This time: Outline

- **Game playing**
  - The minimax algorithm
  - Resource limitations
  - alpha-beta pruning
  - Elements of chance

What kind of games?

- **Abstraction**: To describe a game we must capture every relevant aspect of the game. Such as:
  - Chess
  - Tic-tac-toe
  - …
- **Accessible environments:** Such games are characterized by perfect information
- **Search:** game-playing then consists of a search through possible game positions
- **Unpredictable opponent:** introduces **uncertainty** thus game-playing must deal with **contingency problems**

Searching for the next move

- **Complexity:** many games have a huge search space
  - **Chess:** $b = 35, m=100 \Rightarrow nodes = 35^{100}$
    if each node takes about 1 ns to explore
    then each move will take about ***10^{50}* millennia**
    to calculate.

- **Resource (e.g., time, memory) limit:** optimal solution not feasible/possible, thus must approximate

1. **Pruning:** makes the search more efficient by discarding portions of the search tree that cannot improve quality result.

2. **Evaluation functions:** heuristics to evaluate utility of a state without exhaustive search.

Two-player games

- A game formulated as a search problem:
  - Initial state: ?
  - Operators: ?
  - Terminal state: ?
  - Utility function: ?

Two-player games

- A game formulated as a search problem:

• Initial state:	board position and turn
• Operators:	definition of legal moves
• Terminal state:	conditions for when game is over
• Utility function:	a numeric value that describes the outcome of the game. E.g., -1, 0, 1 for loss, draw, win. (AKA **payoff function**)

Game vs. search problem

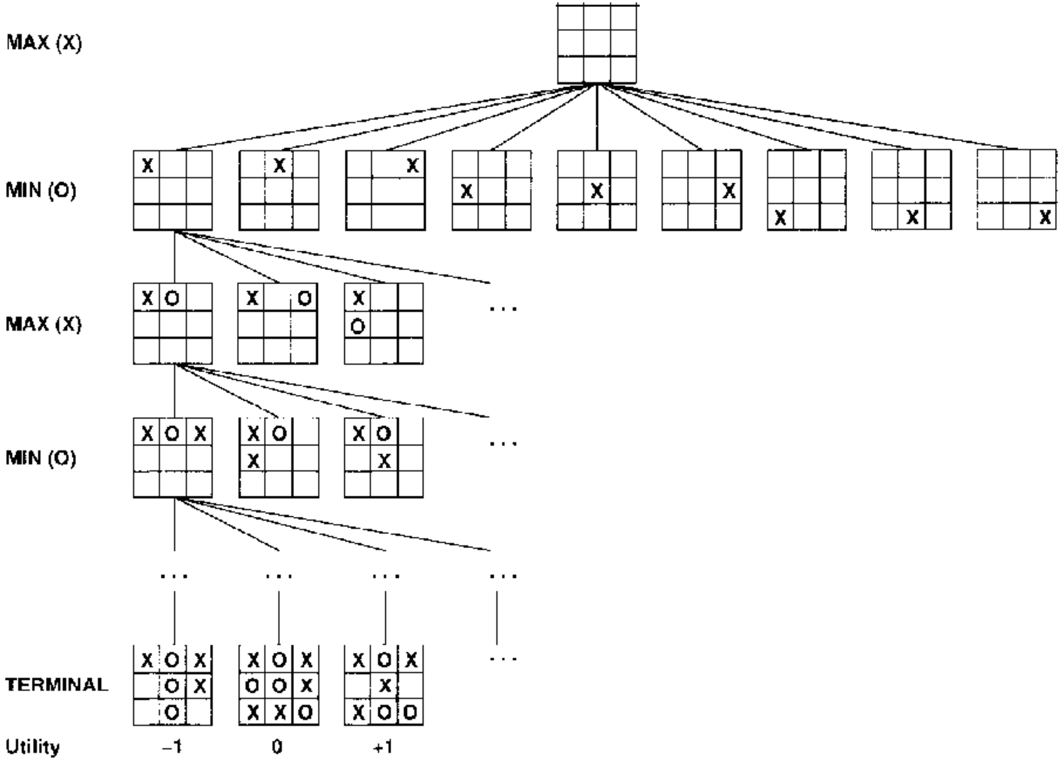
"Unpredictable" opponent $\Rightarrow$ solution is a contingency plan

Time limits $\Rightarrow$ unlikely to find goal, must approximate

Plan of attack:

- algorithm for perfect play (Von Neumann, 1944)
- finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952–57)
- pruning to reduce costs (McCarthy, 1956)

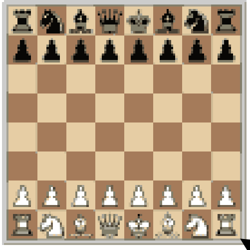
Example: Tic-Tac-Toe



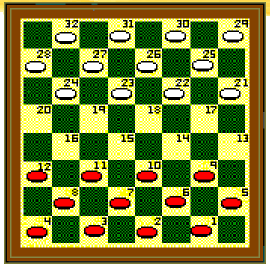
Type of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war

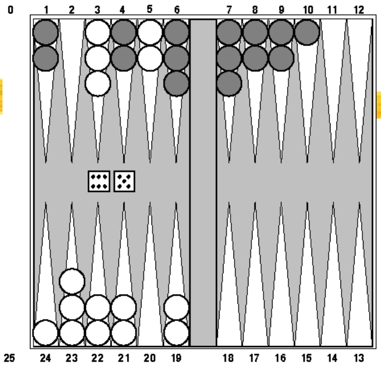
Type of games



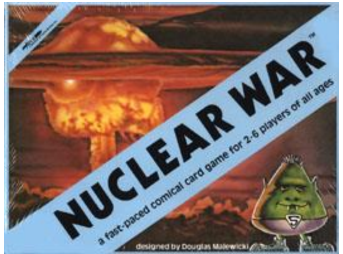
The board set for play



Red to play



	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information		bridge, poker, scrabble nuclear war

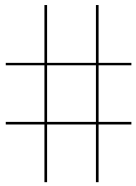


The minimax algorithm

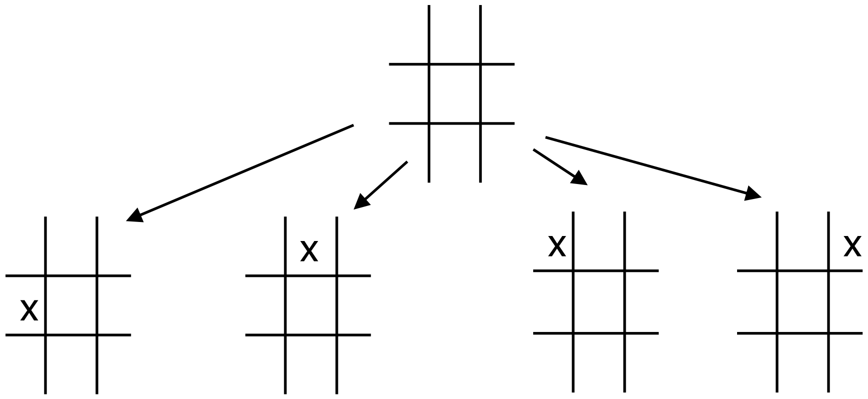
- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
  1. Generate game tree completely
  2. Determine utility of each terminal state
  3. Propagate the utility values upward in the three by applying MIN and MAX operators on the nodes in the current level
  4. At the root node use minimax decision to select the move with the max (of the min) utility value

- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

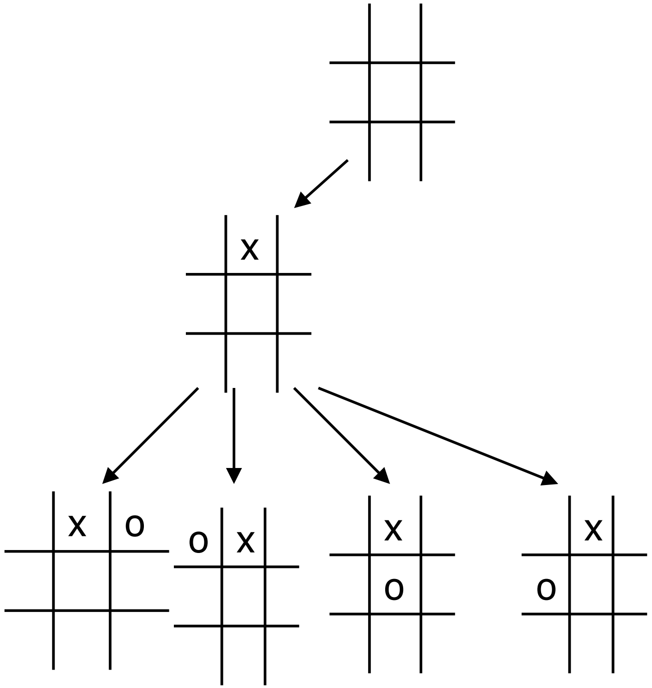
Generate Game Tree



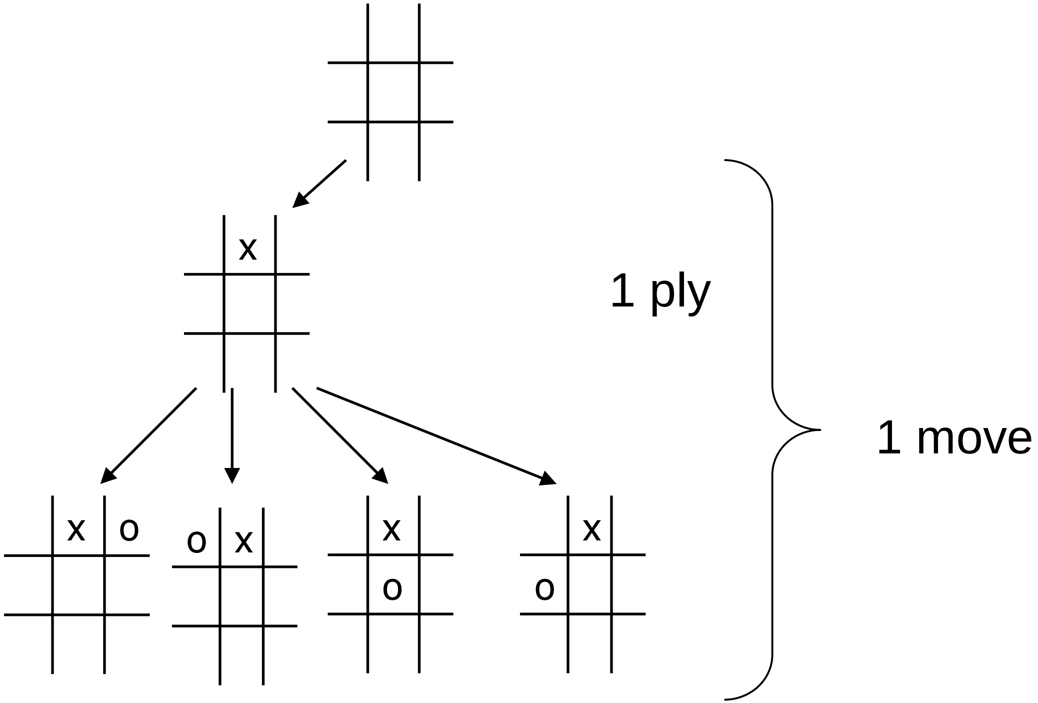
Generate Game Tree



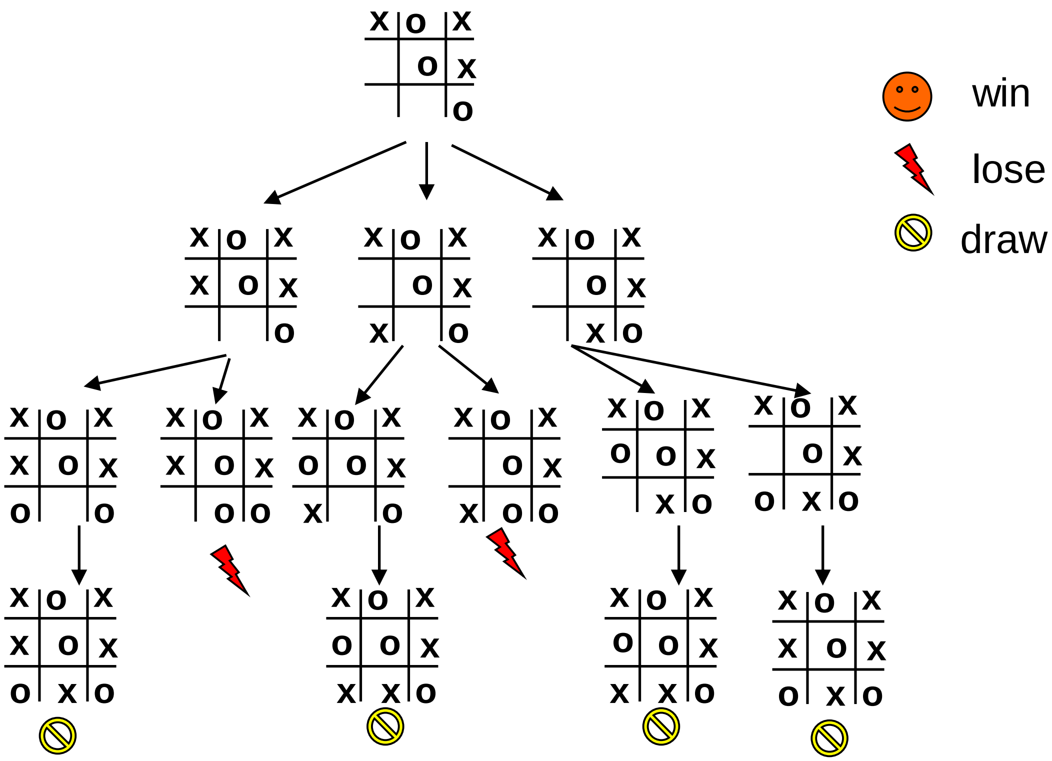
Generate Game Tree



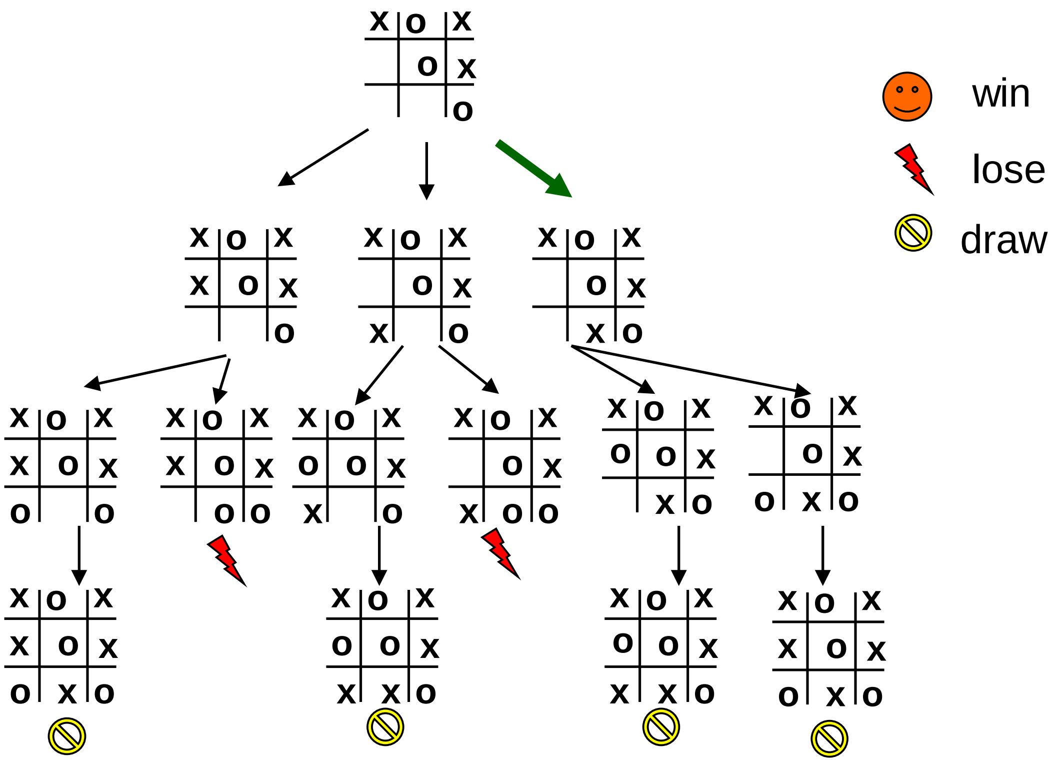
Generate Game Tree



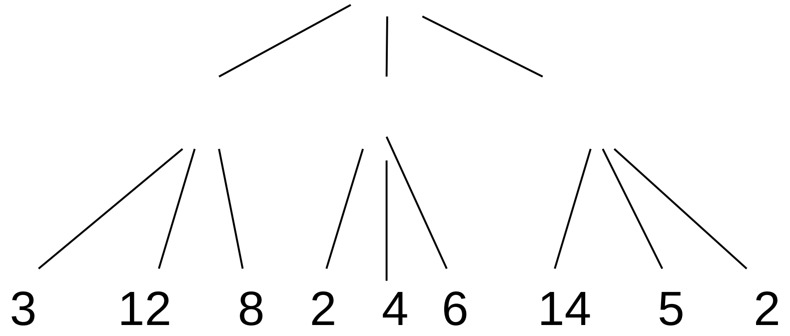
A subtree



What is a good move?



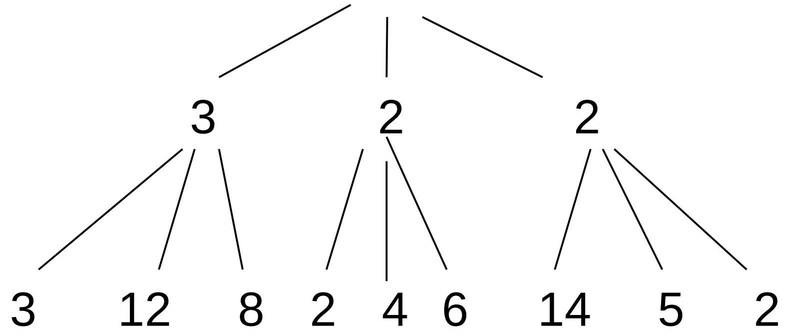
Minimax



- Minimize opponent's chance
- Maximize your chance

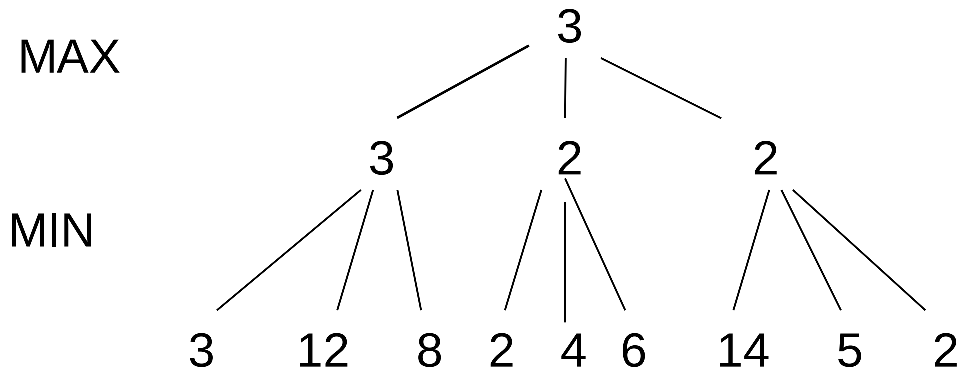
Minimax

MIN



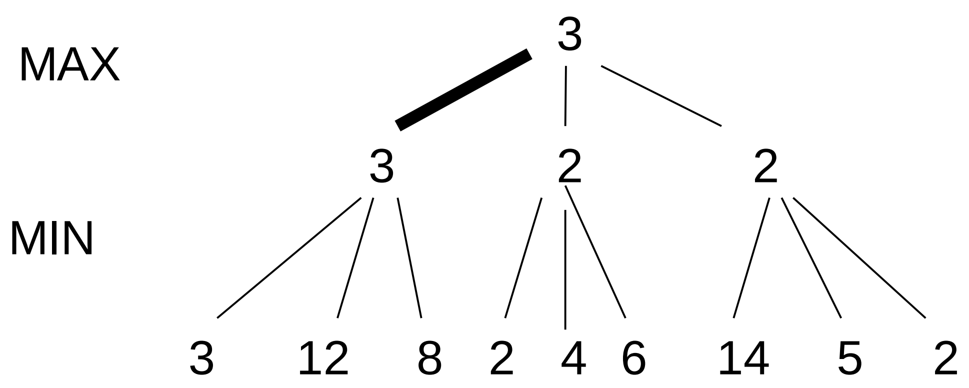
- Minimize opponent’s chance
- Maximize your chance

Minimax



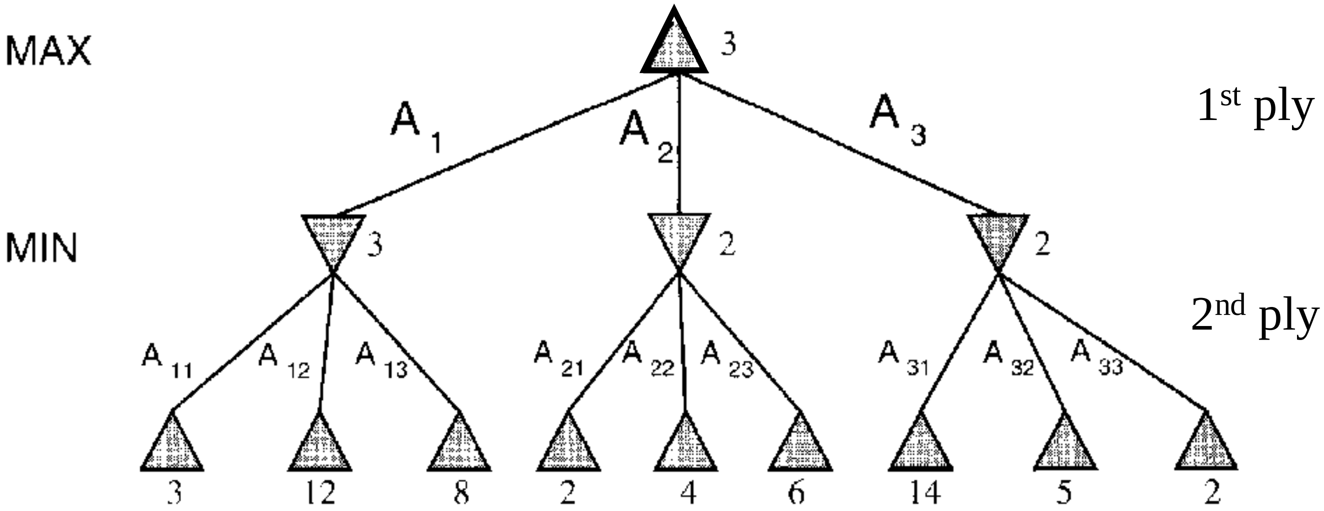
- Minimize opponent’s chance
- Maximize your chance

Minimax



- Minimize opponent's chance
- Maximize your chance

minimax = maximum of the minimum



Minimax: Recursive implementation

```
function MINIMAX-DECISION(state) returns an action
   return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s, a)))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s, a)))
   return v
```

Complete: ?
Optimal: ?

Time complexity: ?
Space complexity: ?

Minimax: Recursive implementation

```
function MINIMAX-DECISION(state) returns an action
   return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s, a)))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s, a)))
   return v
```

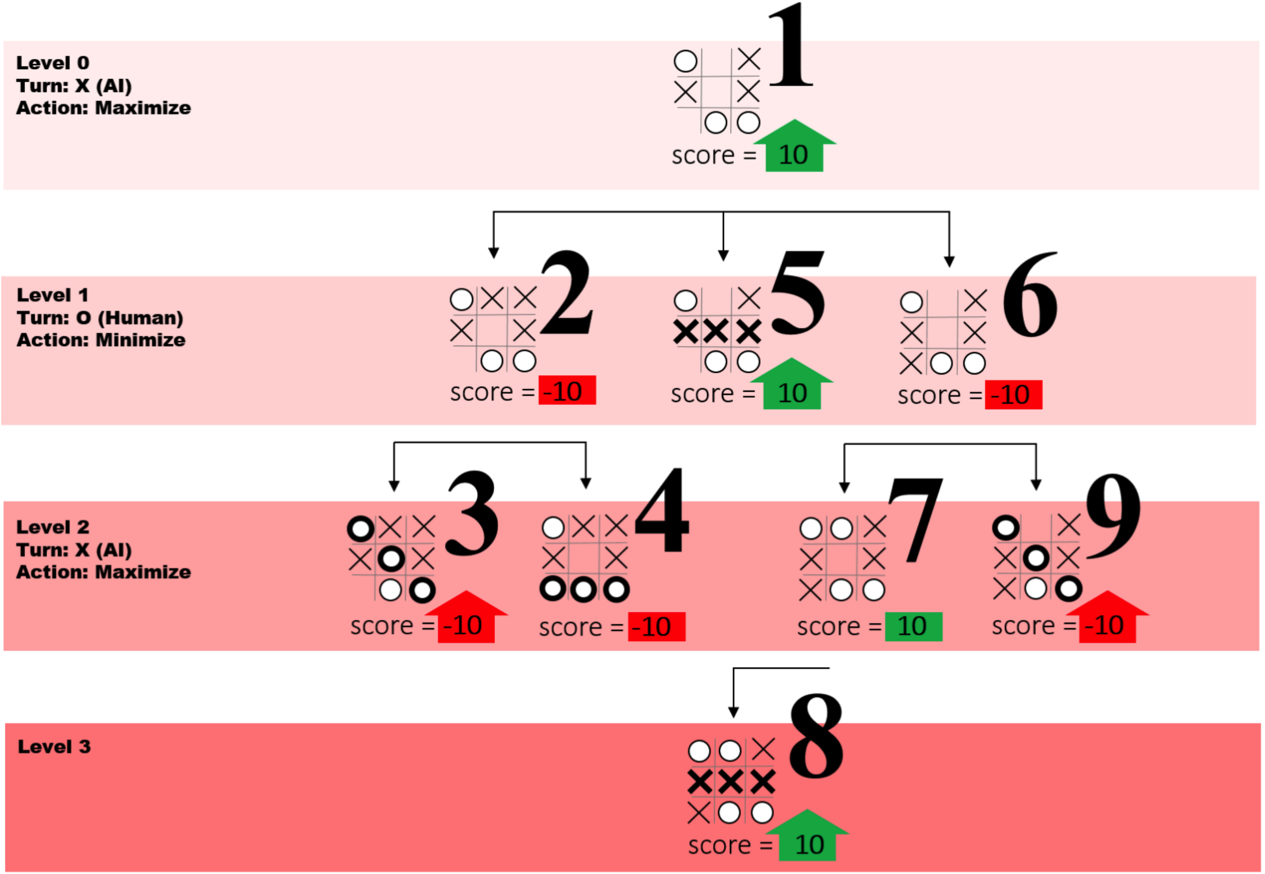
Complete: Yes, for finite state-space
Optimal: Yes

Time complexity: $O(b^m)$
Space complexity: $O(bm)$ (= DFS Does not keep all nodes in memory.)

Minimax: Recursive implementation

https://www.freecodecamp.org/news/how-to-make-your-tic-tac-toe-game-unbeatable-by-using-the-minimax-algorithm-9d690bad4b37

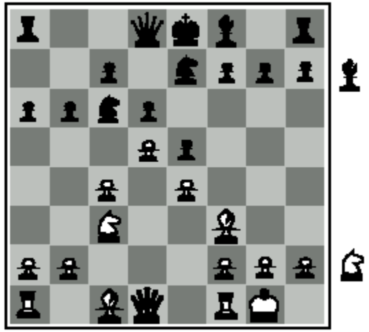
Function call Order during One ply



1. Move evaluation without complete search

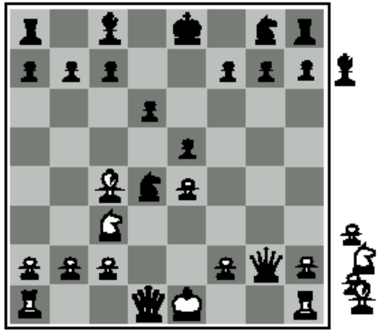
- Complete search is too complex and impractical
- **Evaluation function:** evaluates value of state using heuristics and cuts off search
- **New MINIMAX:**
  - CUTOFF-TEST: cutoff test to replace the termination condition (e.g., deadline, depth-limit, etc.)
  - EVAL: evaluation function to replace utility function (e.g., number of chess pieces taken)

Evaluation functions



Black to move

White slightly better



White to move

Black winning

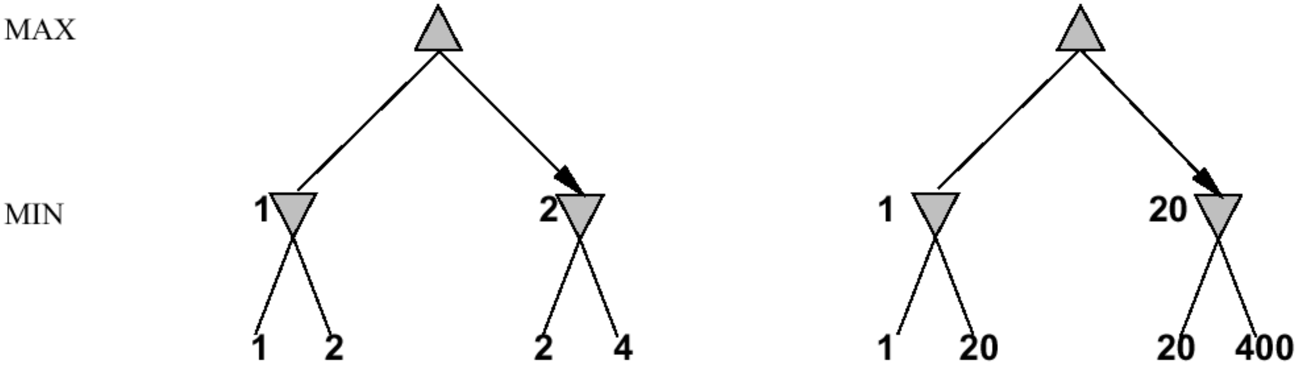
- **Weighted linear evaluation function:** to combine *n* heuristics

$$f = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$$

E.g, w's could be the values of pieces (1 for prawn, 3 for bishop etc.)

f's could be the number of type of pieces on the board

Note: exact values do not matter



Behaviour is preserved under any *monotonic* transformation of EVAL

Only the order matters:
payoff in deterministic games acts as an *ordinal utility* function

Minimax with cutoff: viable algorithm?

MINIMAXCUTOFF is identical to MINIMAXVALUE except

1. TERMINAL? is replaced by CUTOFF?
2. UTILITY is replaced by EVAL

Does it work in practice?

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

4-ply lookahead is a hopeless chess player!

4-ply $\approx$ human novice
8-ply $\approx$ typical PC, human master
12-ply $\approx$ Deep Blue, Kasparov

Assume we have 100 seconds, evaluate 10^4 nodes/s; can evaluate 10^6 nodes/move

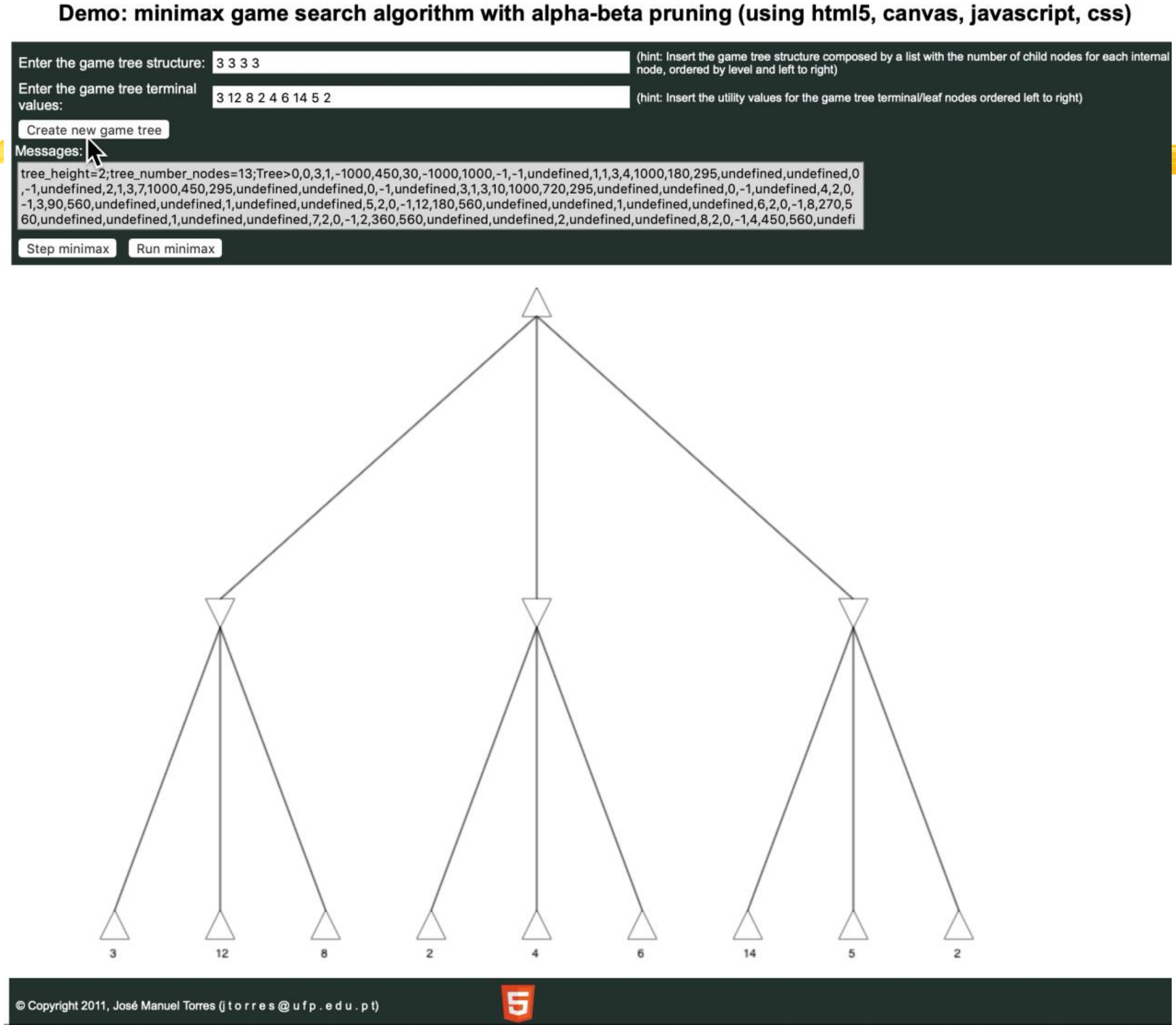
2. α-β pruning: search cutoff

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node

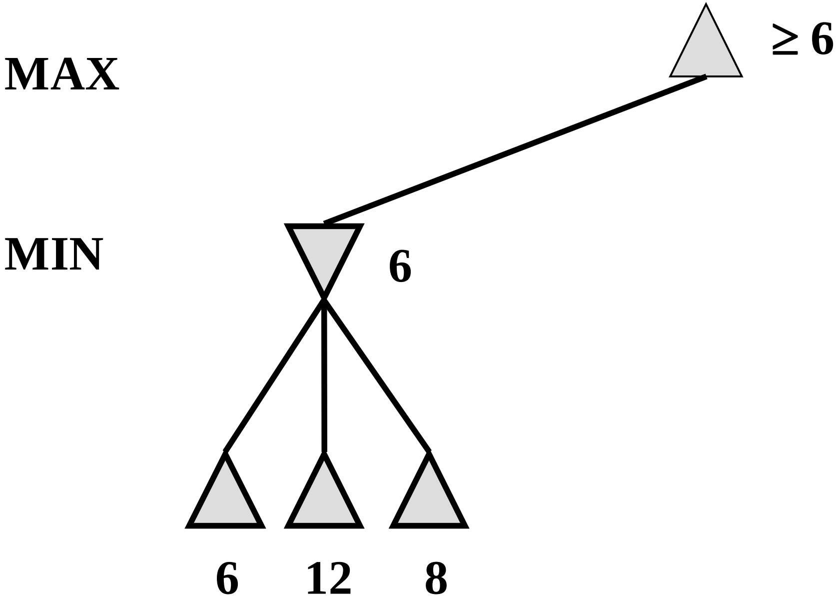
- **α-β pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.

- Does it work? Yes, in roughly cuts the branching factor from b to $\sqrt{b}$ resulting in double as far look-ahead than pure minimax http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html

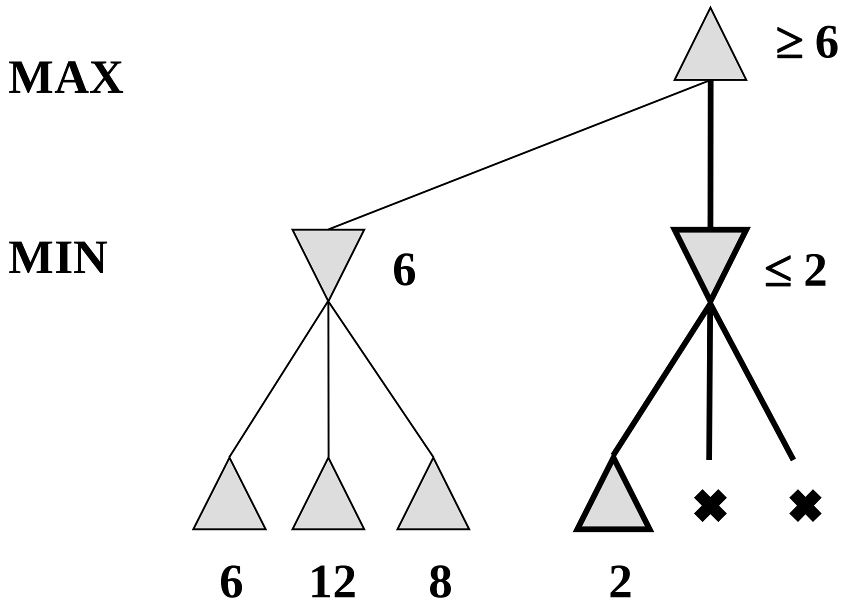
Demo



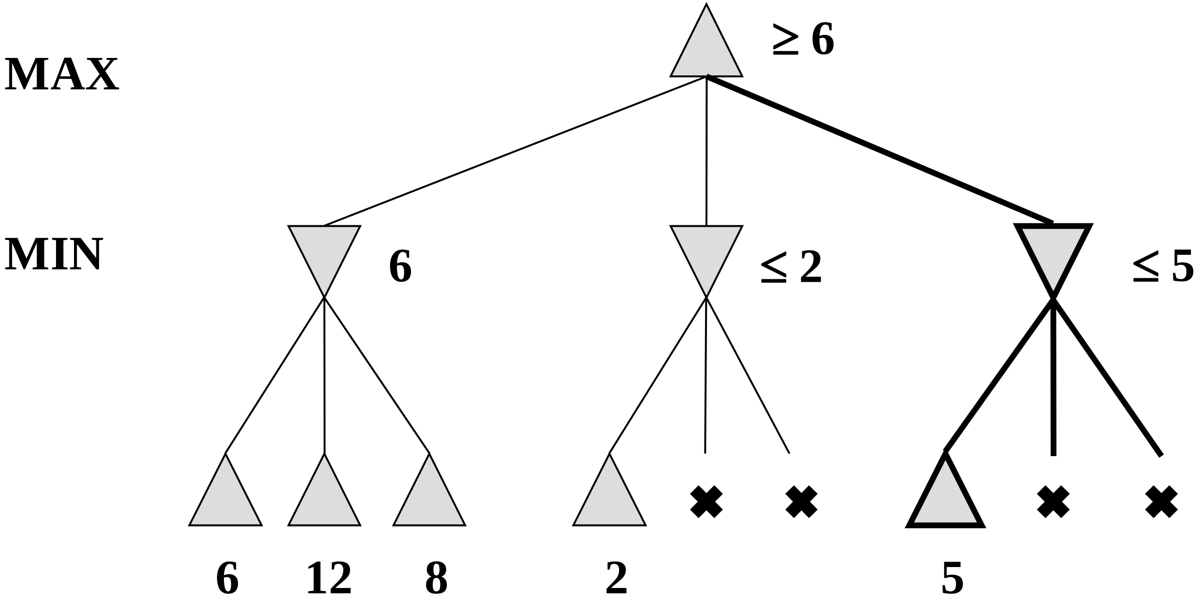
α-β pruning: example



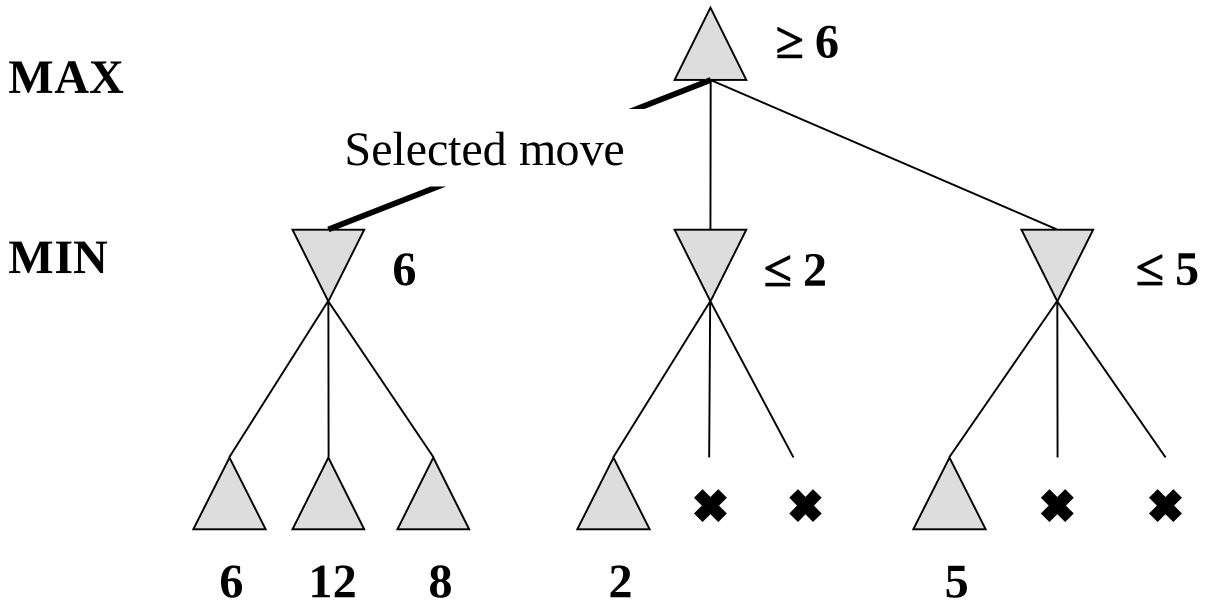
α-β pruning: example



α-β pruning: example

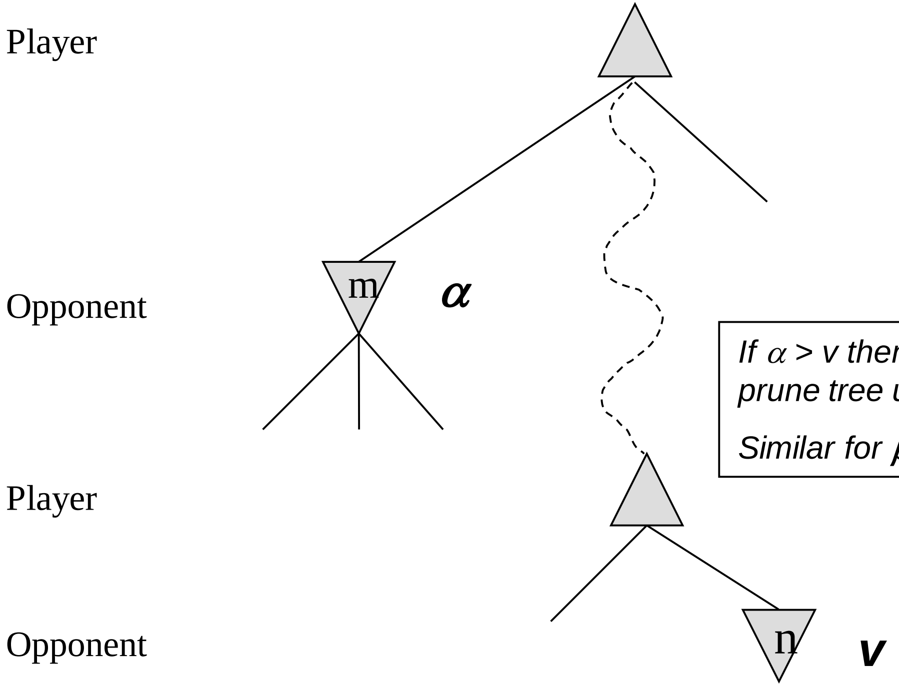


α-β pruning: example



Interactive demo:
https://www.yosenspace.com/posts/computer-science-game-trees.html

α-β pruning: general principle



If $\alpha > v$ then MAX will chose m so prune tree under n

Similar for β for MIN

Properties of α-β

Pruning *does not* affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$

$\Rightarrow$ *doubles* depth of search

$\Rightarrow$ can easily reach depth 8 and play good chess

A simple example of the value of reasoning about which computations are relevant (a form of *metareasoning*)

The α-β algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

More on the α-β algorithm

- Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution.

- We know a branch will not contain a solution once we know a better outcome has already been discovered in a previously explored branch.

Remember: Minimax: Recursive implementation

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

Complete: Yes, for finite state-space
Optimal: Yes

Time complexity: $O(b^m)$
Space complexity: $O(bm)$ (= DFS Does not keep all nodes in memory.)

The α-β algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    ~~**if** $v \geq \beta$ **then return** v~~
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    ~~**if** $v \leq \alpha$ **then return** v~~
    $\beta \leftarrow$ MIN(β, v)
  return v

More on the α-β algorithm

- Same basic idea as minimax, but prune (cut away) branches of the tree that we know will not contain the solution.

- Because minimax is depth-first, let's consider nodes along a given path in the tree. Then, as we go along this path, we keep track of:
  - α : Best choice so far for MAX
  - β : Best choice so far for MIN

The α-β algorithm

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

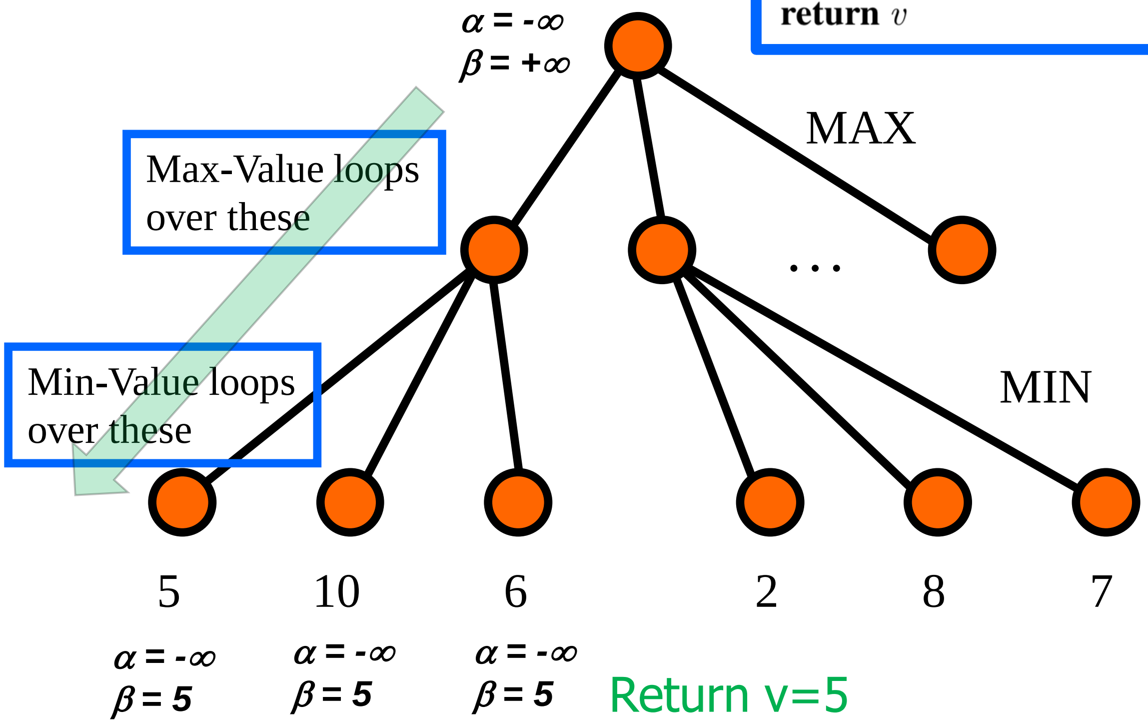
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Note: ***α* and *β*** are both Local variables. At the Start of the algorithm, We initialize them to ***α = -∞* and *β = +∞***

More on the α-β algorithm

In Min-Value:

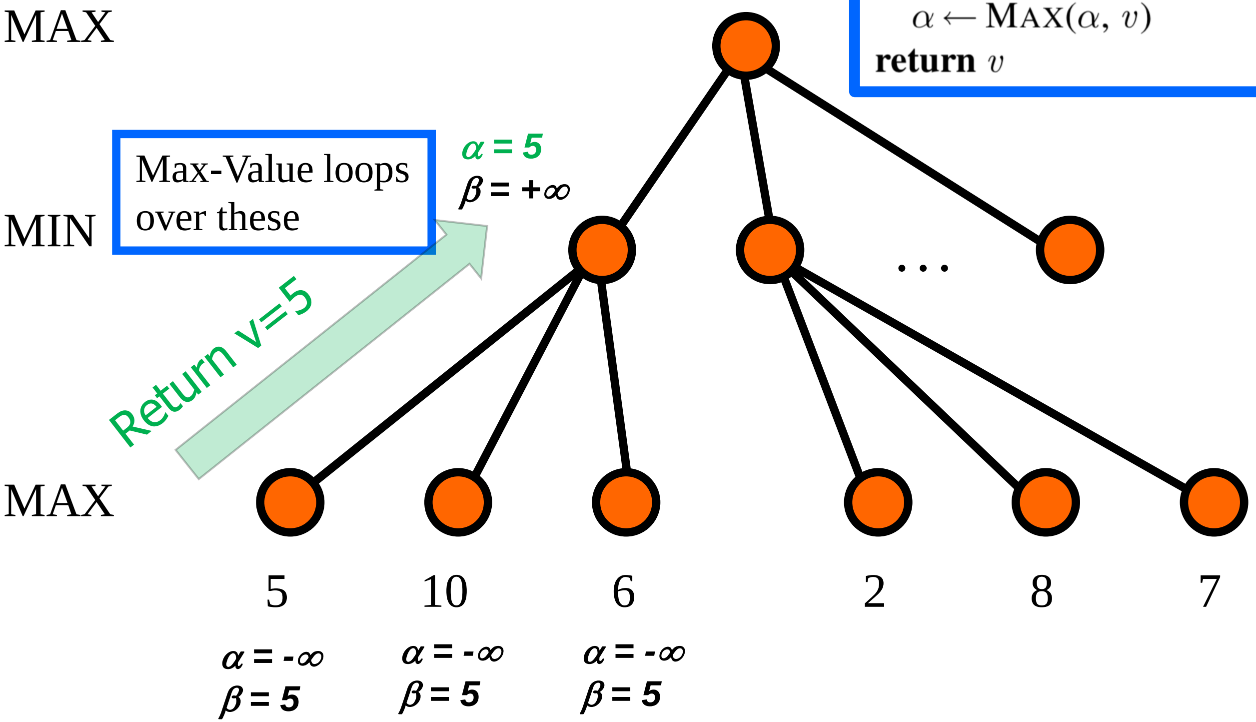
$v \leftarrow +\infty$
for each a **in** ACTIONS($state$) **do**
  $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
  if $v \leq \alpha$ **then return** v
  $\beta \leftarrow$ MIN(β, v)
return v



More on the α-β algorithm

In Max-Value:

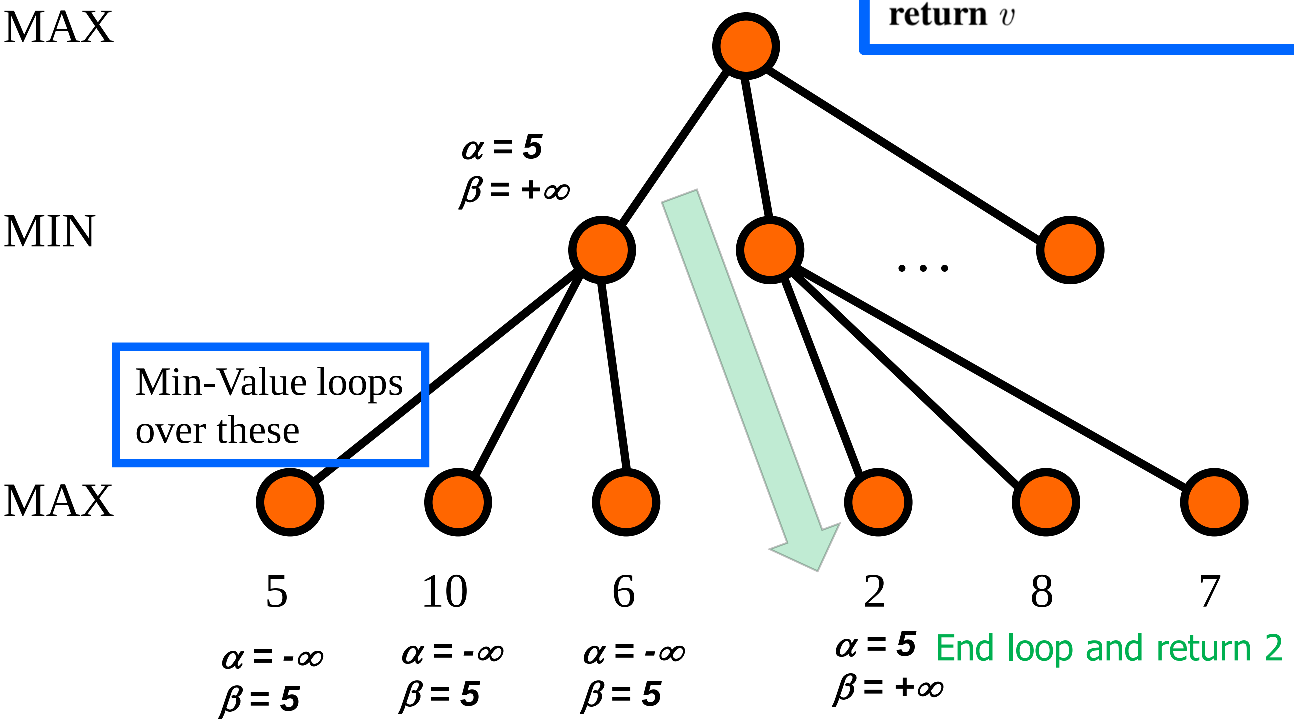
$v \leftarrow -\infty$
for each a **in** ACTIONS($state$) **do**
  $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
  if $v \geq \beta$ **then return** v
  $\alpha \leftarrow$ MAX(α, v)
return v



More on the α-β algorithm

In Min-Value:

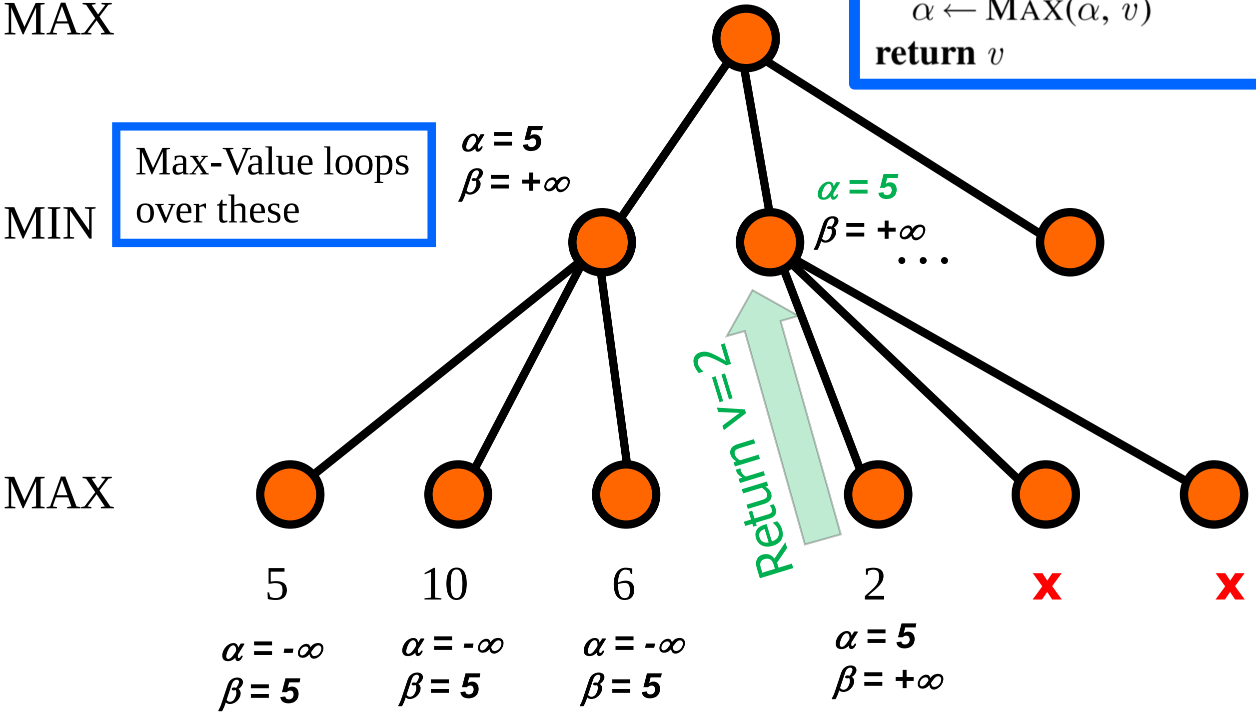
$v \leftarrow +\infty$
for each a **in** ACTIONS($state$) **do**
  $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) ,α, β))
  if $v \leq \alpha$ **then return** v
  $\beta \leftarrow$ MIN(β, v)
return v



More on the α-β algorithm

In Max-Value:

$v \leftarrow -\infty$
for each a **in** ACTIONS($state$) **do**
  $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
  if $v \geq \beta$ **then return** v
  $\alpha \leftarrow$ MAX(α, v)
return v

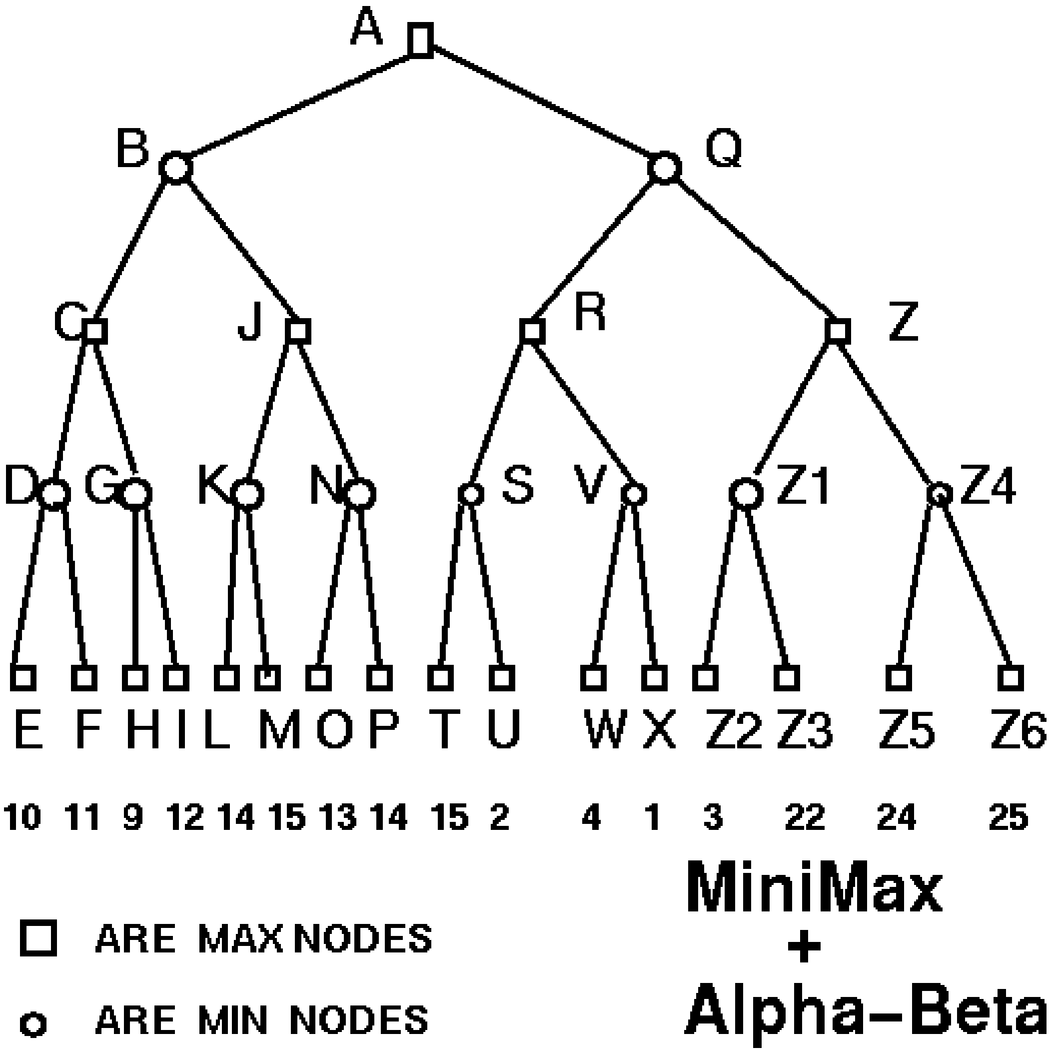


Another way to understand the algorithm

- For a given node N,

  α is the value of N to MAX
  β is the value of N to MIN

Example



MiniMax + Alpha-Beta

α-β algorithm: slight variant (from earlier version of textbook)

Basically MINIMAX + keep track of α, β + prune

```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
    inputs: state, current state in game
            game, game description
            α, the best score for MAX along the path to state
            β, the best score for MIN along the path to state

    if CUTOFF-TEST(state) then return EVAL(state)
    for each s in SUCCESSORS(state) do
        α ← MAX(α, MIN-VALUE(s, game, α, β))
        if α ≥ β then return β
    end
    return α
```

```
function MIN-VALUE(state, game, α, β) returns the minimax value of state

    if CUTOFF-TEST(state) then return EVAL(state)
    for each s in SUCCESSORS(state) do
        β ← MIN(β, MAX-VALUE(s, game, α, β))
        if β ≤ α then return α
    end
    return β
```

Is this wrong compared to latest version of textbook?

Please always use latest version of the algorithm as in 3rd or 4th edition of textbook.

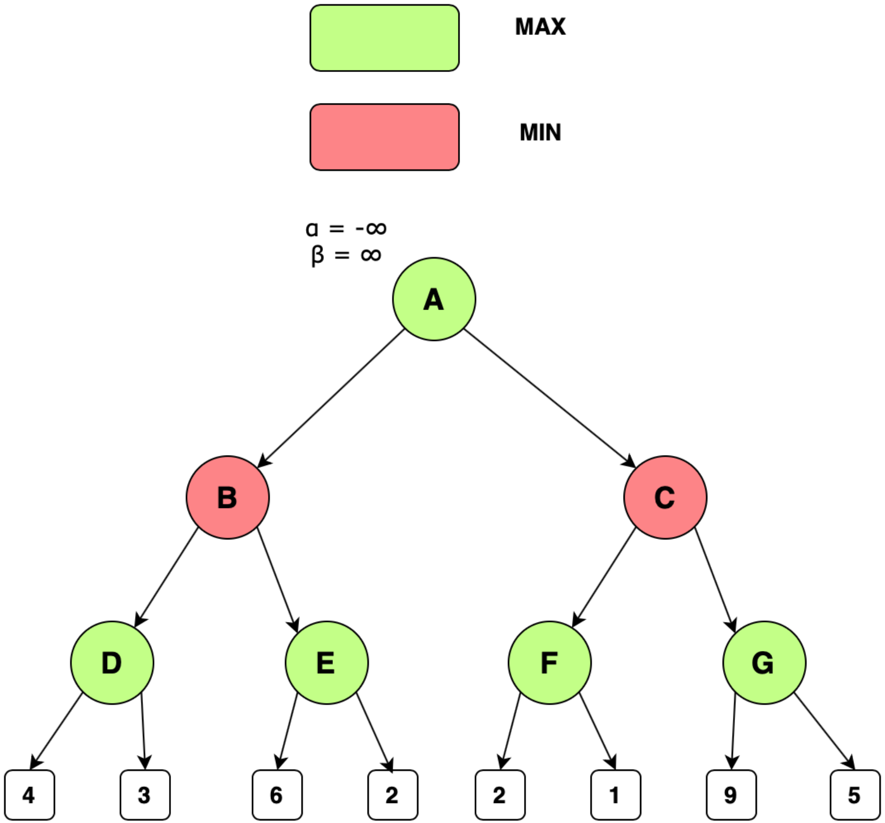
Solution

NODE	TYPE	ALPHA	BETA	SCORE
A	MAX	-Inf	Inf	
B	MIN	-Inf	Inf	
C	MAX	-Inf	Inf	
D	MIN	-Inf	Inf	
E	MAX	10	10	10
D	MIN	-Inf	10	
F	MAX	11	11	11
D	MIN	-Inf	10	10
C	MAX	10	Inf	
G	MIN	10	Inf	
H	MAX	9	9	9
G	MIN	10	9	9
C	MAX	10	Inf	10
B	MIN	-Inf	10	
J	MAX	-Inf	10	
K	MIN	-Inf	10	
L	MAX	14	14	14
K	MIN	-Inf	10	
M	MAX	15	15	15
K	MIN	-Inf	10	10
…				

NODE	TYPE	ALPHA	BETA	SCORE
…				
J	MAX	10	10	10
B	MIN	-Inf	10	10
A	MAX	10	Inf	
Q	MIN	10	Inf	
R	MAX	10	Inf	
S	MIN	10	Inf	
T	MAX	15	15	15
S	MIN	10	15	
U	MAX	2	2	2
S	MIN	10	2	2
R	MAX	10	Inf	
V	MIN	10	Inf	
W	MAX	4	4	4
V	MIN	10	4	4
R	MAX	10	Inf	10
Q	MIN	10	10	10
A	MAX	10	Inf	10

https://www.educative.io/answers/what-is-alpha-beta-pruning

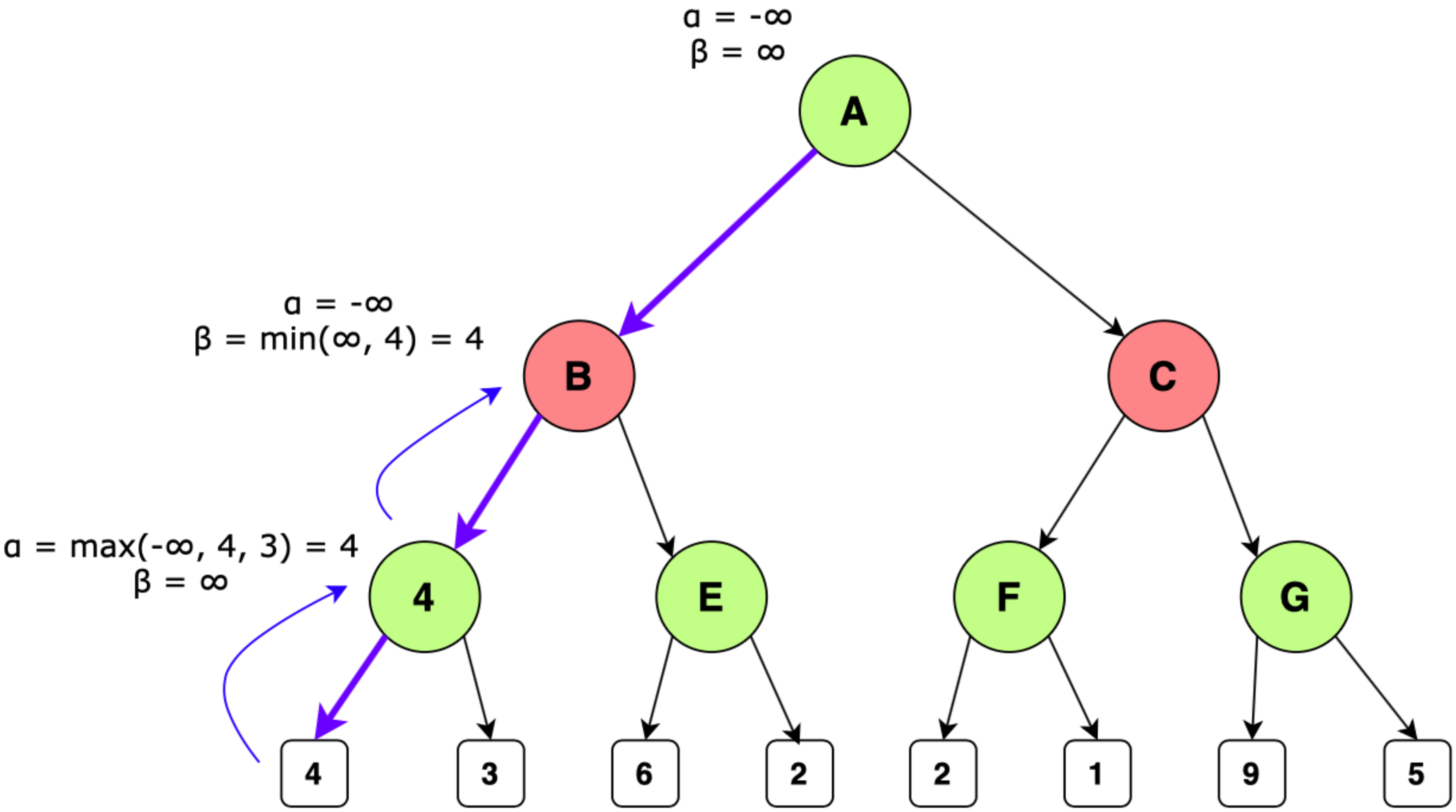
Another example



The values of alpha and beta are passed down until they reach a leaf node – their new value is decided by whichever player's turn it is. The minimum value is chosen if it is min's turn and the maximum value chosen if it is max's turn.

Another example

https://www.educative.io/answers/what-is-alpha-beta-pruning



We move down, depth first, until we reach the left most leaf node. Here, we have to explore both the leaf nodes to determine the maximum value for max's turn. The maximum value is found to be 4, which is the value that alpha is set at. Now, the node's value, i.e., 4 is used to back track. When we reach node B, the value of beta is updated to become 4 as well.

The α-β algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE(*state*, $-\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

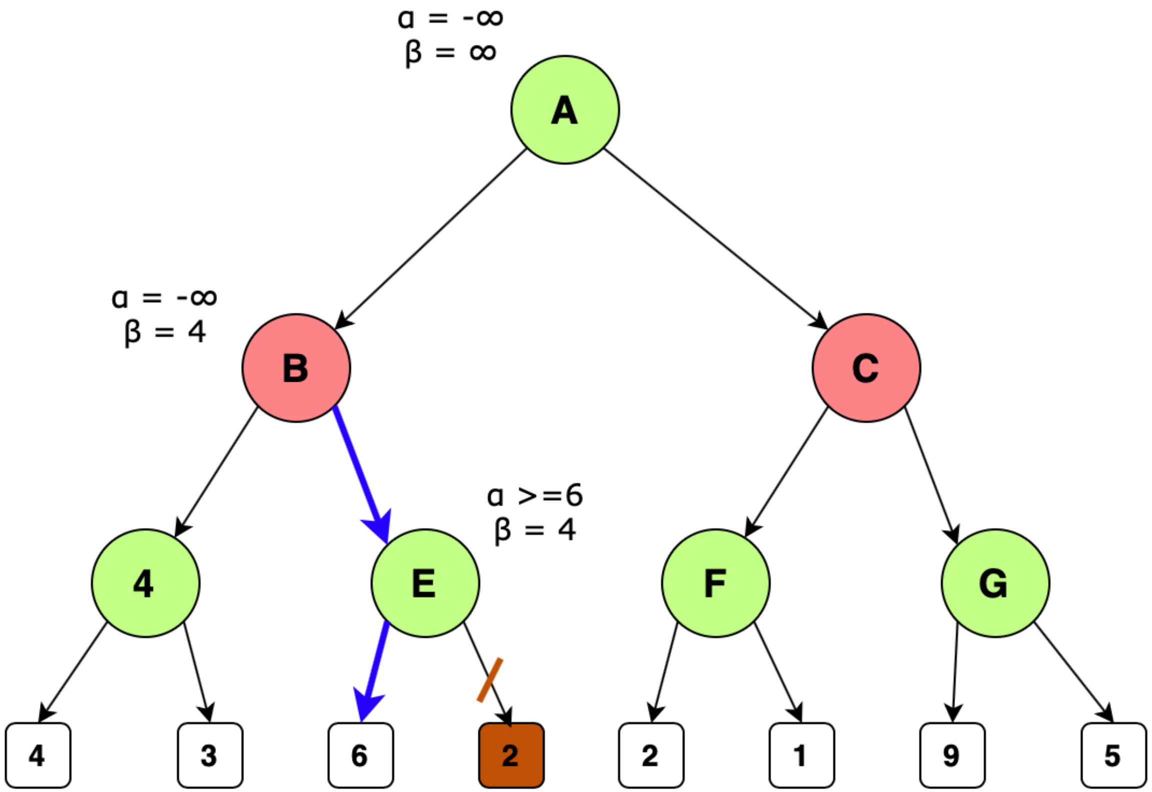
---

function MIN-VALUE(*state*, α, β) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Note: ***α and β*** are both Local variables. At the Start of the algorithm, We initialize them to ***$\alpha = -\infty$ and $\beta = +\infty$***

https://www.educative.io/answers/what-is-alpha-beta-pruning

Another example



Now, moving towards the next branch, we reach the leaf node with value 6. The turn is of max, therefore, it will choose the maximum value from its children node. When we explore 6, we realize that the value of alpha will either be 6 or greater than 6. We do not need to explore the other child node since the value of alpha is now greater than beta. So 2 is pruned.

The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in ACTIONS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

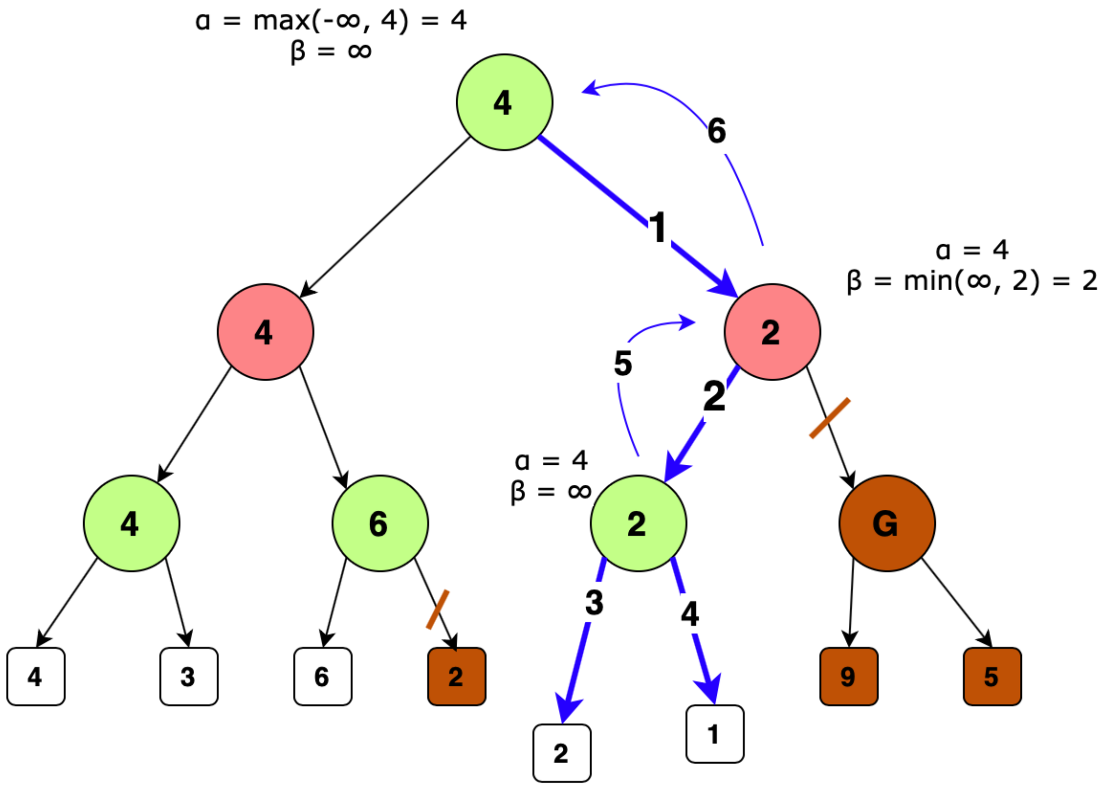
```
function MIN-VALUE(state, α, β) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Note: ***α* and *β*** are both Local variables. At the Start of the algorithm, We initialize them to

α = -∞* and *β = +∞

Another example

https://www.educative.io/answers/what-is-alpha-beta-pruning



B's node value becomes 4 as it is the minimizer and min(4,6) = 4. This value is passed up and the value of alpha is changed by the maximizer to become 4 as well. Now moving towards the next leaf node, we follow the blue path to reach 2. We need to explore both the branches to find the maximum value for the maximizer, which is 2. This node value is passed up and the minimizer changes beta's value to 2. Now alpha is again greater than beta so the entire right branch is pruned.

State-of-the-art for deterministic games

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

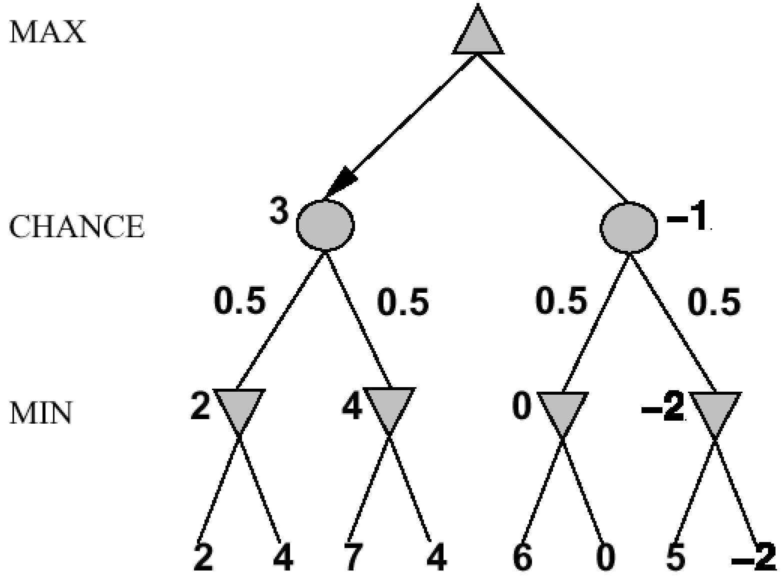
Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

Nondeterministic games

E..g, in backgammon, the dice rolls determine the legal moves

Simplified example with coin-flipping instead of dice-rolling:



Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

. . .
if *state* is a chance node **then**
    return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(*state*)
. . .

A version of α–β pruning is possible
but only if the leaf values are bounded. Why??

Remember: Minimax algorithm

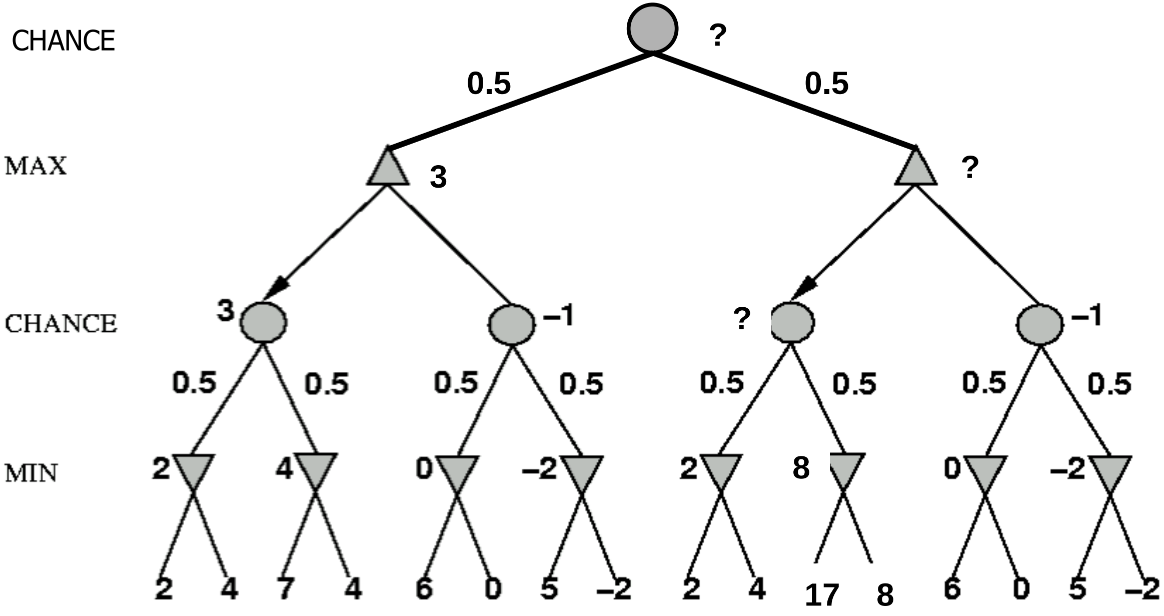
```
function MINIMAX-DECISION(state) returns an action
   return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s, a)))
   return v
```

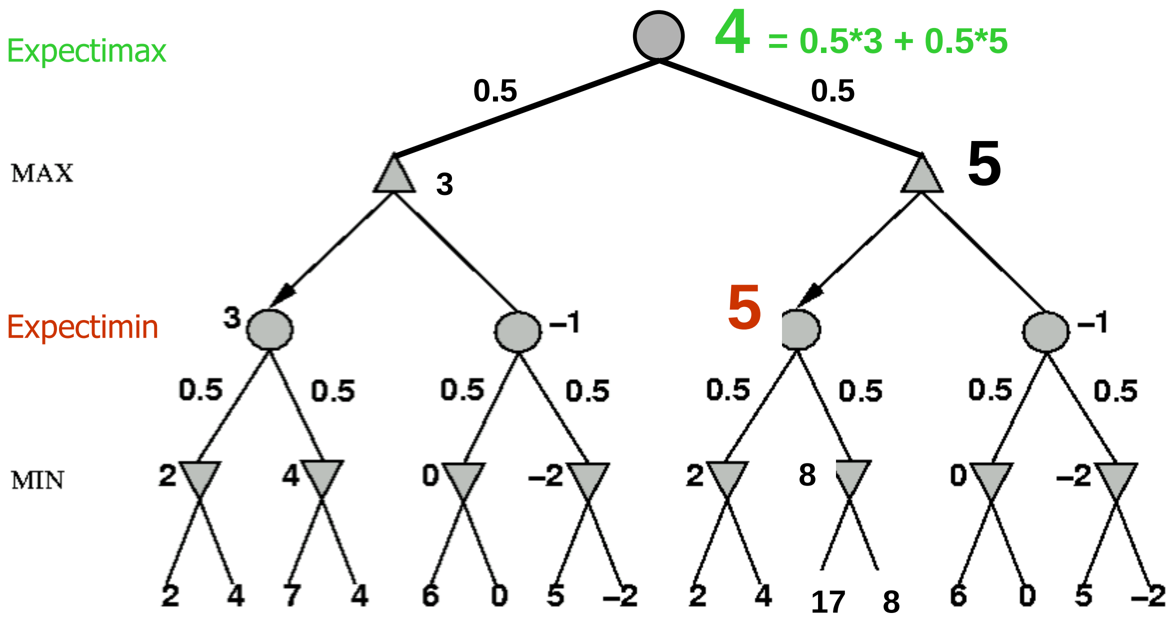
```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s, a)))
   return v
```

Nondeterministic games: the element of chance

expectimax and **expectimin**, expected values over all possible outcomes

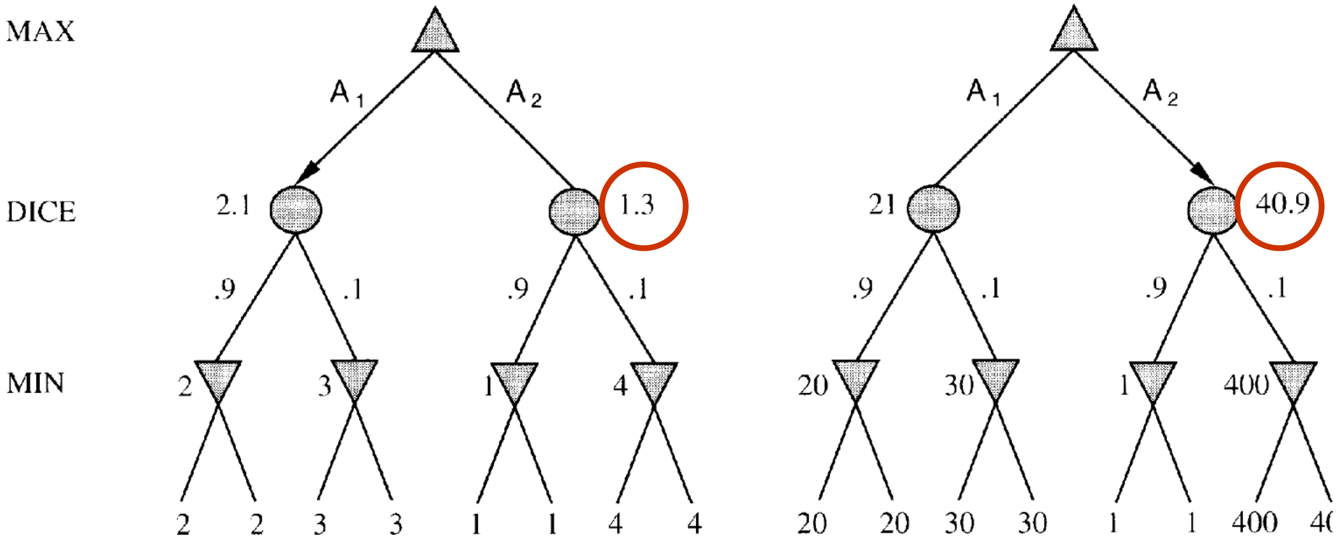


Nondeterministic games: the element of chance



Evaluation functions: Exact values DO matter

Order-preserving transformations do not necessarily behave the same!



State-of-the-art for nondeterministic games

Dice rolls increase b: 21 possible rolls with 2 dice

Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)

$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks

$\Rightarrow$ value of lookahead is diminished

α–β pruning is much less effective

Summary

Games are fun to work on! (and dangerous)

They illustrate several important points about AI

◊ perfection is unattainable ⇒ must approximate

◊ good idea to think about what to think about

◊ uncertainty constrains the assignment of values to states

Games are to AI as grand prix racing is to automobile design

Exercise: Game Playing

Consider the following game tree in which the evaluation function values are shown below each leaf node. Assume that the root node corresponds to the maximizing player. Assume the search always visits children left-to-right.

(a) Compute the backed-up values computed by the minimax algorithm. Show your answer by writing values at the appropriate nodes in the above tree.

(b) Compute the backed-up values computed by the alpha-beta algorithm. What nodes will not be examined by the alpha-beta pruning algorithm?

(c) What move should Max choose once the values have been backed-up all the way?