Games and Adversarial Search

Artificial Intelligence

Slides are mostly adapted from AIMA, MIT Open Courseware and Svetlana Lazebnik (UIUC)

World Champion chess player Garry Kasparov is defeated by IBM’s Deep Blue chess-playing computer in a six-game match in May, 1997
(link)

Why study games?

- Games are a traditional hallmark of intelligence
- Games are easy to formalize
- Games can be a good model of real-world competitive or cooperative activities
  - Military confrontations, negotiation, auctions, etc.

Games – history of chess playing

- 1949 – Shannon paper – originated the ideas
- 1951 – Turing paper – hand simulation
- 1958 – Bernstein program
- 1955-1960 – Simon-Newell program
- 1961 – Soviet program
- 1966 – 1967 – MacHack 6 – defeated a good player
- 1970s – NW chess 4.5
- 1980s – Cray Bitz
- 1990s – Belle, Hitech, Deep Thought,
- 1997 - Deep Blue - defeated Garry Kasparov

Games

- Multi agent environments : any given agent will need to consider the actions of other agents and how they affect its own welfare.
- The unpredictability of these other agents can introduce many possible contingencies
- There could be competitive or cooperative environments

- Competitive environments, in which the agent’s goals are in conflict require adversarial search – these problems are called as games

Games vs. single-agent search

- We don’t know how the opponent will act
  - The solution is not a fixed sequence of actions from start state to goal state, but a ***strategy*** or ***policy*** (a mapping from state to best move in that state)
- Efficiency is critical to playing well
  - The time to make a move is limited
  - The branching factor, search depth, and number of terminal configurations are huge
    - In chess, branching factor ≈ 35 and depth ≈ 100, giving a search tree of 10^{154} nodes
      - Number of atoms in the observable universe $\approx 10^{80}$
  - This rules out searching all the way to the end of the game

Types of game environments

	Deterministic	**Stochastic**
Perfect information (fully observable)	Chess, checkers, go	Backgammon, monopoly
Imperfect information (partially observable)	Battleships	Scrabble, poker, bridge

Games

- In game theory (economics), any multiagent environment (either cooperative or competitive) is a game provided that the impact of each agent on the other is significant
- AI games are a specialized kind - deterministic, turn taking, two-player, zero sum games of perfect information
- In our terminology – deterministic, fully observable environments with two agents whose actions alternate and the utility values at the end of the game are always equal and opposite (+1 and –1)

Alternating two-player zero-sum games

- Players take turns
- Each game outcome or **terminal state** has a **utility** for each player (e.g., 1 for win, -1 for loss, 0 for draw)
- The sum of both players' utilities is a constant

Game Tree search

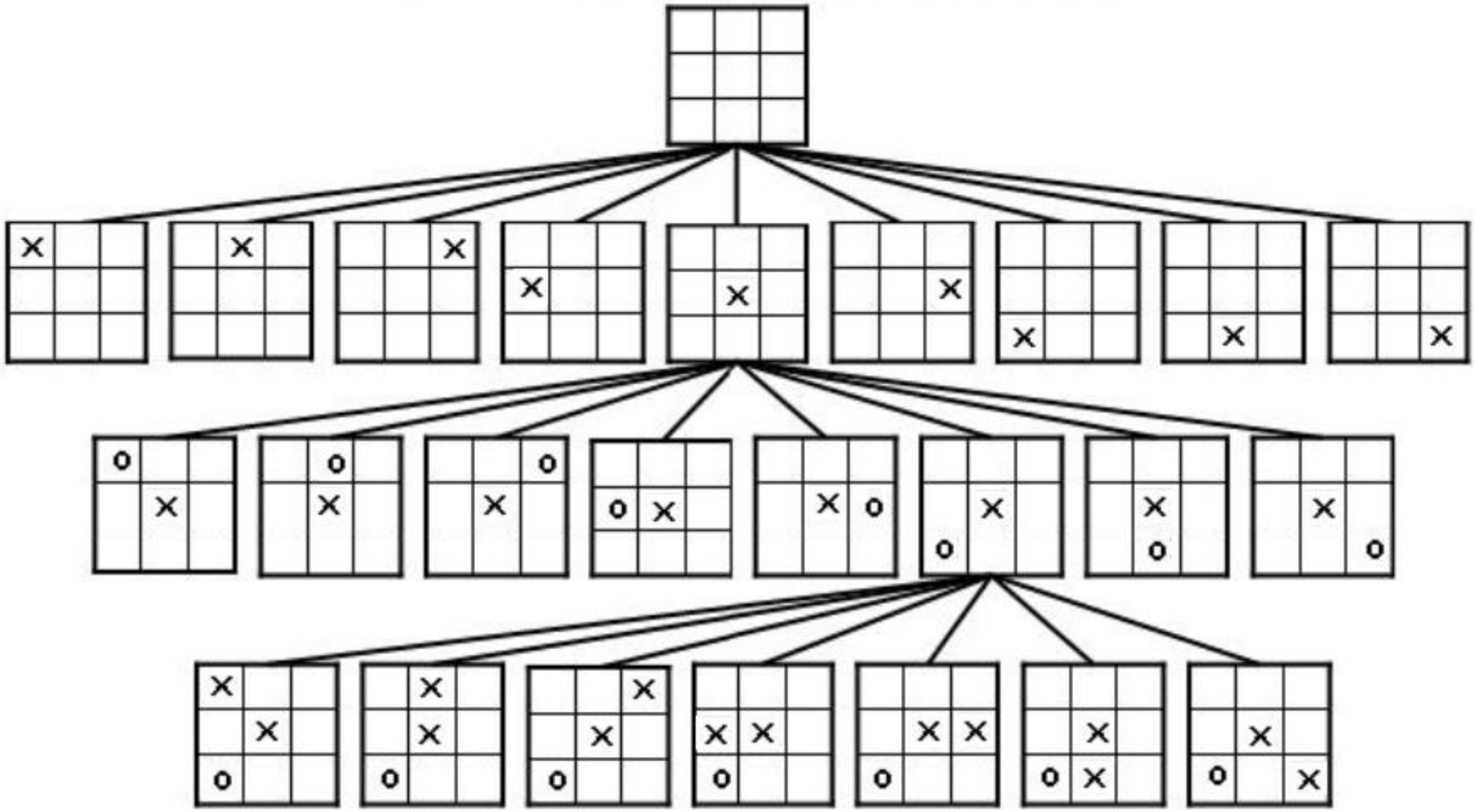
- Initial state: initial board position and player
- Operators: one for each legal move
- Goal states: winning board positions
- Scoring function: assigns numeric value to states
- Game tree: encodes all possible games

- We are not looking for a path, only the next move to make (that hopefully leads to a winning position)
- Our best move depends on what the other player does

Optimal strategies

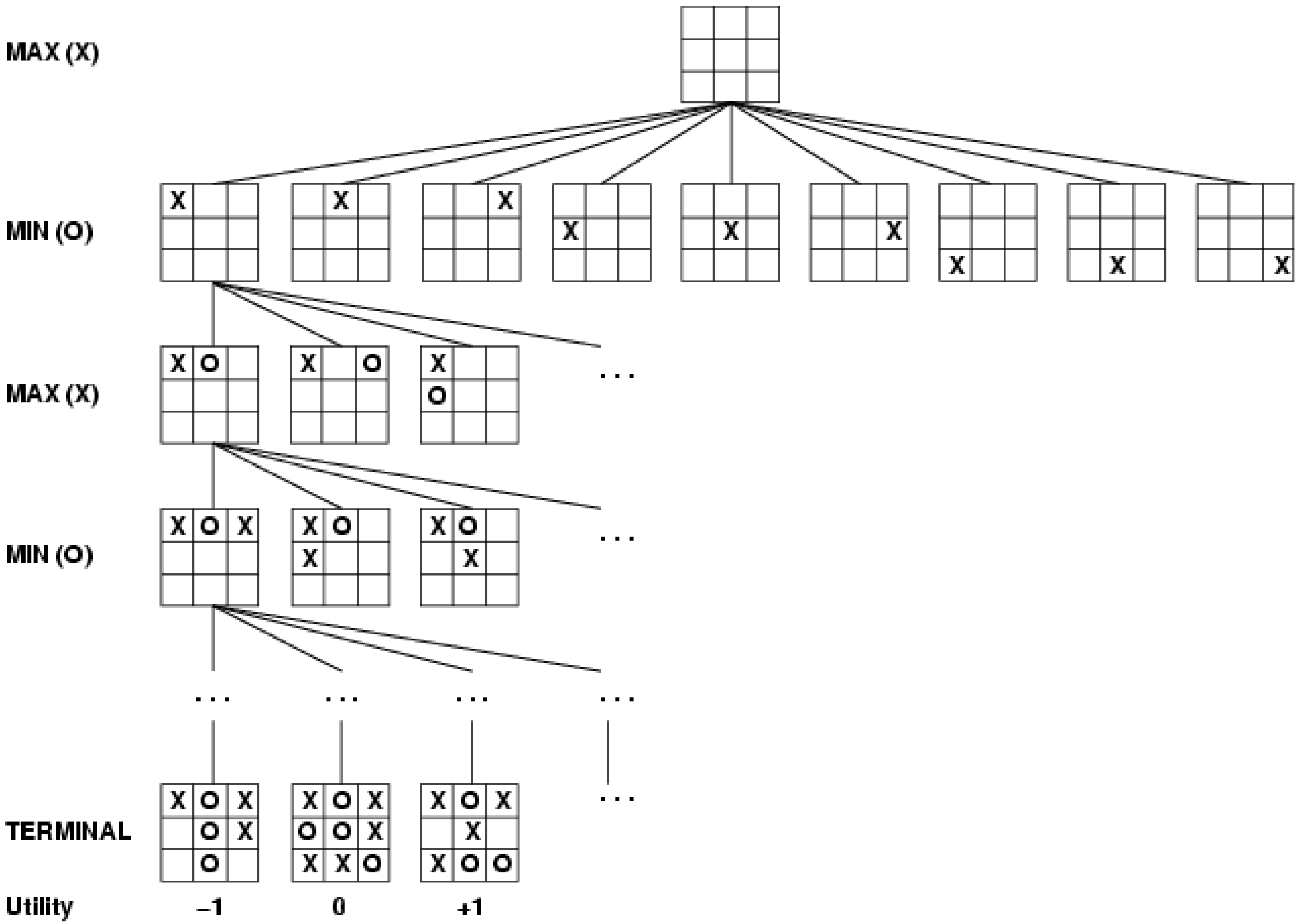
- In a normal search problem, the optimal solution would be a sequence of moves leading to a goal state - a terminal state that is a win
- In a game, MIN has something to say about it and therefore MAX must find a contingent strategy, which specifies
  - MAX's move in the initial state,
  - then MAX's moves in the states resulting from every possible response by MIN,
  - then MAX's moves in the states resulting from every possible response by MIN to those moves
  - …
- An optimal strategy leads to outcomes at least as good as any other strategy when one is playing an infallible opponent

Partial Game Tree for Tic-Tac-Toe



Game tree

- A game of tic-tac-toe between two players, “max” and “min”

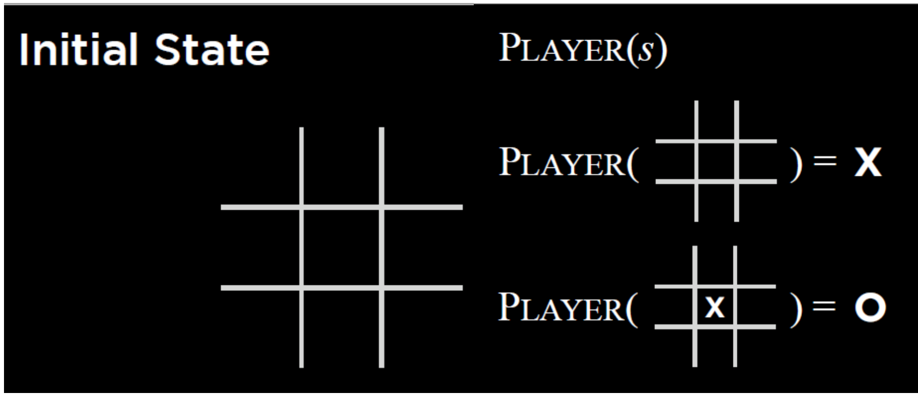


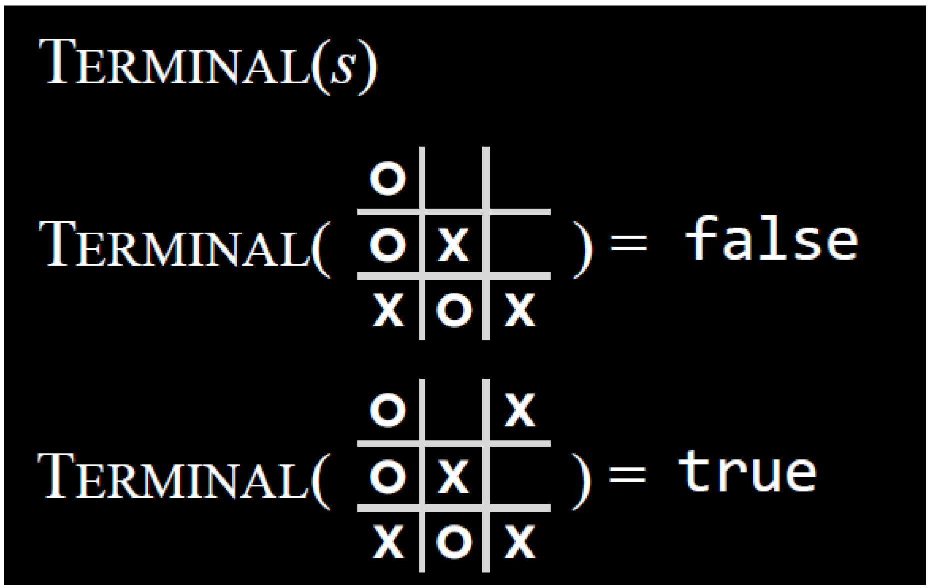
MAX (X) aims to maximize score.
MIN (O) aims to minimize score.

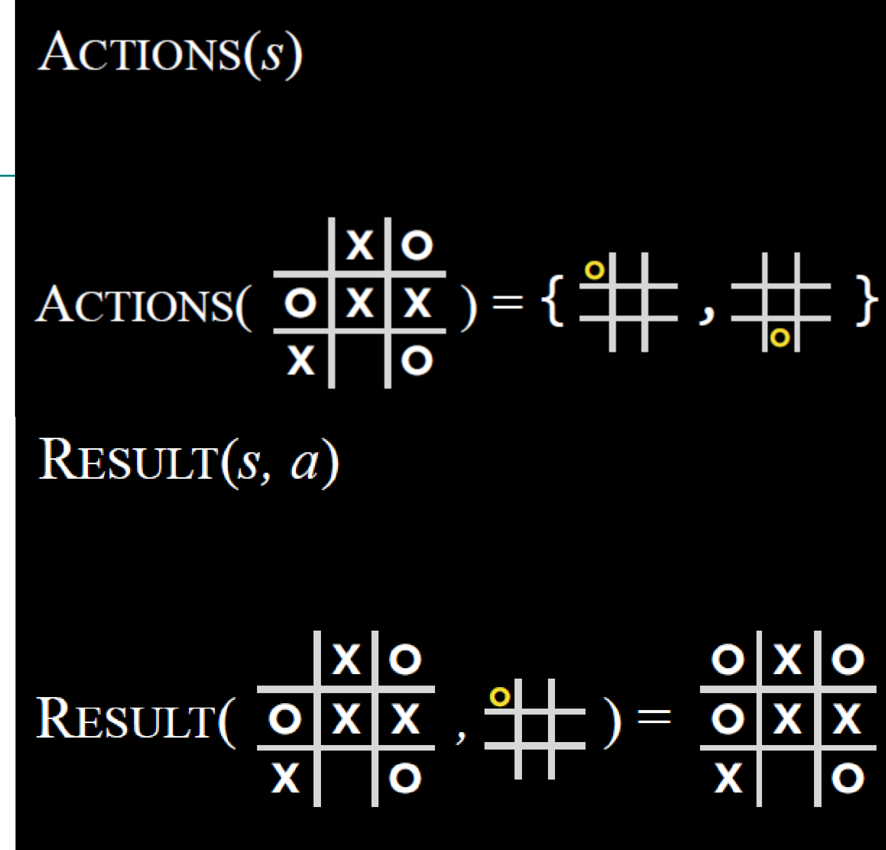
MINIMAX

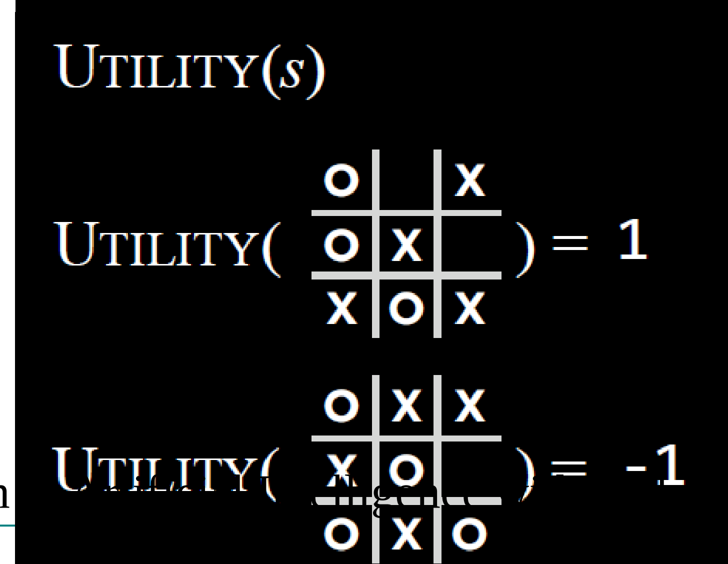
*S*0 : initial state
PLAYER(*s*) : returns which player to move in state *s*
ACTIONS(*s*) : returns legal moves in state *s*
RESULT(*s, a*) : returns state after action *a* taken in state *s*
TERMINAL(*s*) : checks if state *s* is a terminal state
UTILITY(*s*) : final numerical value for terminal state *s*

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



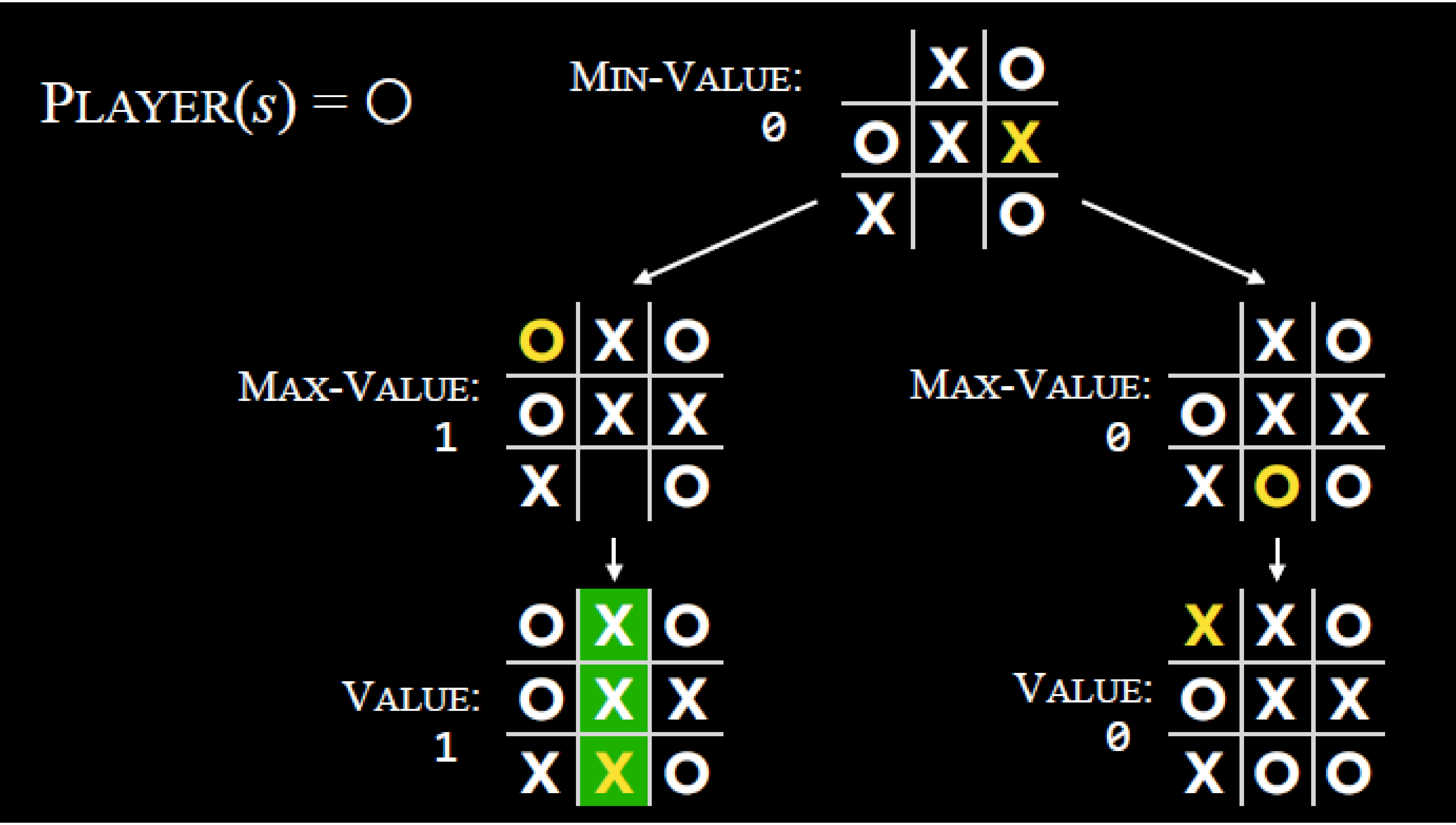




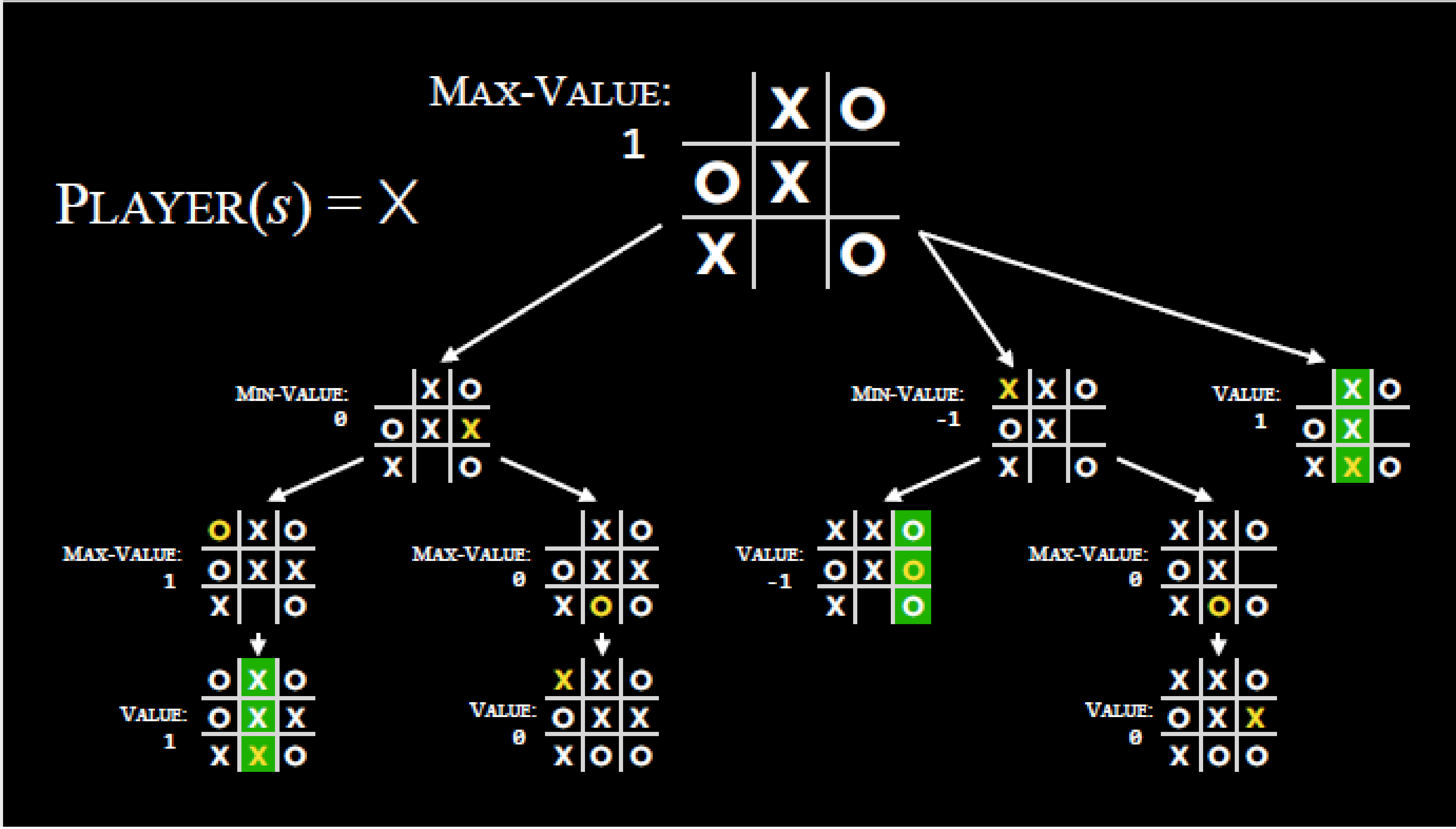


Slide credit : HarvardX CS50AICS50's Introduction

David J. Malan and Brian Yu



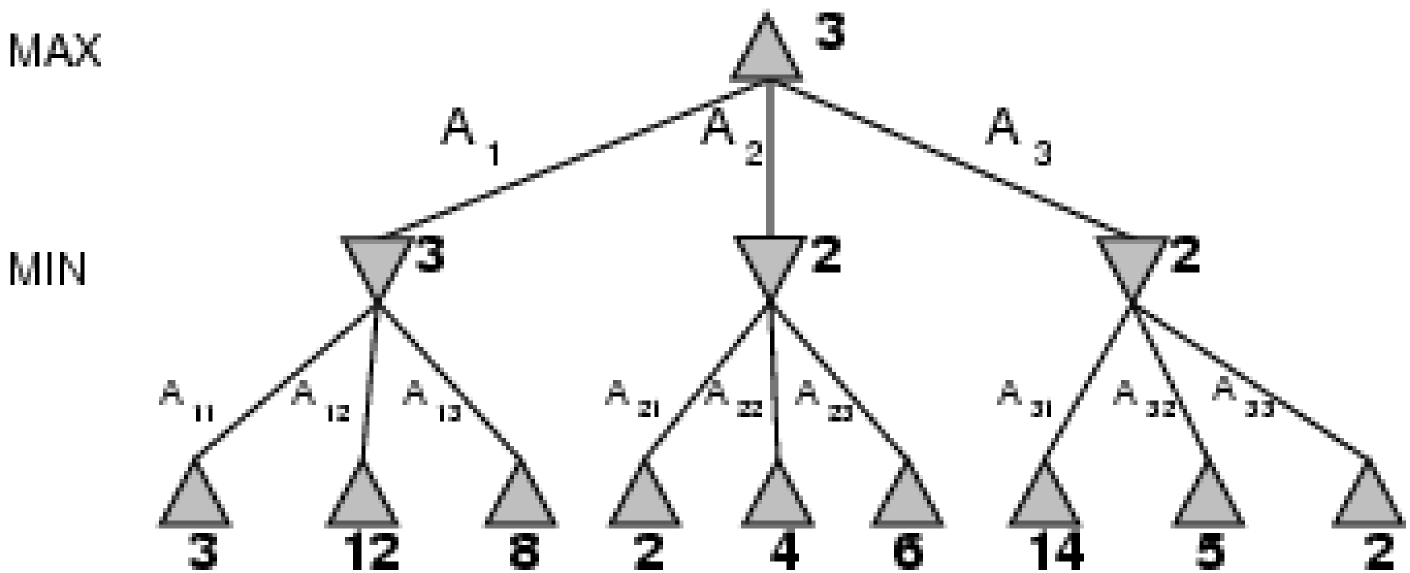
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Minimax

- Perfect play for deterministic games
- Idea: choose move to position with highest minimax value
  = best achievable payoff against best play
- E.g., 2-ply game:



Minimax value

- Given a game tree, the optimal strategy can be determined by examining the minimax value of each node (MINIMAX-VALUE(n))
- The minimax value of a node is the utility of being in the corresponding state, assuming that both players play optimally from there to the end of the game
- Given a choice, MAX prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value

Minimax algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*

  $v \leftarrow$ MAX-VALUE(*state*)
  return the *action* in SUCCESSORS(*state*) with value v

---

function MAX-VALUE(*state*) **returns** *a utility value*

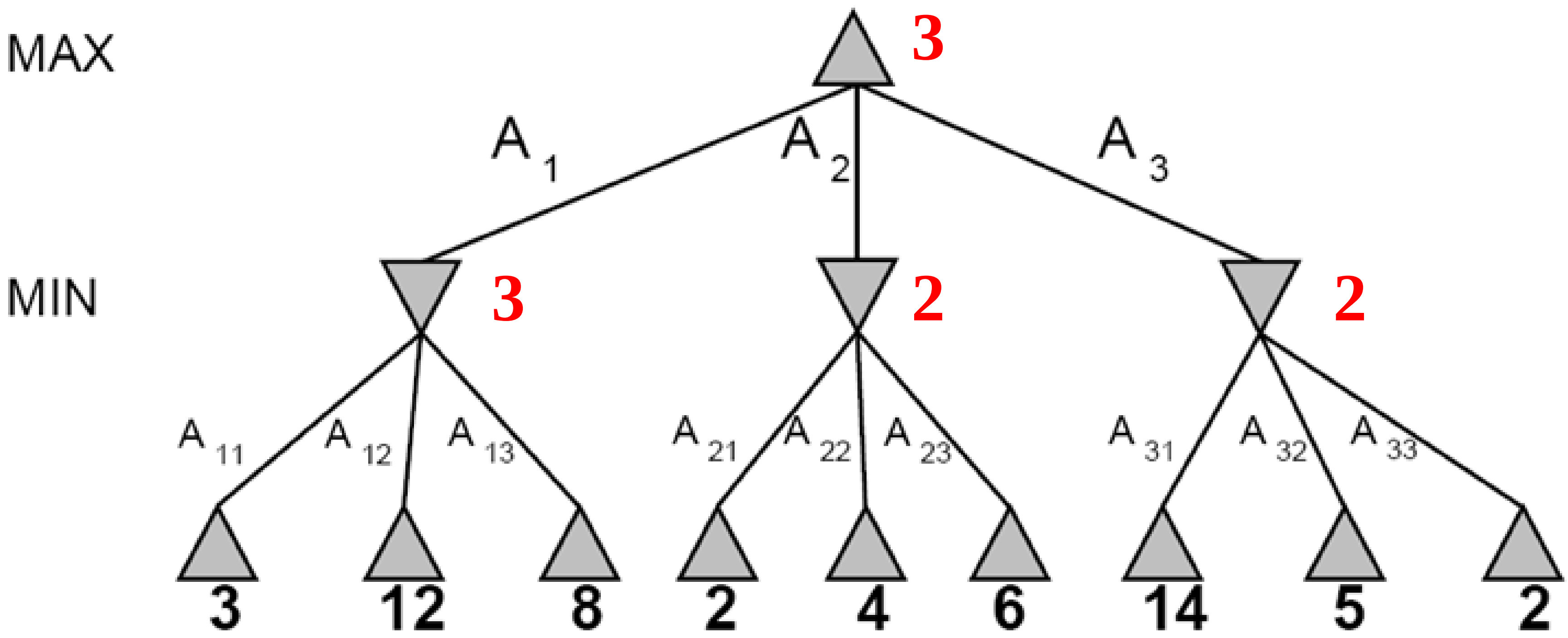
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(s))
  return v

Game tree search

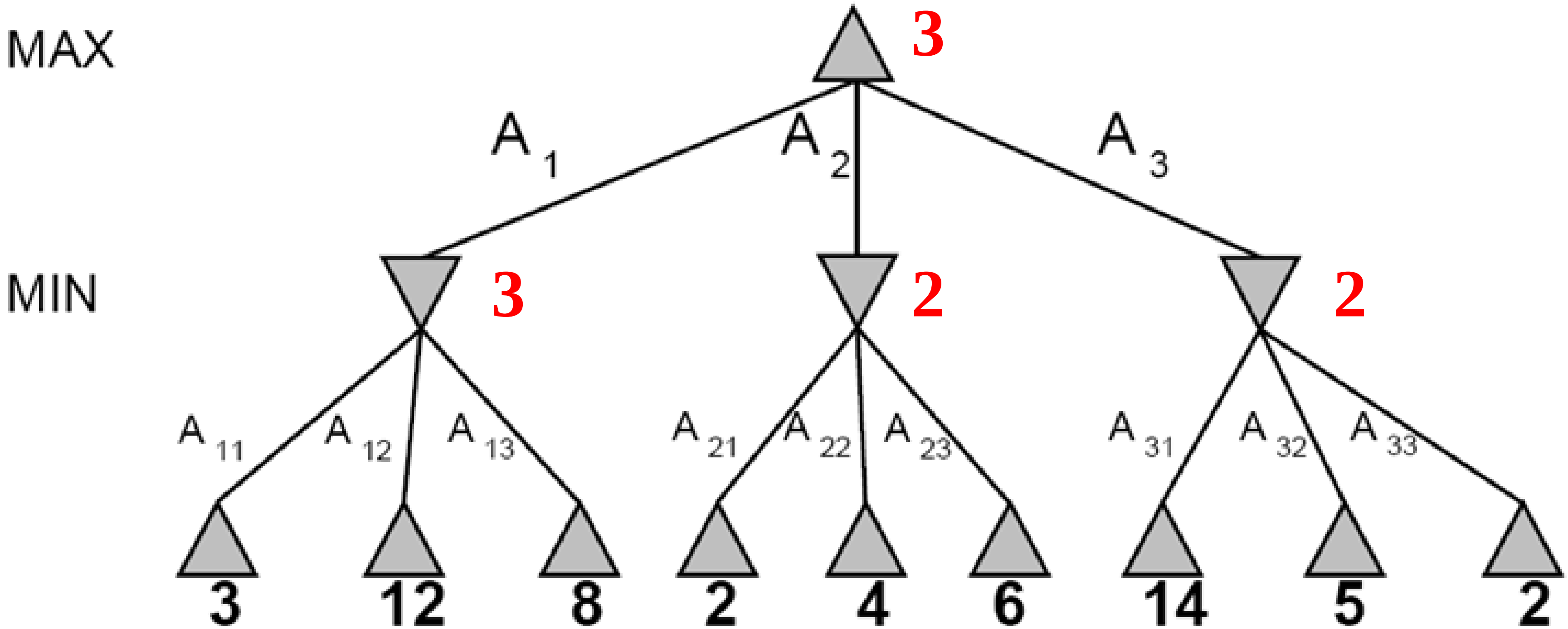


- **Minimax value of a node**: the utility (for MAX) of being in the corresponding state, assuming perfect play on both sides
- **Minimax strategy:** Choose the move that gives the best worst-case payoff

MINIMAX-VALUE(root) = max(min(3,12,8), min(2,4,6), min(14,5,2))

= max(3,2,2)

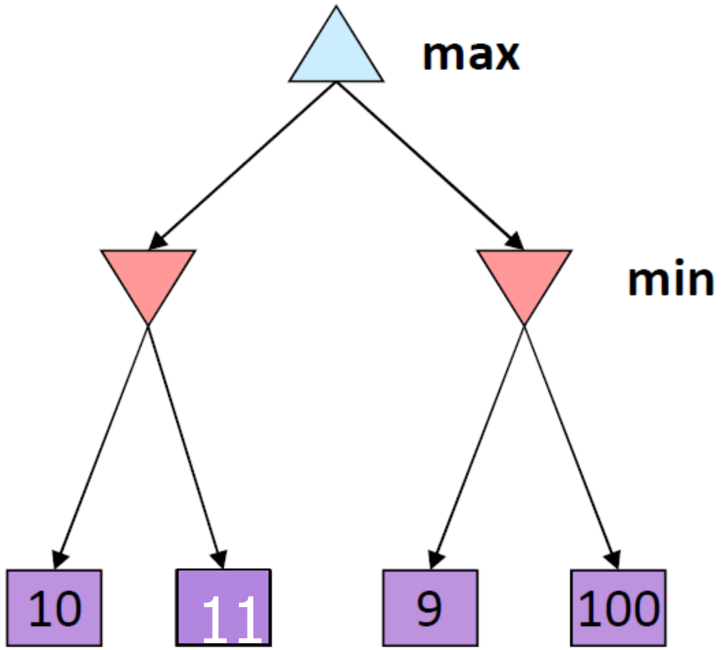
= 3



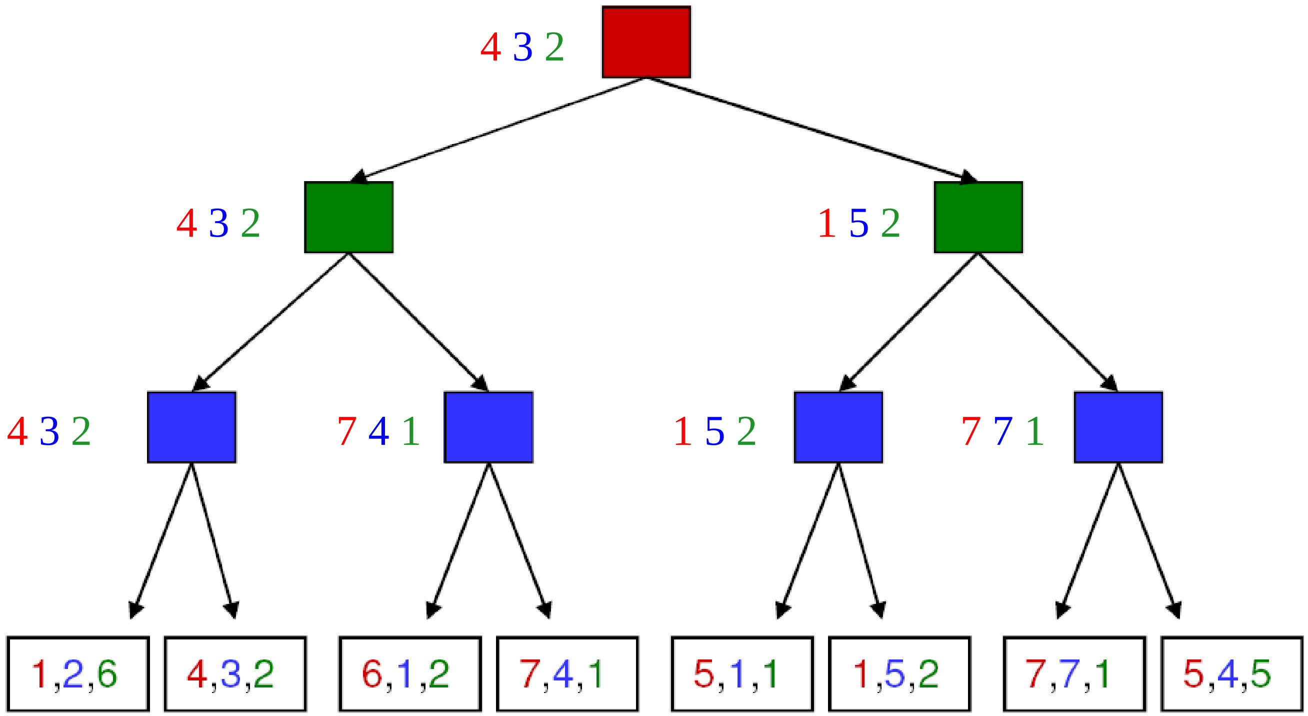
- **Minimax**(*node*) =
  - Utility(*node*) if *node* is terminal
  - $\max_{action}$ **Minimax**(Succ(*node*, *action*)) if *player* = MAX
  - $\min_{action}$ **Minimax**(Succ(*node*, *action*)) if *player* = MIN

Optimality of minimax

- The minimax strategy is optimal against an optimal opponent
- What if your opponent is suboptimal?
  - Your utility can only be higher than if you were playing an optimal opponent!
  - A different strategy may work better for a sub-optimal opponent, but it will necessarily be worse against an optimal opponent

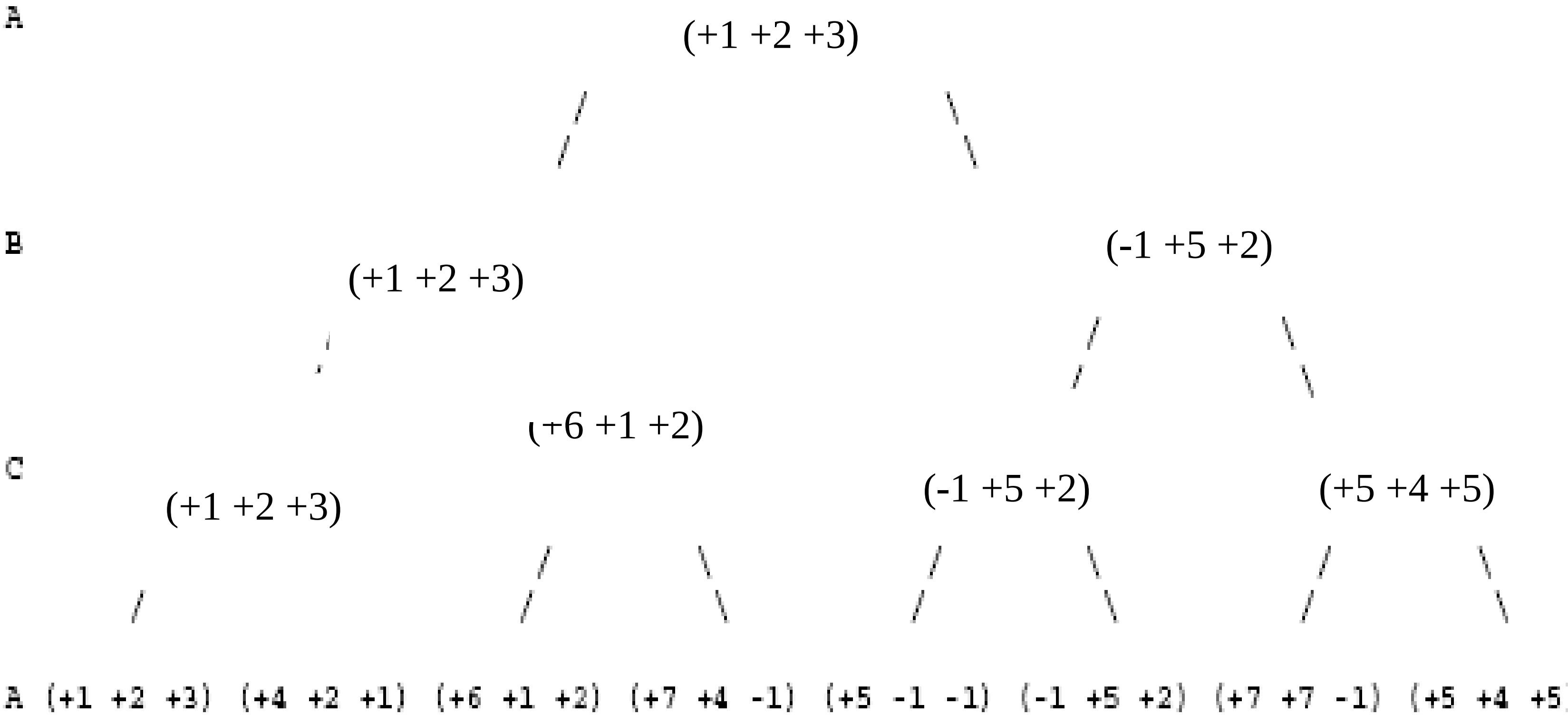


More general games



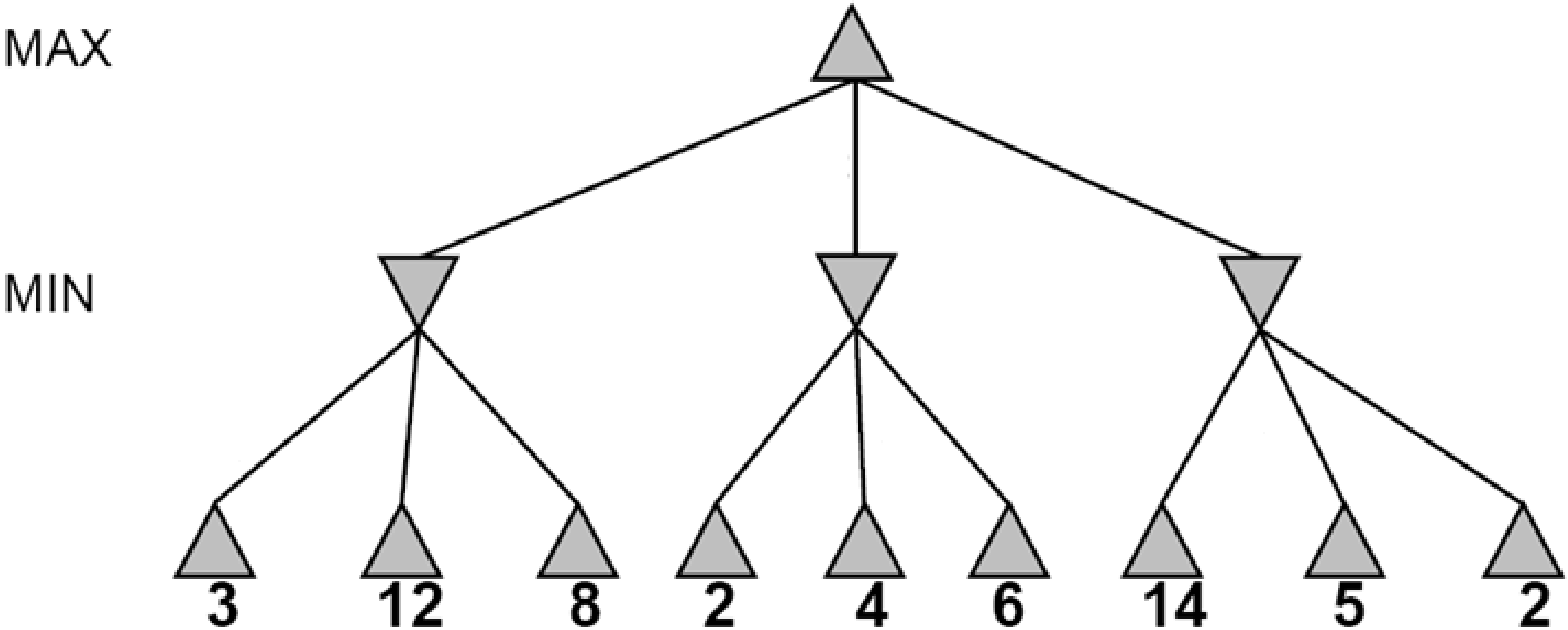
- More than two players, non-zero-sum
- Utilities are now tuples
- Each player maximizes their own utility at their node
- Utilities get propagated (*backed up*) from children to parents

Tree Player and Non-zero sum games



Alpha-beta pruning

- It is possible to compute the exact minimax decision without expanding every node in the game tree



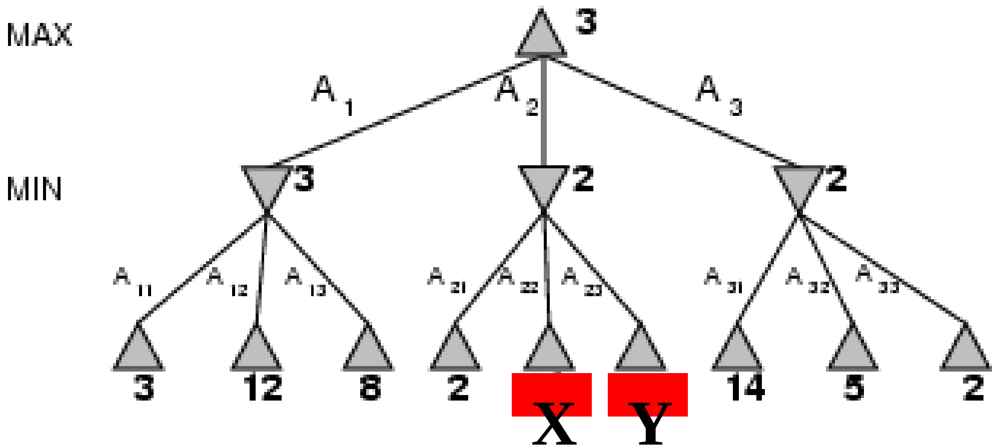
α-β pruning

MINIMAX-VALUE(root) = max(min(3,12,8), min(2,x,y), min(14,5,2))

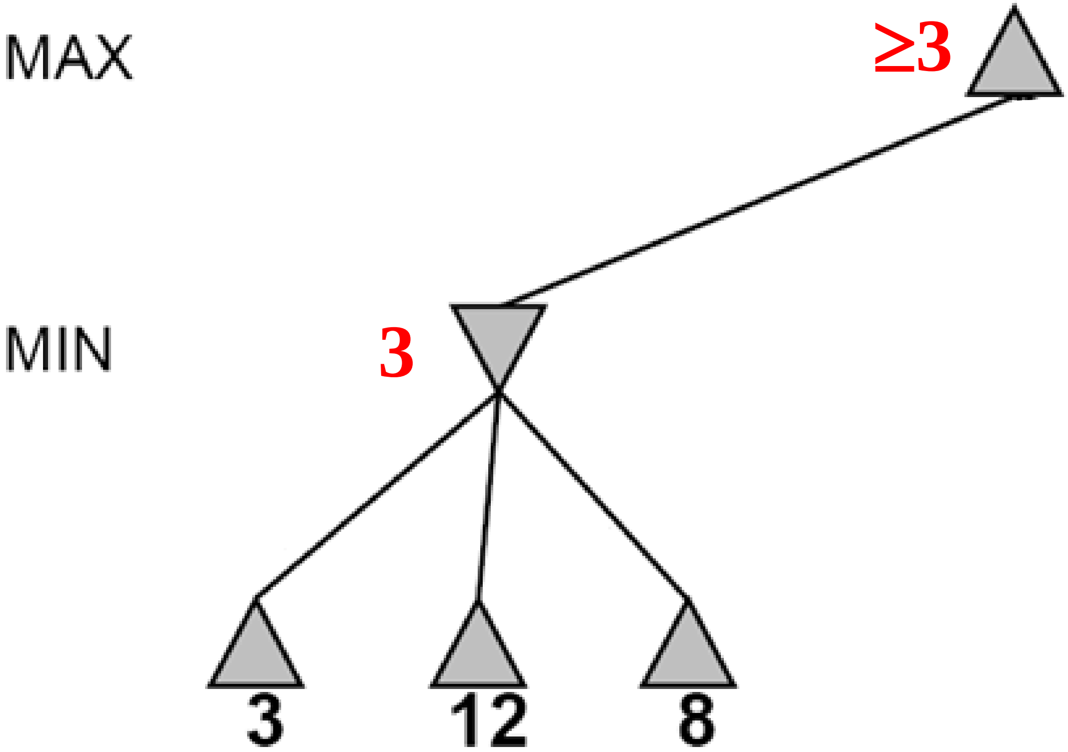
= max(3,min(2,x,y),2)

= max(3,z,2)   where z <=2

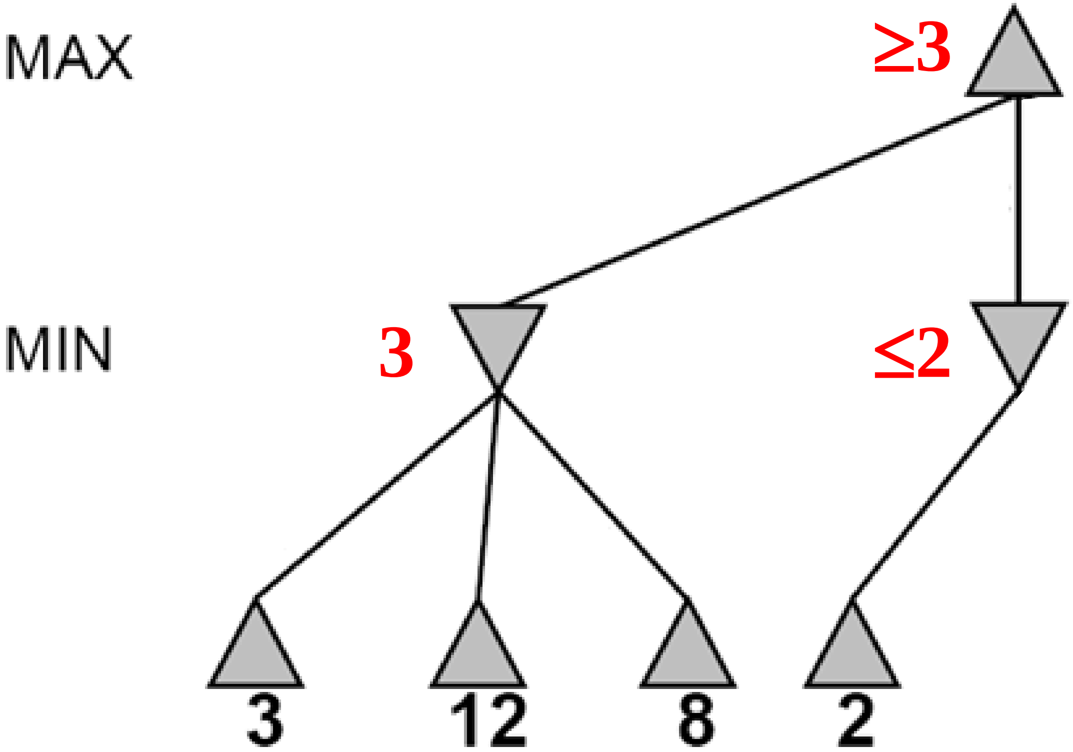
= 3



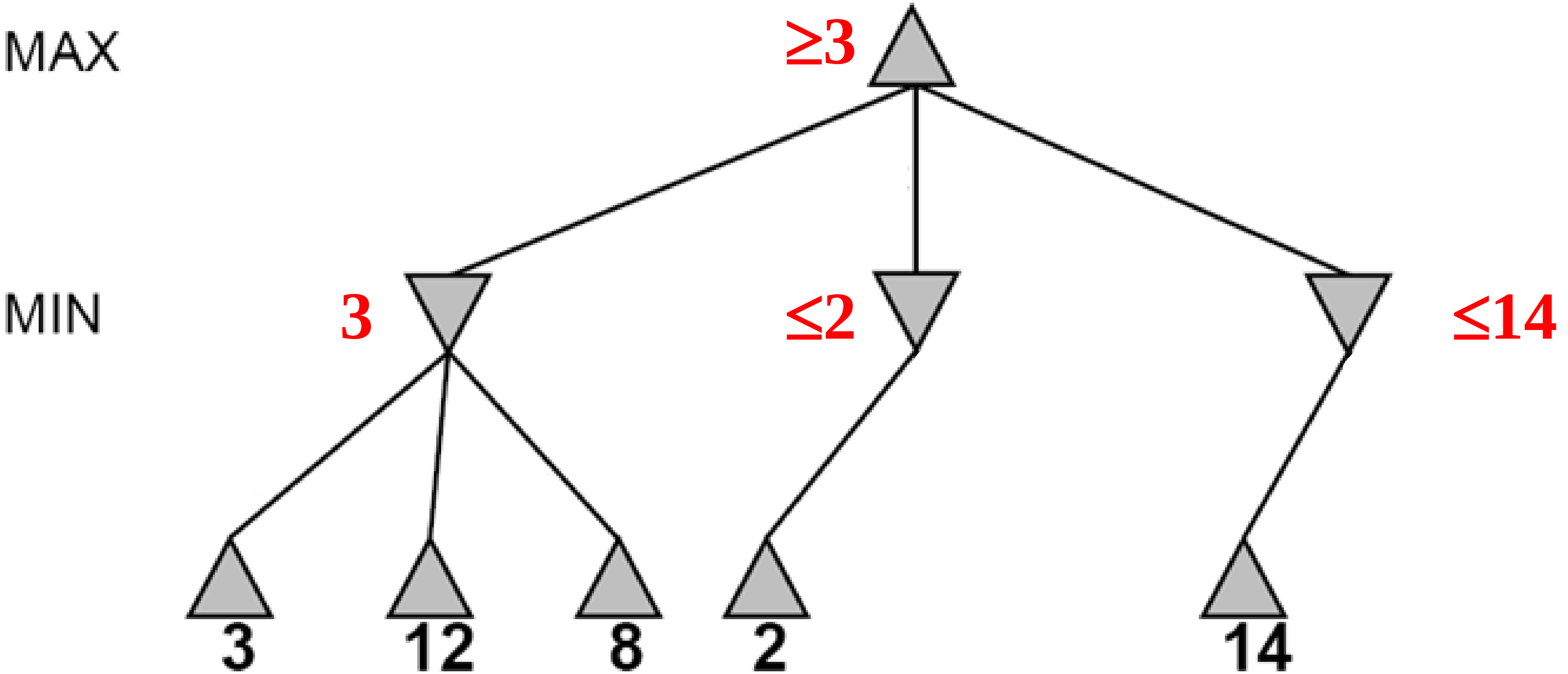
Alpha-beta pruning



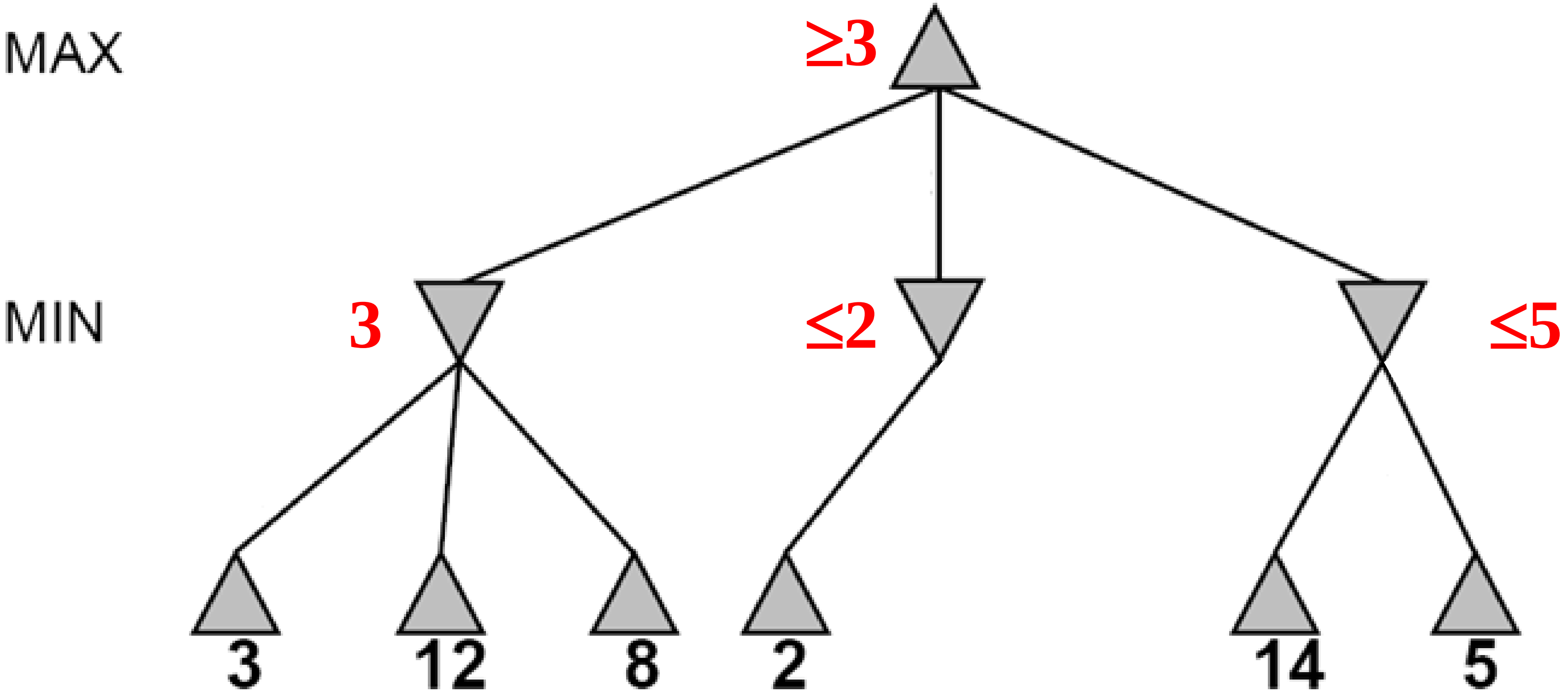
Alpha-beta pruning



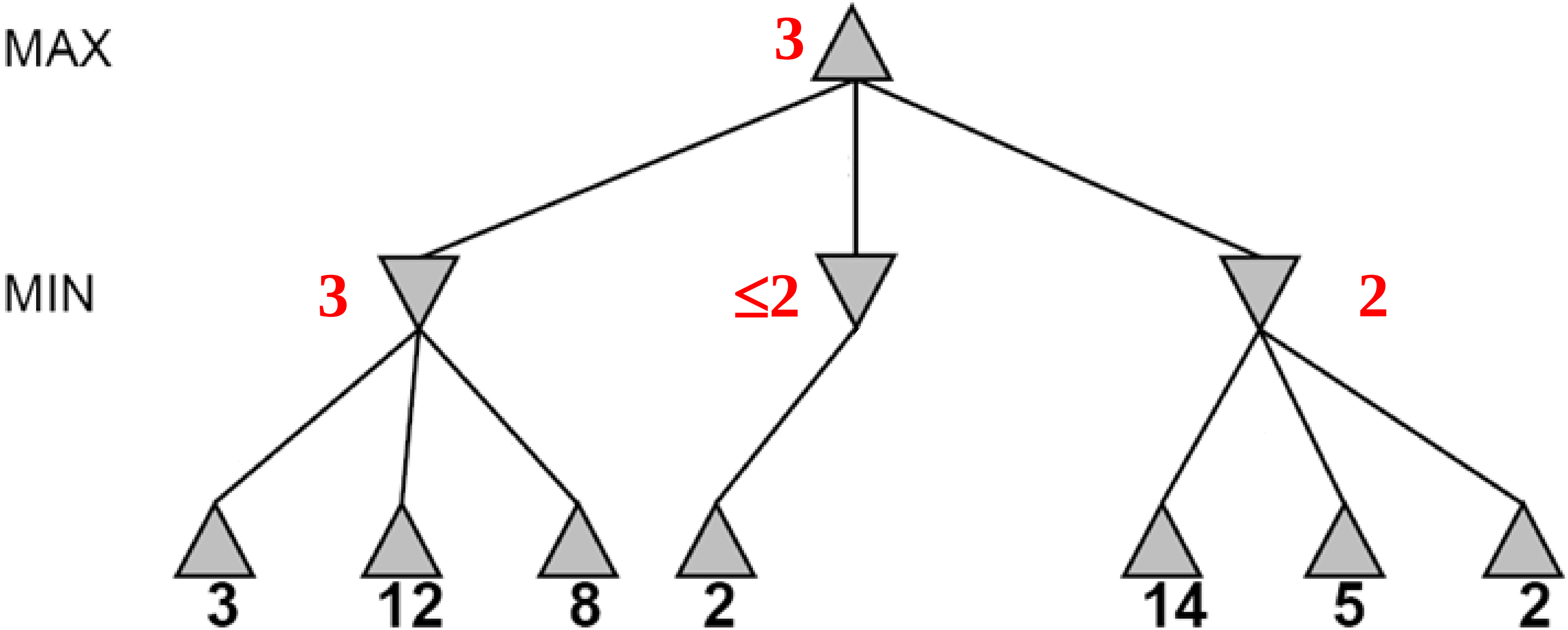
Alpha-beta pruning



Alpha-beta pruning

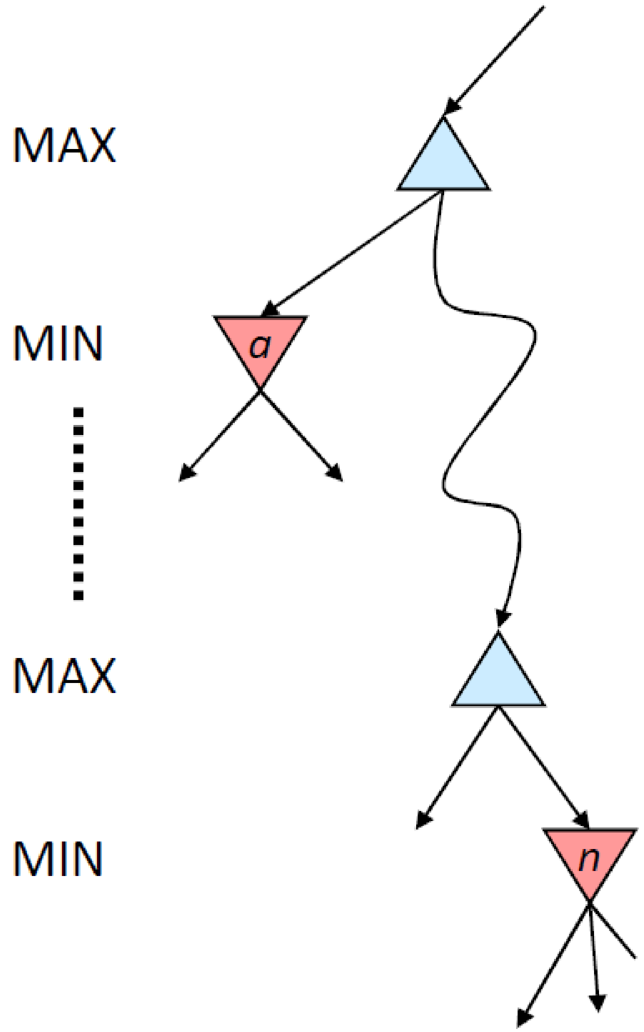


Alpha-beta pruning



Alpha-beta pruning

- **α** is the value of the best choice for the MAX player found so far at any choice point above node *n*
- We want to compute the MIN-value at *n*
- As we loop over *n*'s children, the MIN-value decreases
- If it drops below **α**, MAX will never choose *n*, so we can ignore *n*'s remaining children
- Analogously, **β** is the value of the lowest-utility choice found so far for the MIN player



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
```

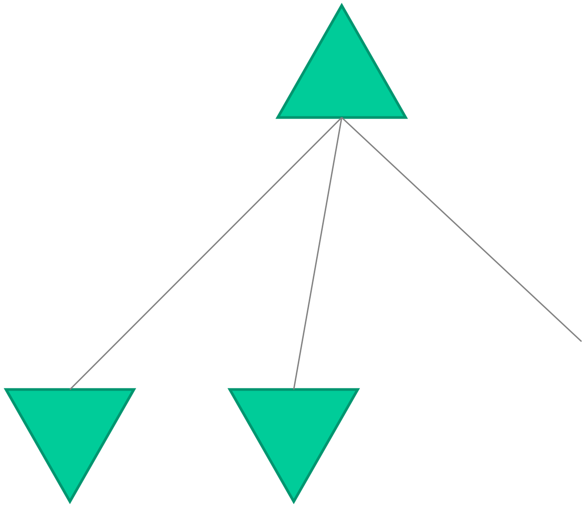
```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-beta pruning

Function *action* = **Alpha-Beta-Search**(*node*)

v = **Max-Value**(*node*, $-\infty$, ∞)

return the *action* from *node* with value v



α: *best alternative available to the Max player*

β: *best alternative available to the Min player*

Function v = **Max-Value**(*node*, α, β)

if Terminal(*node*) return Utility(*node*)

$v = -\infty$

for each *action* from *node*

v = Max(v, **Min-Value**(Succ(*node*, *action*), α, β))

if $v \geq \beta$ return v

α = Max(α, v)

end for

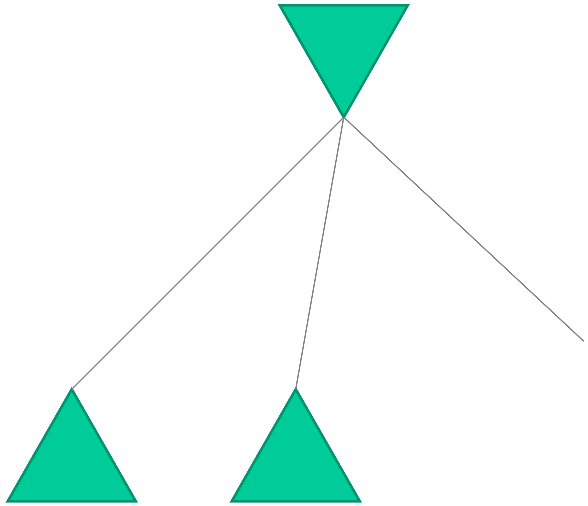
return v

Alpha-beta pruning

Function *action* = **Alpha-Beta-Search**(*node*)

v = **Min-Value**(*node*, $-\infty$, ∞)

return the *action* from *node* with value v



α: *best alternative available to the Max player*

β: *best alternative available to the Min player*

Function v = **Min-Value**(*node*, α, β)

if Terminal(*node*) return Utility(*node*)

$v = +\infty$

for each *action* from *node*

v = Min(v, **Max-Value**(Succ(*node*, *action*), α, β))

if $v \leq \alpha$ return v

β = Min(β, v)

end for

return v

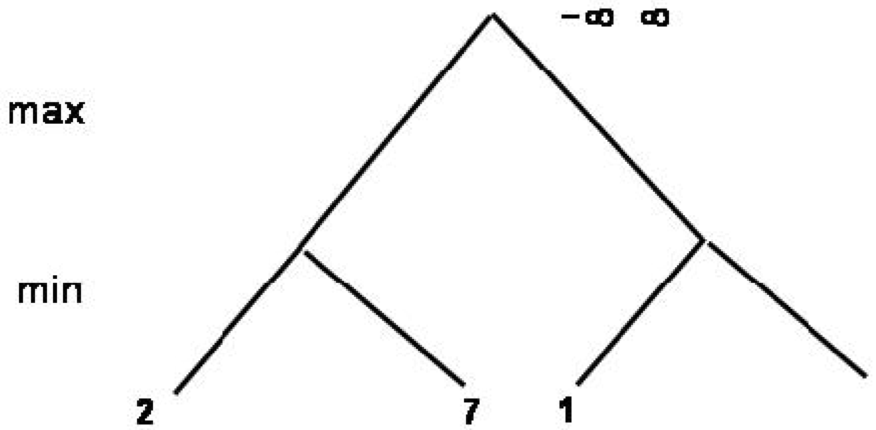
α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

α - β



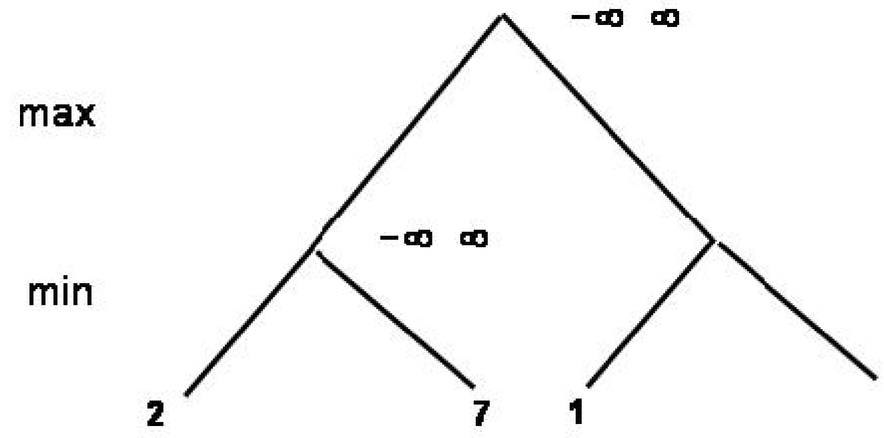
α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

α - β

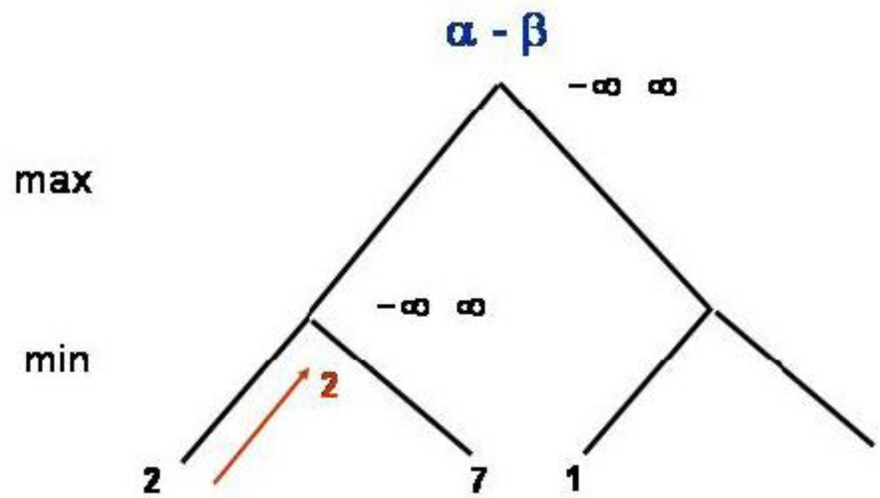


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

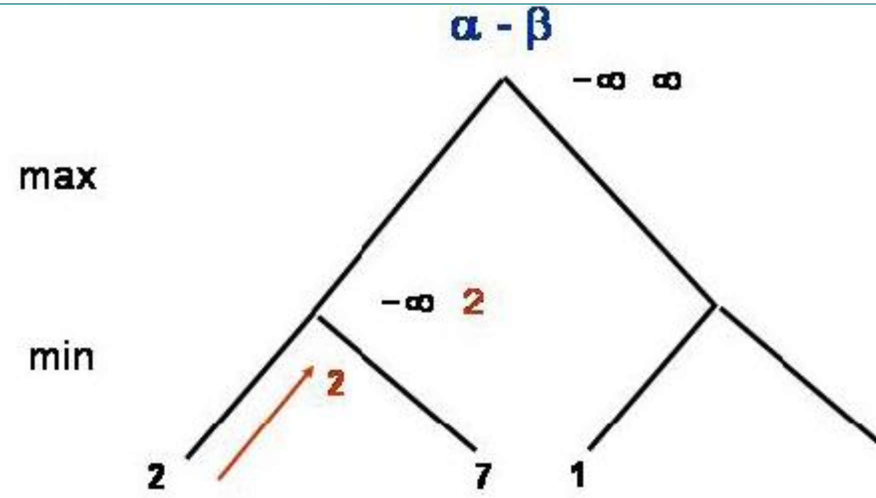


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```



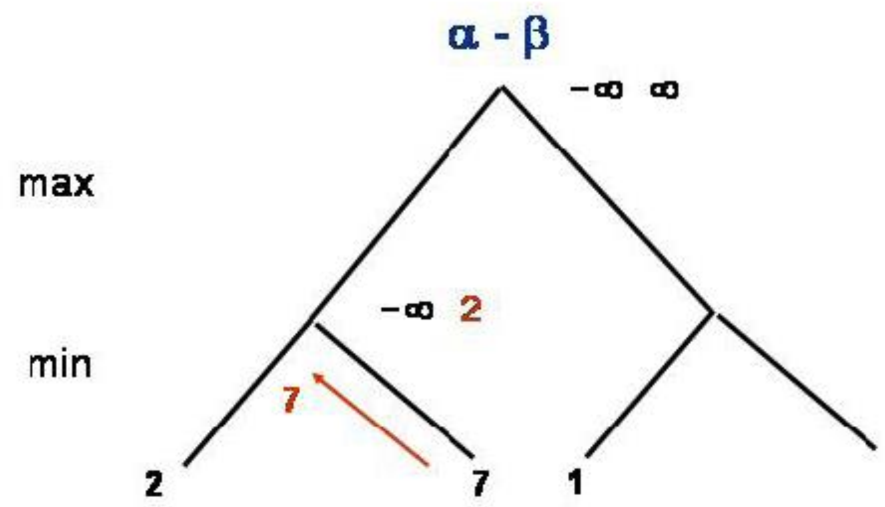
α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

function MAX-VALUE (state, α, β, depth)
if (depth == 0) then return EVAL (state)
for each s in SUCCESSORS (state) do
α = MAX (α, MIN-VALUE (s, α, β,depth-1))
if α ≥ β then return α // cutoff
end
return α

function MIN-VALUE (state, α, β, depth)
if (depth == 0) then return EVAL (state)
for each s in SUCCESSORS (state) do
β = MIN (β, MAX-VALUE (s, α, β,depth-1))
if β ≤ α then return β // cutoff
end
return β

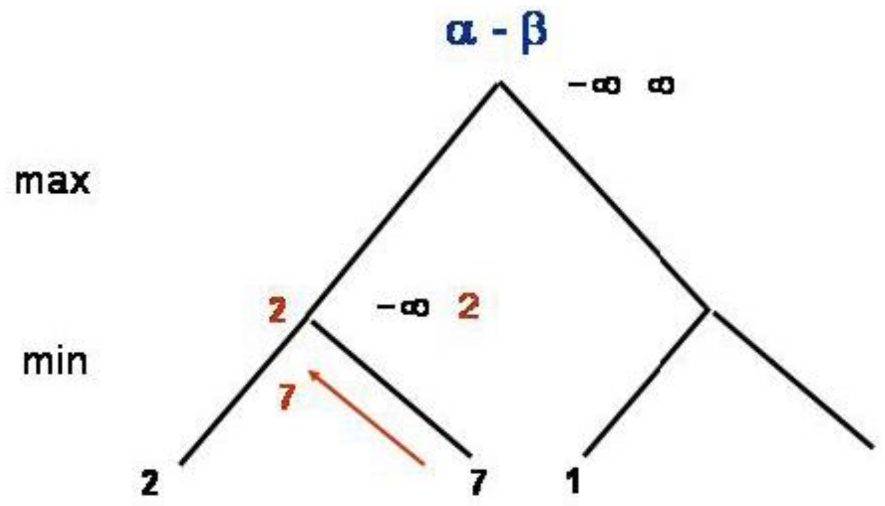


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

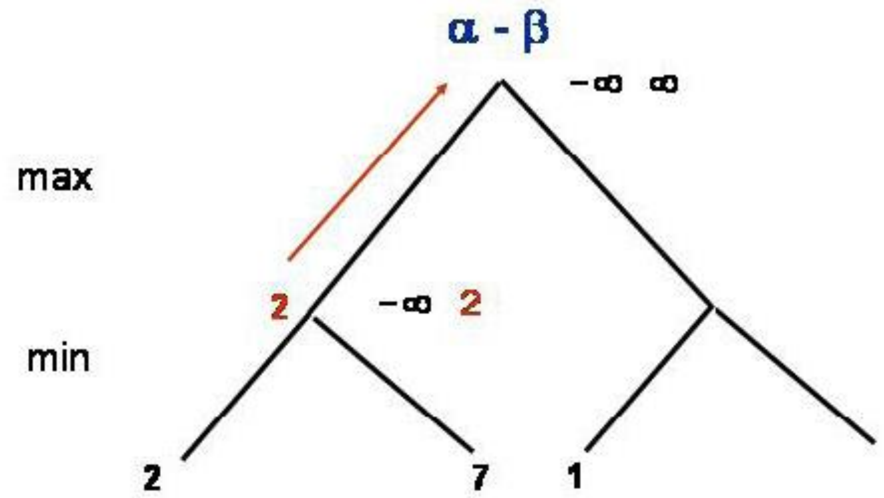


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

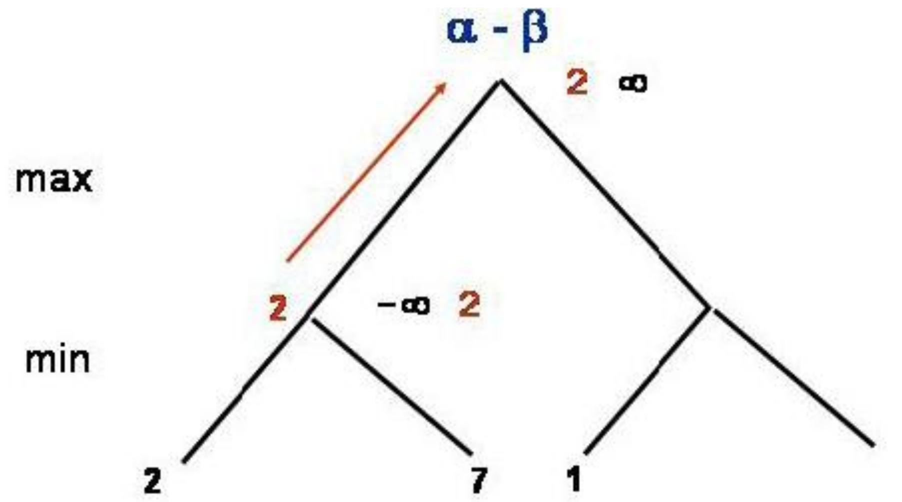


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

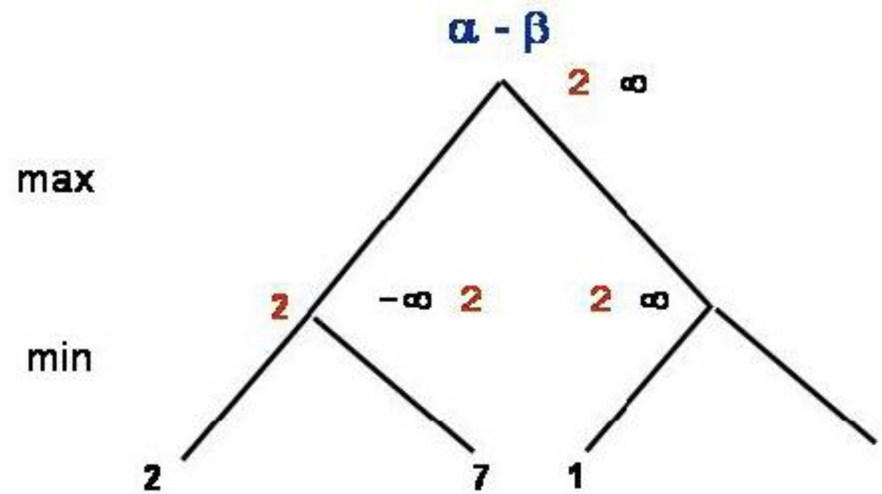


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
     end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

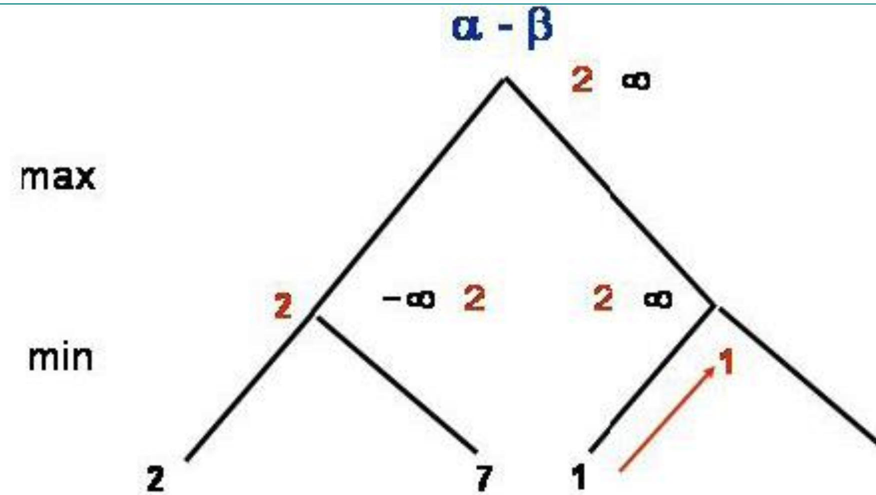


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

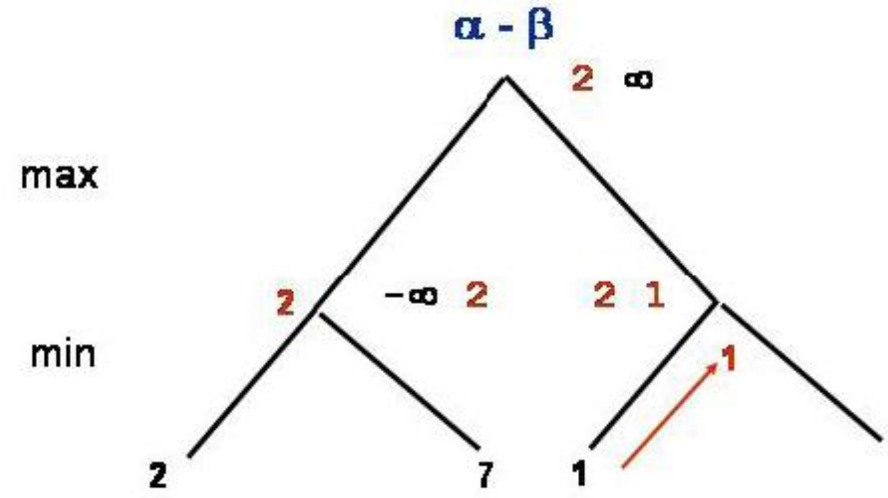


α-β pruning example

α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

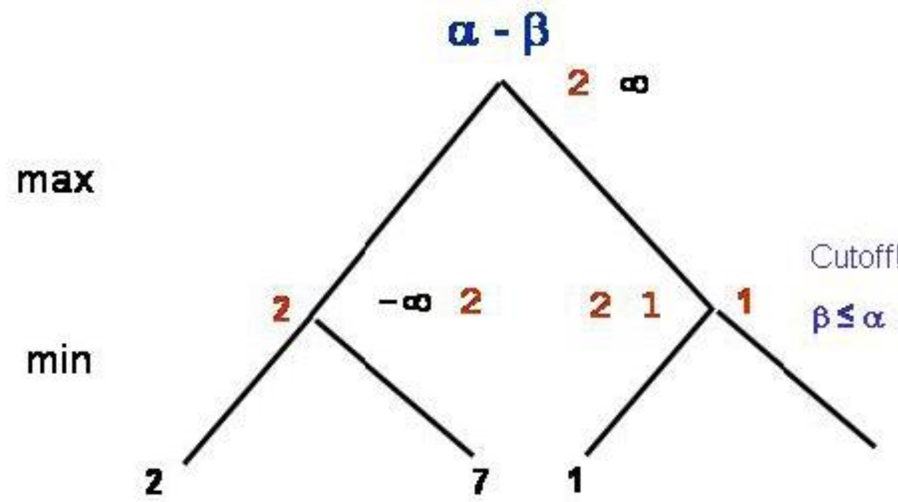


α-β pruning example

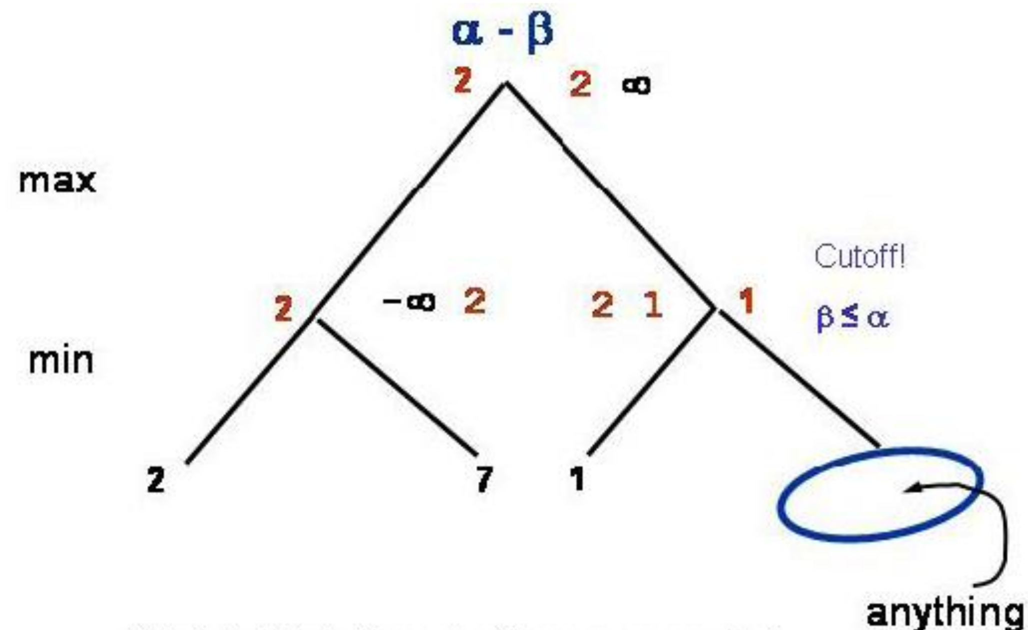
α - β

// α = best score for MAX, β = best score for MIN
// initial call is MAX-VALUE(state,-∞, ∞,MAX-DEPTH)

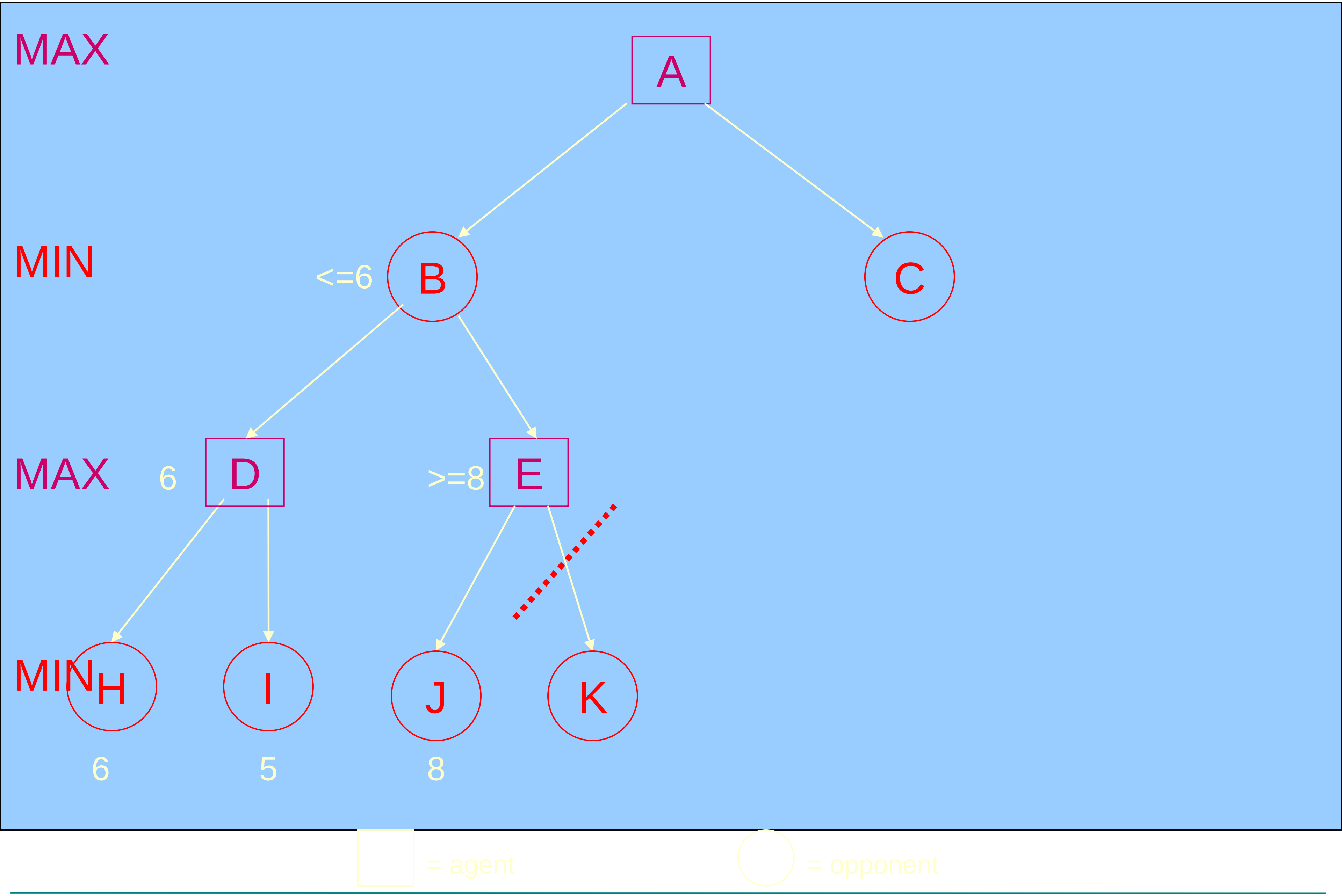
```
function MAX-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        α = MAX (α, MIN-VALUE (s, α, β,depth-1))
        if α ≥ β then return α // cutoff
    end
    return α

function MIN-VALUE (state, α, β, depth)
    if (depth == 0) then return EVAL (state)
    for each s in SUCCESSORS (state) do
        β = MIN (β, MAX-VALUE (s, α, β,depth-1))
        if β ≤ α then return β // cutoff
    end
    return β
```

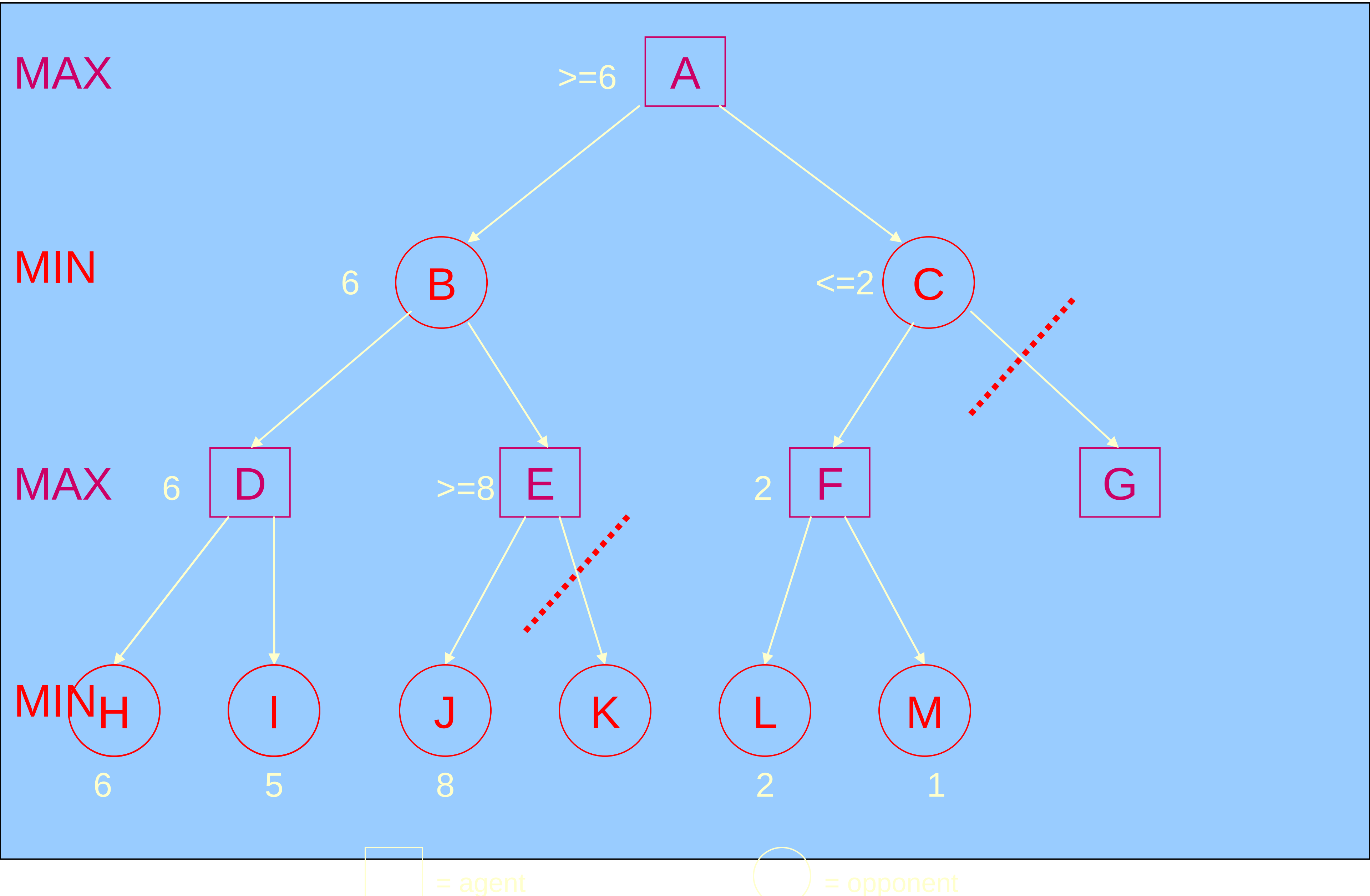


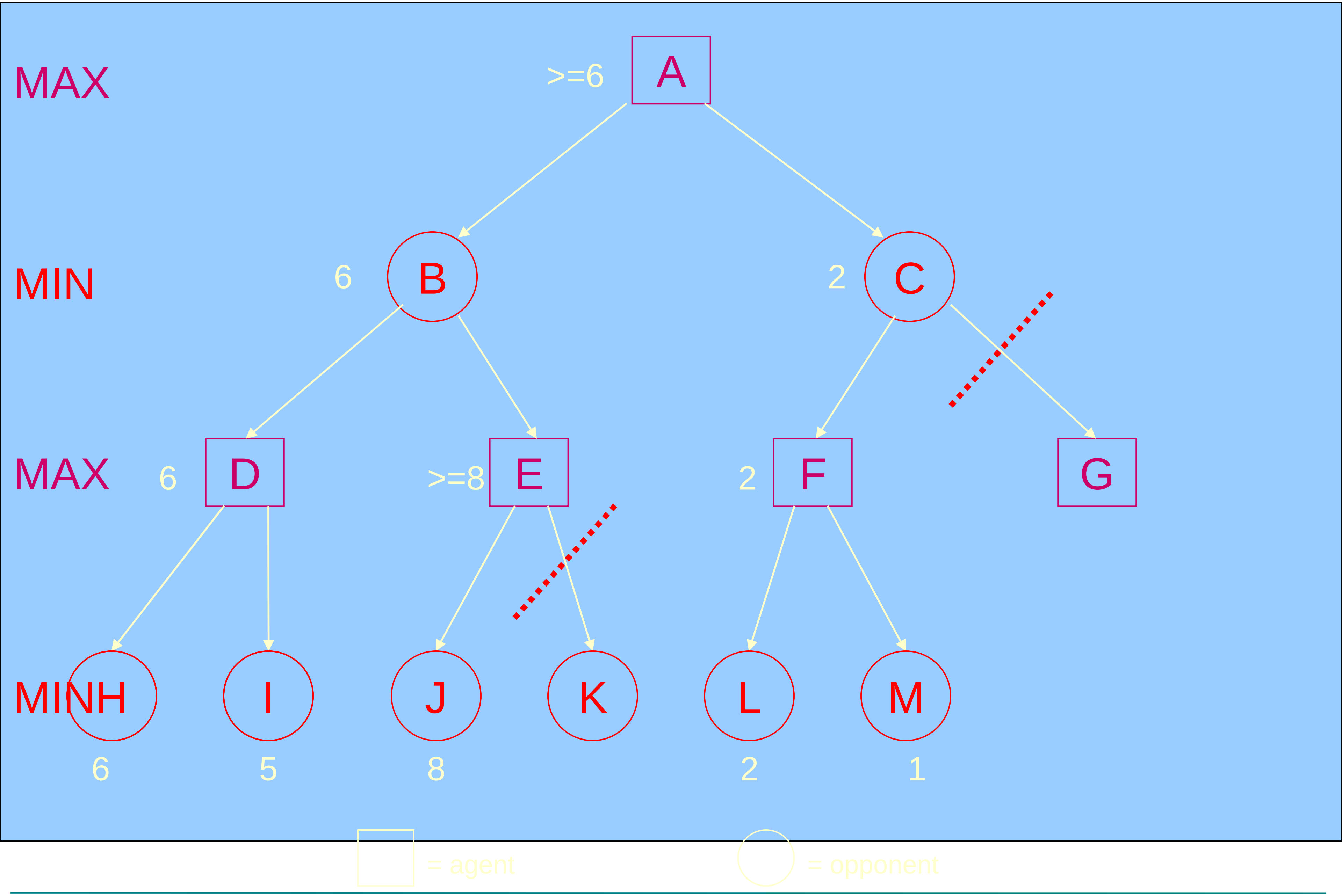
α-β pruning example



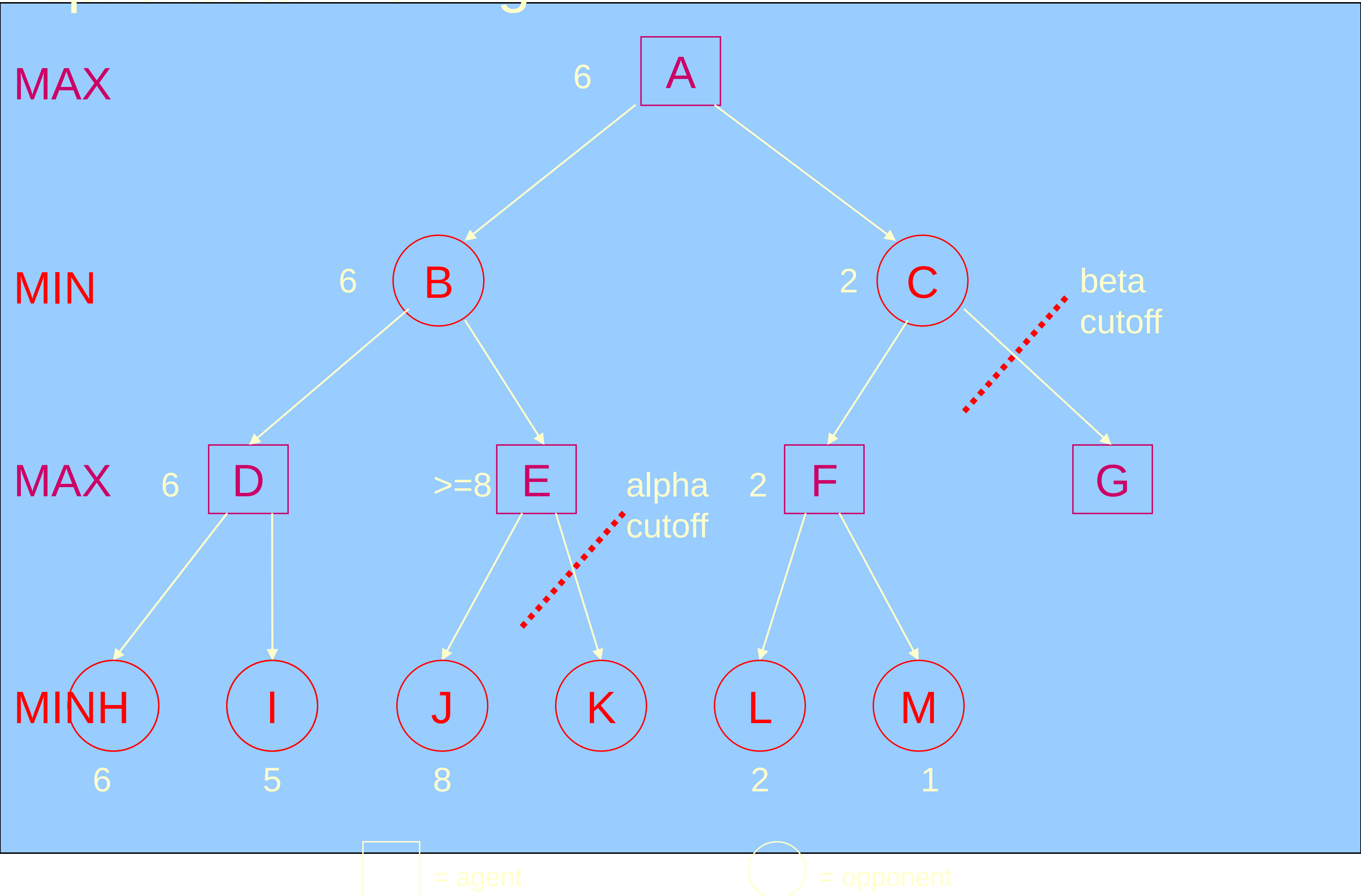
A total of 3 static evaluations were needed to obtain the value for the tree.

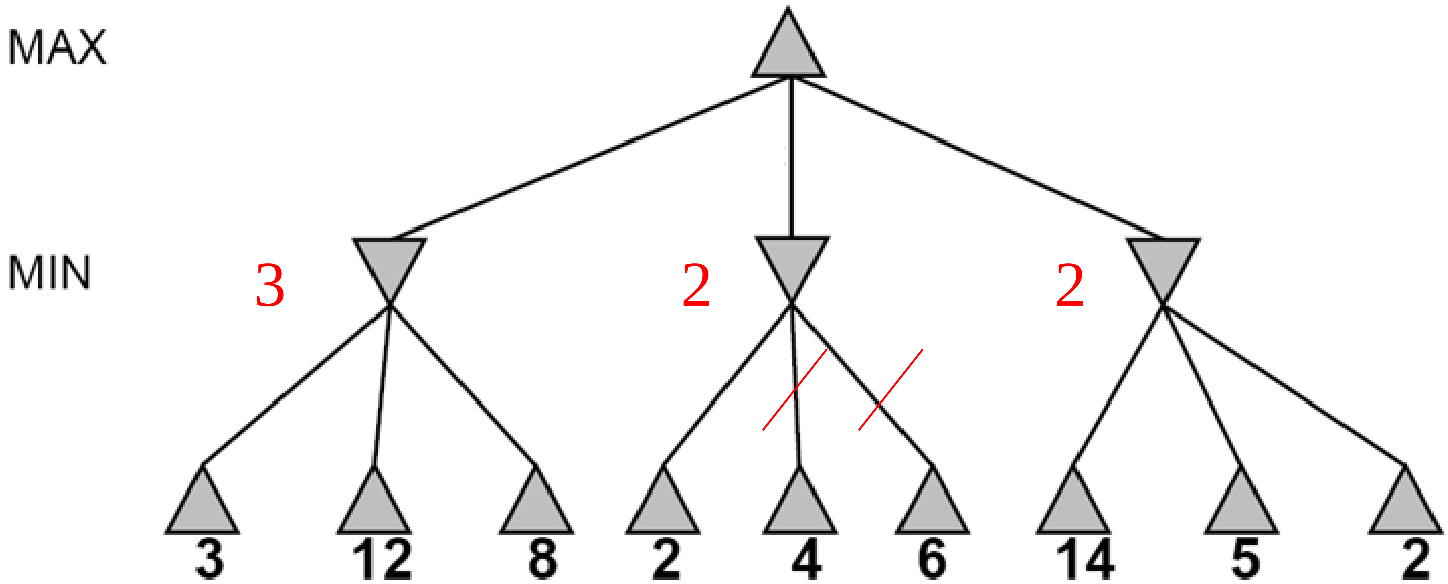

MAX
A
MIN
<=6
B
C
MAX
6
D
>=8
E
MIN
H
I
J
K
6
5
8
= agent
= opponent

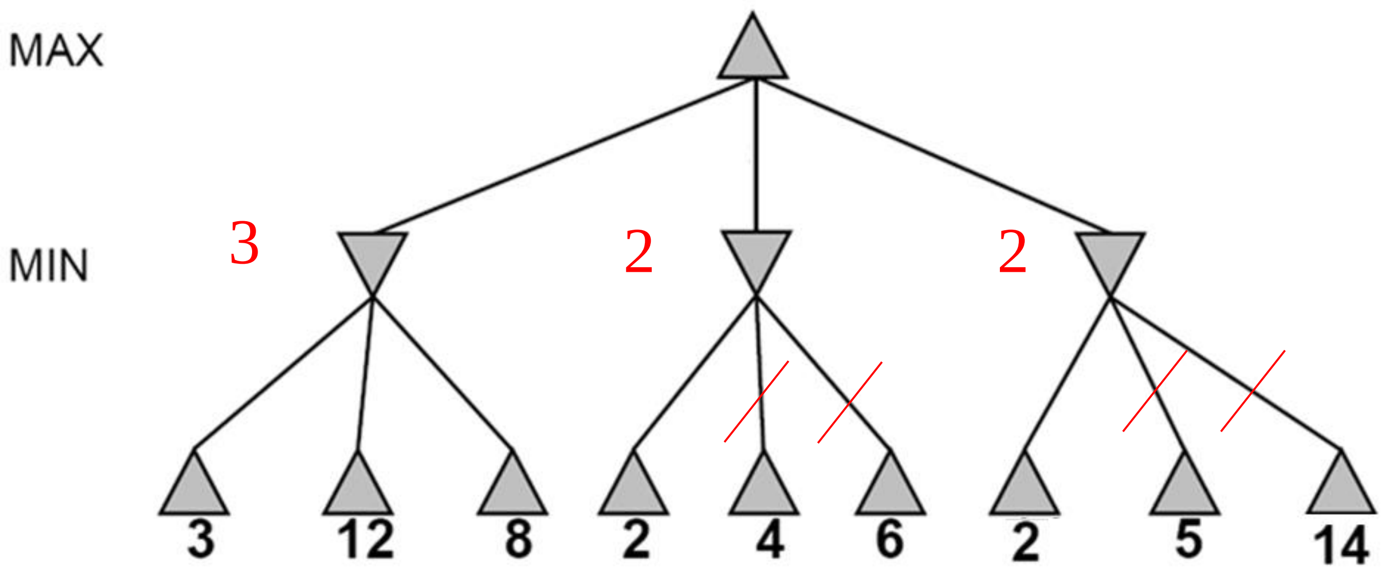

MAX
MIN
MAX
MIN
>=6
A
6
B
<=2
C
6
D
>=8
E
2
F
G
H
I
J
K
L
M
6
5
8
2
1
= agent
= opponent


MAX
>=6
A
MIN
6
B
2
C
MAX
6
D
>=8
E
2
F
G
MIN
H
I
J
K
L
M
6
5
8
2
1
= agent
= opponent

Alpha-beta Pruning



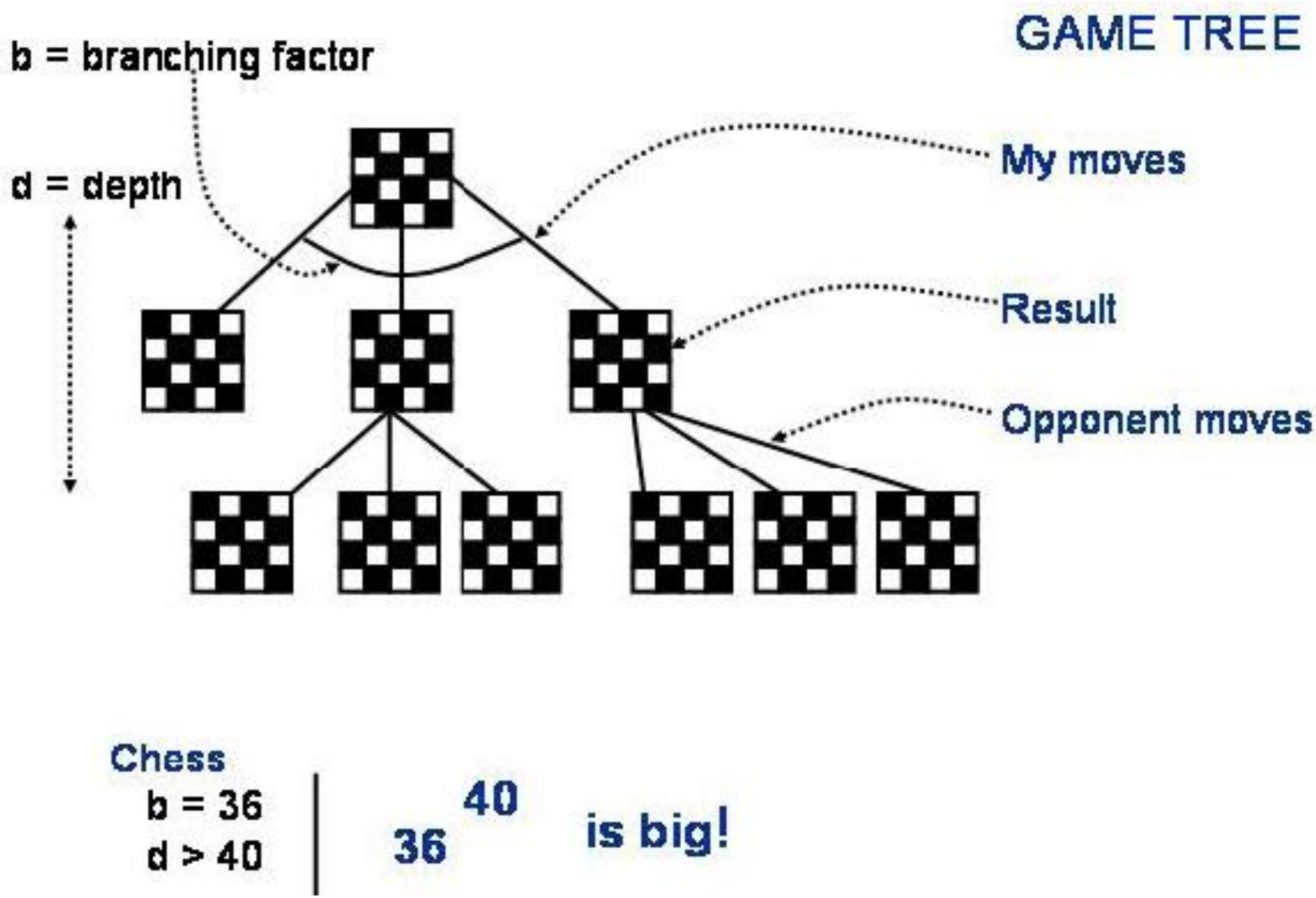

MAX
MIN
3
2
2
3
12
8
2
4
6
14
5
2


MAX
MIN
3
2
2
3
12
8
2
4
6
2
5
14

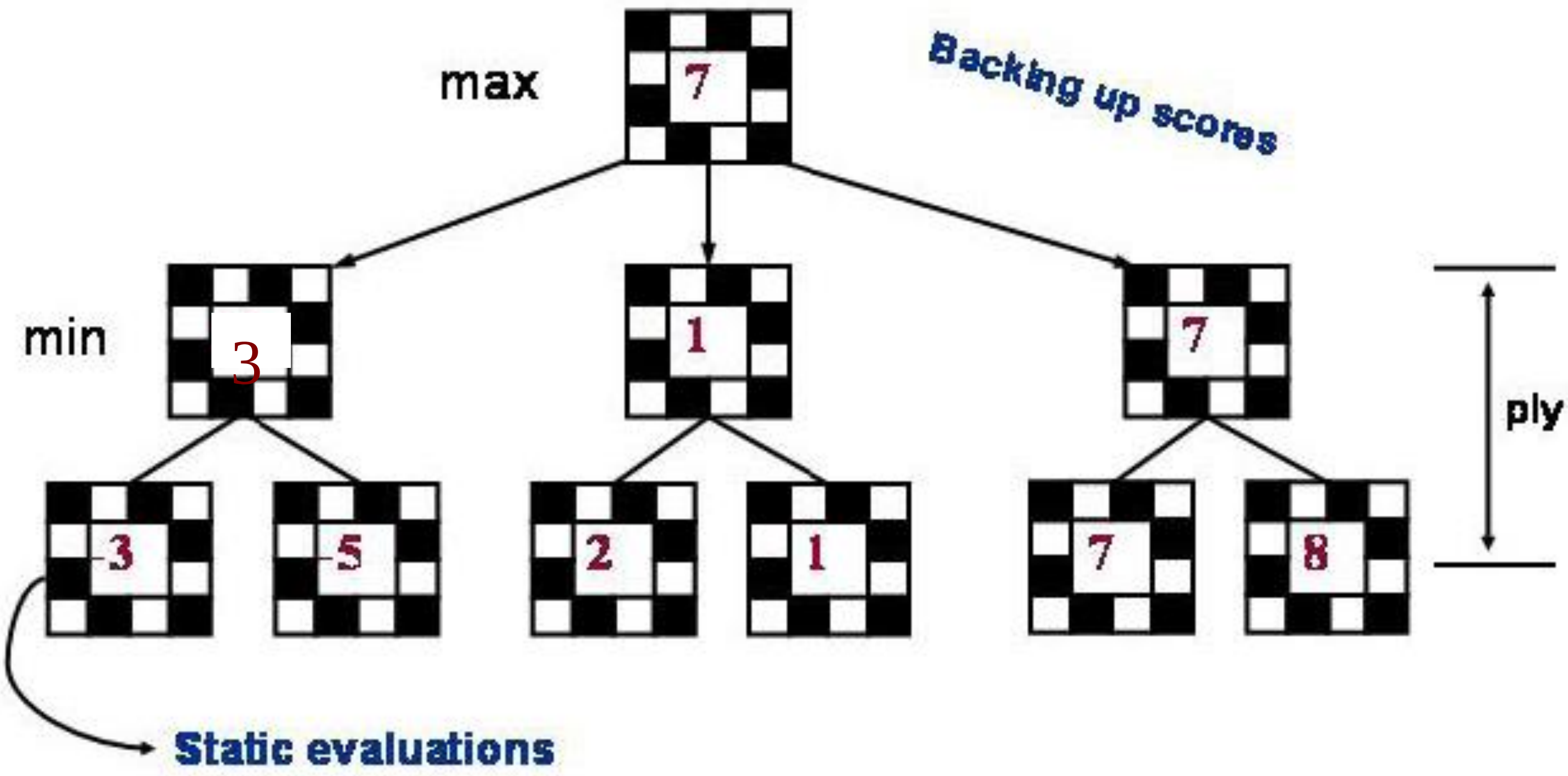
Alpha-beta pruning

- Pruning does not affect final result
- Amount of pruning depends on move ordering
  - Should start with the "best" moves (highest-value for MAX or lowest-value for MIN)
  - For chess, can try captures first, then threats, then forward moves, then backward moves
  - Can also try to remember "killer moves" from other branches of the tree
- With perfect ordering, the time to find the best move is reduced to $O(b^{m/2})$ from $O(b^m)$
  - Depth of search is effectively doubled

Move generation



Min-Max


max
7
Backing up scores
min
3
1
7
ply
3
5
2
1
7
8
Static evaluations

Resource limits

Suppose we have 100 secs, explore 10^4 nodes/sec

→ 10^6 nodes per move

Standard approach:

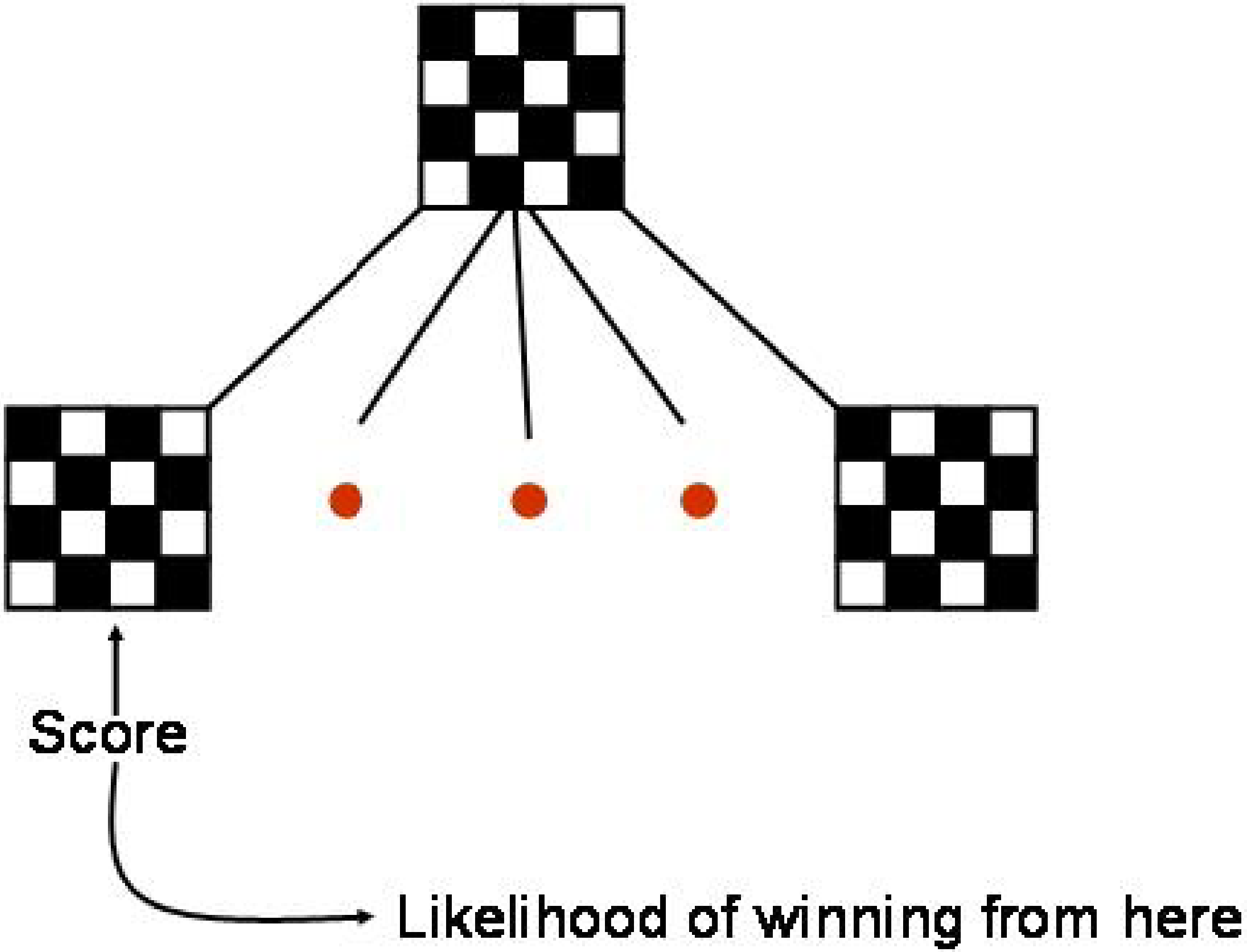
- cutoff test:

  e.g., depth limit (perhaps add quiescence search)

- evaluation function

  = estimated desirability of position

Evaluation function



Evaluation function

$$
\begin{aligned}
S = \;& c_1 \times \text{material} \\
+ \;& c_2 \times \text{pawn structure} \\
+ \;& c_3 \times \text{mobility} \\
+ \;& c_4 \times \text{king safety} \\
+ \;& c_5 \times \text{center control} \\
+ \;& \ldots
\end{aligned}
$$

P	1
K	3
B	3.5
R	5
Q	9

- "material", : some measure of which pieces one has on the board.
- A typical weighting for each type of chess piece is shown
- Other types of features try to encode something about the distribution of the pieces on the board.

Evaluation functions

- A typical evaluation function is a linear function in which some set of coefficients is used to weight a number of "features" of the board position.

- weighted sum of *features*:

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

  – For chess, w_k may be the **material value** of a piece (pawn = 1, knight = 3, rook = 5, queen = 9) and $f_k(s)$ may be the advantage in terms of that piece

  – Eg. $w_1 = 9$ with

  $f_1(s)$ = (number of white queens) – (number of black queens)

Evaluation function

- Cut off search at a certain depth and compute the value of an **evaluation function** for a state instead of its minimax value

- The evaluation function may be thought of as the probability of winning from a given state or the *expected value* of that state
  - If a position A has a 100% chance of winning it should have the evaluation 1
  - If position B have a 50% chance of winning and 25% os loosing and 25% of being a draw, the evaluation value would be $+1x0.50+ -1x0.25+ 0x0.25 = 0.25$

- Evaluation functions may be *learned* from game databases or by having the program play many games against itself

Cutting off search

MinimaxCutoff is identical to *MinimaxValue* except

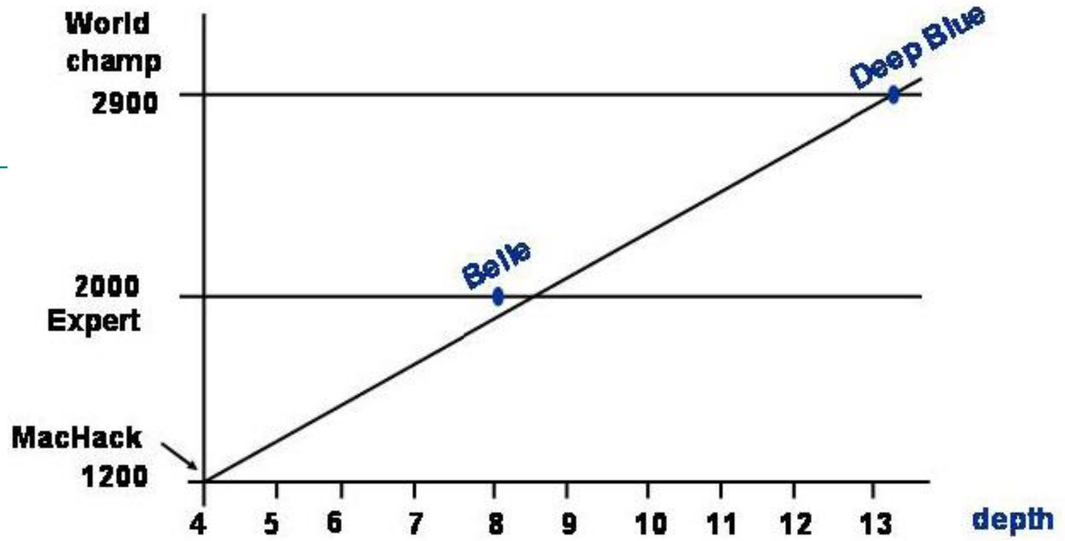
1. *Terminal?* is replaced by *Cutoff?*
2. *Utility* is replaced by *Eval*

Does it work in practice?

$b^m = 10^6$, b=35 m=4

4-ply lookahead is a hopeless chess player!

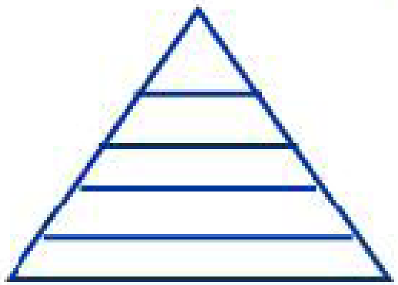
- 4-ply ≈ human novice
- 8-ply ≈ typical PC, human master
- 12-ply ≈ Deep Blue, Kasparov



Chess playing systems

- Baseline system: 200 million node evalutions per move (3 min), minimax with a decent evaluation function and quiescence search
  - 5-ply ≈ human novice
- Add alpha-beta pruning
  - 10-ply ≈ typical PC, experienced player
- Deep Blue: 30 billion evaluations per move, singular extensions, evaluation function with 8000 features, large databases of opening and endgame moves
  - 14-ply ≈ Garry Kasparov
- More recent state of the art (Hydra, ca. 2006): 36 billion evaluations per second, advanced pruning techniques
  - 18-ply ≈ better than any human alive?

Practical issues



Variable branching

Iterative deepening

- order best move from last search first
- use previous backed up value to initialize [α, β]
- keep track of repeated positions (transposition tables)

Horizon effect

- quiescence
- Pushing the inevitable over search horizon

Parallelization

Cutting off search

- **Horizon effect:** you may incorrectly estimate the value of a state by overlooking an event that is just beyond the depth limit
  - For example, a damaging move by the opponent that can be delayed but not avoided
- Possible remedies
  - **Quiescence search:** do not cut off search at positions that are unstable – for example, are you about to lose an important piece?
  - **Singular extension:** a strong move that should be tried when the normal depth limit is reached

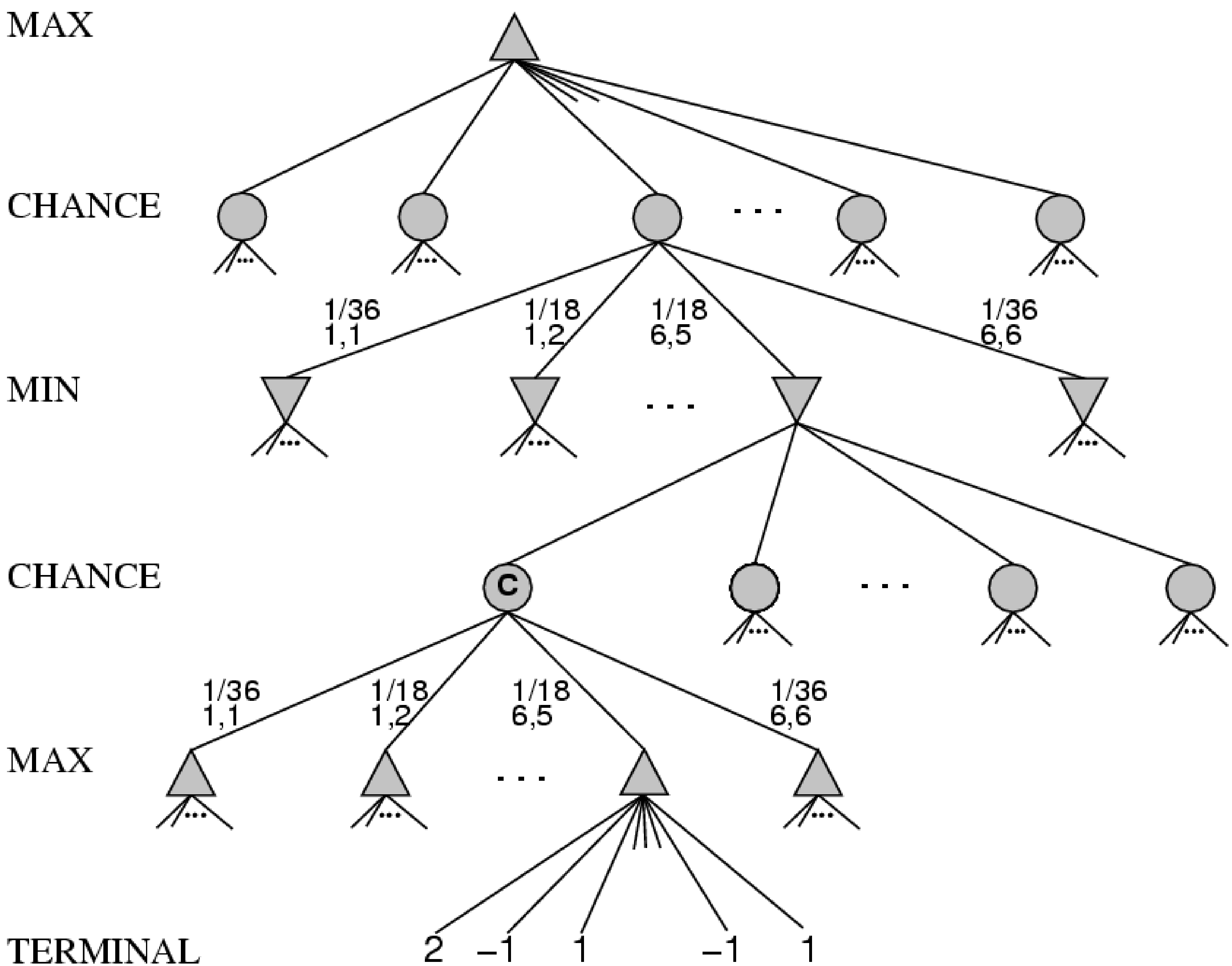
Types of game environments

	Deterministic	**Stochastic**
Perfect information (fully observable)	Chess, checkers, go	Backgammon, monopoly
Imperfect information (partially observable)	Battleships	Scrabble, poker, bridge

Stochastic games

- How to incorporate dice throwing into the game tree?

Stochastic games



Minimax vs. Expectiminimax

- **Minimax:**
  - **Maximize** (over all possible moves I can make) the
  - **Minimum** (over all possible moves Min can make) of the
  - Reward

$$Value(node) = \max_{my\ moves}\left(\min_{Min's\ moves}(Reward)\right)$$

- **Expectiminimax:**
  - **Maximize** (over all possible moves I can make) the
  - **Minimum** (over all possible moves Min can make) of the
  - **Expected** reward

$$Value(node) = \max_{my\ moves}\left(\min_{Min's\ moves}(\mathbb{E}[Reward])\right)$$

$$\mathbb{E}[Reward] = \sum_{outcomes} Probability(outcome) \times Reward(outcome)$$

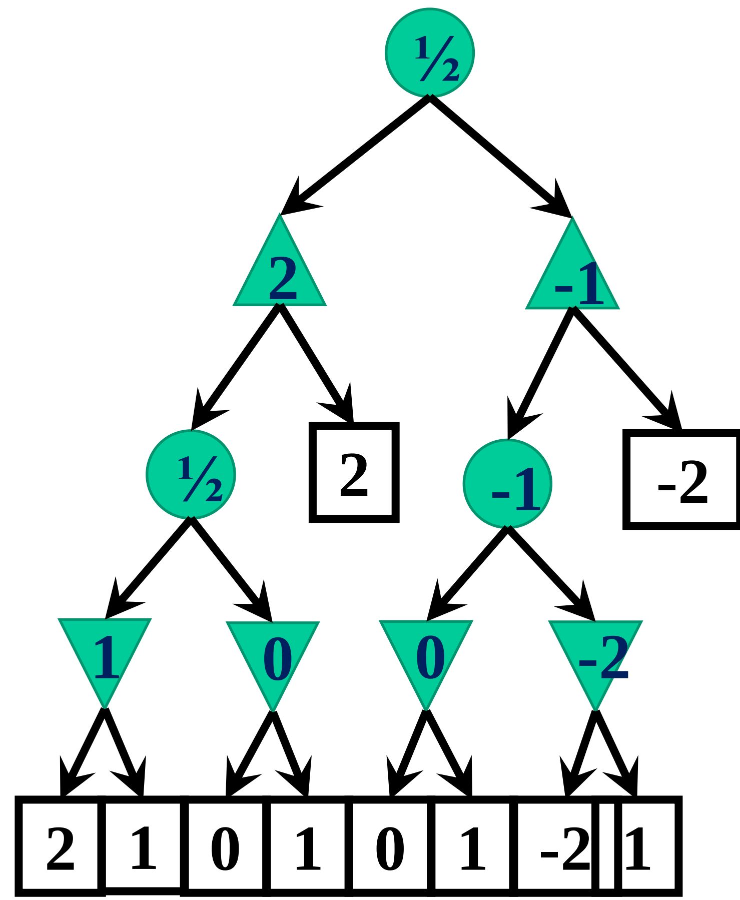
Stochastic games

- **Expectiminimax:** for chance nodes, sum values of successor states weighted by the probability of each successor

- **Value**(*node*) =
  - Utility(*node*) if *node* is terminal
  - $\max_{action}$ **Value**(Succ(*node*, *action*)) if *type* = MAX
  - $\min_{action}$ **Value**(Succ(*node*, *action*)) if *type* = MIN
  - sum_{action} P(Succ(*node*, *action*)) * **Value**(Succ(*node*, *action*)) if *type* = CHANCE

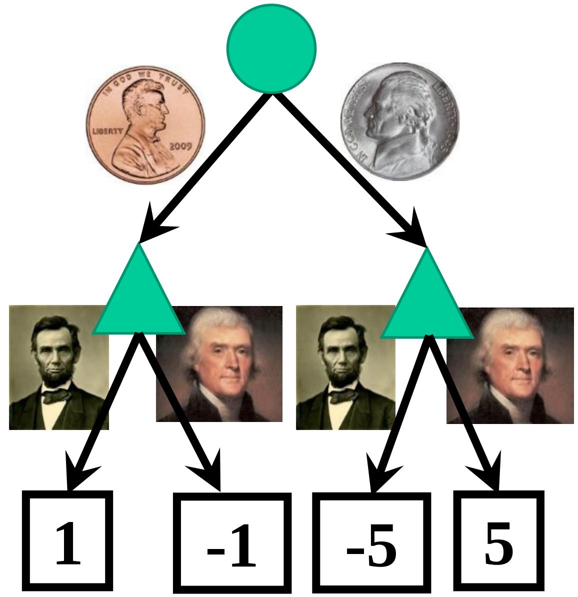
Expectiminimax example

- RANDOM: Max flips a coin. It's heads or tails.
- MAX: Max either stops, or continues.
  - Stop on heads: Game ends, Max wins (value = $2).
  - Stop on tails: Game ends, Max loses (value = -$2).
  - Continue: Game continues.
- RANDOM: Min flips a coin.
  - HH: value = $2
  - TT: value = -$2
  - HT or TH: value = 0
- MIN: Min decides whether to keep the current outcome (value as above), or pay a penalty (value=$1).



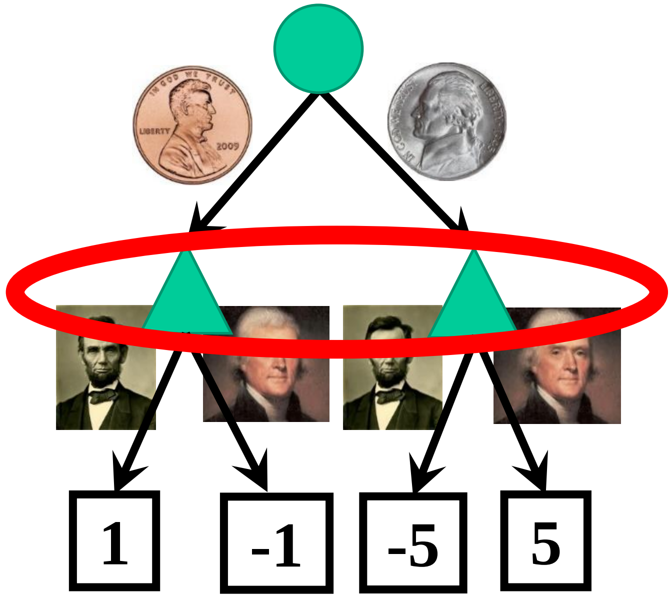
Imperfect information example

- Min chooses a coin.
- I say the name of a U.S. President.
  - If I guessed right, she gives me the coin.
  - If I guessed wrong, I have to give her a coin to match the one she has.



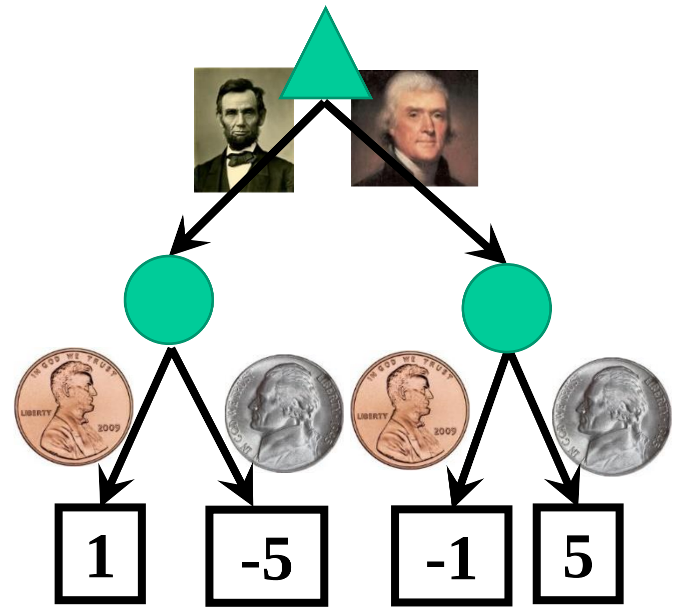
Method #1: Treat "unknown" as "unknown"

- The problem: I don't know which state I'm in. I only know it's one of these two.
- The solution: choose the policy that maximizes my minimum reward.
  - "Lincoln": minimum reward is -5.
  - "Jefferson": minimum reward is -1.
- Miniminimax policy: say "Jefferson".



Method #2: Treat "unknown" as "random"

- Expectiminimax: treat the unknown information as random.
- Choose the policy that maximizes my expected reward.
  - "Lincoln": $\frac{1}{2} \times 1 + \frac{1}{2} \times (-5) = -2$
  - "Jefferson": $\frac{1}{2} \times (-1) + \frac{1}{2} \times 5 = 2$
- Expectiminimax policy: say "Jefferson".
- BUT WHAT IF:  and  are not equally likely?



How to deal with imperfect information

- If you think you know the probabilities of different settings, and if you want to maximize your average winnings (for example, you can afford to play the game many times): **expectiminimax**
- If you have no idea of the probabilities of different settings; or, if you can only afford to play once, and you can't afford to lose: **miniminimax**
- If the unknown information has been selected intentionally by your opponent: use **game theory**

Stochastic games

- **Expectiminimax:** for chance nodes, sum values of successor states weighted by the probability of each successor
  - Nasty branching factor, defining evaluation functions and pruning algorithms more difficult
- **Monte Carlo simulation:** when you get to a chance node, simulate a large number of games with random dice rolls and use win percentage as evaluation function
  - Can work well for games like Backgammon

Stochastic games of imperfect information

- Simple Monte Carlo approach: run multiple simulations with random cards pretending the game is fully observable
  - "Averaging over clairvoyance"
  - Problem: this strategy does not account for bluffing, information gathering, etc.

Stochastic search for stochastic games

- The problem with expectiminimax: huge branching factor (many possible outcomes)

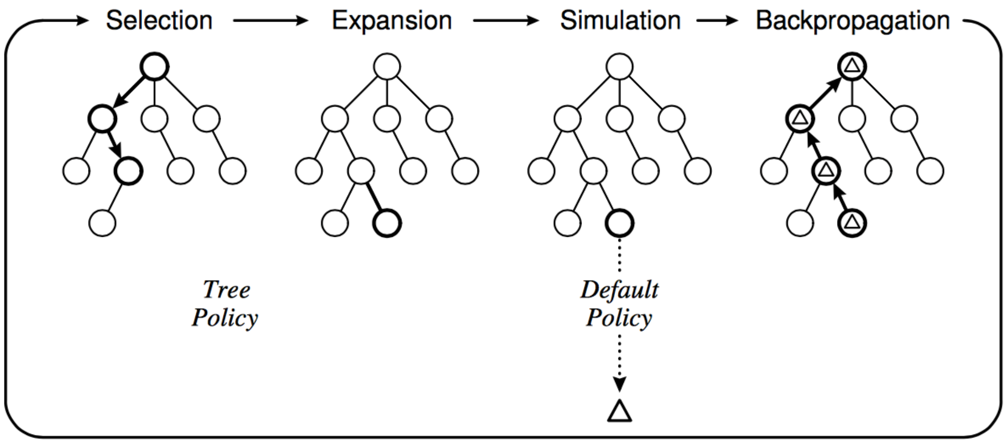
$$\mathbb{E}[Reward] = \sum_{outcomes} Probability(outcome) \times Reward(outcome)$$

- An approximate solution: Monte Carlo search

$$\mathbb{E}[Reward] \approx \frac{1}{n}\sum_{i=1}^{n} Reward(i'th\, random\, game)$$

Monte Carlo Tree Search

- What about ***deterministic*** games with deep trees, large branching factor, and no good heuristics – like Go?
- Instead of depth-limited search with an evaluation function, use randomized simulations
- Starting at the current state (root of search tree), iterate:
  - Select a leaf node for expansion using a *tree policy* (trading off *exploration* and *exploitation*)
  - Run a simulation using a *default policy* (e.g., random moves) until a terminal state is reached
  - Back-propagate the outcome to update the value estimates of internal tree nodes

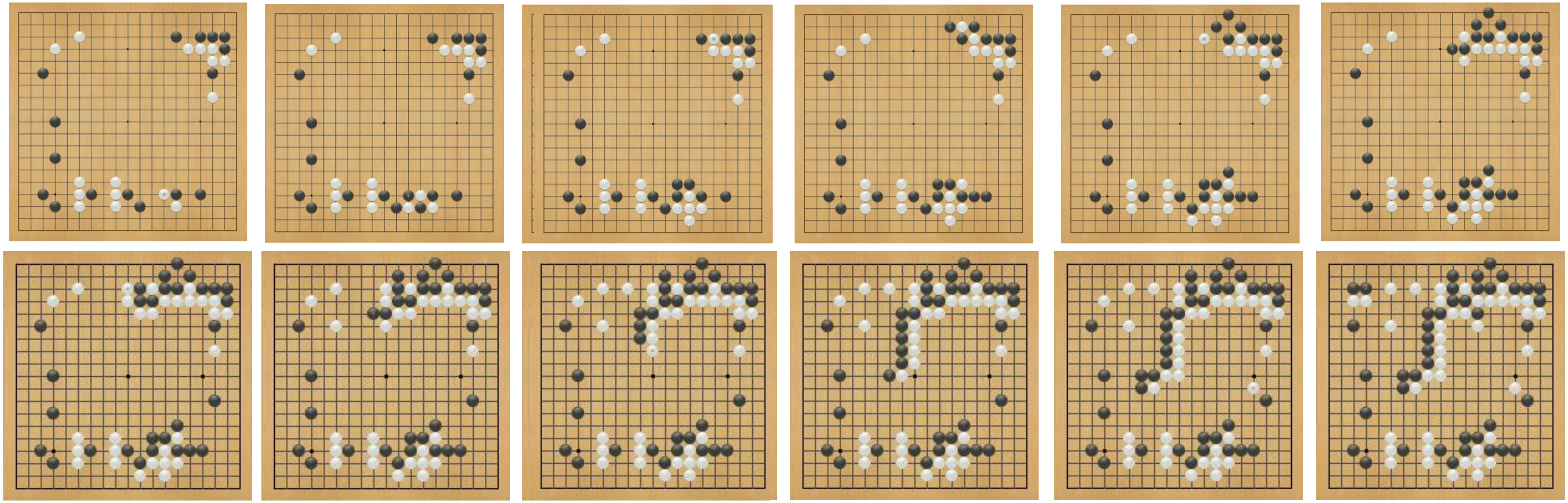


Case study: AlphaGo

- **“Gentlemen should not waste their time on trivial games -- they should play go.”**
- ***-- Confucius,***
- ***The Analects***
- ***ca. 500 B. C. E.***

Anton Ninno Roy Laird, Ph.D.
antonninno@yahoo.com roylaird@gmail.com
special thanks to Kiseido Publications

Learned evaluation functions



Stochastic search off-line

Training phase:

- Spend a few weeks allowing your computer to play billions of random games from every possible starting state
- Value of the starting state = average value of the ending states achieved during those billion random games

Testing phase:

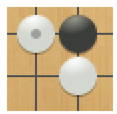
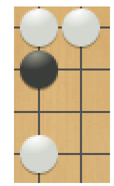
- During the alpha-beta search, search until you reach a state whose value you have stored in your value lookup table
- Oops…. Why doesn’t this work?

Evaluation as a pattern recognition problem

Training phase:

- Spend a few weeks allowing your computer to play billions of random games from billions of possible starting states.
- Value of the starting state = average value of the ending states achieved during those billion random games

Generalization:

- Featurize (e.g., x1=number of  patterns, x2 = number of  patterns, etc.)
- Linear regression: find a1, a2, etc. so that Value(state) ≈ a1*x1+x2*x2+…

Testing phase:

- During the alpha-beta search, search as deep as you can, then estimate the value of each state at your horizon using Value(state) ≈ a1*x1+x2*x2+…

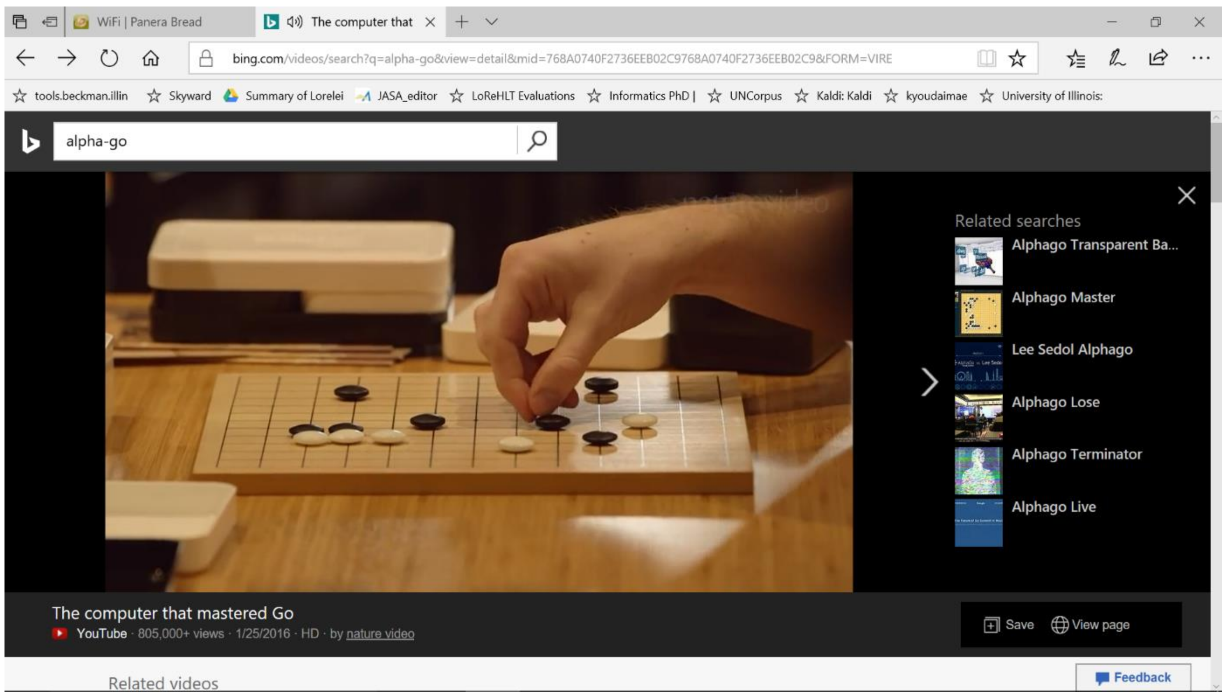
Pros and Cons

- Learned evaluation function
  - Pro: off-line search permits lots of compute time, therefore lots of training data
  - Con: there's no way you can evaluate every starting state that might be achieved during actual game play. Some starting states will be missed, so generalized evaluation function is necessary
- On-line stochastic search
  - Con: limited compute time
  - Pro: it's possible to estimate the value of the state you've reached during actual game play

AlphaGo

- SL policy network
  - Idea: perform *supervised learning* (SL) to predict human moves
  - Given state s, predict probability distribution over moves a, P(a|s)
  - Trained on 30M positions, 57% accuracy on predicting human moves
  - Also train a smaller, faster *rollout policy* network (24% accurate)
- RL policy network
  - Idea: fine-tune policy network using *reinforcement learning* (RL)
  - Initialize RL network to SL network
  - Play two snapshots of the network against each other, update parameters to maximize expected final outcome
  - RL network wins against SL network 80% of the time, wins against open-source Pachi Go program 85% of the time

Alpha-Go video



Game AI: Origins

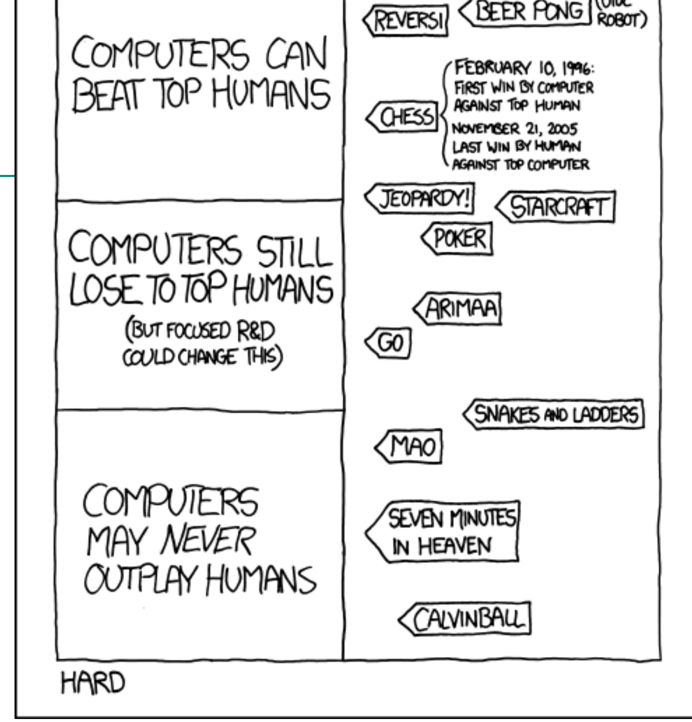
- Minimax algorithm: Ernst Zermelo, 1912
- Chess playing with evaluation function, quiescence search, selective search:
  Claude Shannon, 1949 (paper)
- Alpha-beta search: John McCarthy, 1956
- Checkers program that learns its own evaluation function by playing against itself: Arthur Samuel, 1956 (Rodney Brooks blog post)

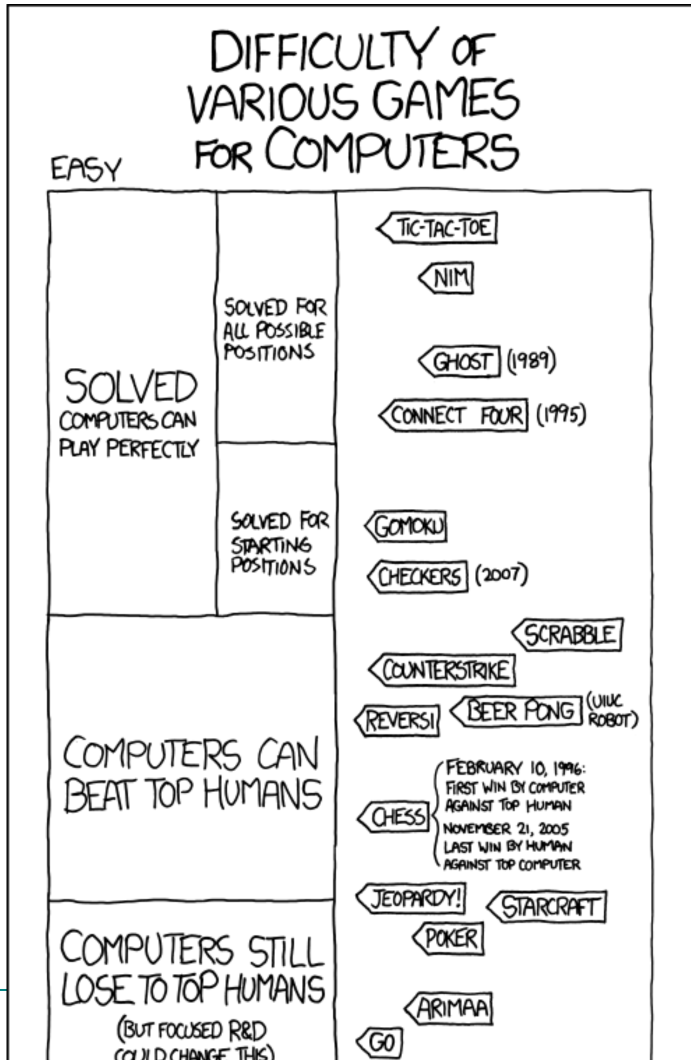
Game AI: State of the art

- Computers are better than humans:
  - **Checkers:** solved in 2007
  - **Chess:**
    - State-of-the-art search-based systems now better than humans
    - Deep learning machine teaches itself chess in 72 hours, plays at International Master Level (arXiv, September 2015)
- Computers are competitive with top human players:
  - **Backgammon:** TD-Gammon system (1992) used *reinforcement learning* to learn a good evaluation function
  - **Bridge:** top systems use Monte Carlo simulation and alpha-beta search
  - **Go:** computers were not considered competitive until AlphaGo in 2016

Game AI: State of the art

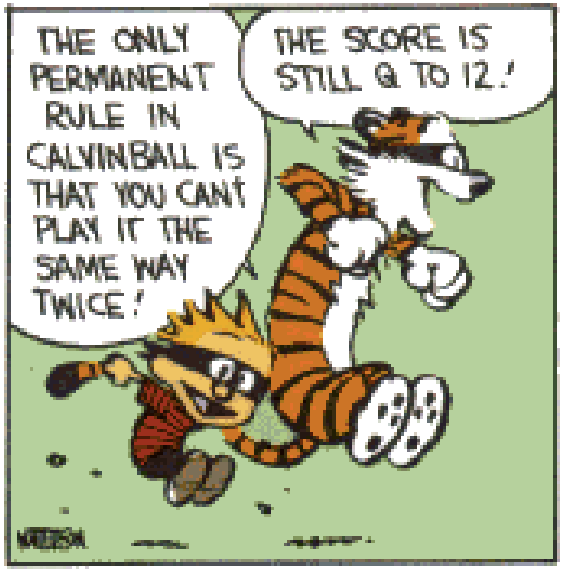
- Computers are ~~not~~ competitive with top human players:
  - **Poker**
    - Heads-up limit hold'em poker is solved (2015)
      - Simplest variant played competitively by humans
      - Smaller number of states than checkers, but partial observability makes it difficult
      - *Essentially weakly solved* = cannot be beaten with statistical significance in a lifetime of playing
    - CMU's Libratus system beats four of the best human players at no-limit Texas Hold'em poker (2017)





http://xkcd.com/1002/

http://xkcd.com/1263/



Calvinball:

- Play it online
- Watch an instructional video