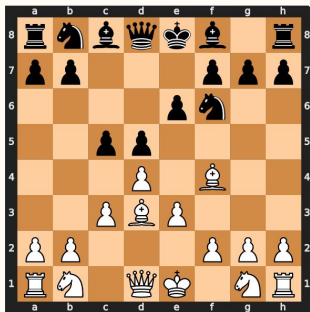




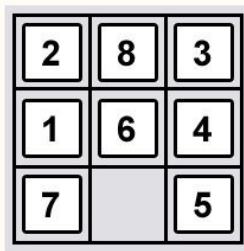
Adversarial Search?

- Search in competitive environments (where agents' goals are in conflict)
- Games are good examples, reasons?
 - simplified state representations
 - Limited Agent's action sets
 - Outcomes of agent's actions are clearly and precisely
 - Usually hard to solve

Games that are adversarial search



Examples of Games that are NOT adversarial search



Making them easier to model and analyze

How to Formally Define a game?

Consider a game with 2 players MAX and MIN

- MAX moves first (place X) followed by MIN(place O)

A game can be formally defined with the following elements:

- **So**: Initial State; Specifies the game setup at the start.
- **TO-MOVE(s)**: Player to Move; Indicates whose turn it is to move in state s.
- **ACTIONS(s)**: Defines the legal moves in state s.
- **RESULT(s, a)**: Transition Model; Defines the state resulting from taking action a in state s.
- **IS-TERMINAL(s)**: Terminal Test; Determines when the game is over.
- **UTILITY(s, p)**: Utility Function; Defines the final numeric value to player p when the game ends in terminal state s.

The initial state, actions, and result function define game tree for a game.

A game tree - tree where nodes are games states and edges are moves



Game Tree for a tic-tac-toe

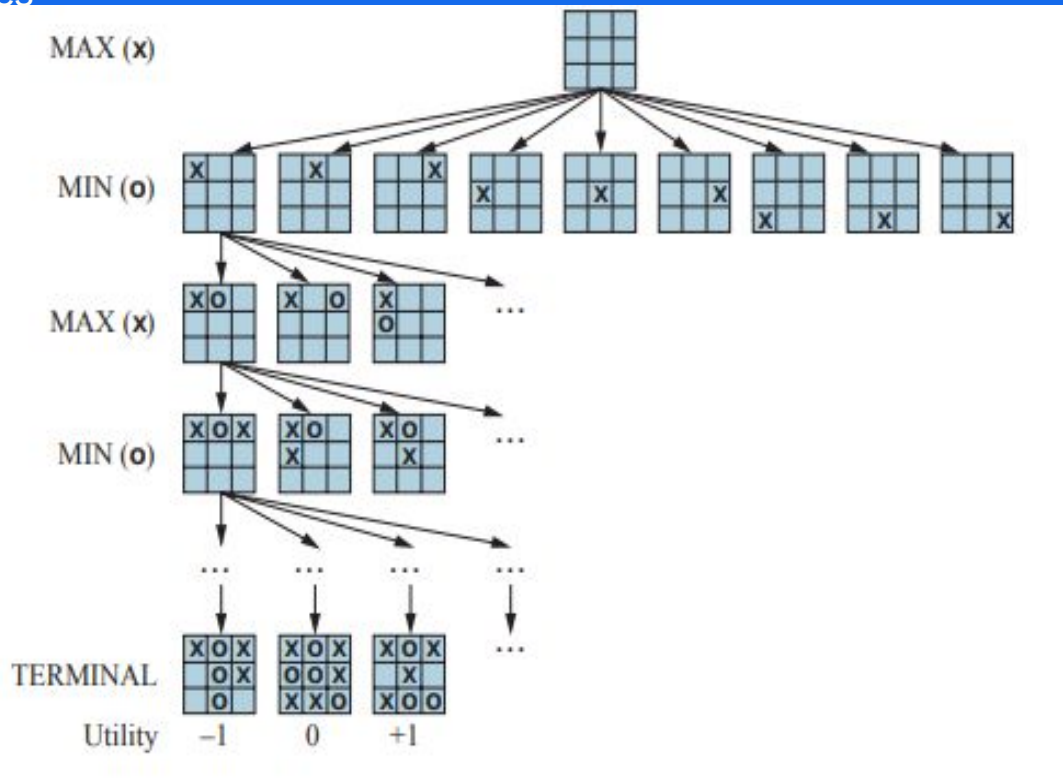
of Nodes

L0: 1

L1: 9

L2: 8×9

L3: $7 \times 8 \times 9$



MAX has 9 possible moves

Play alternates between MAX placing an X and MIN placing an O

The number of in each of the leaf node indicates the utility value of the terminal state from point of view of MAX

- High value are assumed to be good for MAX and bad for MIN

of nodes are in the game tree?

Optimal Decisions in games

A normal search problem:

- An optimal solution is a sequence of actions leading to a goal state

An Adversarial Search

- 'MIN' interferes with the sequence of actions
- Strategy for 'MAX'
 - Specify the move in the initial state
 - Observe every possible response of MIN
 - Specify moves in response
 - and so on...

This optimal strategy can be determined from the **Minimax value** of each node.

How to Calculate minimax value?

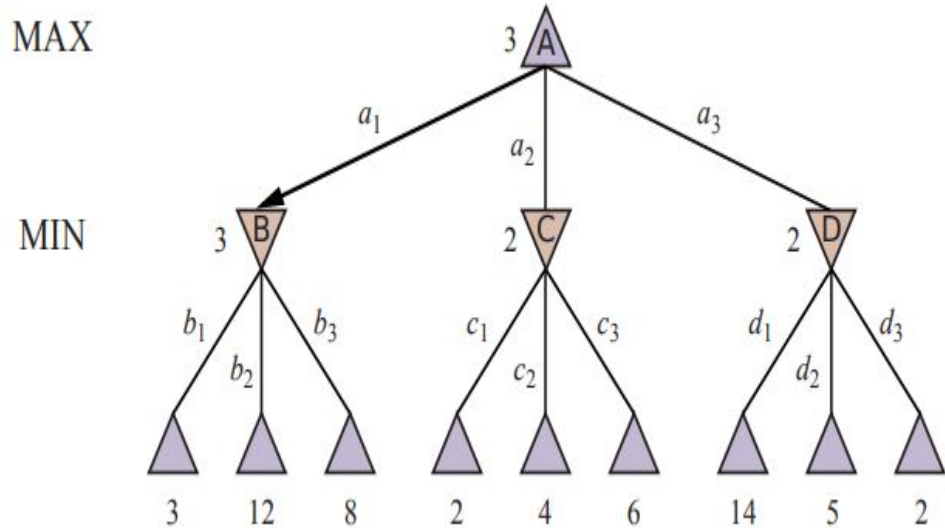
Consider the game tree

- The possible moves for MAX at the root are a_1 , a_2 , and a_3
- Possible replies to a_1 for MIN are b_1 , b_2 , b_3 , and so on

Optimal strategy can be determined using minimax value of each node

MINIMAX(n)

- **MINIMAX(n)** for the user MAX is the utility of being in the corresponding state
- So, MAX will always prefer to move to a state of maximum value



MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if To-MOVE}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if To-MOVE}(s) = \text{MIN} \end{cases}$$

How to Calculate minimax value?

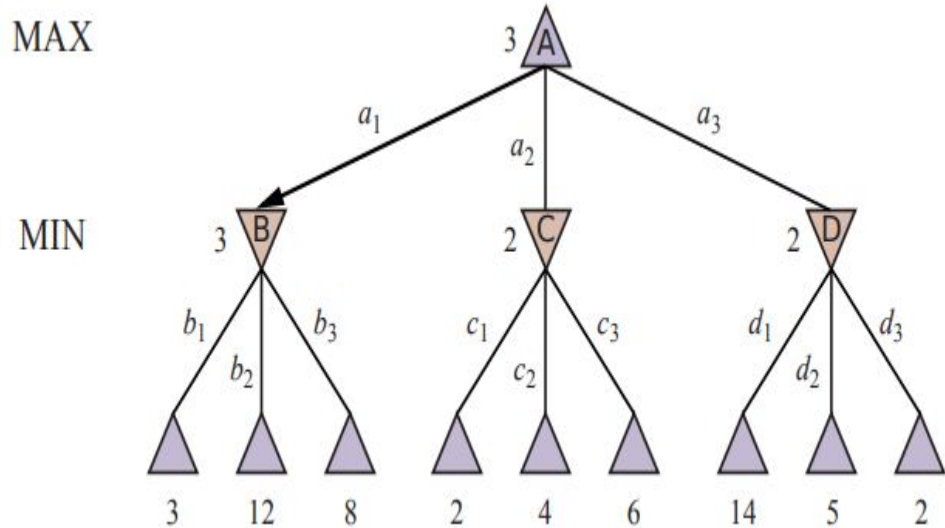
The minimax values at nodes A, B, C, and D for the player MAX given the game tree below. The numbers at the leaf nodes represent the values of Utility (leafnode, MAX)?

$$\text{MINIMAX}(B) = \min(3, 12, 8) = 3$$

$$\text{MINIMAX}(C) = \min(2, 4, 6) = 2$$

$$\text{MINIMAX}(D) = \min(14, 5, 2) = 2$$

$$\text{MINIMAX}(A) = \max(\text{MINIMAX}(B), \text{MINIMAX}(C), \text{MINIMAX}(D)) = \max(3, 2, 2) = 3$$



$\text{MINIMAX}(s) =$

$$\begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if To-MOVE}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if To-MOVE}(s) = \text{MIN} \end{cases}$$

Utility function vs Minimax function

Utility (s, p): defines the final numeric value for a game that ends in a terminal state s for player p

Minimax (s, p): defines the numeric values at all other nodes

- Works both terminal and non-terminal nodes. But terminal node returns utility of that node directly

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

MINIMAX ALGORITHM

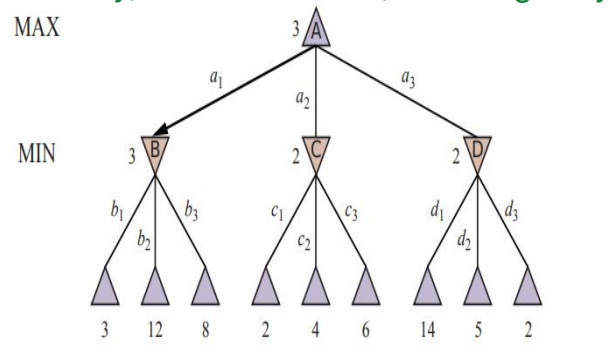
It is a recursive algorithm that explores the game tree in a depth-first manner, proceeding to the leaves and then backing up the minimax values through the tree as the recursion unwinds.

MINIMAX ALGORITHM

Purpose : Finds the best move for MAX by evaluating all possible actions and selecting the one with the highest MINIMAX value.

Process :

- Recursive algorithm exploring the entire game tree.
- Backs up minimax values from leaf nodes to the root.
- Alternates between MAX-VALUE (maximizing utility) and MIN-VALUE (minimizing utility).



function MINIMAX-SEARCH(*game, state*) **returns** an action
 player \leftarrow game.TO-MOVE(*state*)
 value, move \leftarrow MAX-VALUE(*game, state*)
return move

function MAX-VALUE(*game, state*) **returns** a (utility, move) pair
if game.IS-TERMINAL(*state*) **then return** game.UTILITY(*state, player*), null
 v, move $\leftarrow -\infty$
for each a in game.ACTIONS(*state*) **do**
 v2, a2 \leftarrow MIN-VALUE(*game, game.RESULT(state, a)*)
 if v2 > v **then**
 v, move \leftarrow v2, a
return v, move

function MIN-VALUE(*game, state*) **returns** a (utility, move) pair
if game.IS-TERMINAL(*state*) **then return** game.UTILITY(*state, player*), null
 v, move $\leftarrow +\infty$
for each a in game.ACTIONS(*state*) **do**
 v2, a2 \leftarrow MAX-VALUE(*game, game.RESULT(state, a)*)
 if v2 < v **then**
 v, move \leftarrow v2, a
return v, move

MINIMAX ALGORITHM

1. Call MINIMAX-SEARCH(game, A)

- player = MAX
- Calls **MAX-VALUE(game, A)**

2. Execute MAX-VALUE(game, A)

- A is a MAX node, initialize: $v = -\infty$, move = null
- Iterate over actions $\{a_1, a_2, a_3\}$ leading to nodes $\{B, C, D\}$

For action a_1 (B): Call MIN-VALUE(game, B)

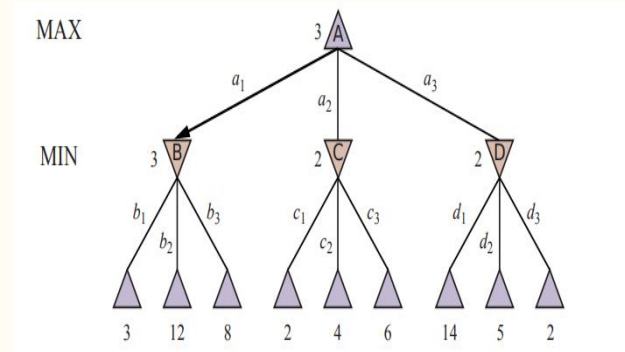
3. Execute MIN-VALUE(game, B)

- B is a MIN node, initialize: $v = +\infty$, move = null
- Iterate over $\{b_1, b_2, b_3\}$ leading to $\{3, 12, 8\}$
- Compute $v = \min(3, 12, 8) = 3$
- Return $(3, b_1)$

Back to MAX-VALUE(A), update:

if $v_2 > v \rightarrow v = 3$, move = a_1

For action a_2 (C): Call MIN-VALUE(game, C)



```
function MINIMAX-SEARCH(game, state) returns an action
  player  $\leftarrow$  game.TO-MOVE(state)
  value, move  $\leftarrow$  MAX-VALUE(game, state)
  return move
```

```
function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move  $\leftarrow -\infty$ 
  for each a in game.ACTIONS(state) do
    v2, a2  $\leftarrow$  MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move  $\leftarrow$  v2, a
  return v, move
```

```
function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move  $\leftarrow +\infty$ 
  for each a in game.ACTIONS(state) do
    v2, a2  $\leftarrow$  MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move  $\leftarrow$  v2, a
  return v, move
```

4. Execute MIN-VALUE(game, C)

- C is a MIN node, initialize: $v = +\infty$, move = null
- Iterate over $\{c_1, c_2, c_3\}$ leading to $\{2, 4, 6\}$
- Compute $v = \min(2, 4, 6) = 2$
- Return (2, c1)

Back to MAX-VALUE(A), update:

if $v_2 > v \rightarrow$ No change ($3 > 2$)

For action a_3 (D): Call MIN-VALUE(game, D)

5. Execute MIN-VALUE(game, D)

- D is a MIN node, initialize: $v = +\infty$, move = null
- Iterate over $\{d_1, d_2, d_3\}$ leading to $\{14, 5, 2\}$
- Compute $v = \min(14, 5, 2) = 2$
- Return (2, d3)

Back to MAX-VALUE(A), update:

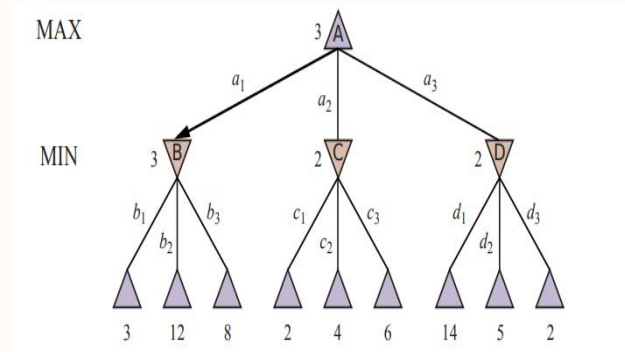
if $v_2 > v \rightarrow$ No change ($3 > 2$)

6. Return from MAX-VALUE(A)

- $v = 3$, move = a_1
- Optimal move = $a_1 \rightarrow B$

Final Output

- Best move for MAX: a_1 (choosing B)
- Minimax value of A: 3

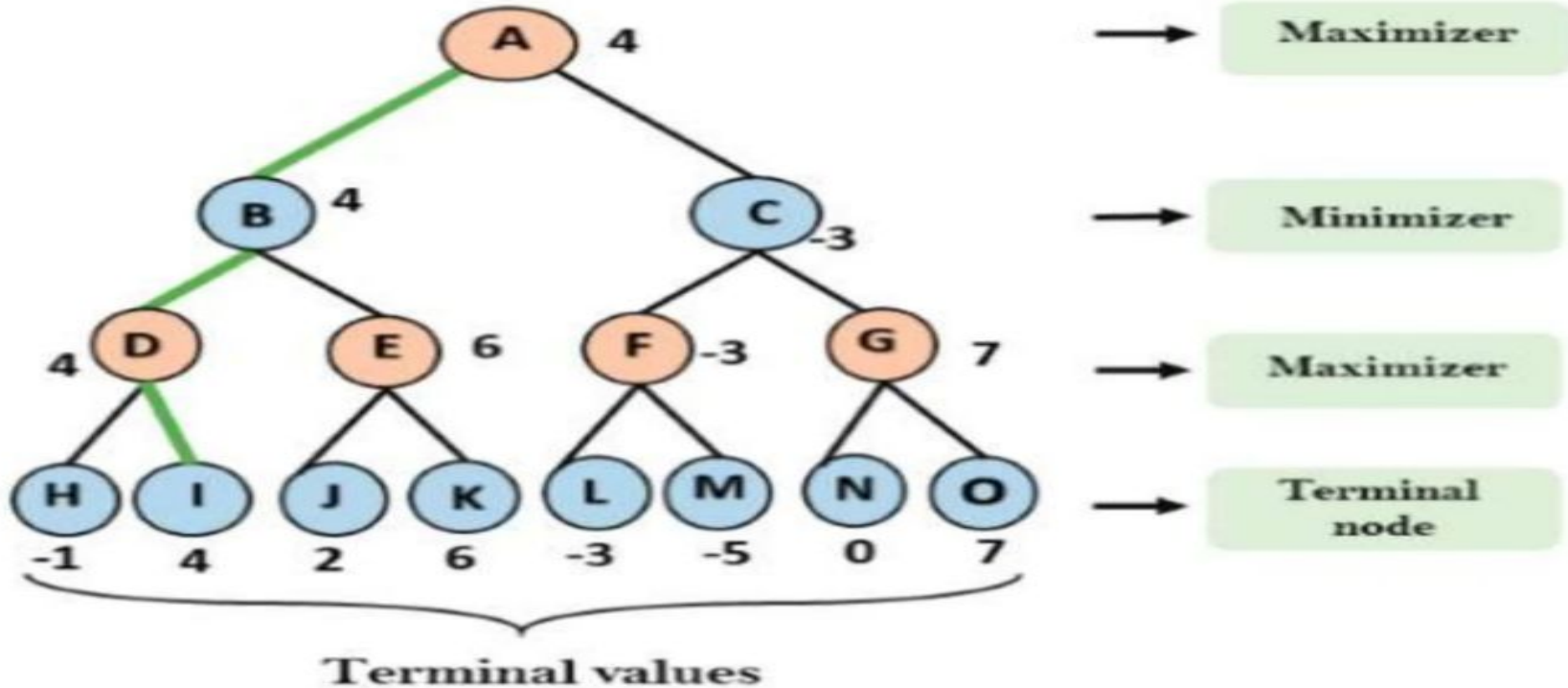


```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move
```

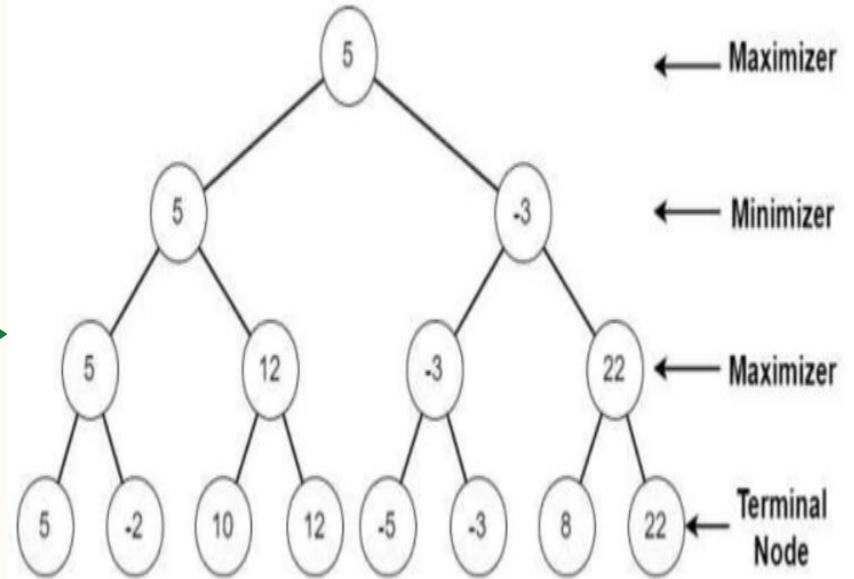
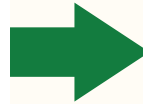
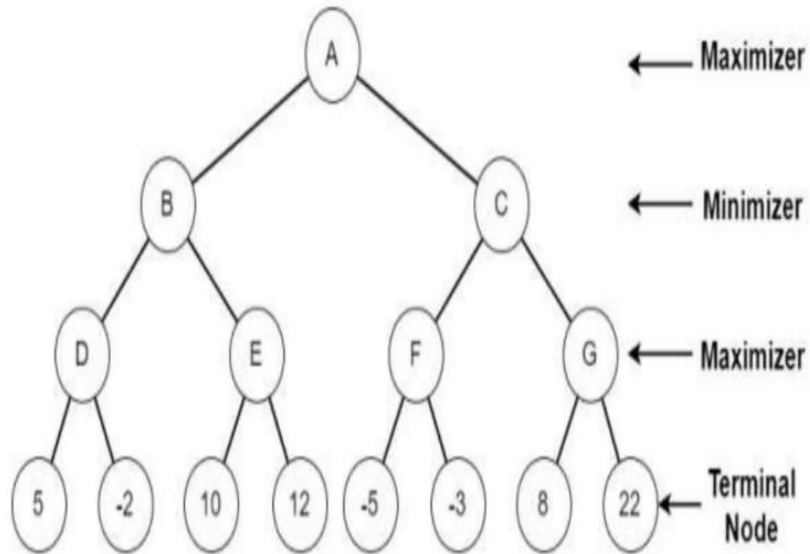
```
function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← -∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move
```

```
function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v, move ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

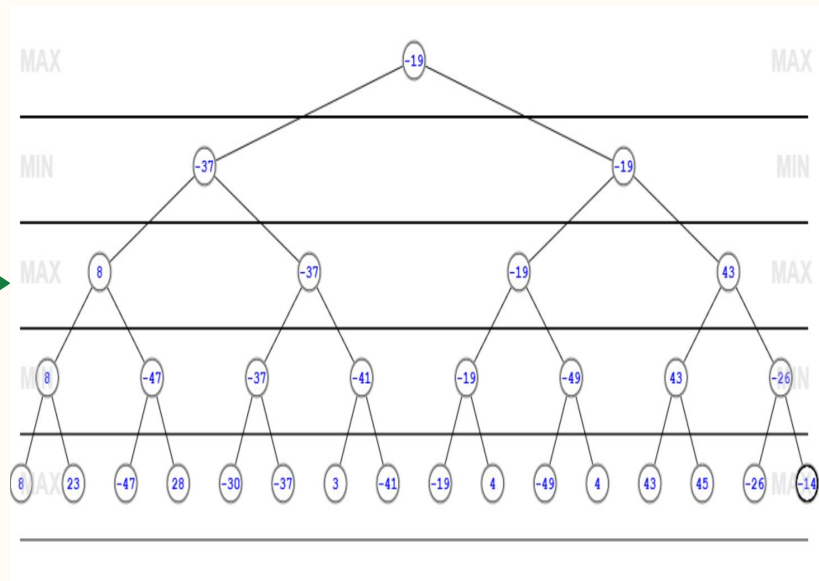
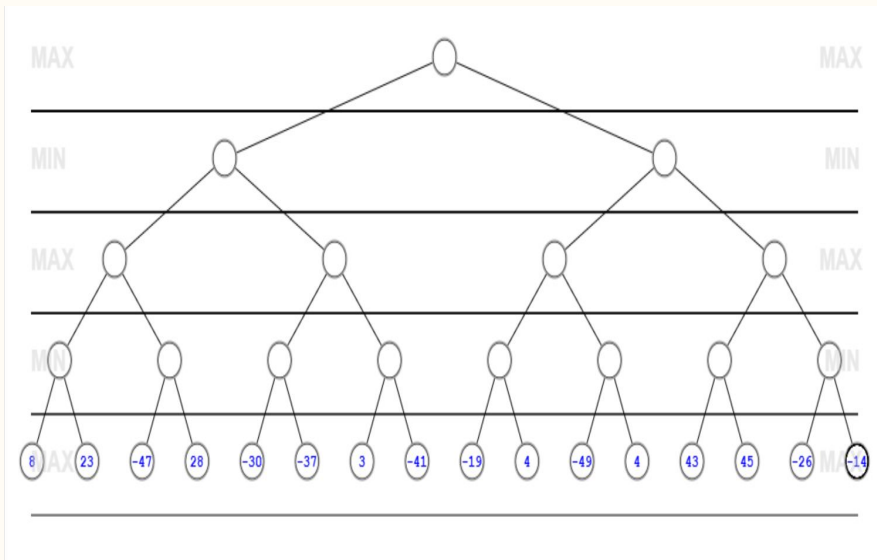
EXAMPLE OF MINIMAX ALGORITHM



Example 2



Example 3



Complexity of Minimax Algorithm

The **time complexity** of the minimax algorithm is $O(b^m)$, where

- m is the maximum depth of the tree and
- b is the number of legal moves at each point.

The **space complexity** is:

- $O(bm)$ for an algorithm that generates all actions at once, or
- $O(m)$ for an algorithm that generates actions one at a time.

Limitations

Exponential complexity makes it impractical for deep or large trees (e.g. in complex games, chess with $b \approx 35$, $m \approx 80$ ply and it is not feasible to search $35^{80} \approx 10^{123}$ states).

Optimal Decisions in Multiplayer Games

- Extend minimax to multiplayer games using utility vectors (e.g., $[v_A, v_B, v_C]$).
- Nonterminal nodes: Backed-up value = successor state with max utility for the acting player.

Alliances (whether formal or informal, among the players) form naturally when collaboration benefits players (e.g., weaker players vs. stronger).

Alliances can be temporary; breaking them depends on short-term vs. long-term gains.

In non-zero-sum games, cooperation may emerge for mutually beneficial outcomes. **Example:** If a terminal state offers max utility $[v_A = 1000, v_B = 1000]$, both players will cooperate to reach it.

(Optimal strategy = joint effort for mutual gain.)

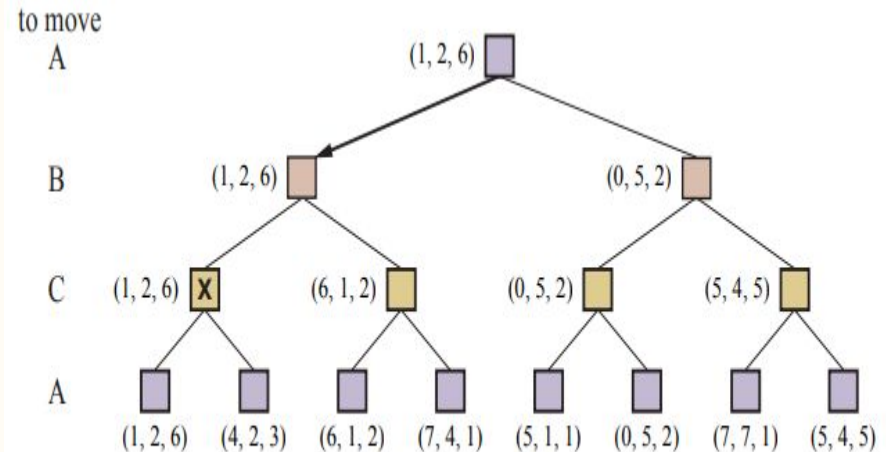


Figure 6.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

Alpha-Beta Pruning

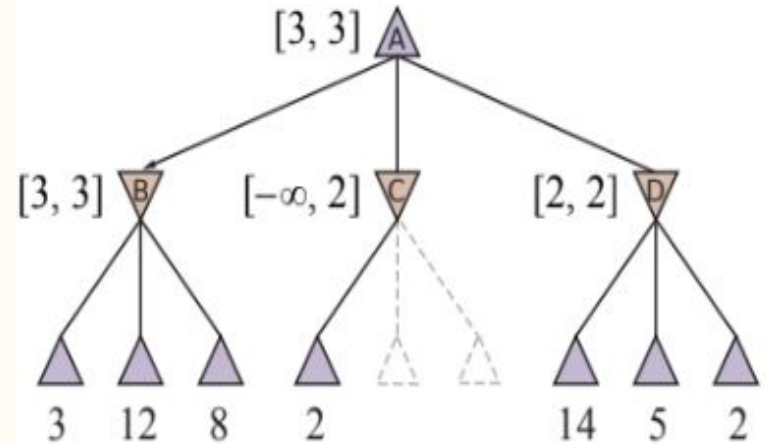
Alpha-beta pruning is a technique used to make minimax search more efficient by ignoring parts of the search tree that do not affect the optimal move.

Do we need to compute all 'MINIMAX' Value?

Let the two unevaluated successors of node C have values x and y . Then the value of the root node is given by:

$$\begin{aligned} \text{MINIMAX}(\text{root}) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\ &= \max(3, \min(2, x, y), 2) \\ &= \max(3, z, 2) \text{ where } z = \min(2, x, y) \leq 2 \\ &= 3. \end{aligned}$$

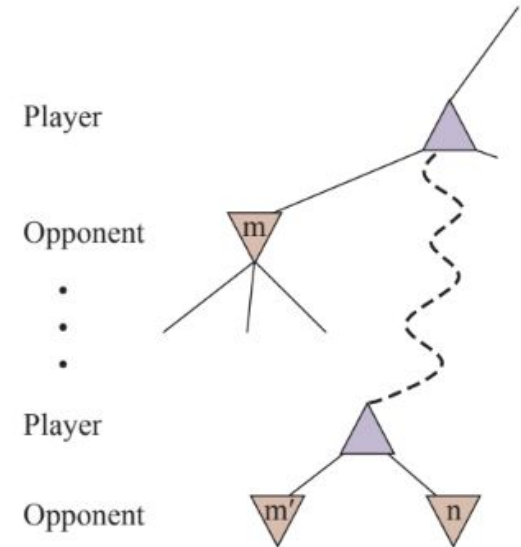
Is minimax dependent on the value of x and y ? NO, the value of the root and hence the minimax decision are independent of the values of the leaves x and y , and therefore they can be pruned.



Which nodes to prune?

The general principle is this:

- consider a node n somewhere in the tree, such that Player has a choice of moving to n . If Player has a better choice either at the same level (e.g. m') or at any point higher up in the tree (e.g. m), then Player will never move to n .
- So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.



Take-Away?

"If you have an idea that is surely bad, do not take time to see how truly awful it is"

Alpha-Beta Pruning

Minimax search is depth-first, so only nodes along a single path need to be considered at any time.

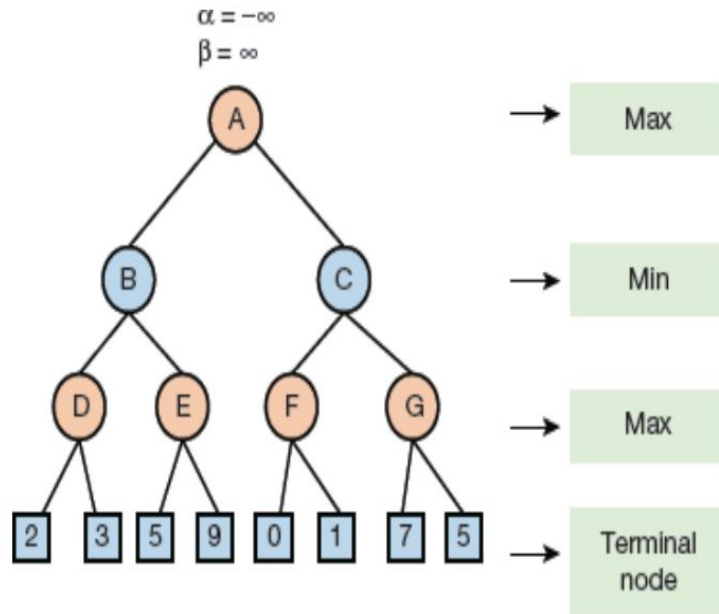
Alpha-beta pruning gets its name from the two extra parameters:

Parameters :

- α (alpha) : The best (highest-value) choice found so far for MAX along the path ("at least").
- β (beta) : The best (lowest-value) choice found so far for MIN along the path ("at most").

Pruning Mechanism : Updates α and β during the search and prunes branches when the current node's value is worse than the current α or β , terminating further exploration of that path.

ALPHA-BETA ALGORITHM

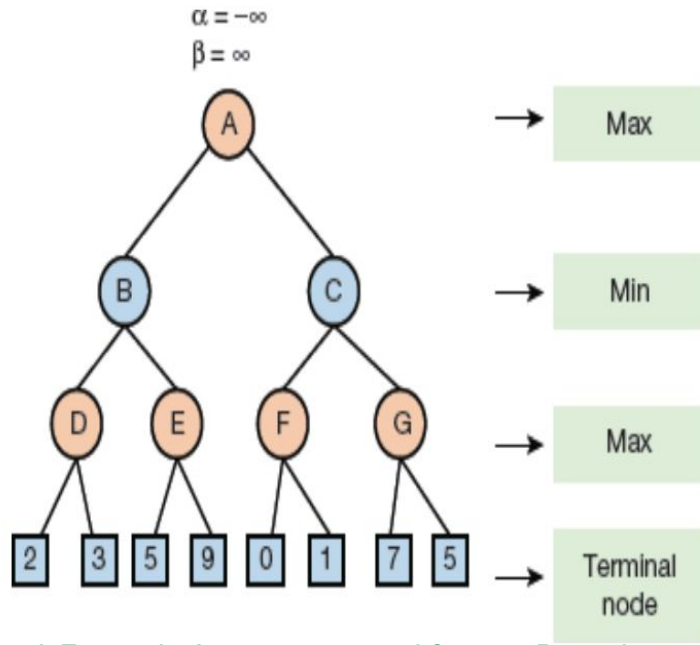


function ALPHA-BETA-SEARCH(*game, state*) **returns** an action
 player \leftarrow game.TO-MOVE(*state*)
 value, move \leftarrow MAX-VALUE(*game, state, $-\infty, +\infty$*)
return move

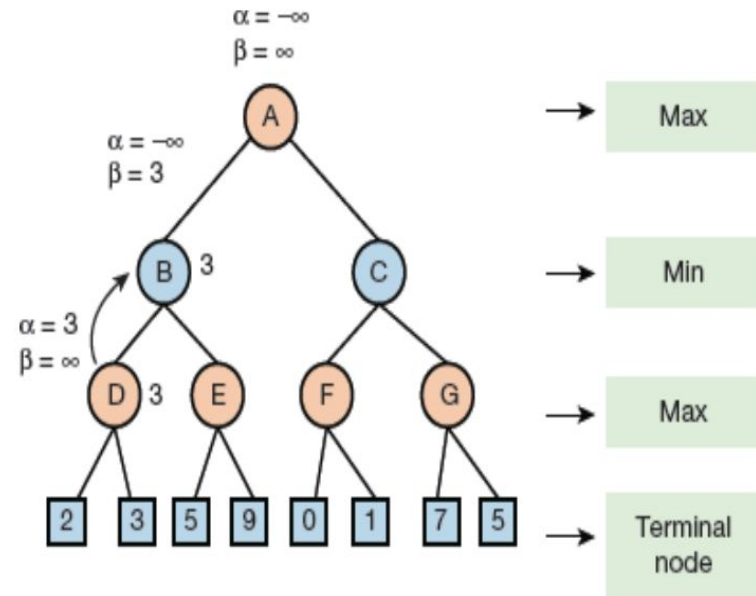
function MAX-VALUE(*game, state, α, β*) **returns** a (utility, move) pair
if game.IS-TERMINAL(*state*) **then return** game.UTILITY(*state, player*), null
 $v \leftarrow -\infty$
for each *a* **in** game.ACTIONS(*state*) **do**
 $v2, a2 \leftarrow$ MIN-VALUE(*game, game.RESULT(state, a), α, β*)
 if $v2 > v$ **then**
 $v, move \leftarrow v2, a$
 $\alpha \leftarrow \text{MAX}(\alpha, v)$
 if $v \geq \beta$ **then return** $v, move$
return $v, move$

function MIN-VALUE(*game, state, α, β*) **returns** a (utility, move) pair
if game.IS-TERMINAL(*state*) **then return** game.UTILITY(*state, player*), null
 $v \leftarrow +\infty$
for each *a* **in** game.ACTIONS(*state*) **do**
 $v2, a2 \leftarrow$ MAX-VALUE(*game, game.RESULT(state, a), α, β*)
 if $v2 < v$ **then**
 $v, move \leftarrow v2, a$
 $\beta \leftarrow \text{MIN}(\beta, v)$
 if $v \leq \alpha$ **then return** $v, move$
return $v, move$

ALPHA-BETA ALGORITHM Example 1

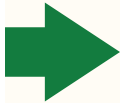
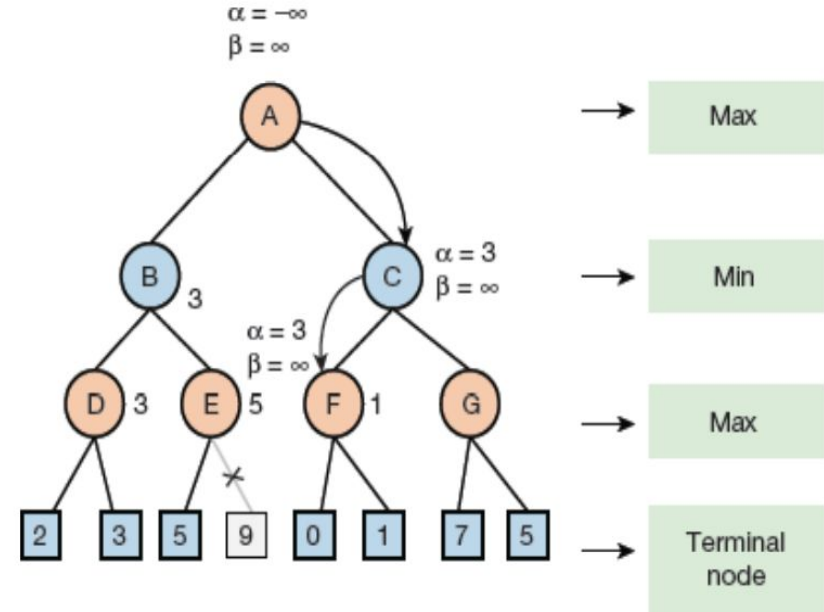
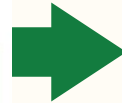
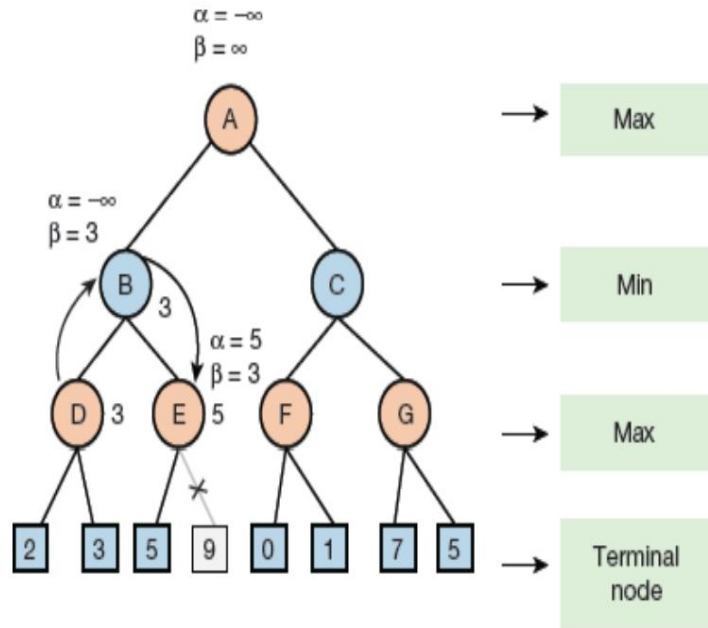


Step 1: For node A, set $\alpha = -\infty$ and $\beta = +\infty$. Pass down these values to node B and node D. Now, all three nodes have $\alpha = -\infty$ and $\beta = +\infty$



Step 2: Node D will play the role of maximizer and will therefore choose a value of α . The value of α at node D = 3 since $\max(2, 3) = 3$.

ALPHA-BETA ALGORITHM Example 1 (continued)



Step 3: Algorithm backtracks to node B which plays the role of a minimizer. So, node B will set the value of β . Recall that $\beta = +\infty$ at B. After comparing this value with the available subsequent node's value, we set $\beta = 3$ as $\min(\infty, 3) = 3$. The algorithm also passes values $\alpha = -\infty$, and $\beta = 3$ to its successor node E

Step 4: At node E, maximizer will update the value of alpha. The current value of alpha will be compared with 5. Since $\max(-\infty, 5) = 5$, $\alpha = 5$ and $\beta = 3$, where $\alpha \geq \beta$, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

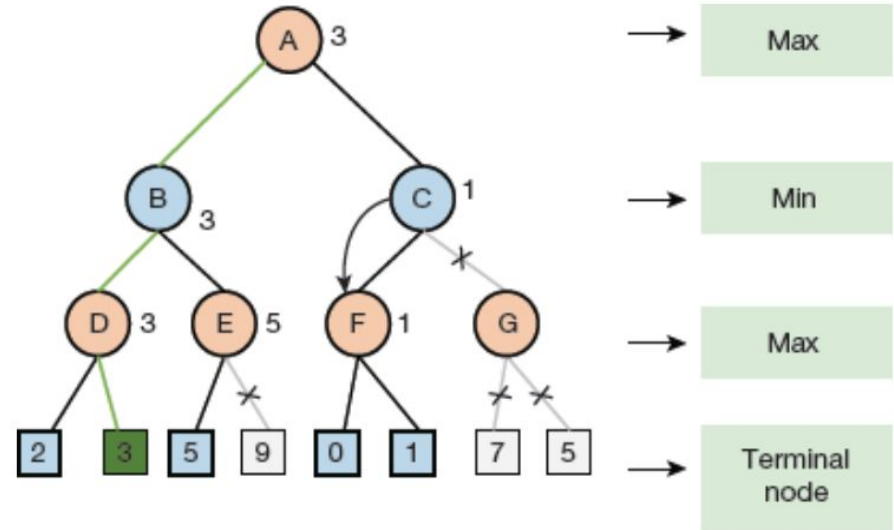
ALPHA-BETA ALGORITHM Example 1 (continued)

Step 5: The algorithm backtracks the tree to node A from node B. At this node, the value of alpha is updated to 3 as $\max(-\infty, 3) = 3$. Now α and β ($+\infty$) values are passed to the right successor of A which is Node C. At node C, $\alpha=3$ and $\beta=+\infty$, and these values are passed to node F.

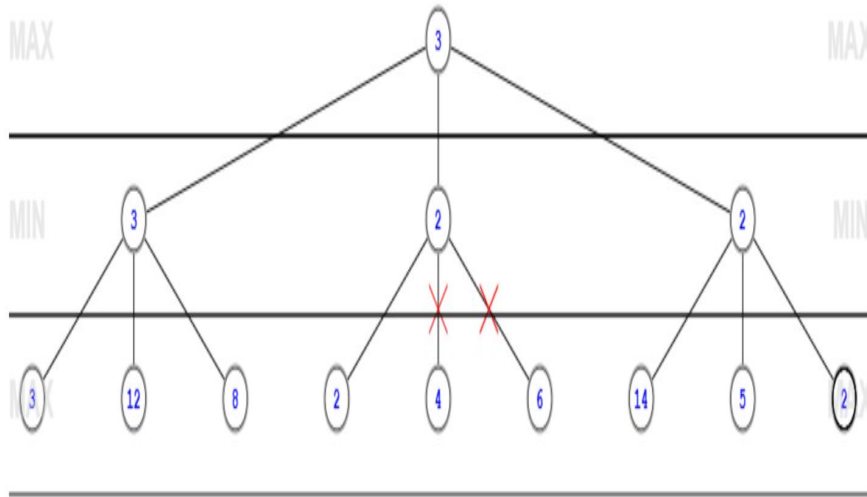
Step 6: At node F, the value of α is compared with the value of its left child which is 0. Since, $\max(3, 0) = 3$, and then with right child which is 1, and $\max(3, 1) = 3$. The node value of F becomes 1.

Step 7: The value of node F is returned to node C. At C, $\alpha=3$ and $\beta=+\infty$. Since node B plays the role of minimizer, value of beta is set to 1 as, $\min(\infty, 1) = 1$. Now, the condition $\alpha >= \beta$ is met. Therefore, the right child of C which is G is pruned. Thus, the algorithm will not compute the entire subtree G.

Step 8: Node C returns the value of 1 to A. Since, A plays a maximizer, it selects value 3 as $\max(3, 1) = 3$.



Example 2 ALPHA-BETA ALGORITHM



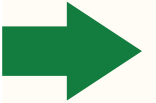
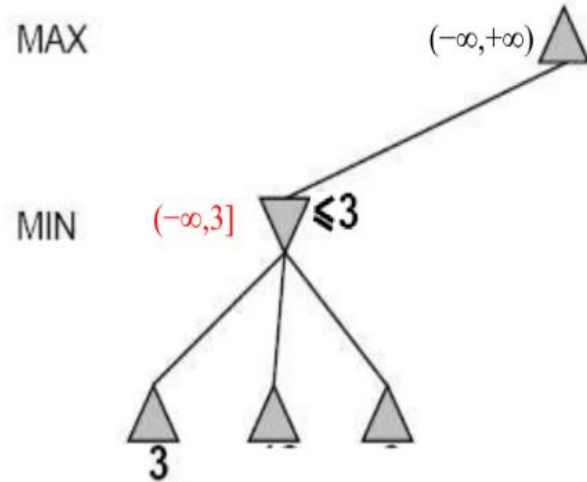
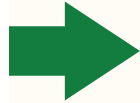
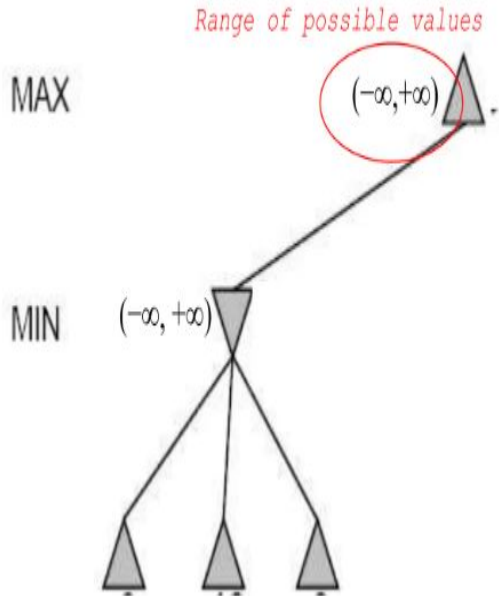
function ALPHA-BETA-SEARCH(*game*, *state*) **returns** an action
 player \leftarrow game.TO-MOVE(*state*)
 value, move \leftarrow MAX-VALUE(*game*, *state*, $-\infty$, $+\infty$)
return move

function MAX-VALUE(*game*, *state*, α , β) **returns** a (utility, move) pair
if game.IS-TERMINAL(*state*) **then return** game.UTILITY(*state*, player), null
 $v \leftarrow -\infty$
for each *a* in game.ACTIONS(*state*) **do**
 $v2, a2 \leftarrow$ MIN-VALUE(*game*, game.RESULT(*state*, *a*), α , β)
 if $v2 > v$ **then**
 $v, move \leftarrow v2, a$
 $\alpha \leftarrow \text{MAX}(\alpha, v)$
 if $v \geq \beta$ **then return** $v, move$
return $v, move$

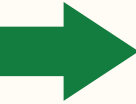
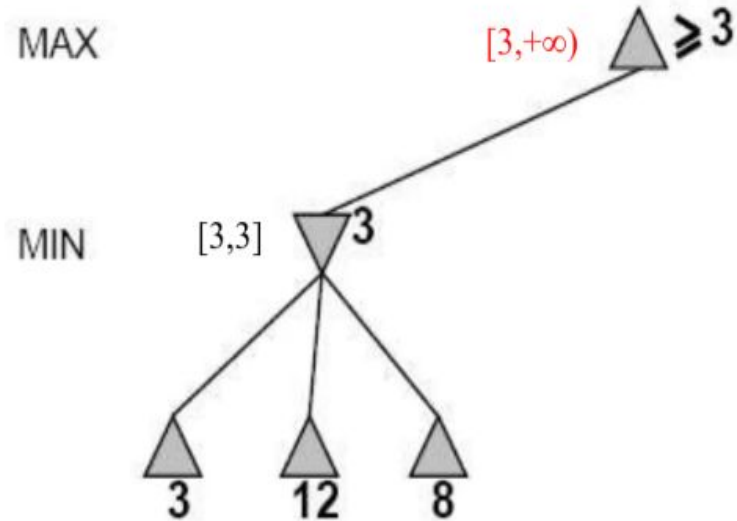
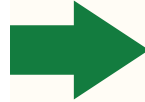
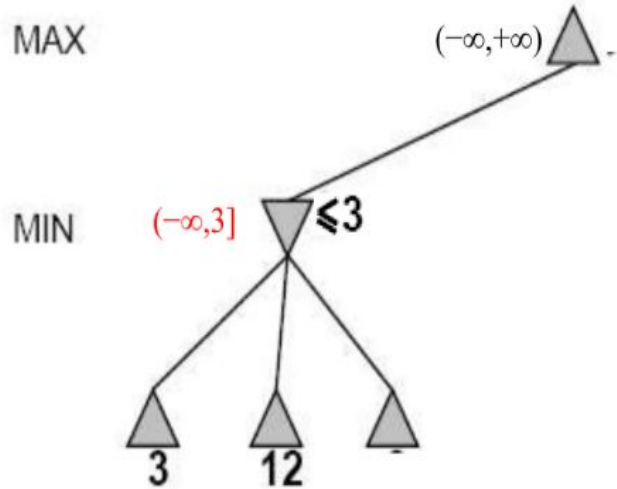
function MIN-VALUE(*game*, *state*, α , β) **returns** a (utility, move) pair
if game.IS-TERMINAL(*state*) **then return** game.UTILITY(*state*, player), null
 $v \leftarrow +\infty$
for each *a* in game.ACTIONS(*state*) **do**
 $v2, a2 \leftarrow$ MAX-VALUE(*game*, game.RESULT(*state*, *a*), α , β)
 if $v2 < v$ **then**
 $v, move \leftarrow v2, a$
 $\beta \leftarrow \text{MIN}(\beta, v)$
 if $v \leq \alpha$ **then return** $v, move$
return $v, move$

ALPHA-BETA ALGORITHM EXAMPLE 2 (continued)

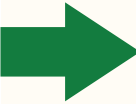
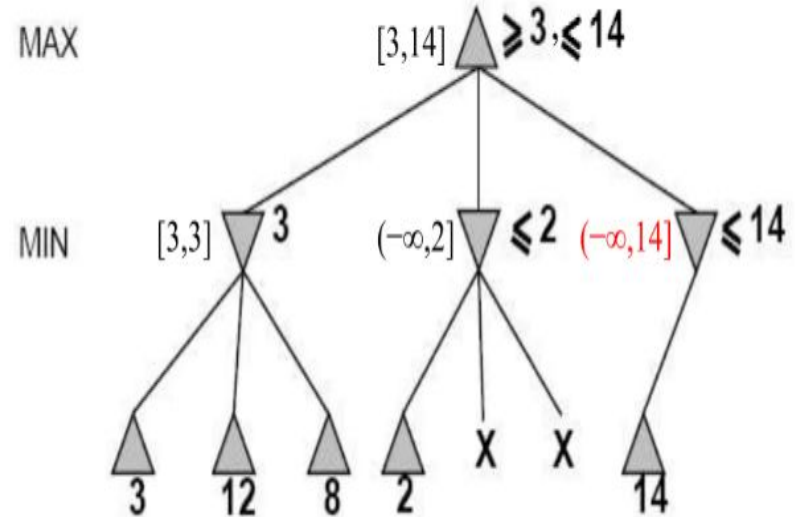
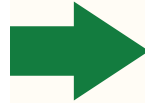
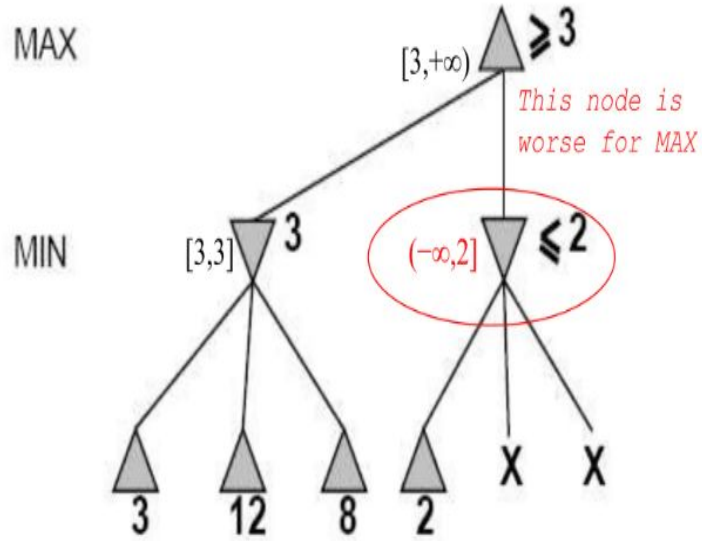
Do DF-search until first leaf



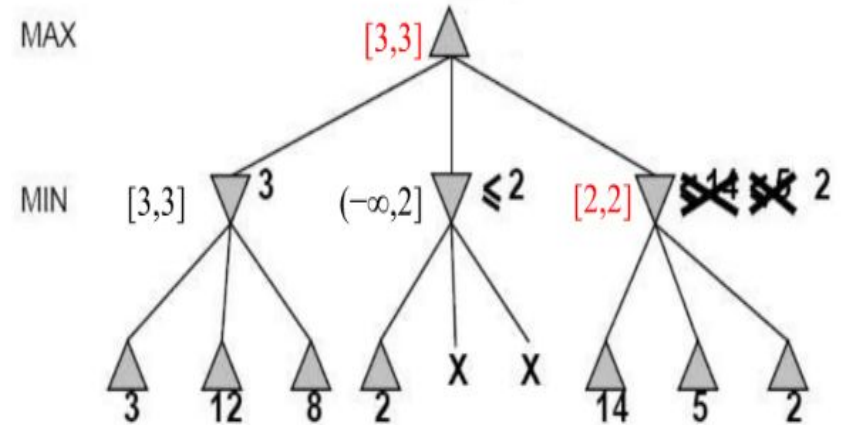
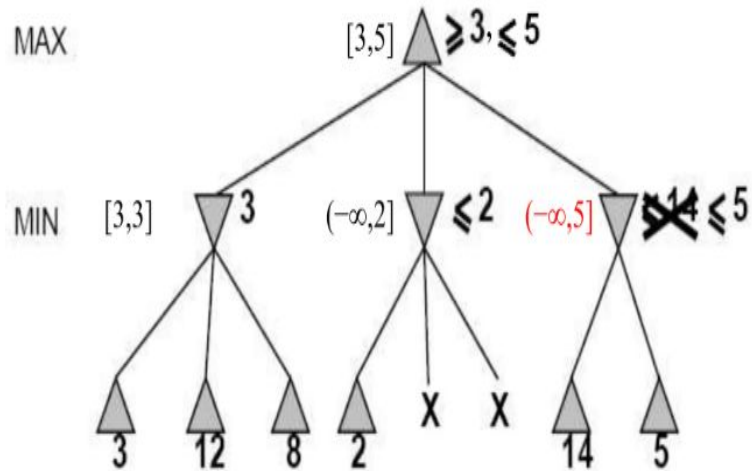
ALPHA-BETA ALGORITHM EXAMPLE 2 (continued)



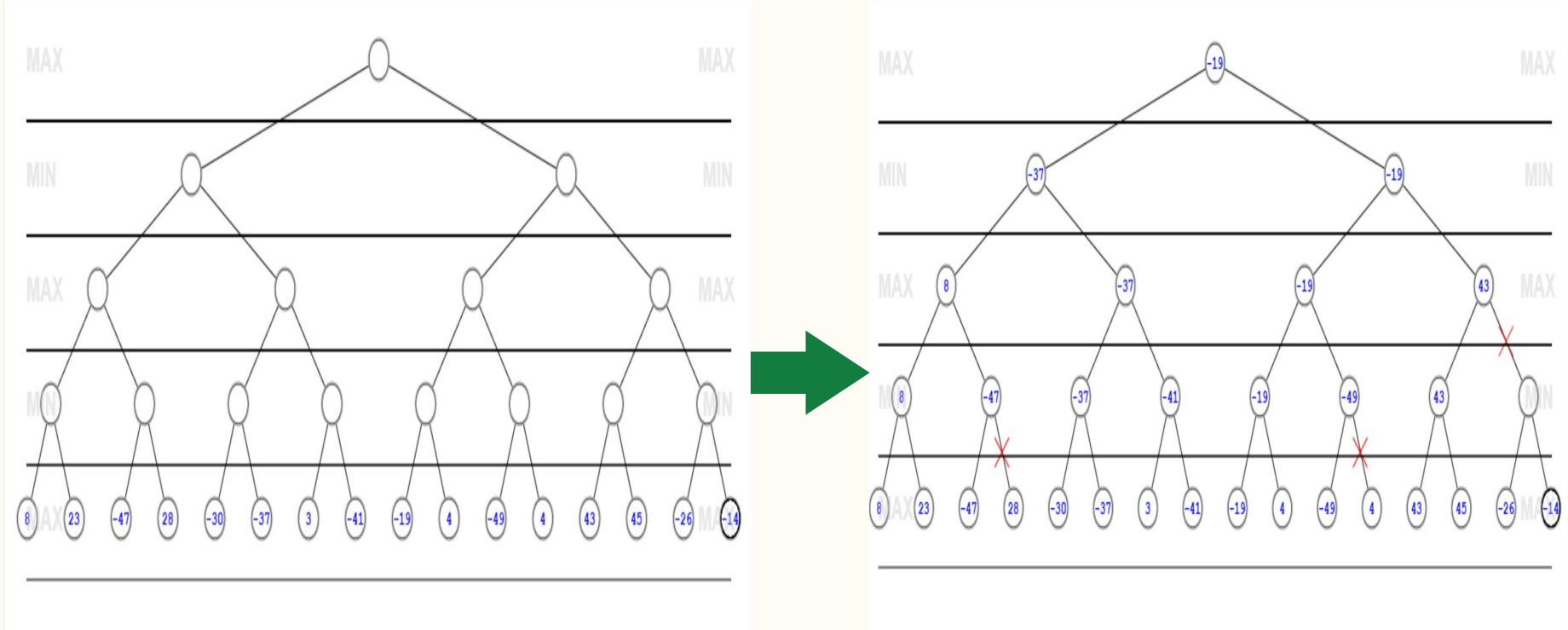
ALPHA-BETA ALGORITHM EXAMPLE 2 (continued)



ALPHA-BETA ALGORITHM EXAMPLE 2 (continued)



Alpha-Beta Algorithm Example 3



Alpha-Beta Pruning Efficiency

Worst-Case: branches are ordered such that no pruning takes place. In this case alpha-beta gives no improvement over Minimax Algorithm. The best move occurs on the right side of the tree and the time complexity for such an order is **$O(b^m)$** .

The effectiveness of alpha-beta pruning depends heavily on the order in which moves are examined. Better move ordering reduces the number of nodes explored.

- **Perfect Move Ordering** : Reduces node exploration to $O(b^{(m/2)})$ instead of $O(b^m)$ for minimax, effectively halving the branching factor ($b \rightarrow \sqrt{b}$).
- **Random Move Ordering** : Results in $O(b^{(3m/4)})$ nodes explored for moderate branching factors.
- **Practical Move Ordering** : Simple heuristics (e.g., prioritizing captures, threats, forward moves) can achieve near-optimal performance, within a factor of 2 of the **best-case scenario** ($O(b^{(m/2)})$).



Questions?



Adiós Muchachos 🖐️

Thanks ❤️