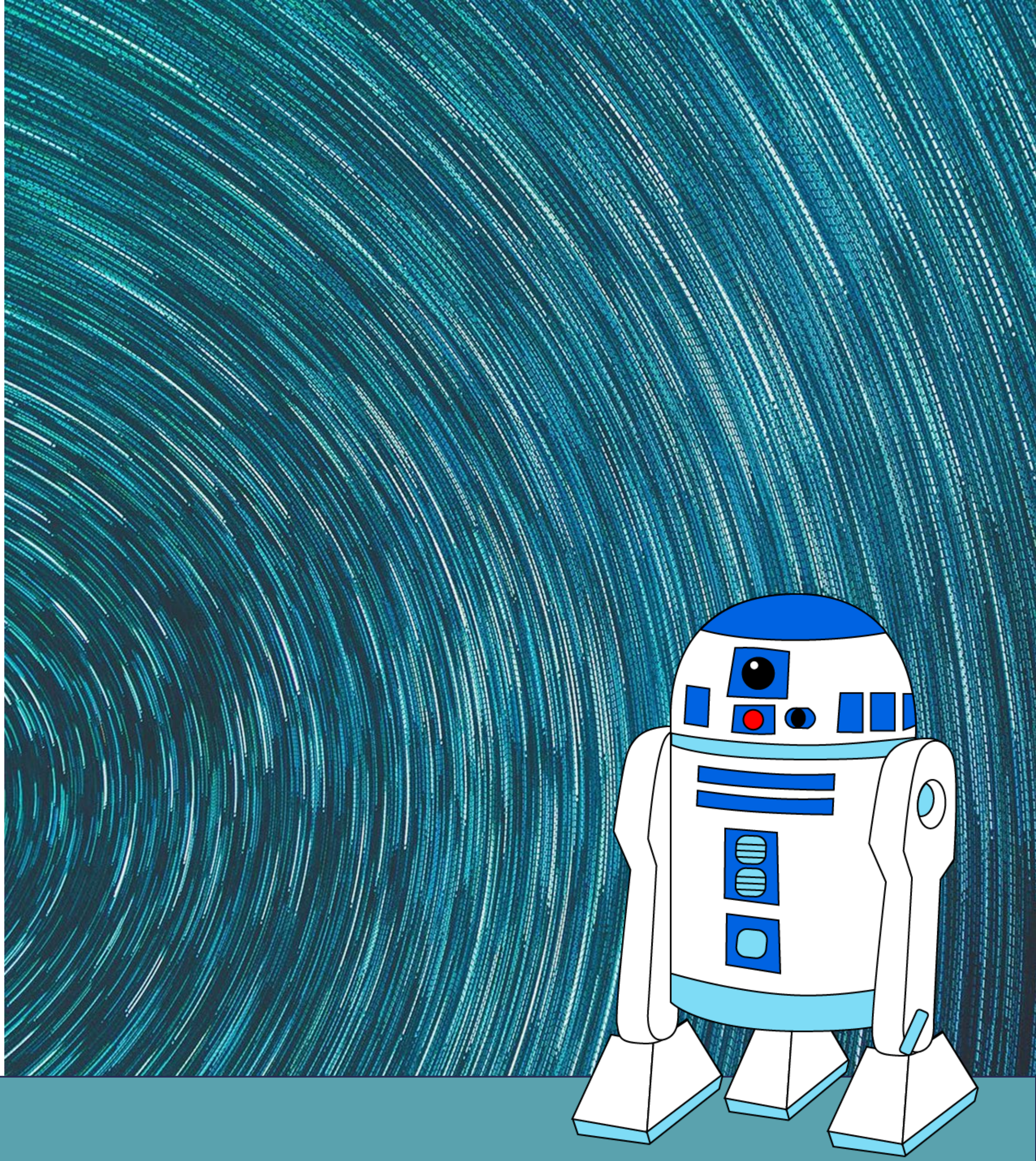
CIS 521:
ARTIFICIAL INTELLIGENCE

Games and Adversarial Search

Harry Smith



Penn Engineering
University of Pennsylvania

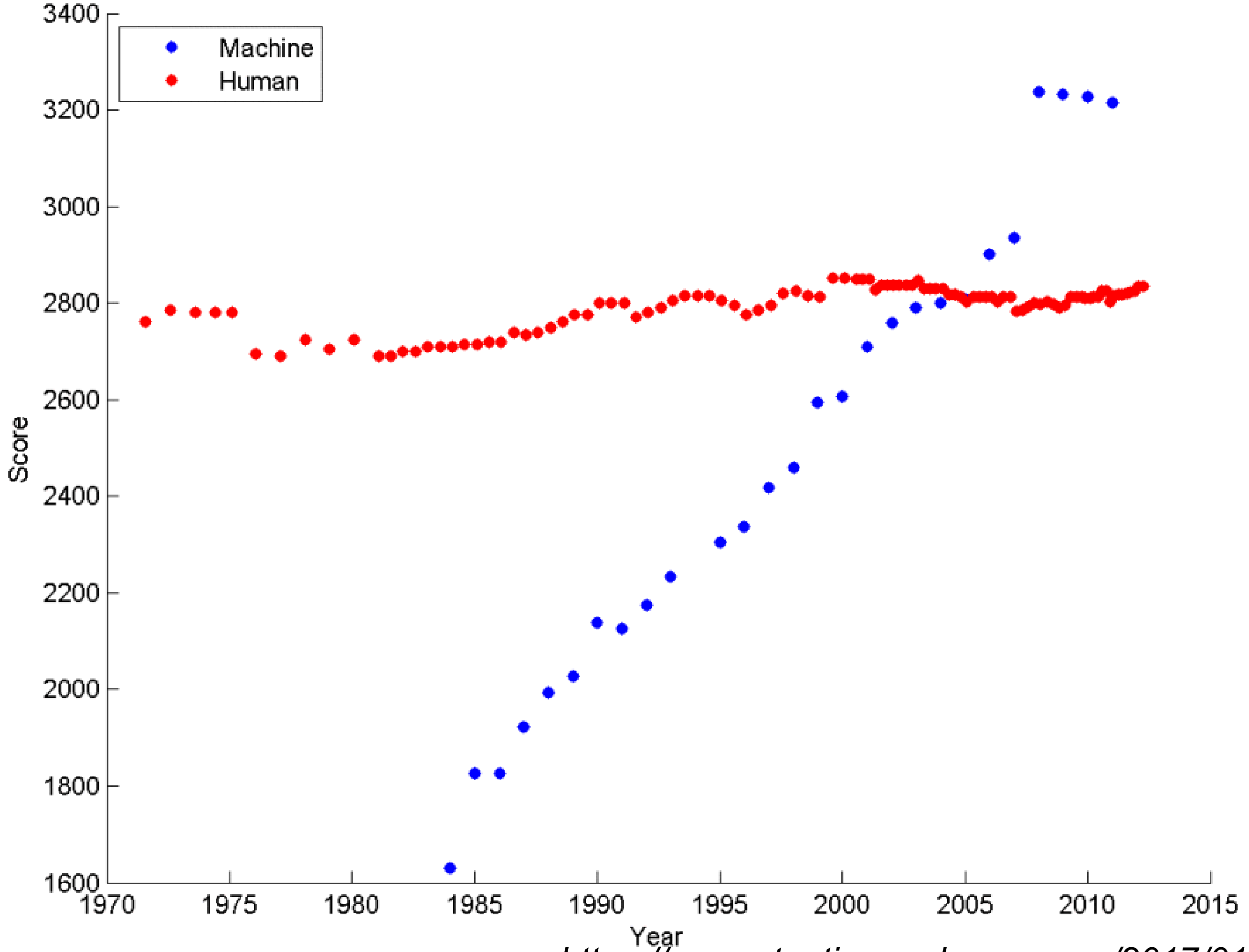
Games: Outline of Unit

- Part 1: Games as Search
  - Motivation
  - Game-playing AI successes
  - Game Trees
  - Evaluation Functions
- Part II: Adversarial Search
  - The Minimax Rule
  - Alpha-Beta Pruning

May 11, 1997


may 11th
game 6 : may 11 @ 3:00PM EDT | 19:00 GMT
kasparov 2.5 deep blue 2.5
Home
The match
The players
The technology
Community
Deep Blue Wins
3.5 to 2.5
KASPAROV VS DEEP BLUE
the rematch
With a dramatic victory in Game 6, Deep Blue won its six-game rematch with Champion Garry Kasparov
OVERVIEW
EVENT COVERAGE
MATCH NEWS
MAIN STORIES
Commentary
George Plimpton on chess, Kasparov, and the limitations of computers
Read the article
Commentary
Vishwanathan Anand on the legacy of Kasparov vs. Deep Blue
Read the article
Club Kasparov
Visit the virtual home of the world's greatest chess player.
Guest essays
Thoughts on chess, computers, and what it all means
Read the essays...
Community
During the rematch, more than 20,000 people from 120 countries joined the community to talk about the match.
Clips from the rematch
Video footage from the games
Highlights from the games

○Ratings of human and computer chess champions



https://srconstantin.wordpress.com/2017/01/28/performance-trends-in-ai/

Artificial intelligence (AI)

AlphaGo seals 4-1 victory over Go grandmaster Lee Sedol

DeepMind's artificial intelligence astonishes fans to defeat human opponent and offers evidence computer software has mastered a major challenge

Steven Borowiec

Tuesday 15 March 2016 06.12 EDT

The world's top Go player, Lee Sedol, lost the final game of the Google DeepMind challenge match. Photograph: Yonhap/Reuters

Google DeepMind's AlphaGo program triumphed in its final game against South Korean Go grandmaster Lee Sedol to win the series 4-1, providing further evidence of the landmark achievement for an artificial intelligence program.

Lee started Tuesday's game strongly, taking advantage of an early mistake by AlphaGo. But in the end, Lee was unable to hold off a comeback by his opponent, which won a narrow victory.

The Simplest Game Environment

- *Multiagent*
- *Static:* No change while an agent is deliberating.
- *Discrete:* A finite set of percepts and actions.
- *Fully observable:* An agent's sensors give it the complete state of the environment.
- *Strategic:* The next state is determined by the current state and the action executed by the agent and the actions of one other agent.

"Strategic" is another way of saying deterministic in the context of a multiagent game

There's tension between "fully observable" and "multiagent." Why?

Key properties of our games

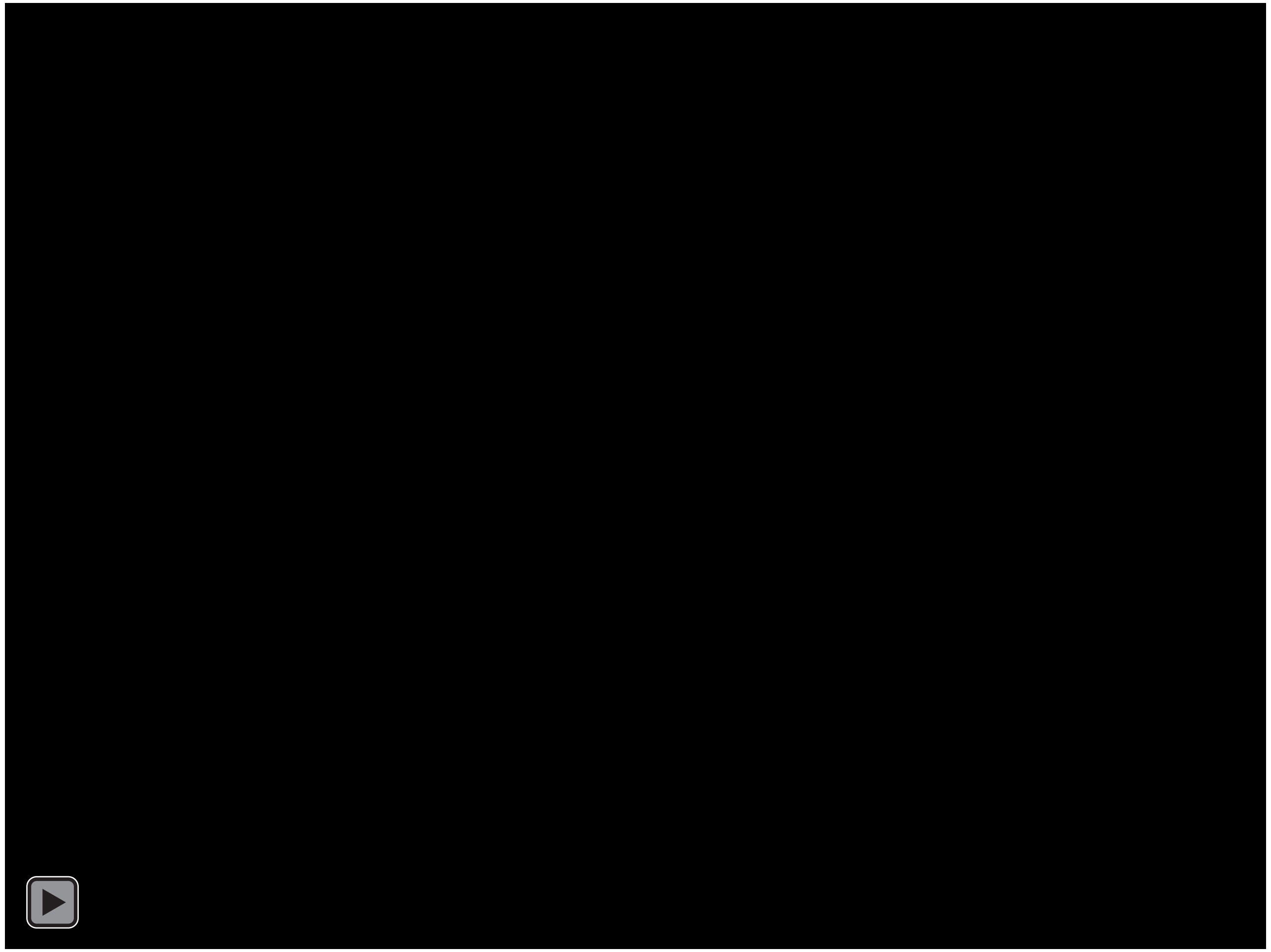
1. **Two players alternate moves**
2. **Zero-sum: one player's loss is another's gain**
3. **Clear set of legal moves**
4. **Well-defined outcomes (e.g. win, lose, draw)**

- **Examples:**
  - Chess, Checkers, Go,
  - Mancala, Tic-Tac-Toe, Othello …

More complicated games

- Most card games (e.g. Hearts, Bridge, etc.) and Scrabble
  - Stochastic, not deterministic
  - Not fully observable: lacking in perfect information
- Real-time strategy games
  - Continuous rather than discrete
  - No pause between actions, don’t take turns
- Cooperative games

Pac-Man



https://youtu.be/-CbyAk3Sn9I

Formalizing the Game setup

1. **Two players: *MAX* and *MIN; MAX* moves first.**
2. ***MAX* and *MIN* take turns until the game is over.**
3. **Winner gets award, loser gets penalty.**

- **Games as *search*:**
  - *Initial state*: e.g. board configuration of chess
  - *Successor function*: list of (move,state) pairs specifying legal moves.
  - *Terminal test*: Is the game finished?
  - *Utility function*: Gives numerical value of terminal states.
    e.g. win (+∞), lose (-∞) and draw (0)
  - *MAX* uses search tree to determine next move.

How to Play a Game by Searching

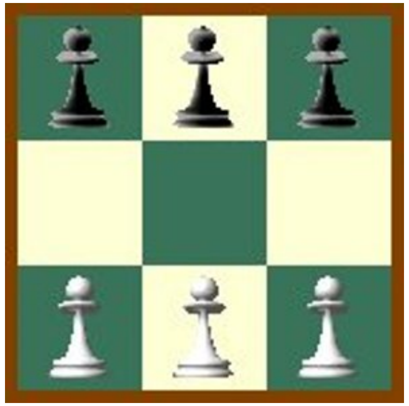
- **General Scheme (from one player's perspective!)**
  1. **Consider all legal successors to the current state ('board position')**
  2. **Evaluate each successor board position**
  3. **Pick the move which leads to the best board position.**
  4. **After your opponent moves,** repeat.

- **Design issues**
  1. Representing the 'board'
  2. Representing legal next boards
  3. Evaluating positions
  4. Looking ahead

Do any of these pose new challenges compared to what we've seen before?

Hexapawn: A very simple Game

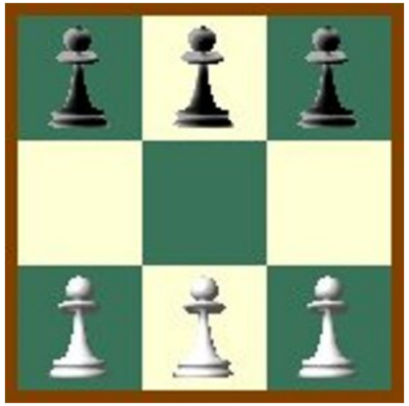
- **Hexapawn is played on a 3x3 chessboard**



- **Only standard pawn moves:**
  1. A pawn moves forward one square onto an empty square
  2. A pawn "captures" an opponent pawn by moving diagonally forward one square, if that square contains an opposing pawn. The opposing pawn is removed from the board.

Hexapawn: A very simple Game

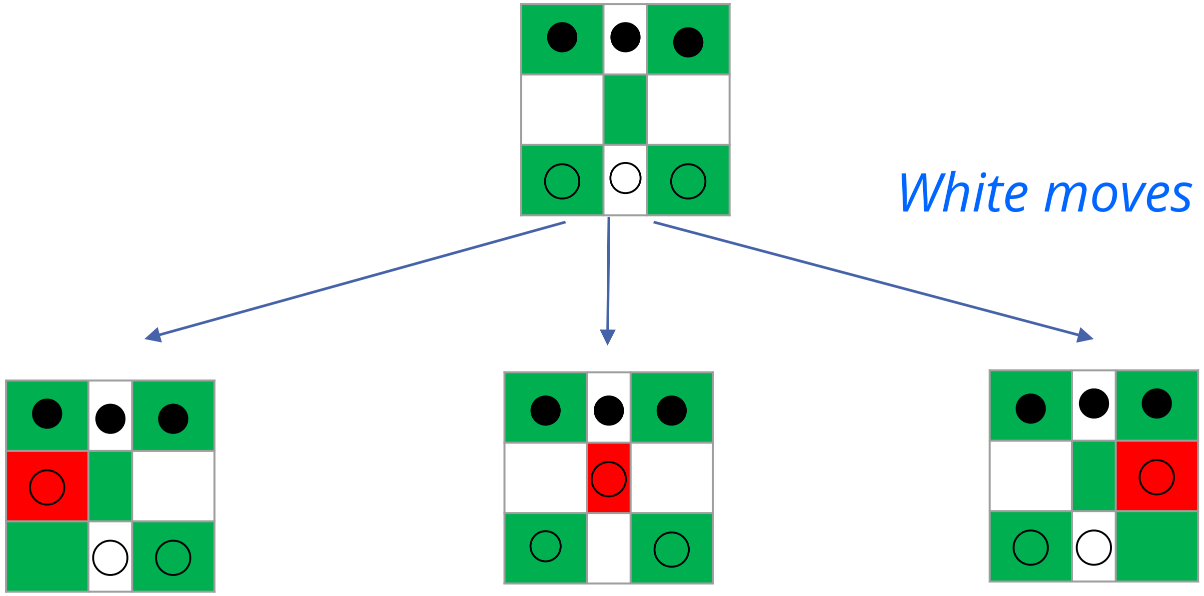
- **Hexapawn is played on a 3x3 chessboard**



- **Player P_1 wins the game against P_2 when:**
  - One of P_1's pawns reaches the far side of the board, or
  - P_2 cannot move because no legal move is possible.
  - P_2 has no pawns left.

  (Invented by Martin Gardner in 1962, with learning "program" using match boxes.)

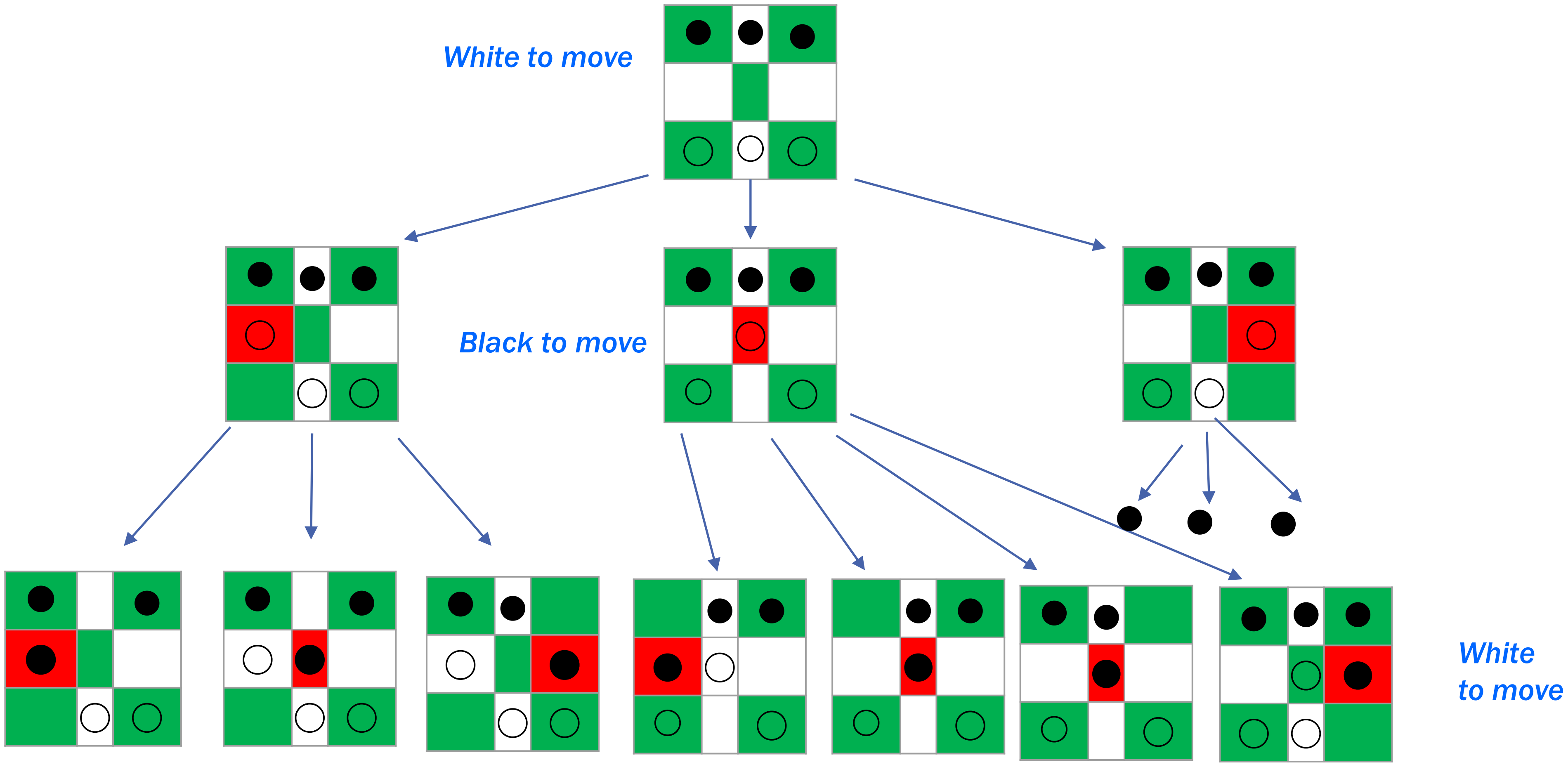
Hexapawn: Three Possible First Moves



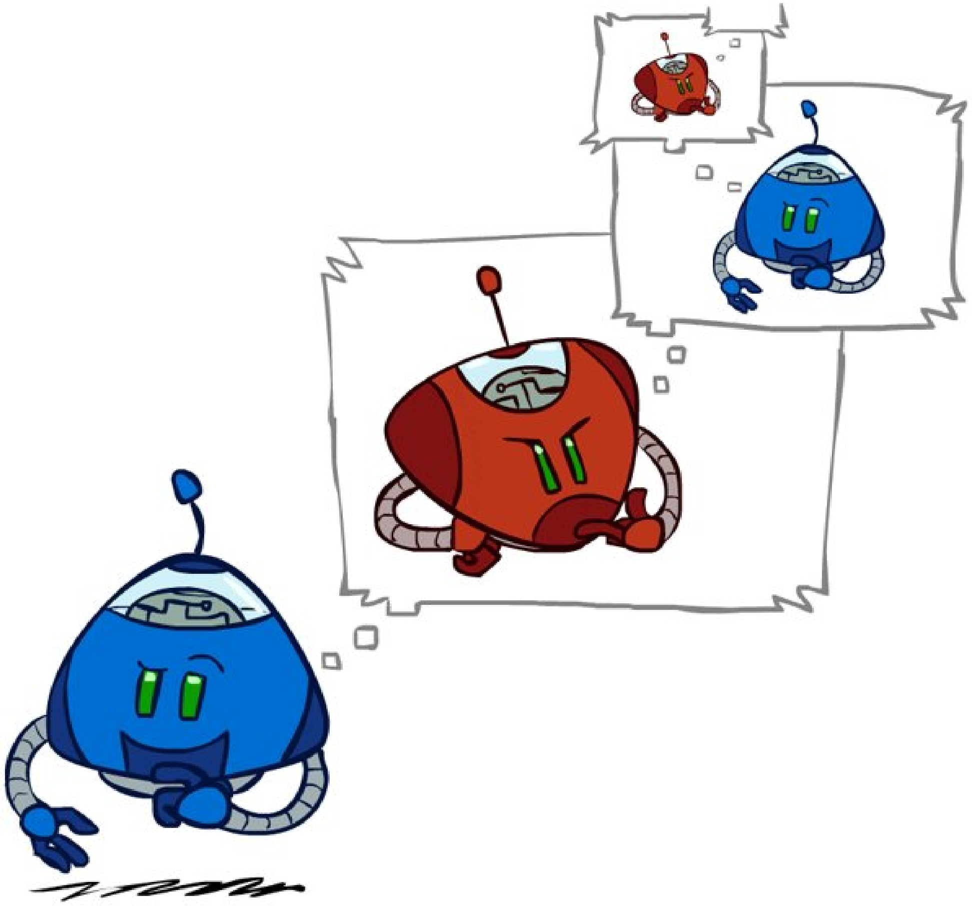
Game Trees

- **Represent the game problem space by a tree:**
  - Nodes represent ‘board positions’; edges represent legal moves.
  - Root node is the first position in which a decision must be made.

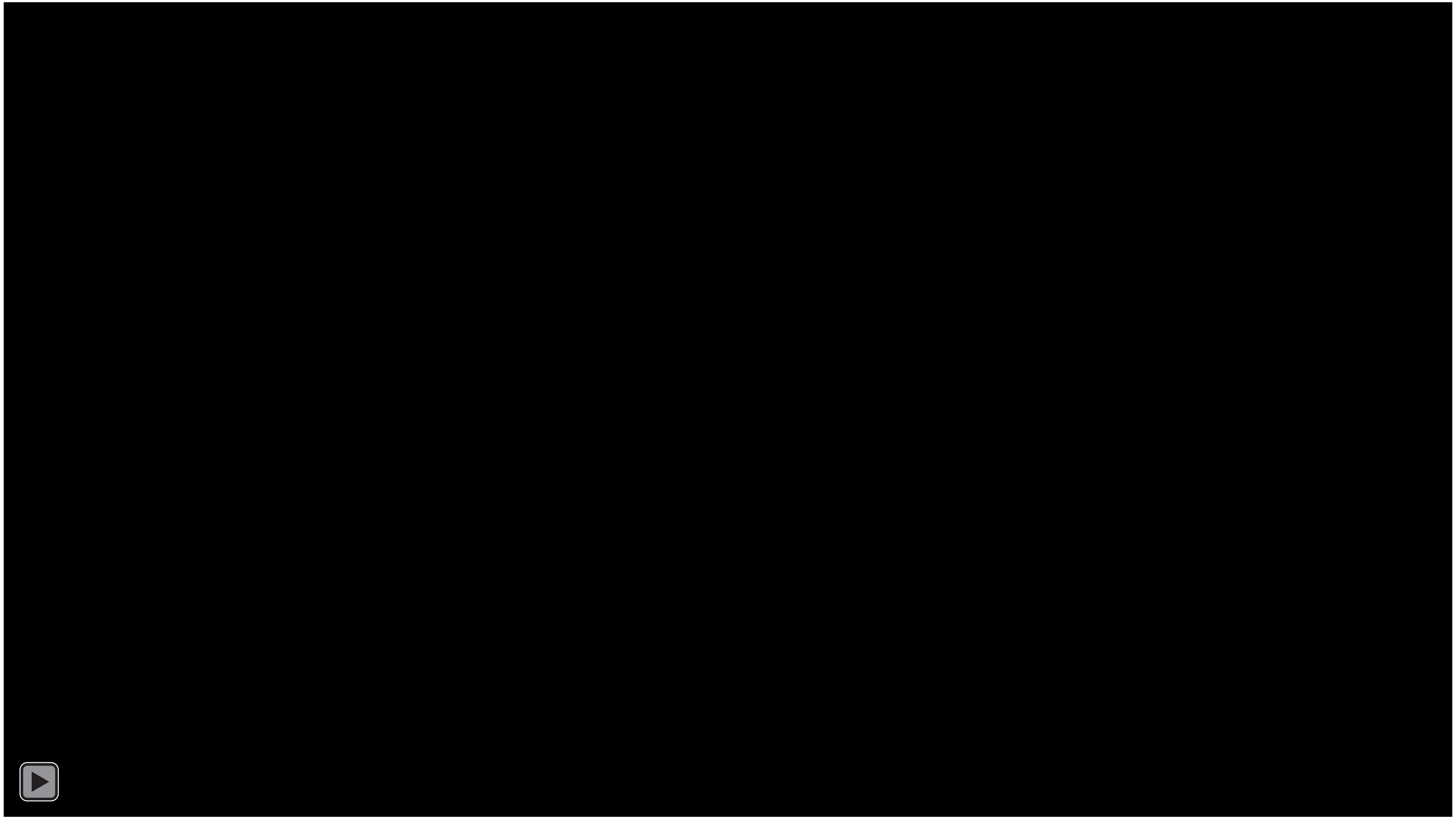
Hexapawn: Simplified Game Tree for 2 Moves



- Adversarial Search



- Battle of Wits



MAX & MIN Nodes : An egocentric view

- **Two players: MAX, MAX's opponent MIN**
- ***All play is computed from MAX's vantage point.***
- **When MAX moves, MAX attempts to MAXimize MAX's outcome.**
- **When MAX's opponent moves, they attempt to MINimize MAX's outcome.**
  - **WE TYPICALLY ASSUME MAX MOVES FIRST:**

- **Label the root (level 0) MAX**
- **Alternate MAX/MIN labels at each successive tree level *(ply)*.**
- ***Even levels* represent turns for MAX**
- ***Odd levels* represent turns for MIN**

Each player's move is a ply, so after two plies, MAX plays again.

Game Trees

- **Represent the game problem space by a tree:**
  - Nodes represent ‘board positions’; edges represent legal moves.
  - Root node is the first position in which a decision must be made.

- **Evaluation function *f* assigns real-number scores to `board positions’ *without reference to path***
- **Terminal nodes represent ways the game could end, labeled with the desirability of that ending (e.g. win/lose/draw or a numerical score)**

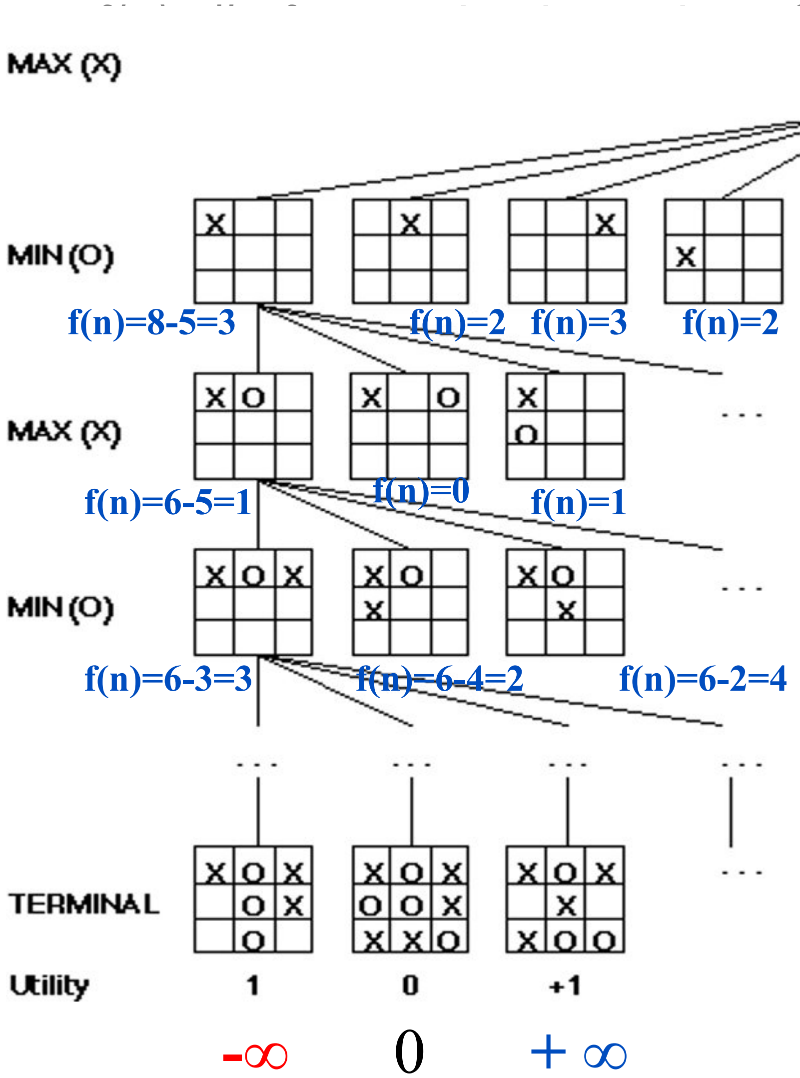
Games are (usually) concerned with the destination rather than the journey.

Evaluation functions: *f(n)*

- **Evaluates how good a 'board position' is**
- **Based on *static features* of that board alone**
- **Zero-sum assumption lets us use one function to describe goodness for both players.**
  - ***f(n)>0*** if MAX is winning in position *n*
  - ***f(n)=0*** if position *n* is tied
  - ***f(n)<0*** if MIN is winning in position *n*
- **Build using expert knowledge,**
  - Tic-tac-toe***: f(n)=(# of 3 lengths open for MAX)- (# open for MIN)***

Evaluation function at a terminal state is usually defined separately (e.g. positive inf, zero, negative inf)

A Partial Game Tree for Tic-Tac-Toe



f(n)=# of potential three-lines for X – # of potential three-line for O

f(n)=0 if n is a terminal tie
f(n)=+ ∞ if n is a terminal win
f(n)=- ∞ if n is a terminal loss

Chess Evaluation Functions

- **Claude Shannon argued for a chess evaluation function in a 1950 paper**
- **Alan Turing defined function in 1948:**
  f(n)=(sum of A's piece values)
  -(sum of B's piece values)
- **More complex: weighted sum of *positional* features:**
  $$\Sigma\, w_i\, feature_i(n)$$
- **Deep Blue had >8000 features**

Pawn	1.0
Knight	3.0
Bishop	3.25
Rook	5.0
Queen	9.0

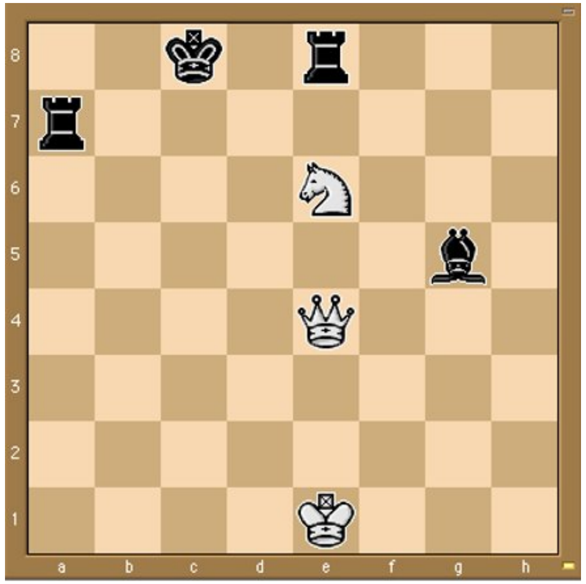
Pieces values for a simple Turing-style evaluation function often taught to novice chess players

Positive: rooks on open files, knights in closed positions, control of the center, developed pieces

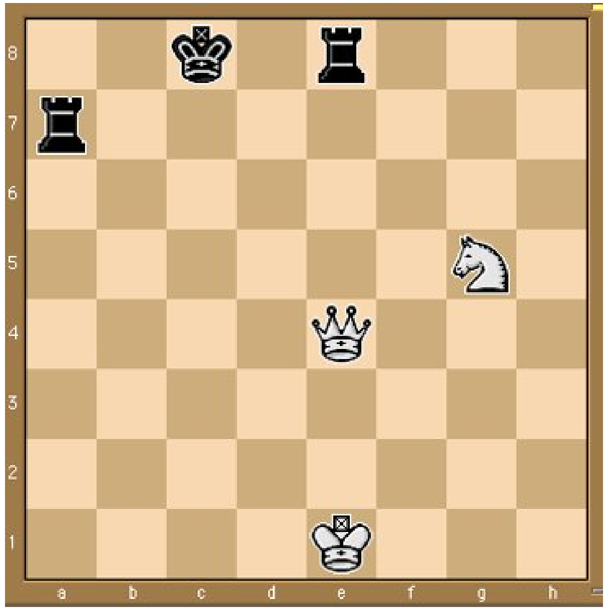
Negative: doubled pawns, wrong-colored bishops in closed positions, isolated pawns, pinned pieces

Examples of more complex features

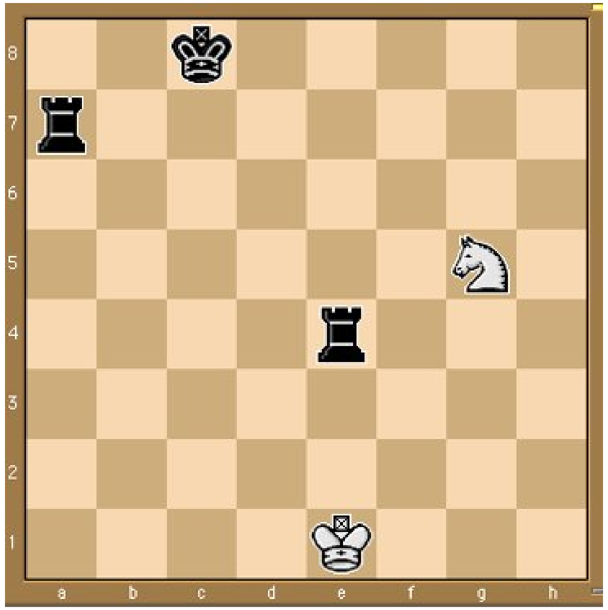
Some Chess Positions and their Evaluations



White to move
f(n)=(9+3)-(5+5+3.25)
=-1.25



… Nxg5??
f(n)=(9+3)-(5+5)
=2



Uh-oh: Rxg4+
f(n)=(3)-(5+5)
=-7
And black may force checkmate

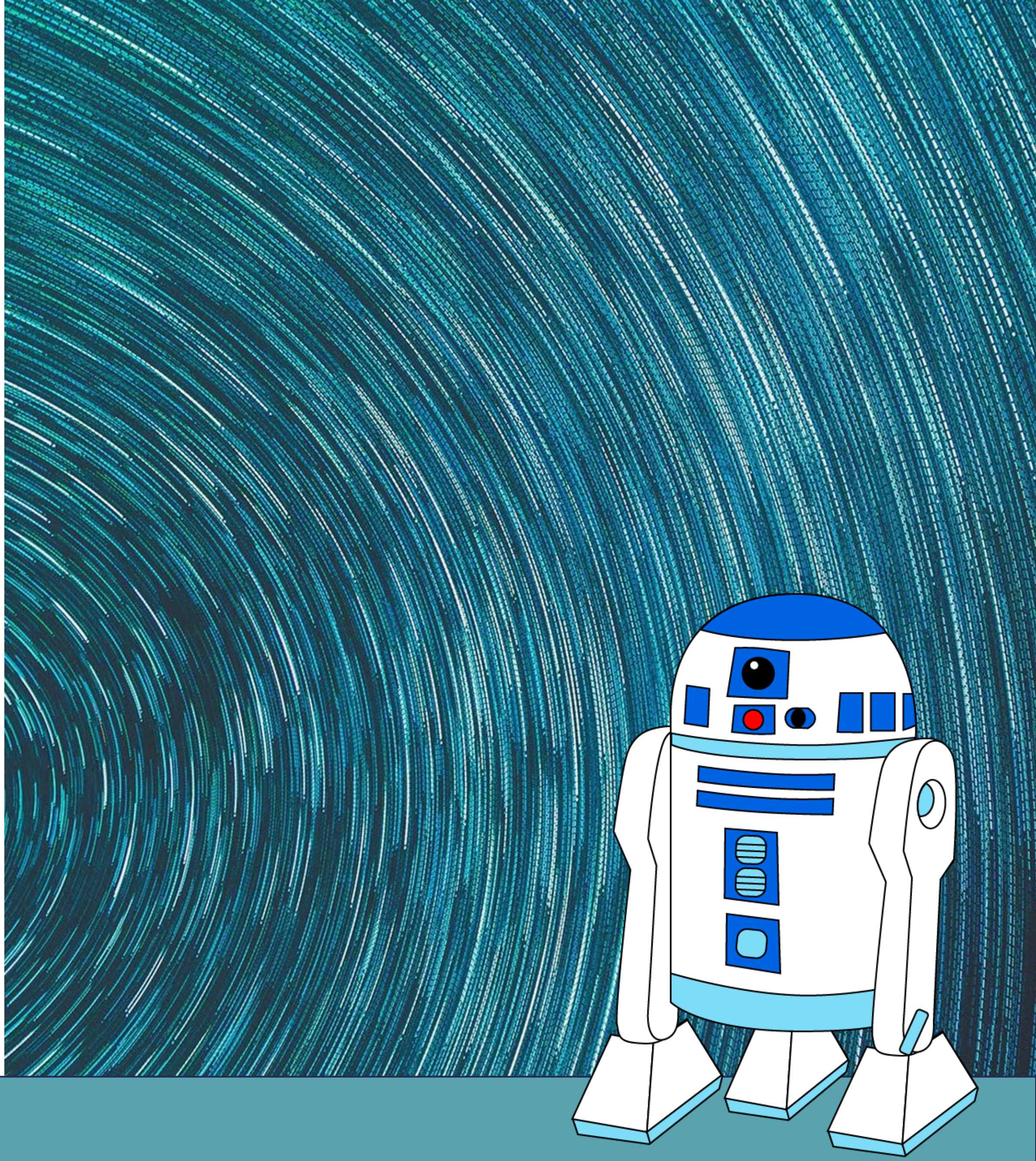
So, considering our opponent's possible responses would be wise.

The Minimax Rule (AIMA 5.2)

CIS 521:
ARTIFICIAL INTELLIGENCE

Minimax and Alpha-Beta Pruning

Harry Smith



Penn Engineering
UNIVERSITY of PENNSYLVANIA

The Minimax Rule: “Don’t play hope chess”

- *Idea*: Make the best move for MAX *assuming that MIN always replies with the best move for MIN*

- **Easily computed by a recursive process**
  - The ***backed-up value*** of each node in the tree is determined by the values of its children:
  - For a **MAX** node, the backed-up value is the ***maximum*** of the values of its children *(i.e. the best for MAX)*
  - For a **MIN** node, the backed-up value is the ***minimum*** of the values of its children *(i.e. the best for MIN)*

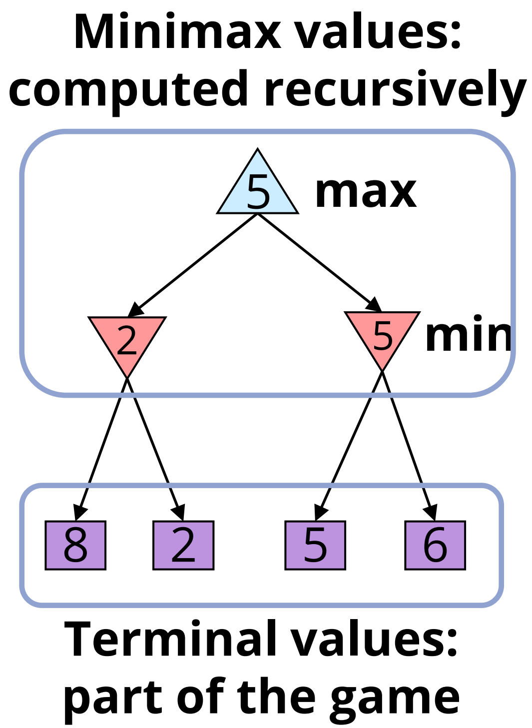
The Minimax Procedure

- Until game is over:

1. Start with the current position as a MAX node.
2. Expand the game tree a fixed number of *ply*.
3. Apply the evaluation function to the leaf positions.
4. Calculate back-up values bottom-up.
5. Pick the move assigned to MAX at the root
6. Wait for MIN to respond

Adversarial Search (Minimax)

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary



Minimax Implementation

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

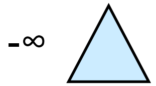
```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

What if MIN does not play optimally?

- **Definition of optimal play for MAX assumes MIN plays optimally:**
  - *Maximizes worst-case outcome* for MAX.
  - (Classic game theoretic strategy)

- **But if MIN does not play optimally, MAX will do even better.**
  - This theorem is not hard to prove

Minimax Example



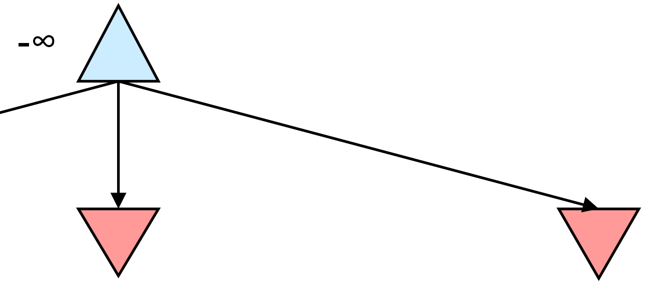

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



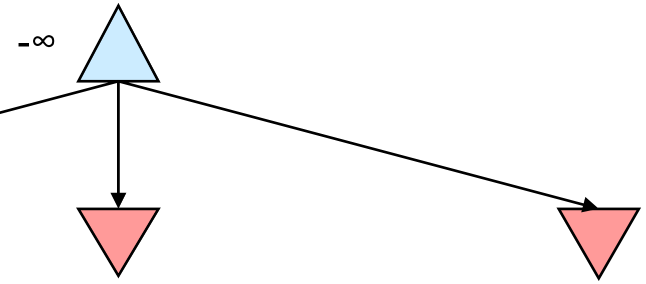

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



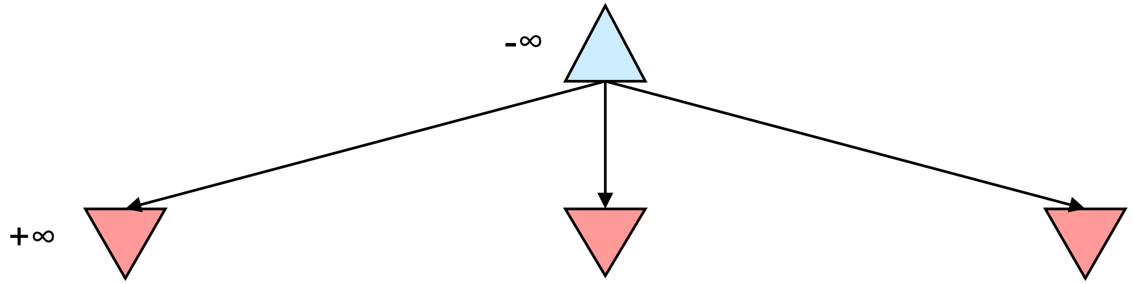

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



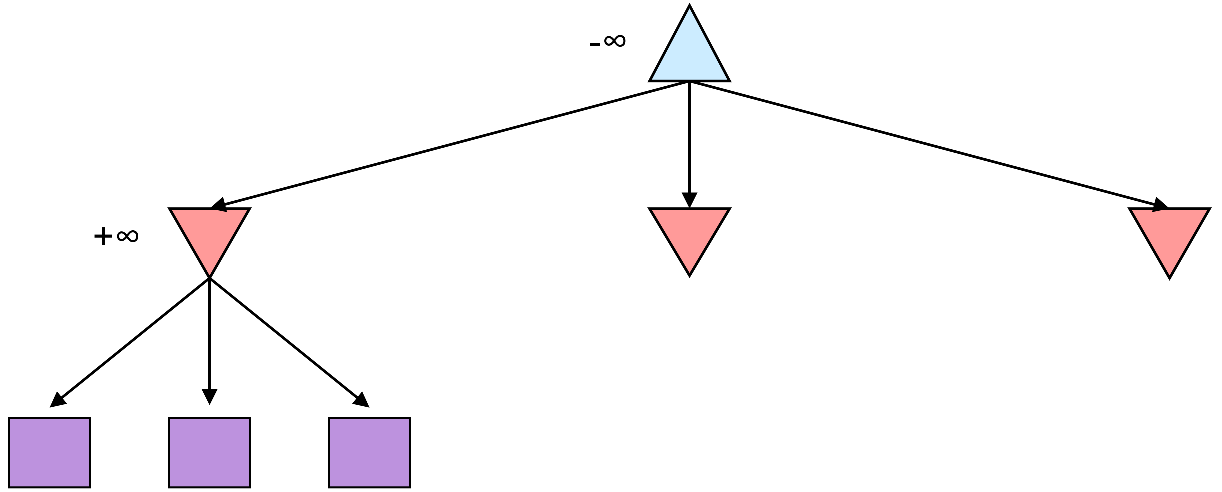

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



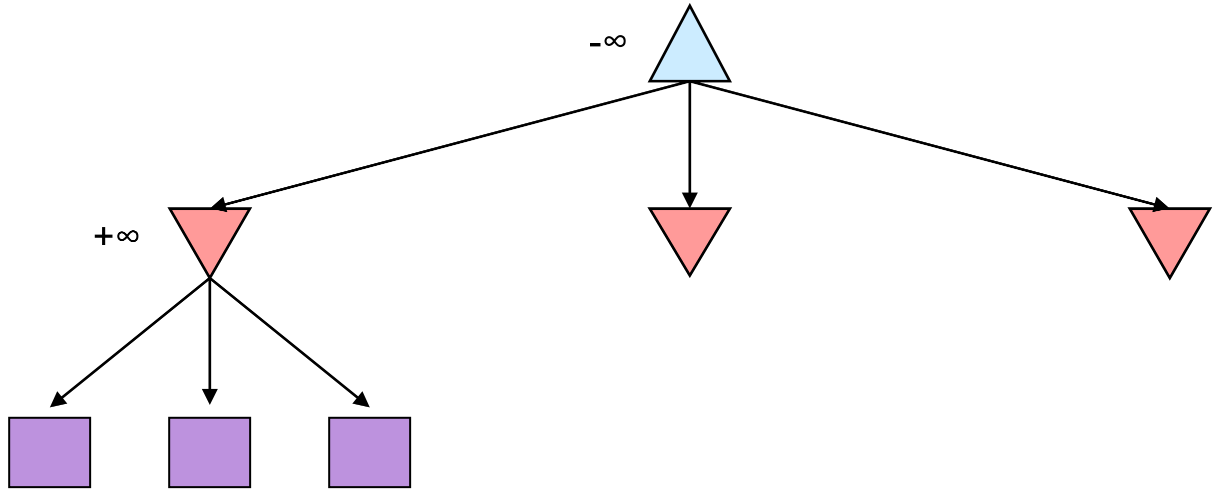

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



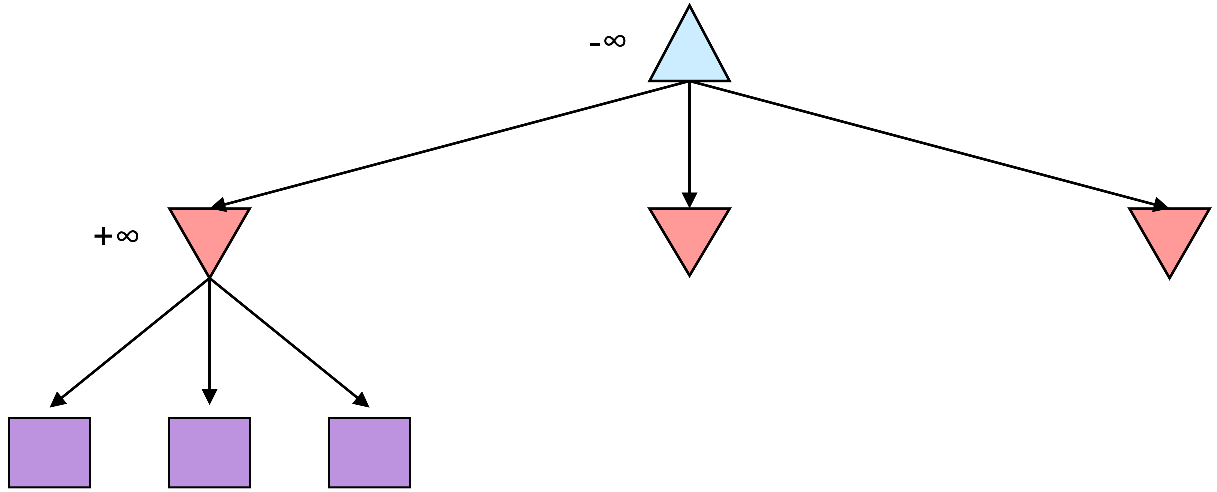

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



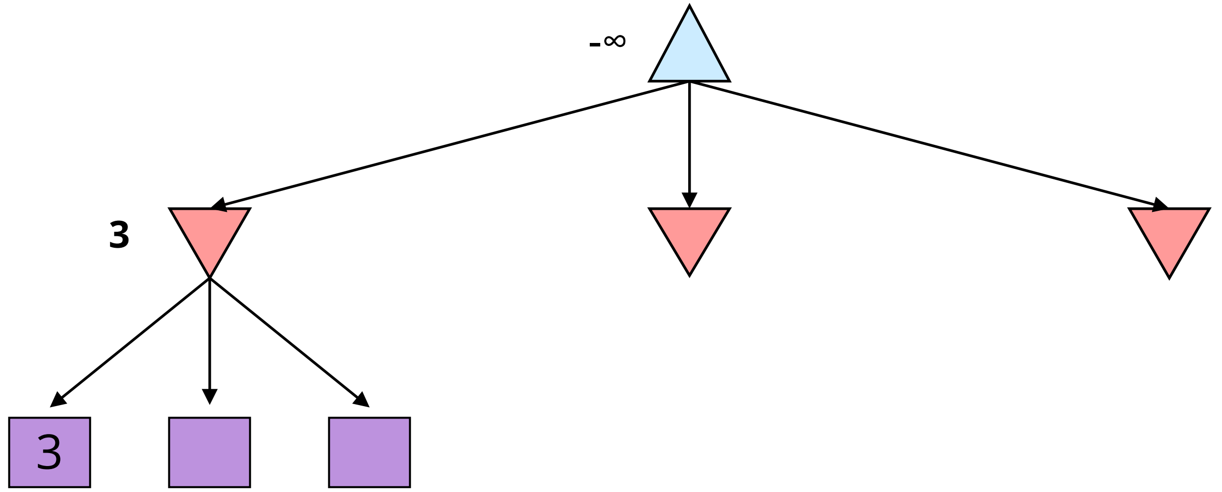

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



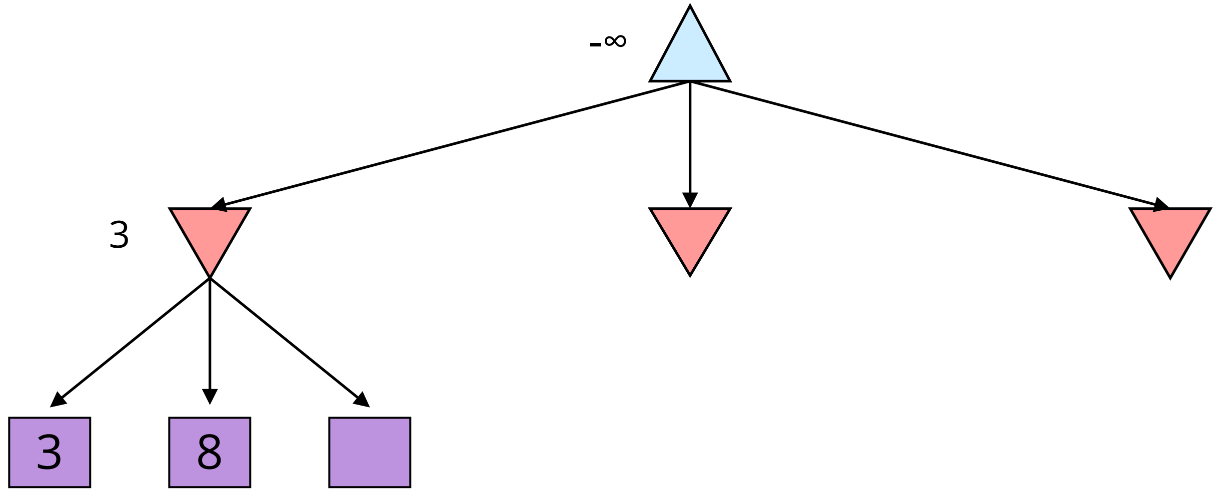

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



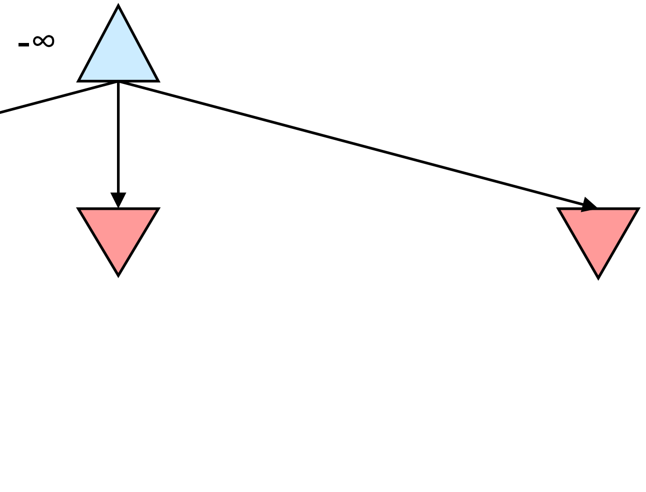

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



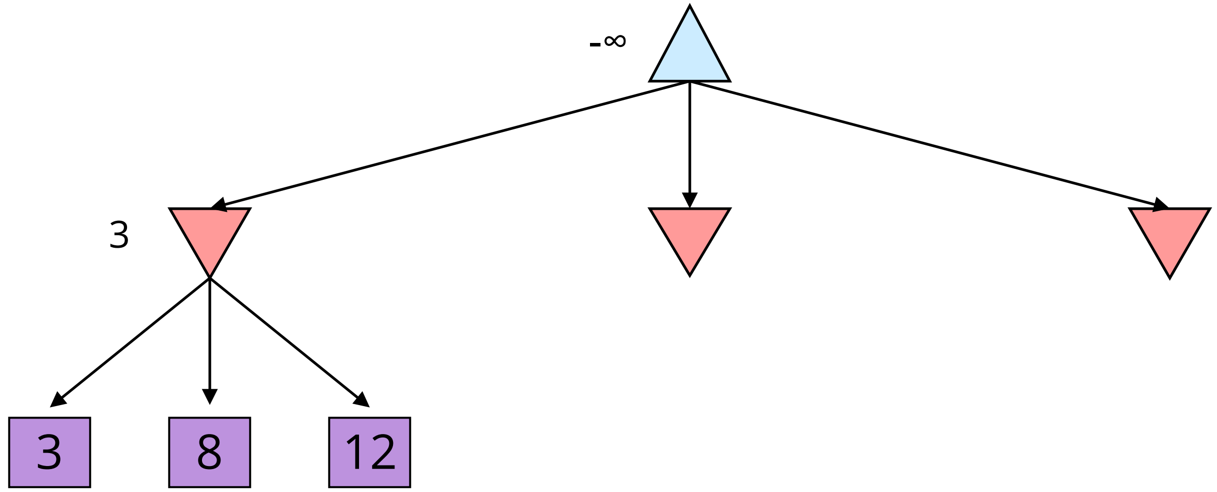

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



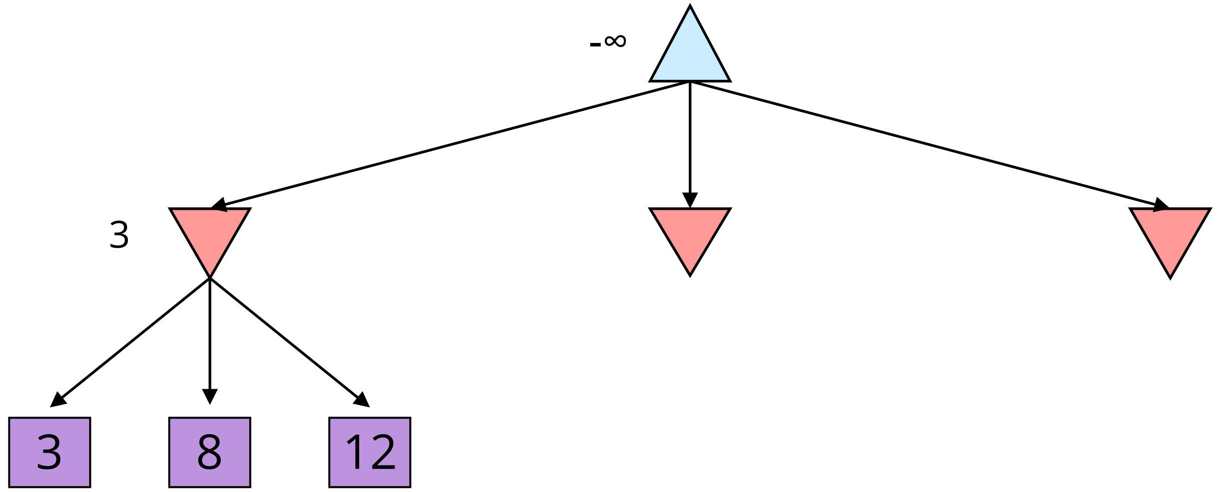

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



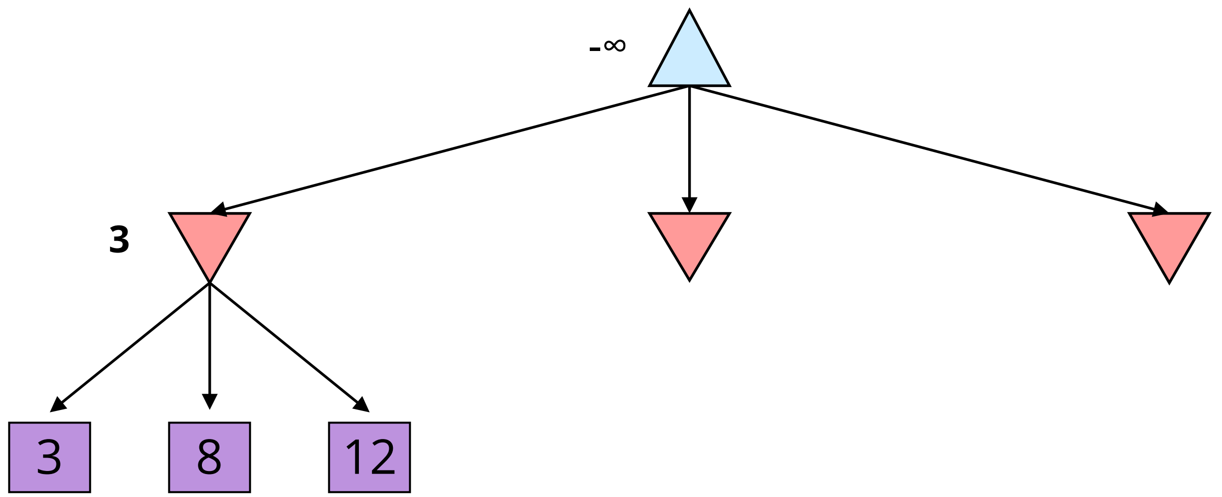

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



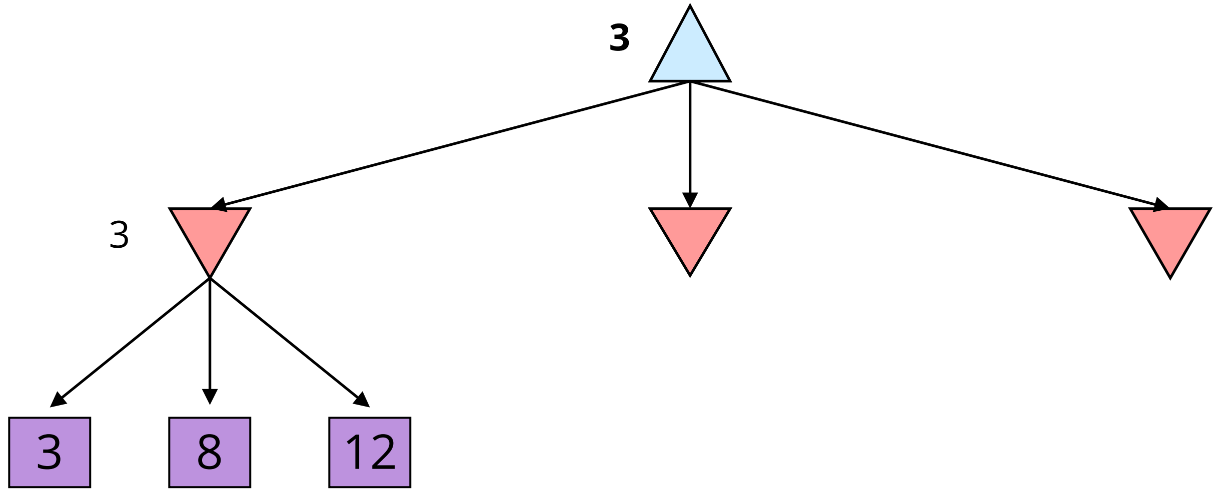

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



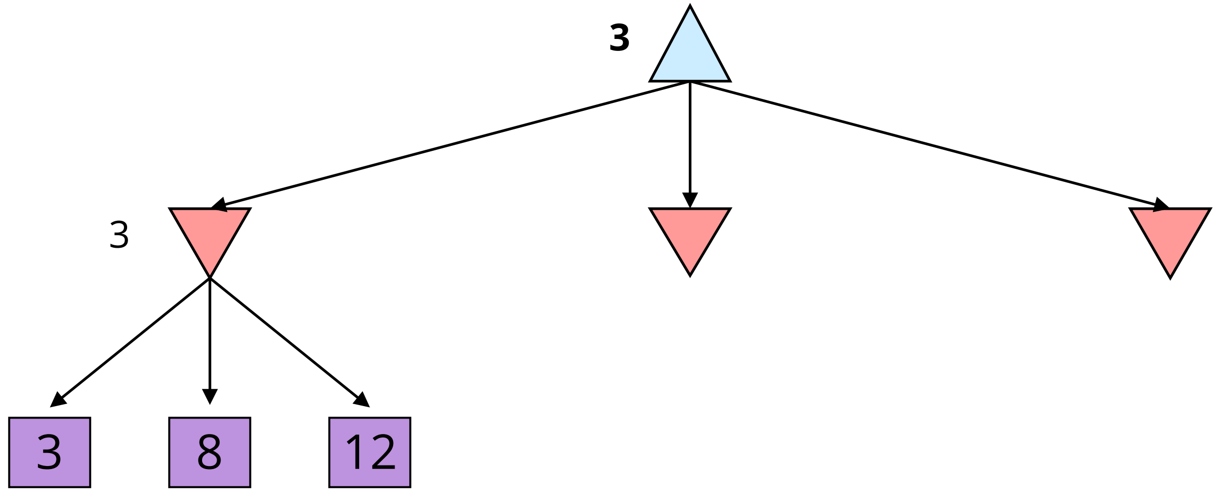

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



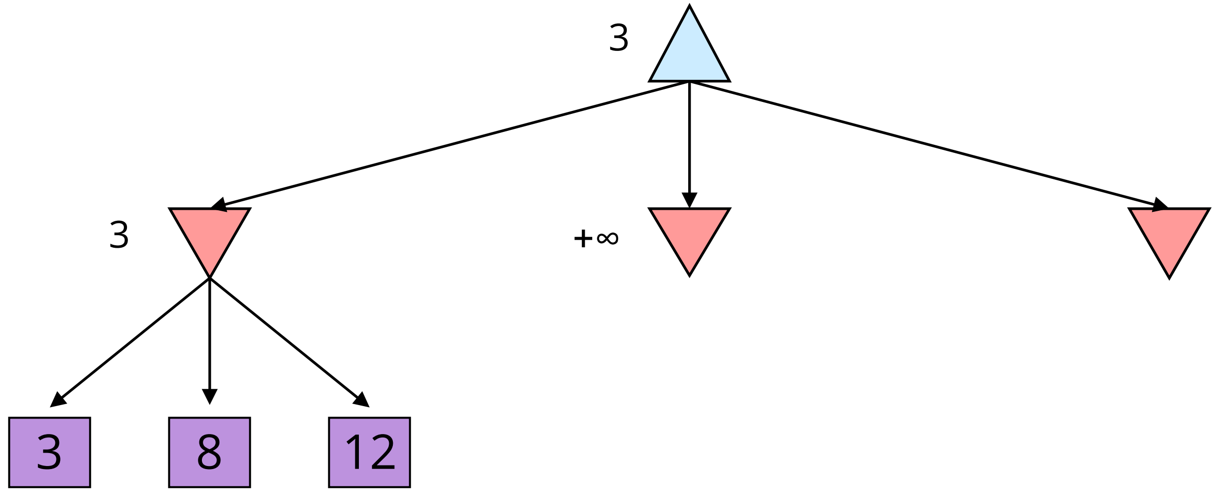

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



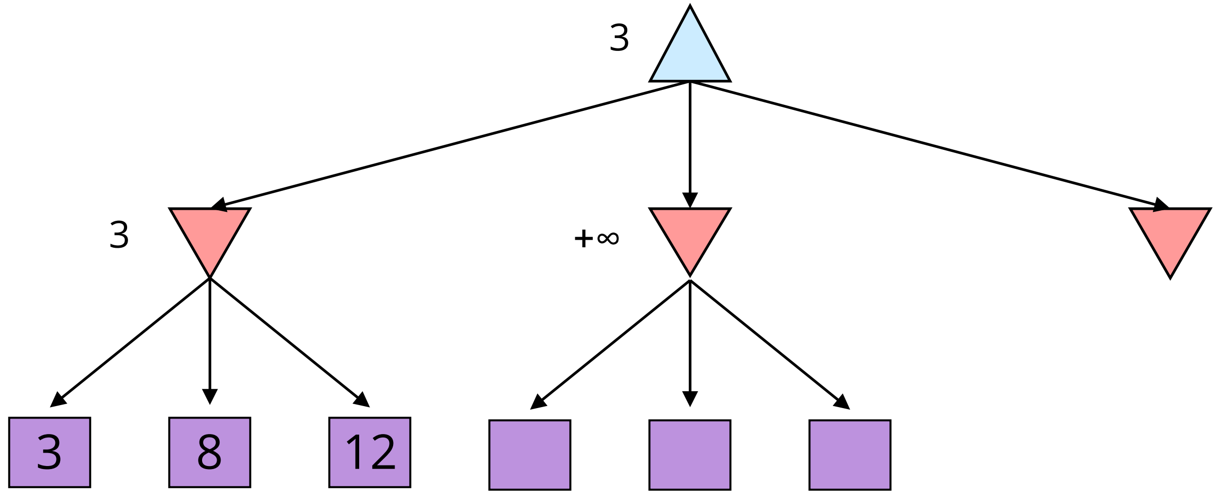

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



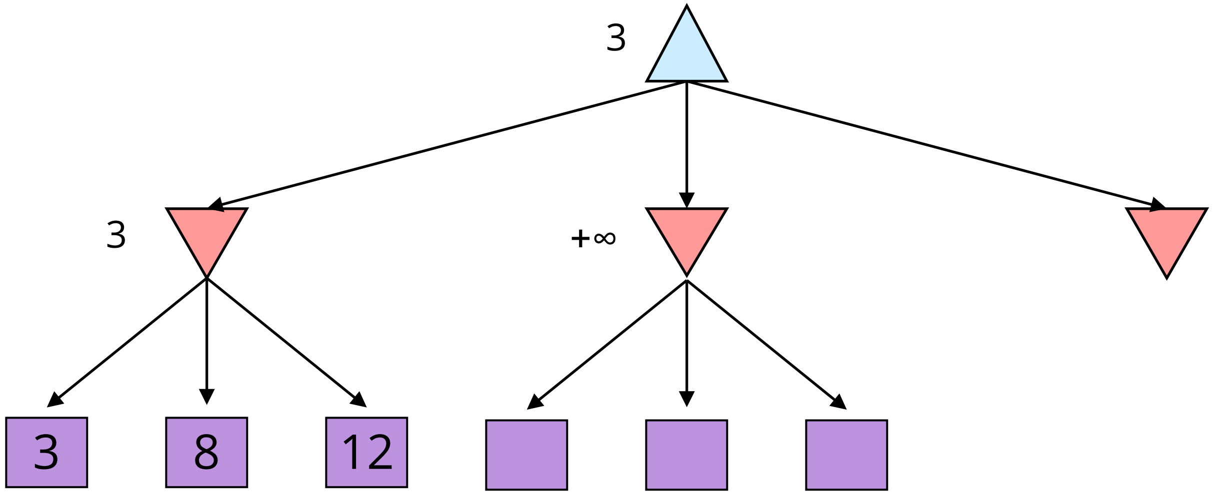
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



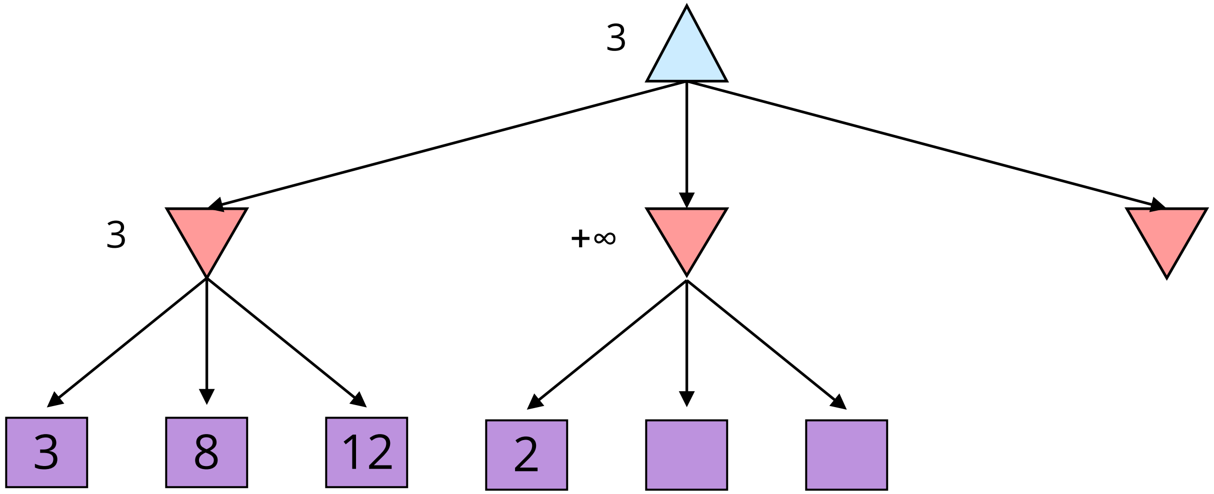
def max-value(state):
 if the state is a terminal state:
  return the state's utility
 initialize v = -∞
 for each successor of state:
  v = max(v, min-value(successor))
 return v

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

def min-value(state):
 if the state is a terminal state:
  return the state's utility
 initialize v = +∞
 for each successor of state:
  v = min(v, max-value(successor))
 return v

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



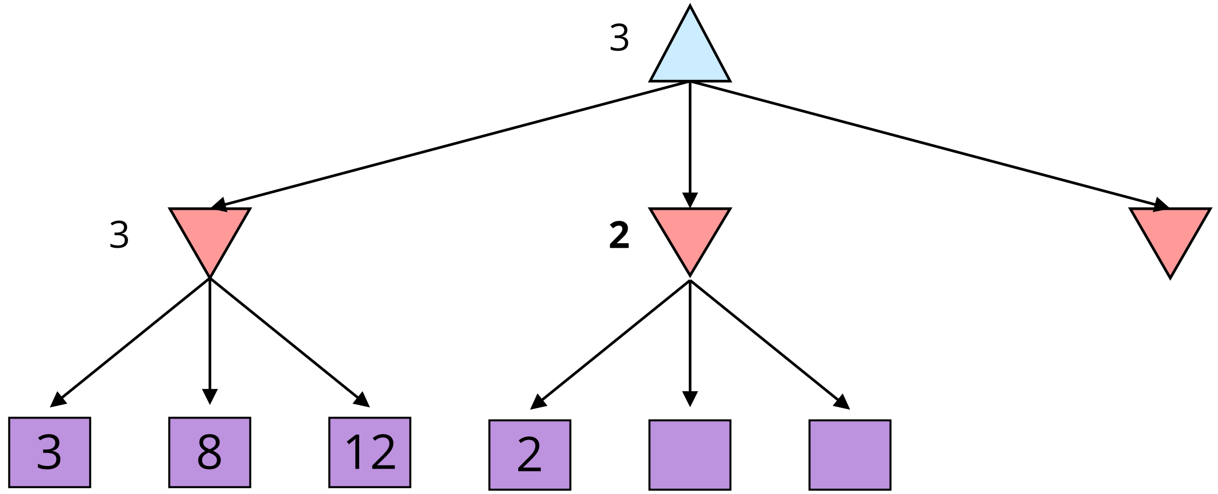

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



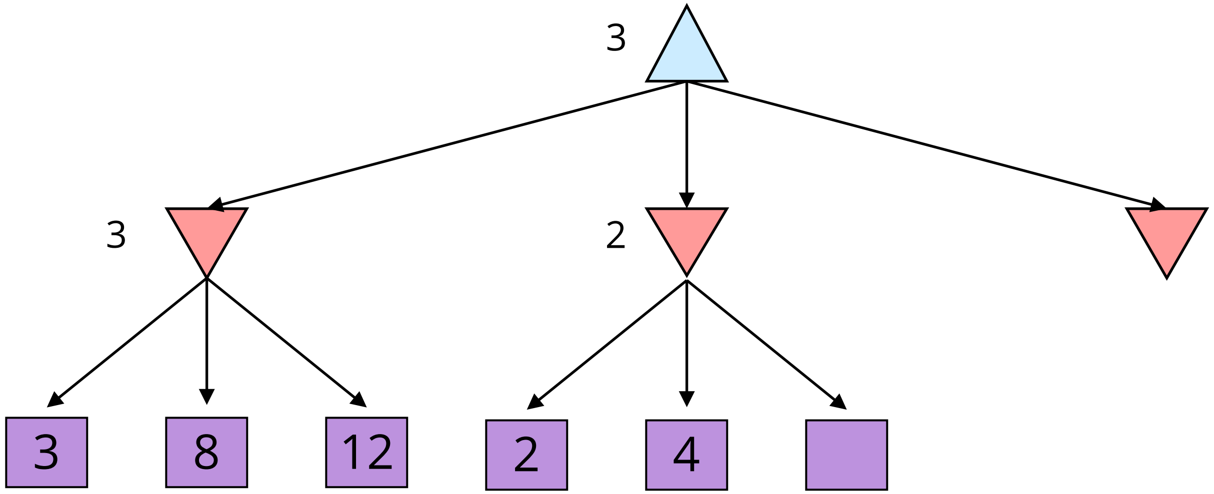

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



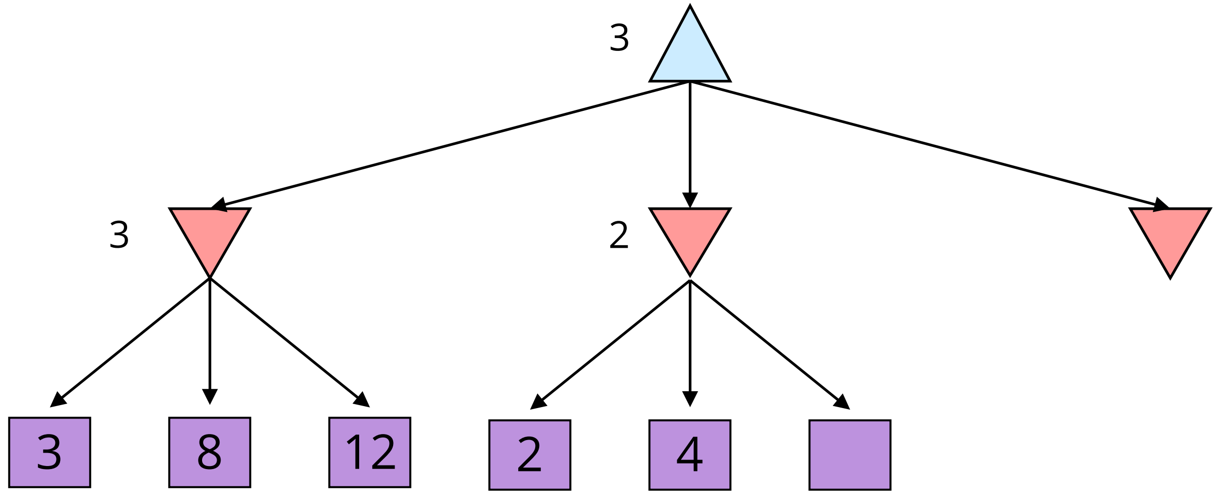

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



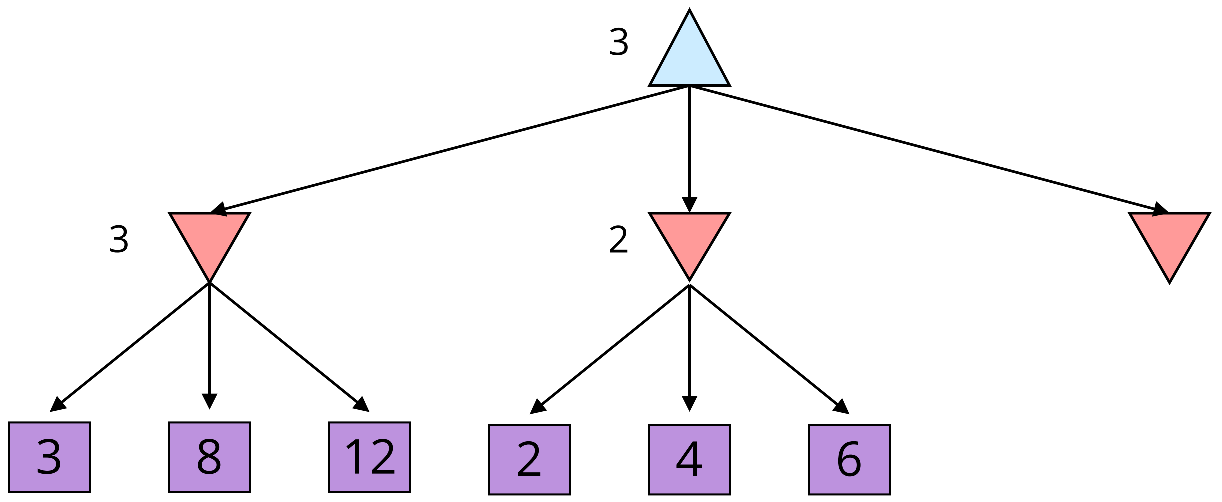

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



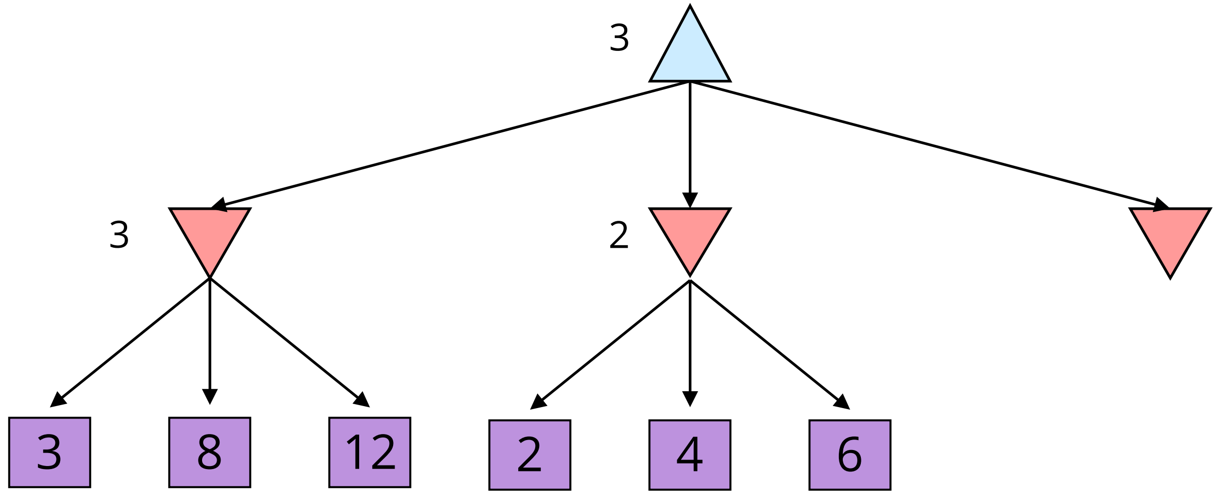

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



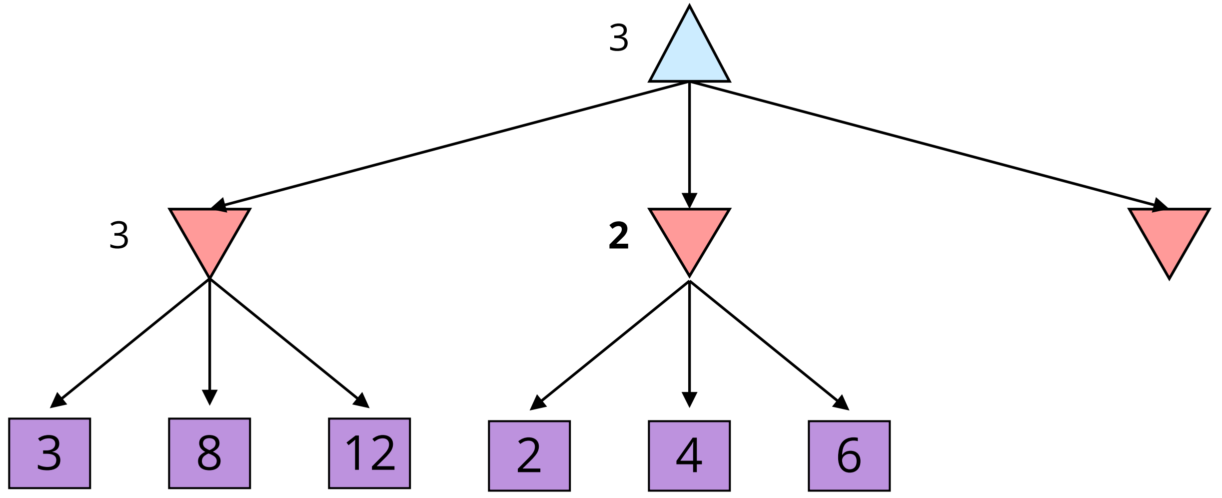

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



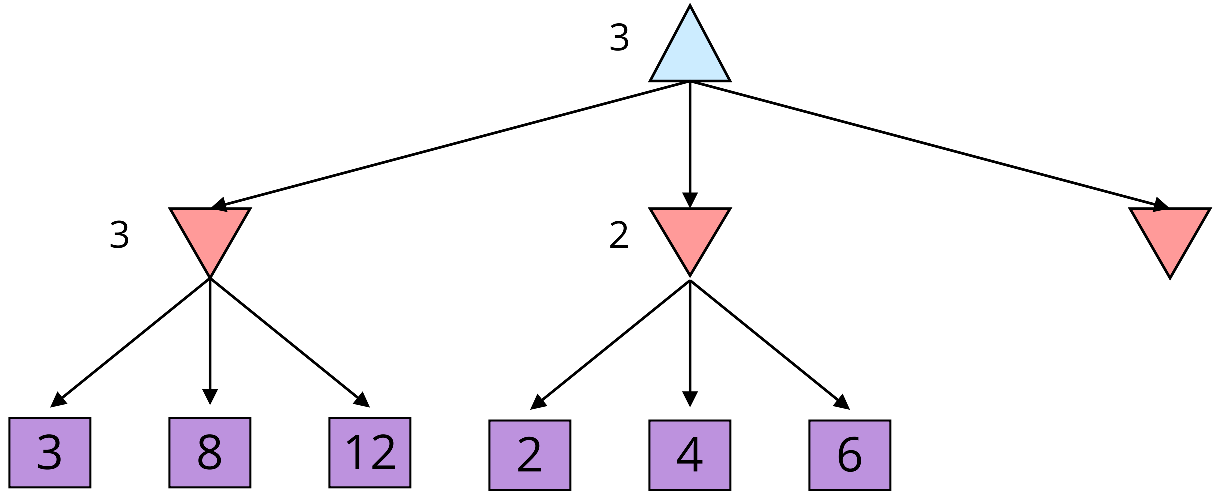

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



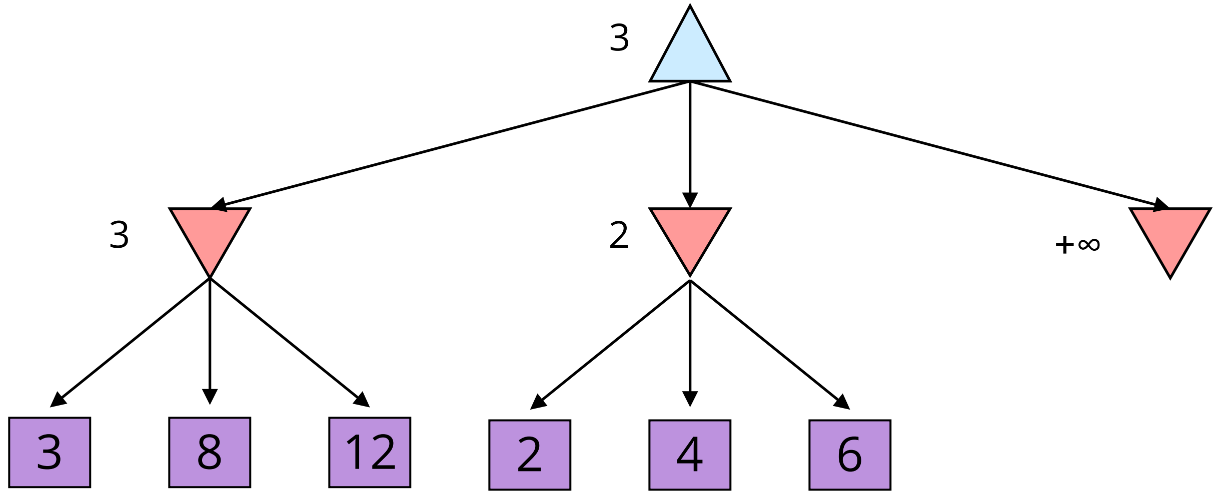

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



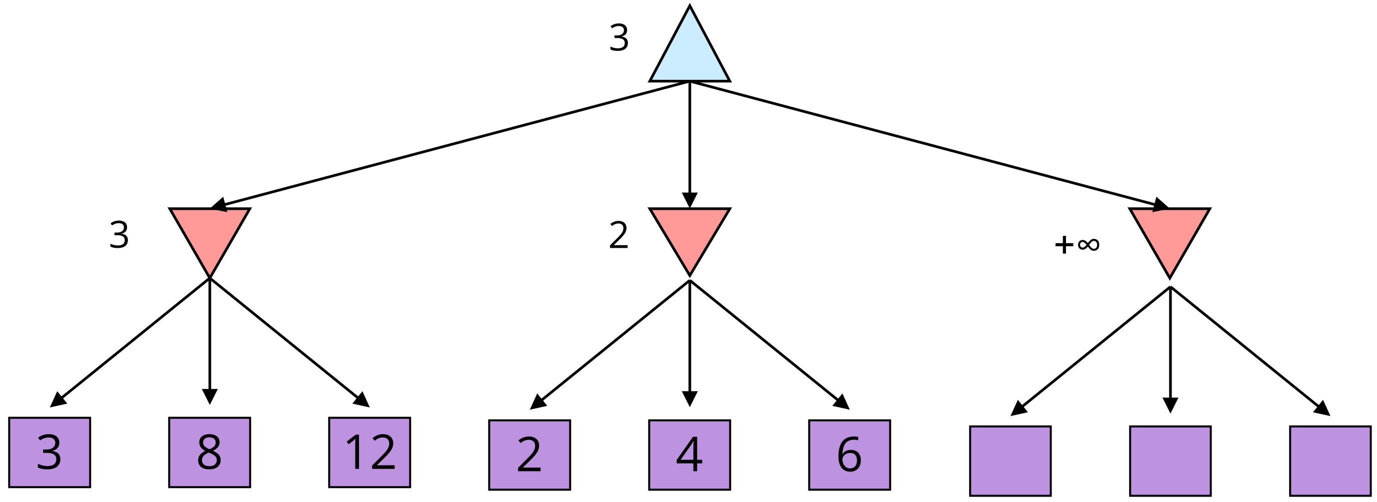

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



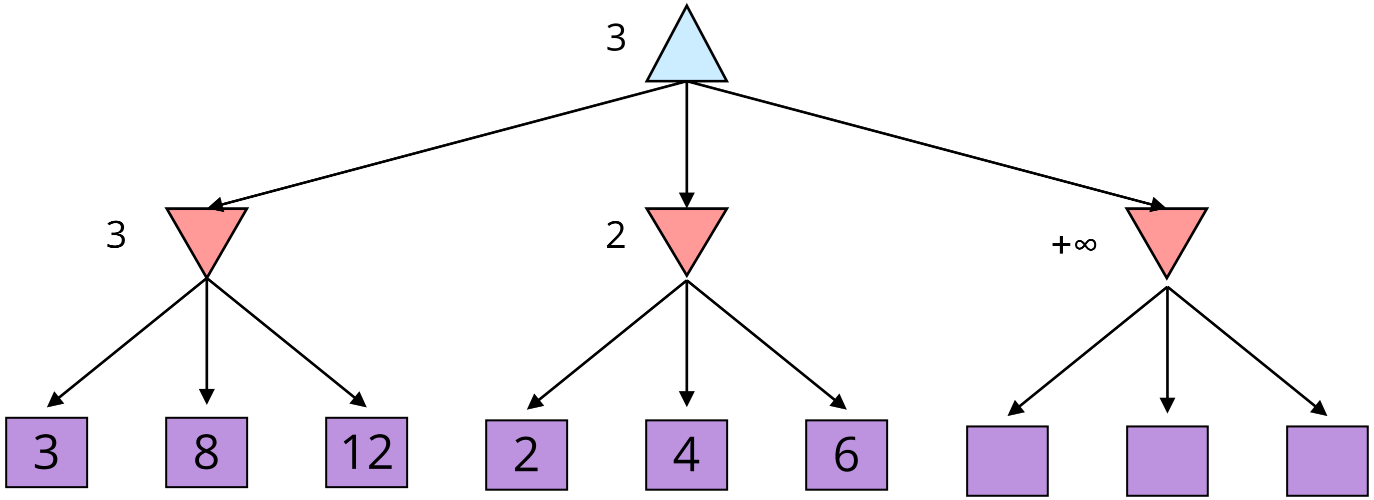

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



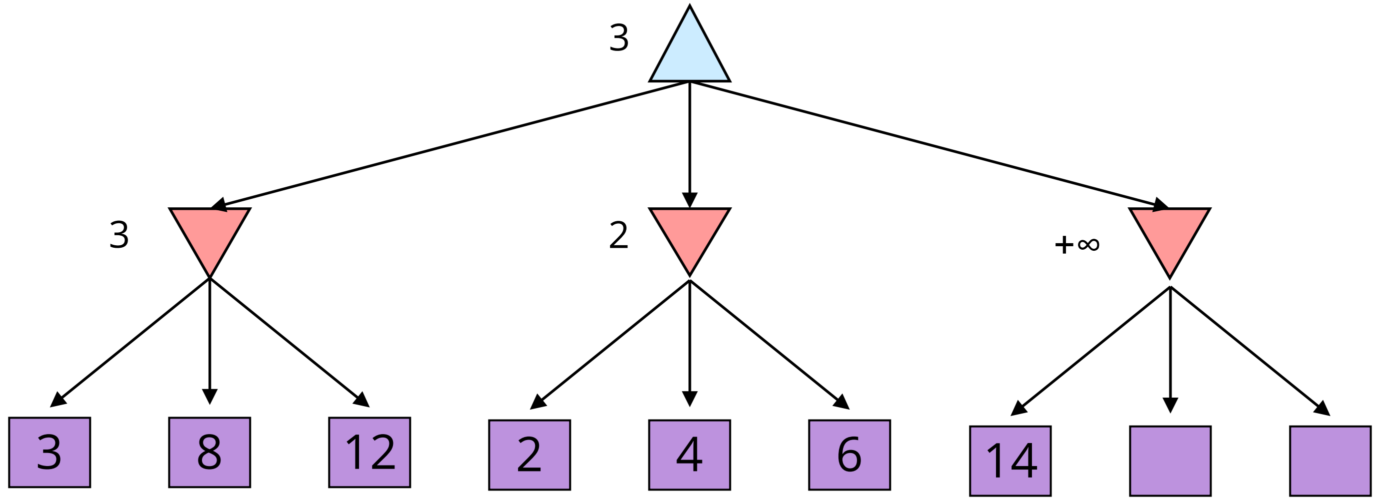

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



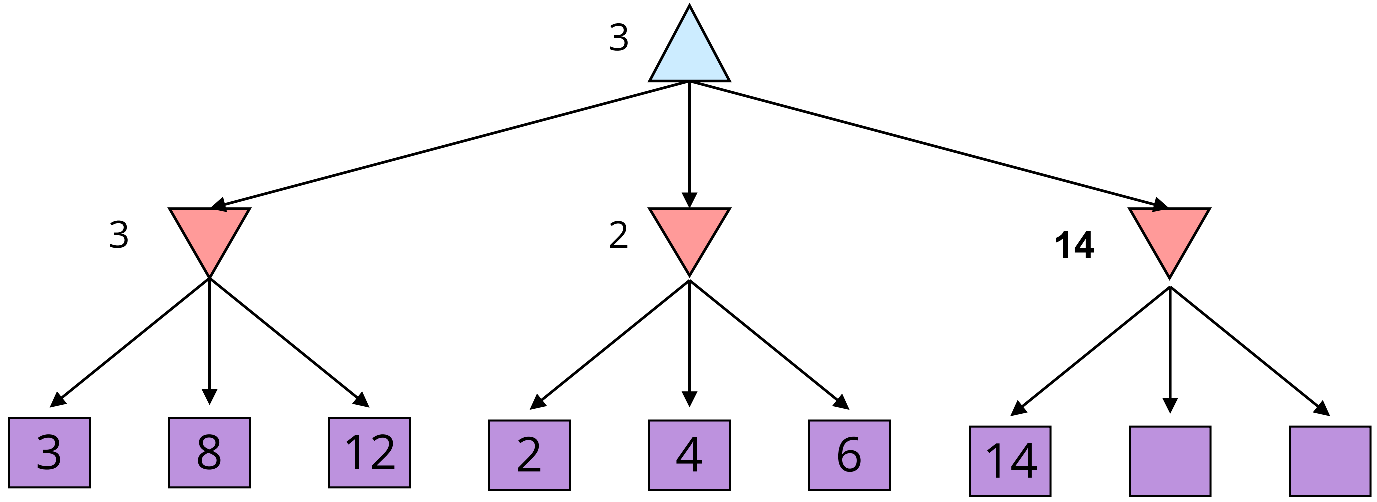

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



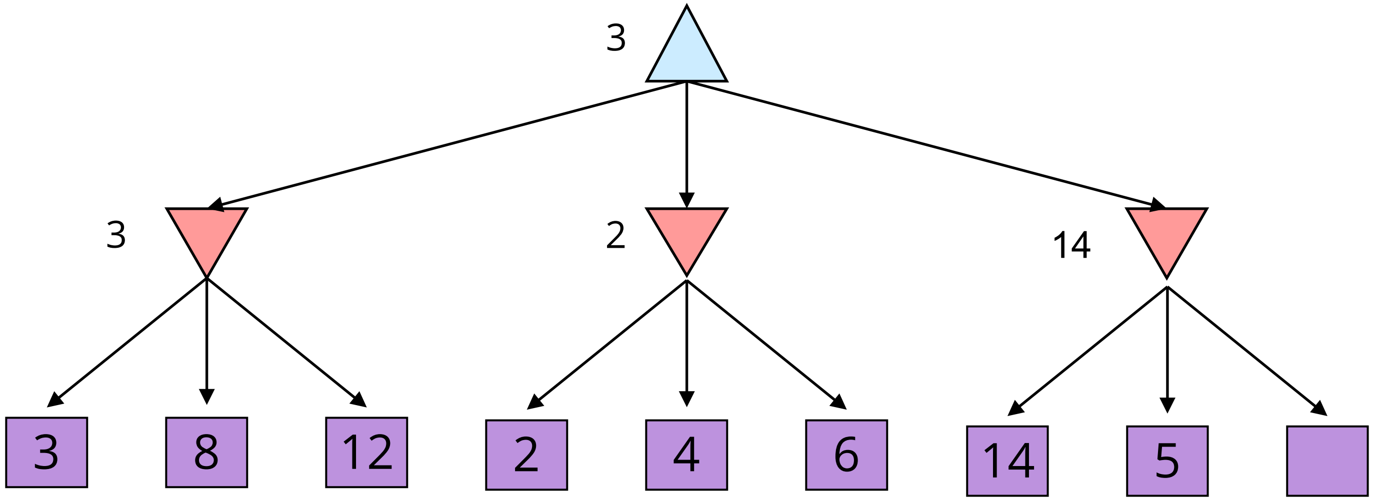

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



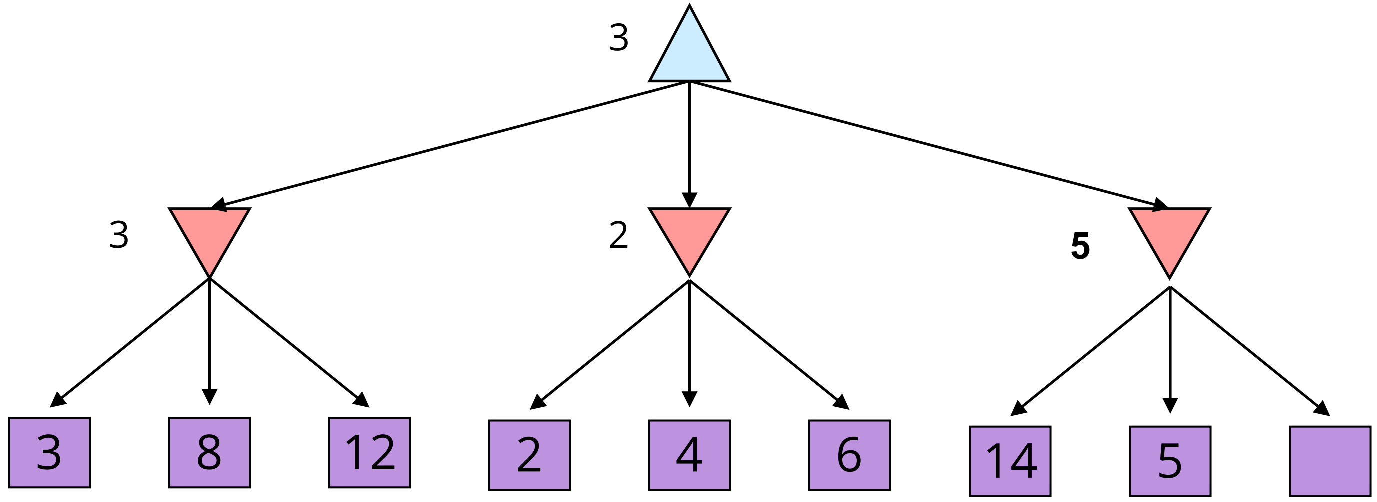

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



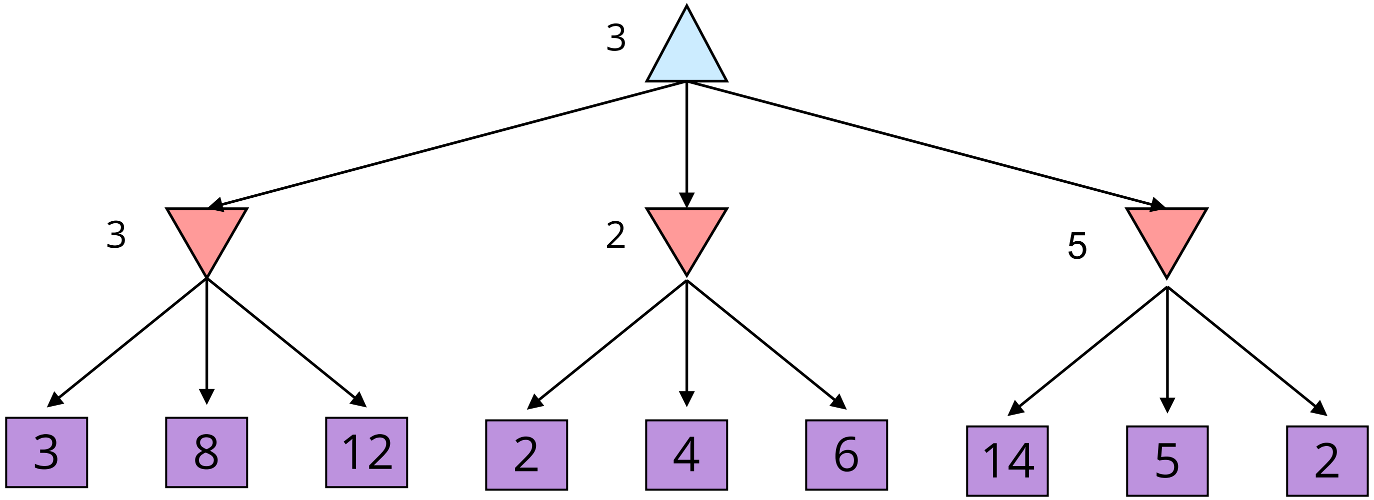

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



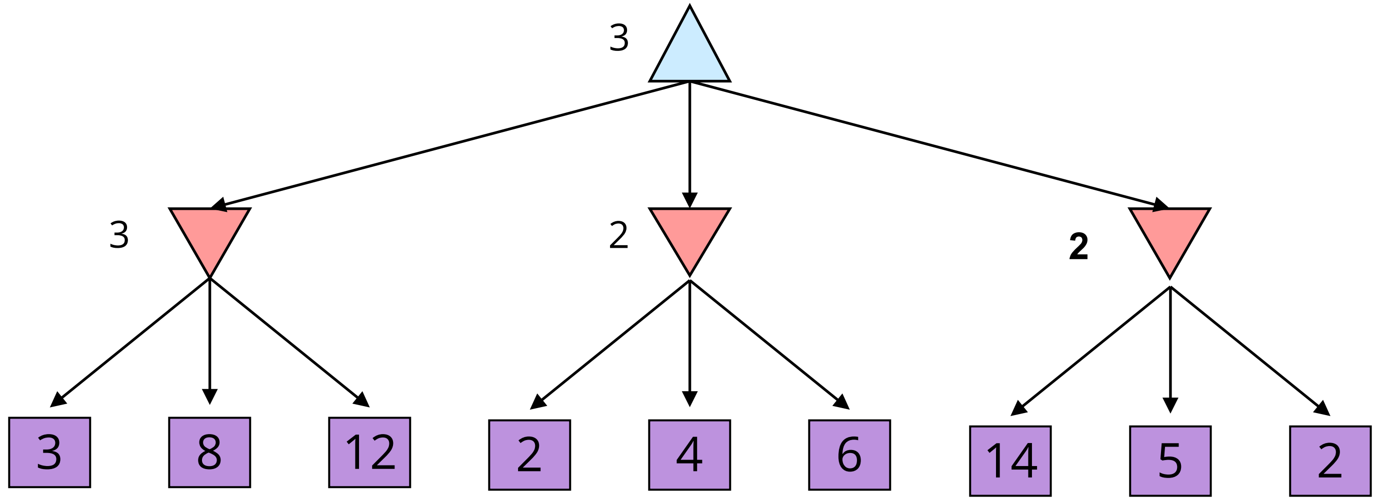

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



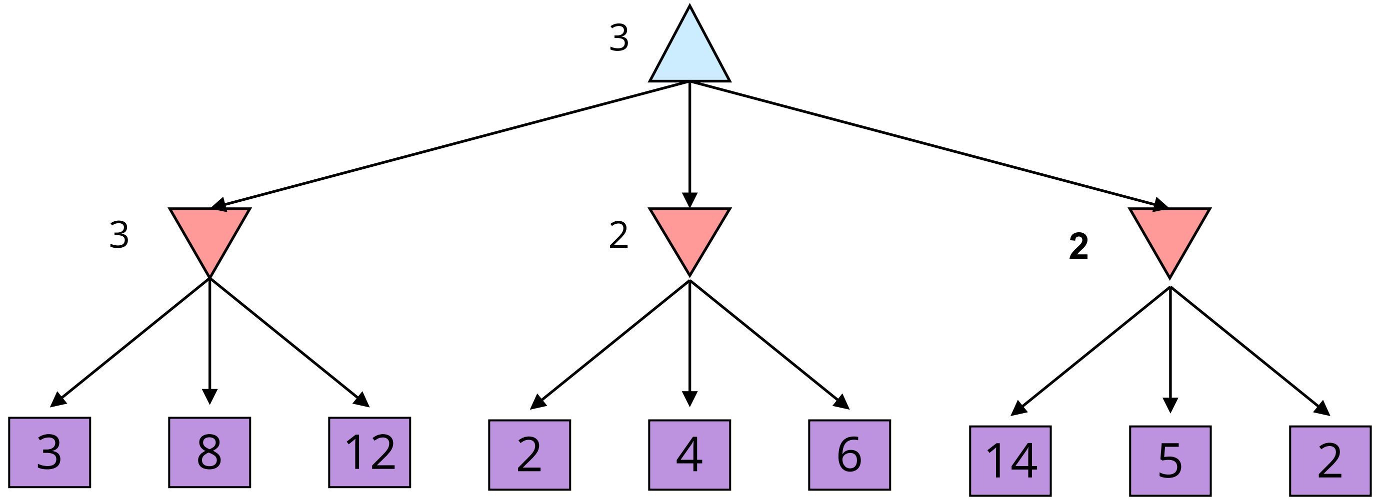

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



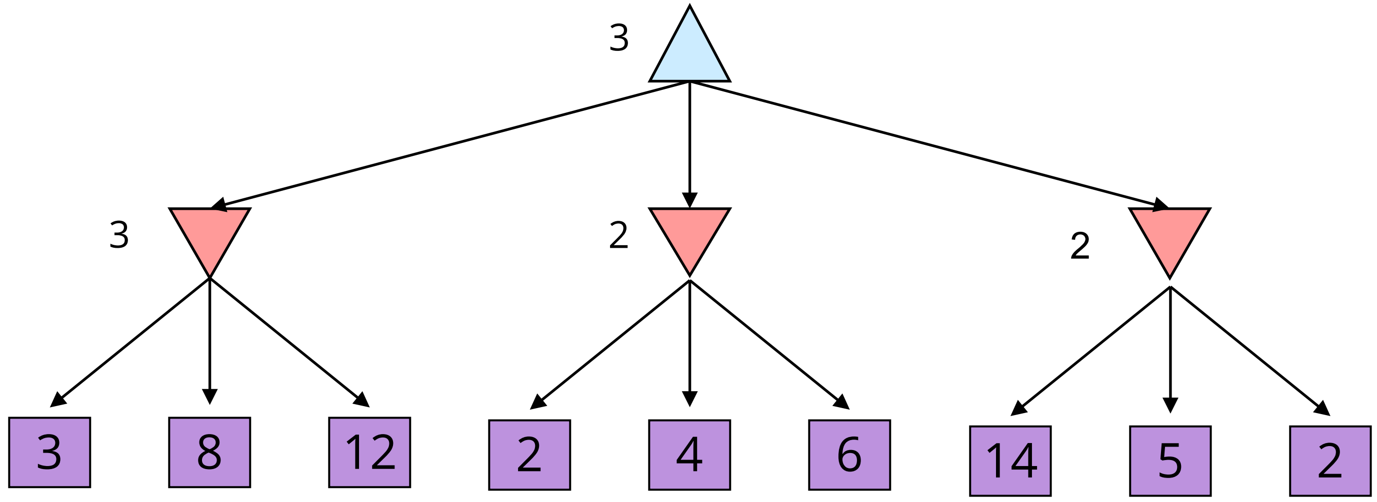

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



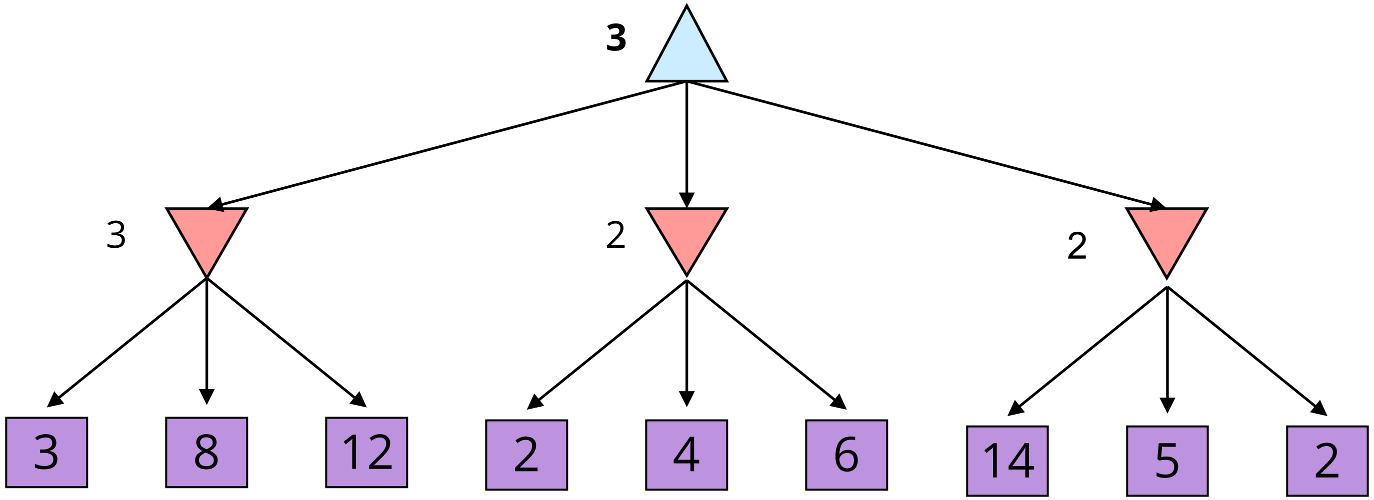

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example



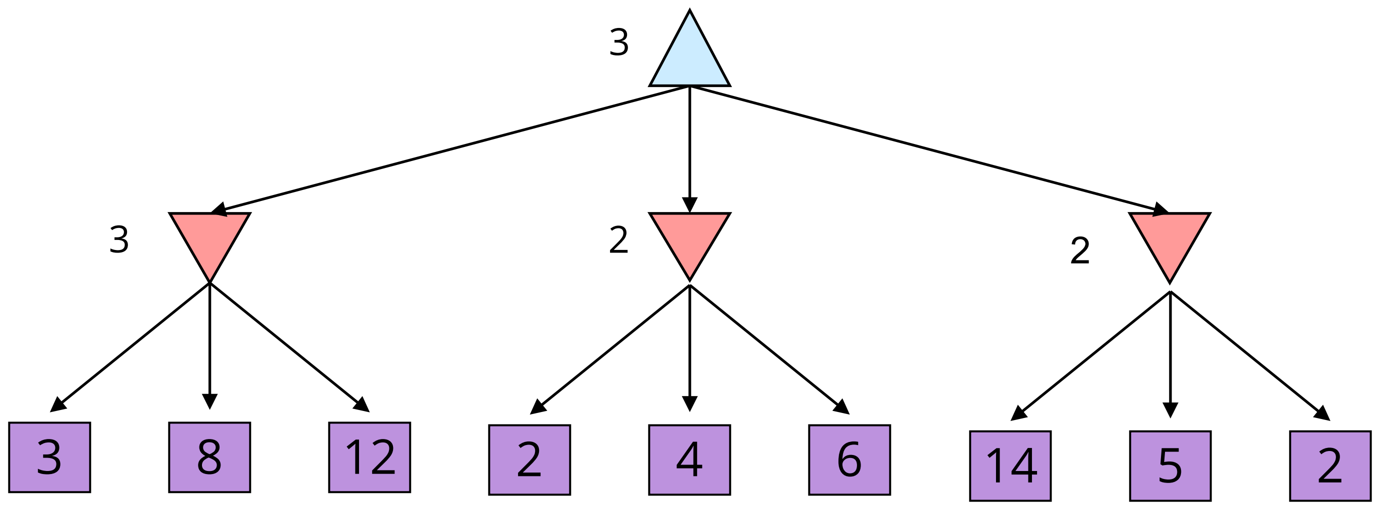

```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Example




```
def max-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    if the state is a terminal state:
        return the state's utility
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Alpha-Beta Pruning

For α think "at least"

For β think "at most"

- During Minimax, keep track of two additional values:
  - **α: MAX's current *lower* bound on MAX's outcome**
  - **β: MIN's current *upper* bound on MIN's outcome**

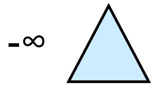
- MAX will never allow a move that could lead to a worse score (for MAX) than α
- MIN will never allow a move that could lead to a better score (for MAX) than β

- Therefore, stop evaluating a branch whenever:
  - When evaluating a MAX node: a value $v \geq \beta$ is backed-up
    - MIN will never select that MAX node
  - When evaluating a MIN node: a value $v \leq \alpha$ is found
    - MAX will never select that MIN node

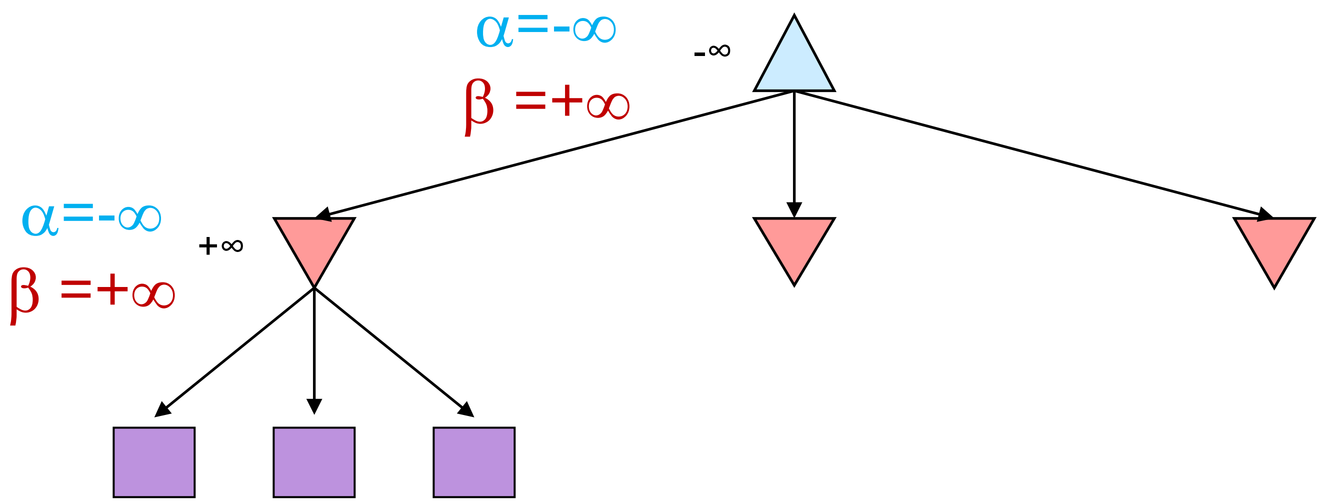
Alpha-Beta Pruning Example

α=-∞

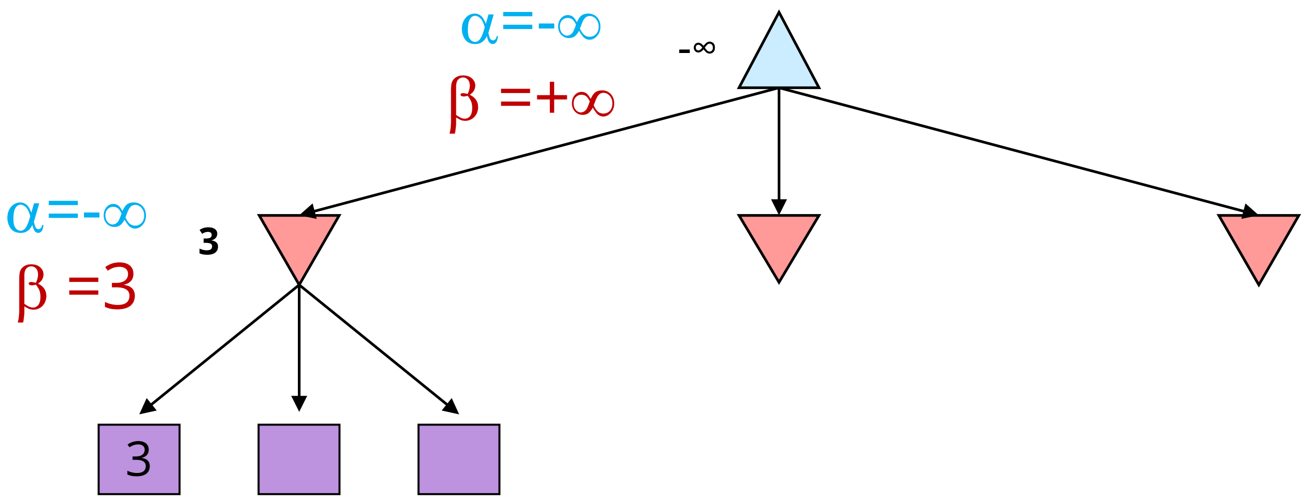
β =+∞



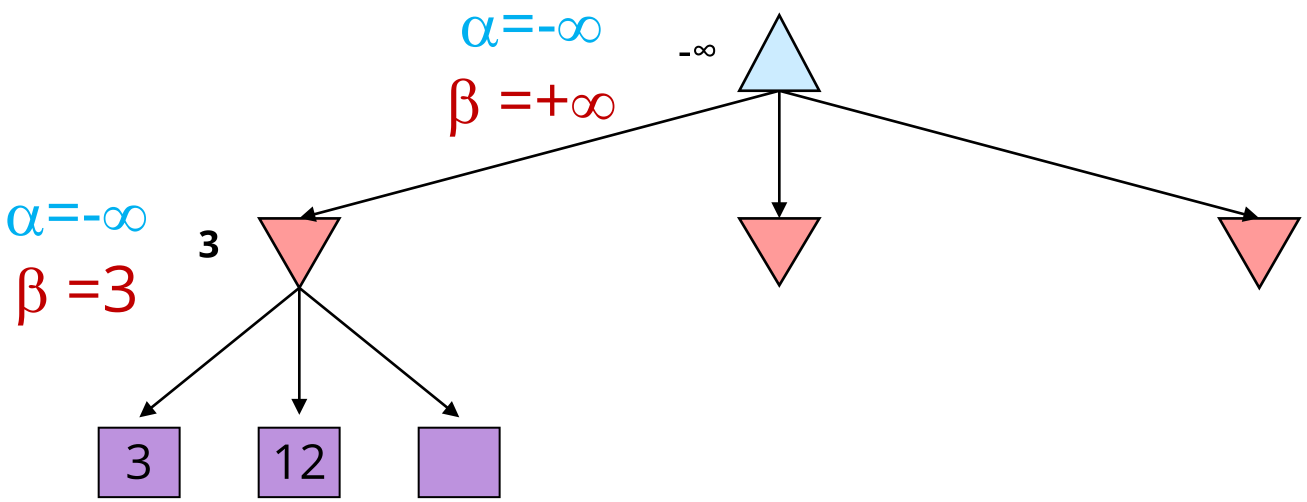
Alpha-Beta Pruning Example



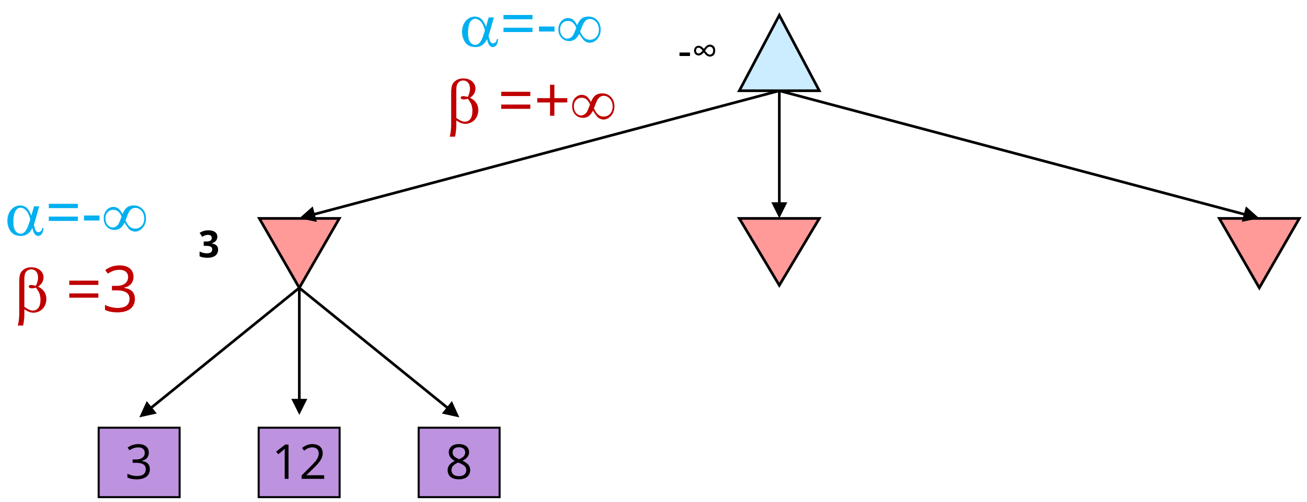
Alpha-Beta Pruning Example



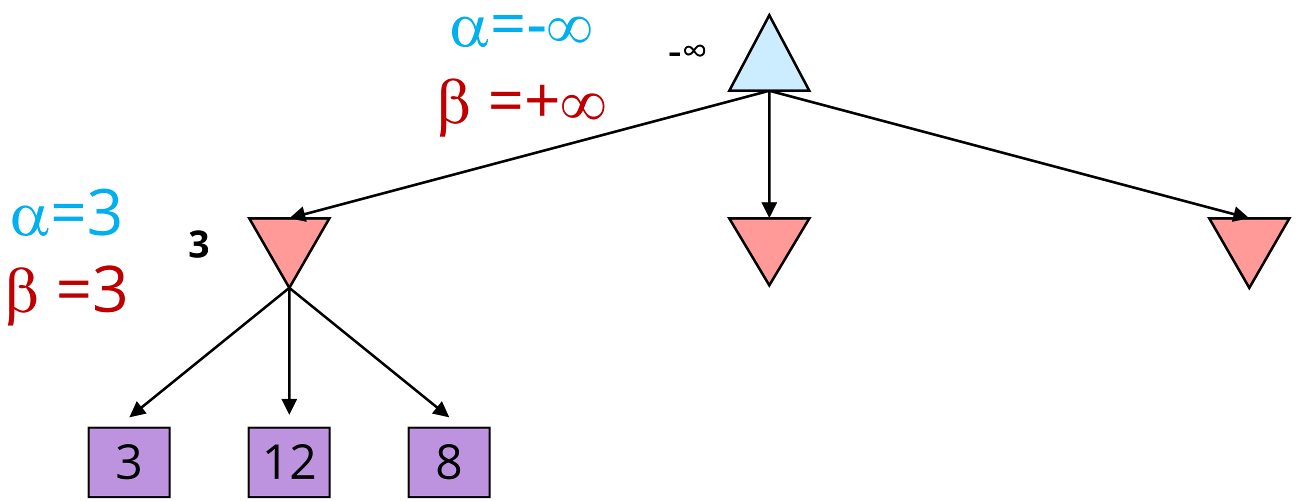
Alpha-Beta Pruning Example



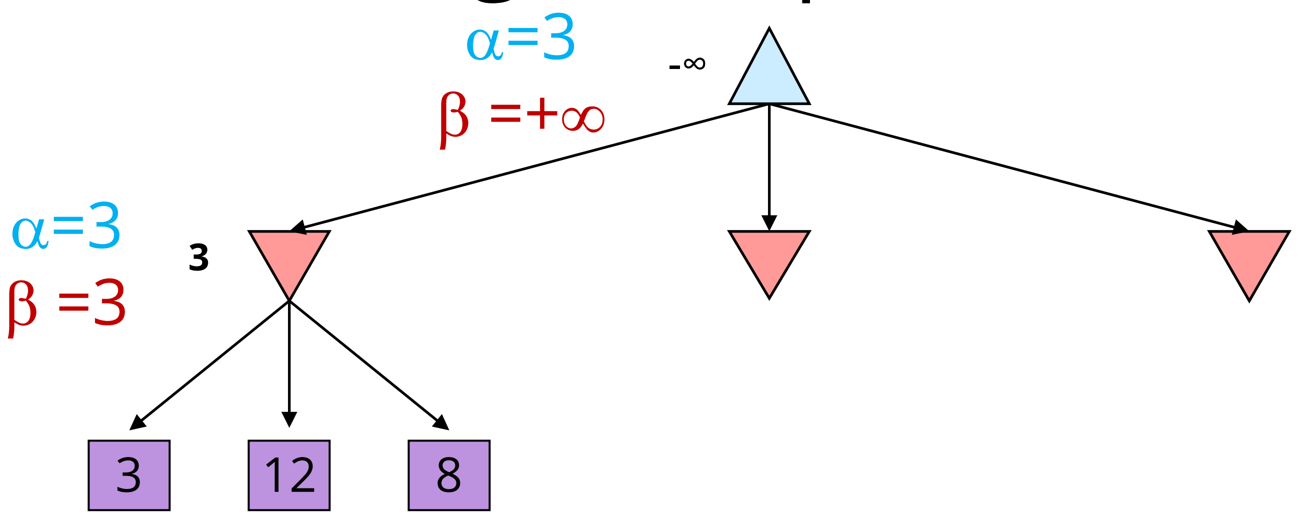
Alpha-Beta Pruning Example



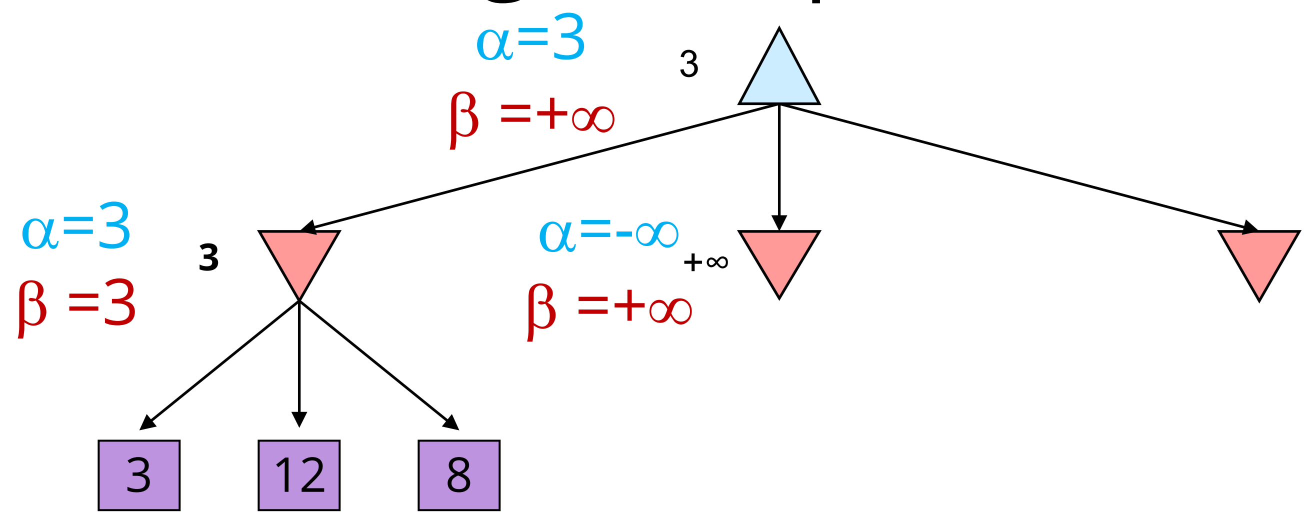
Alpha-Beta Pruning Example



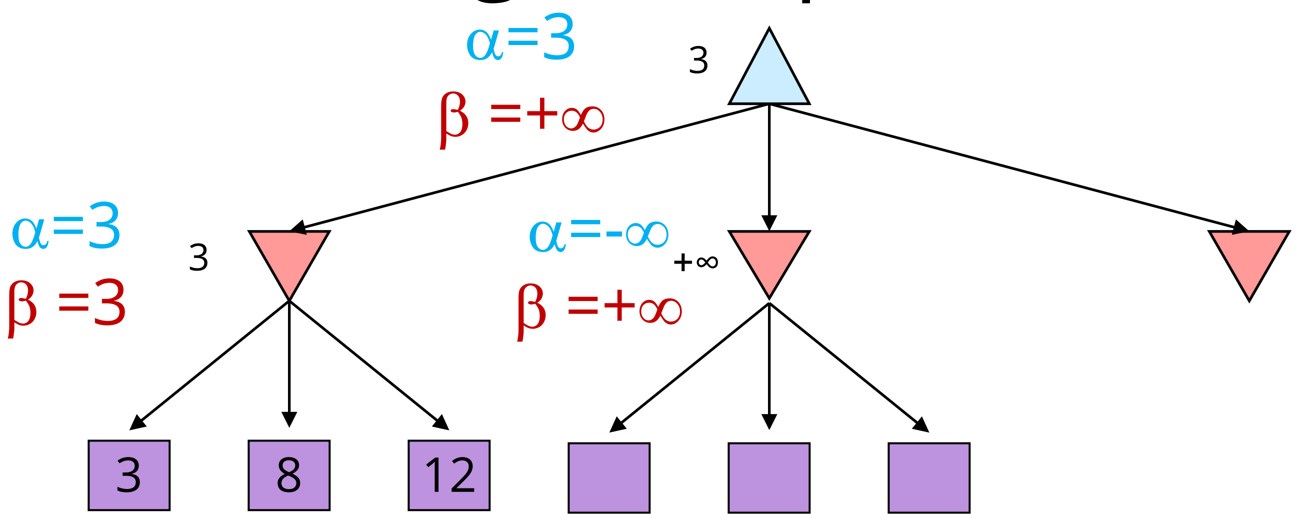
Alpha-Beta Pruning Example



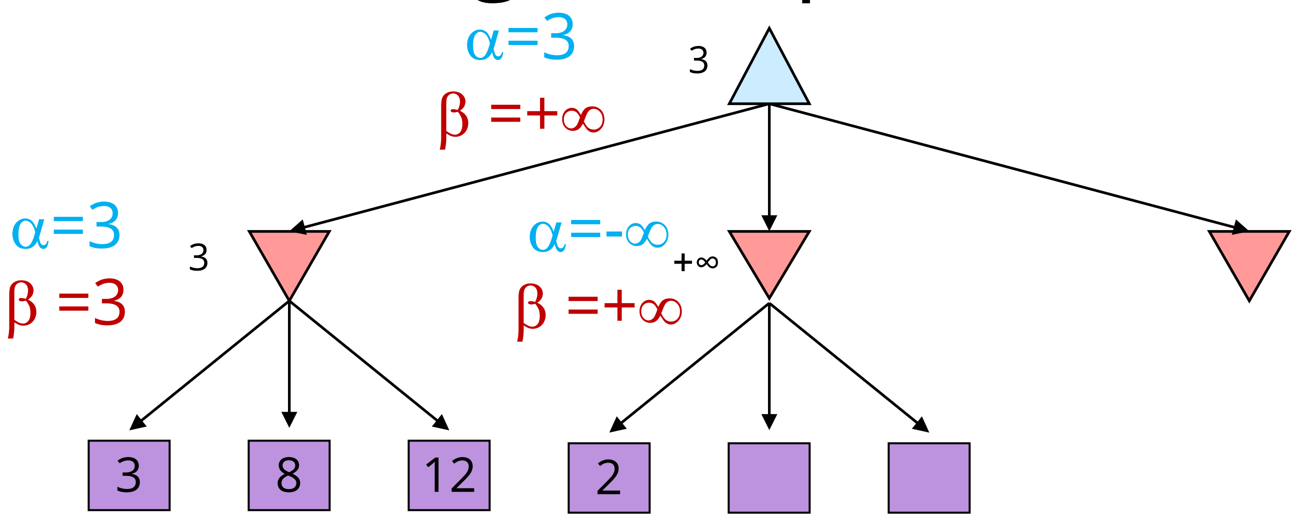
Alpha-Beta Pruning Example



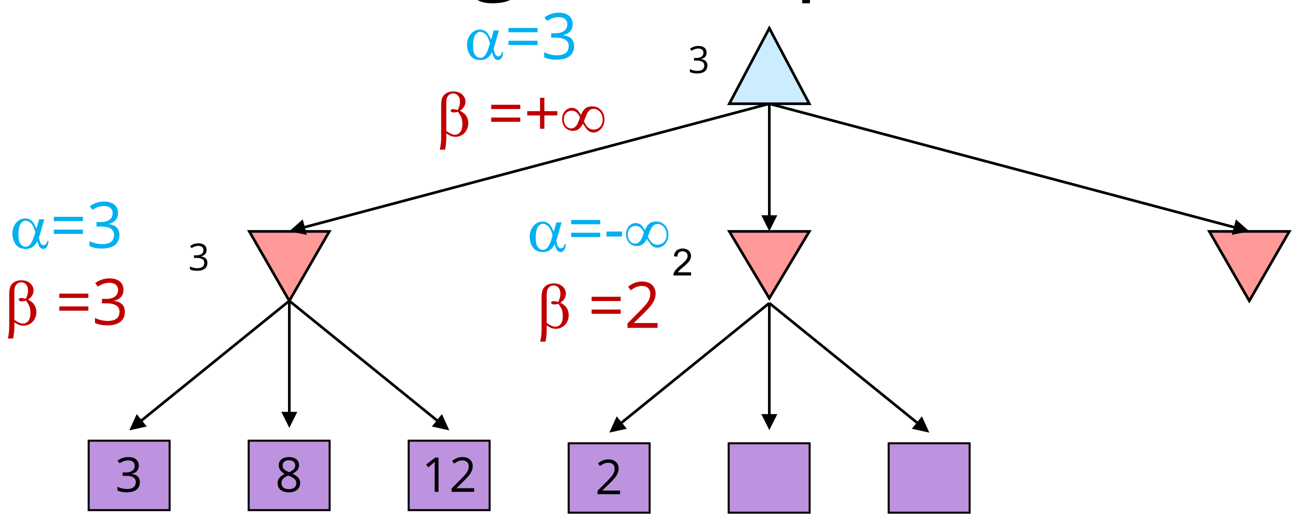
Alpha-Beta Pruning Example



Alpha-Beta Pruning Example

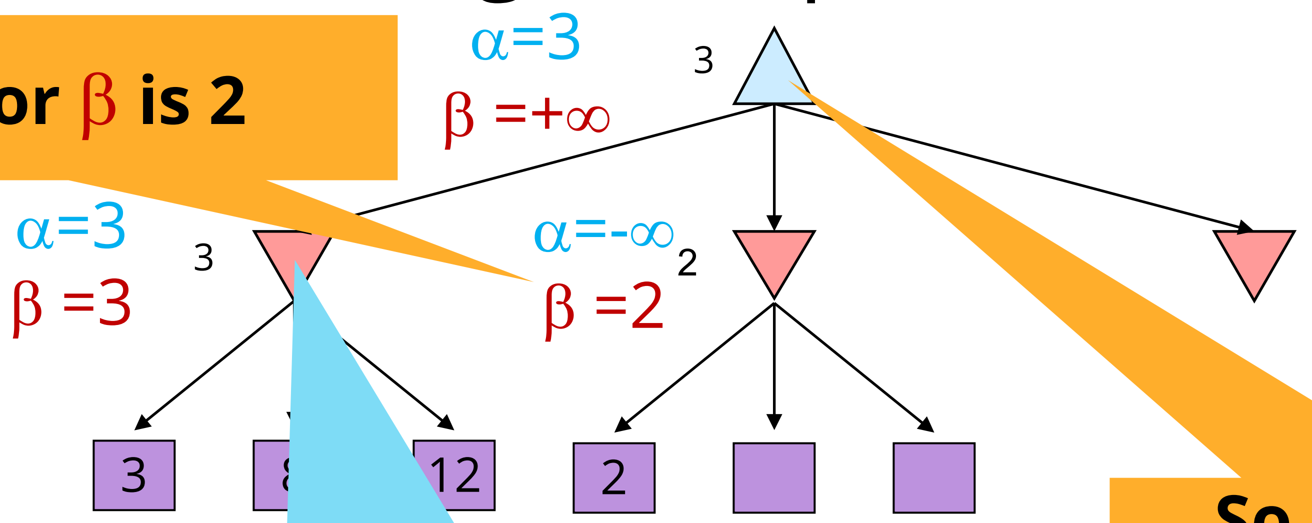


Alpha-Beta Pruning Example



Alpha-Beta Pruning Example

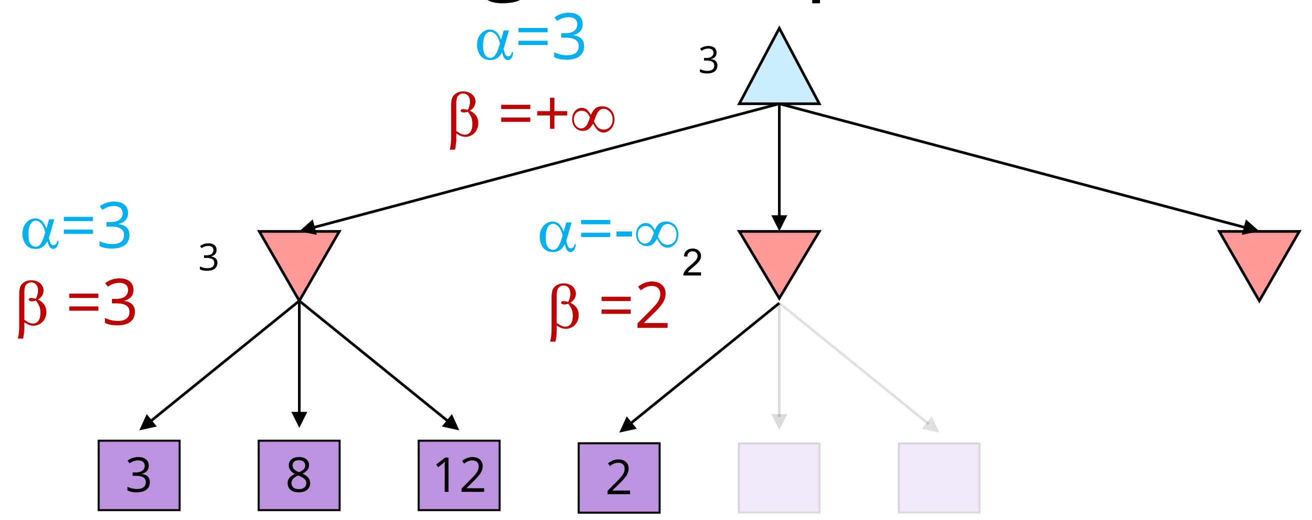
Value for β is 2



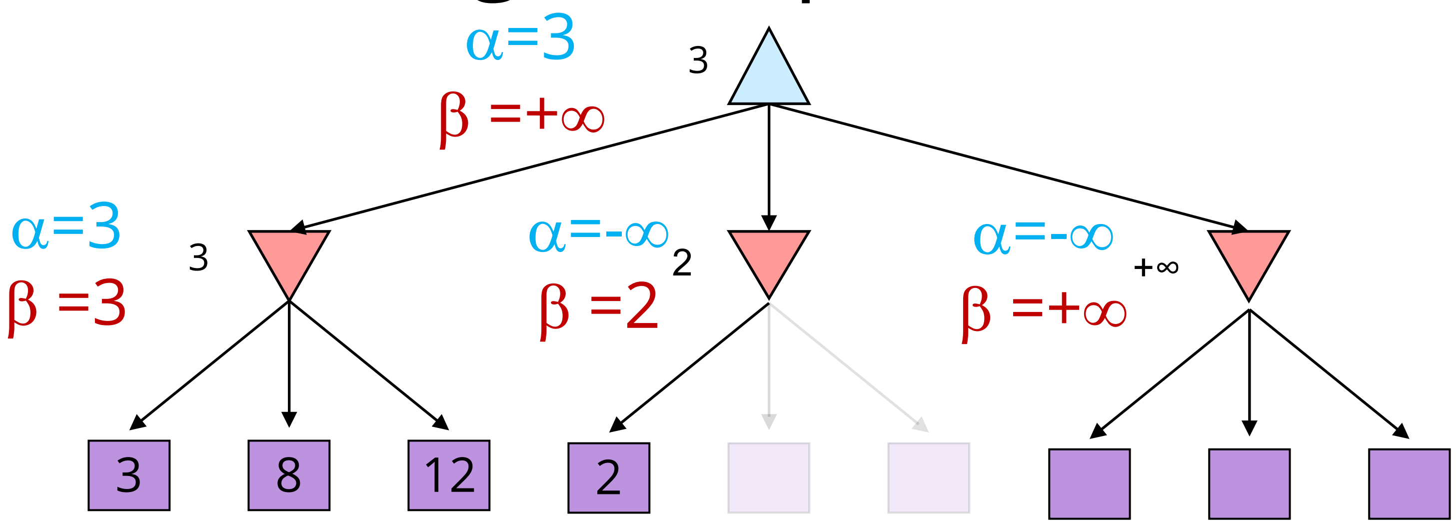
But we know that this node is worth at least 3

So Max will never choose 2.

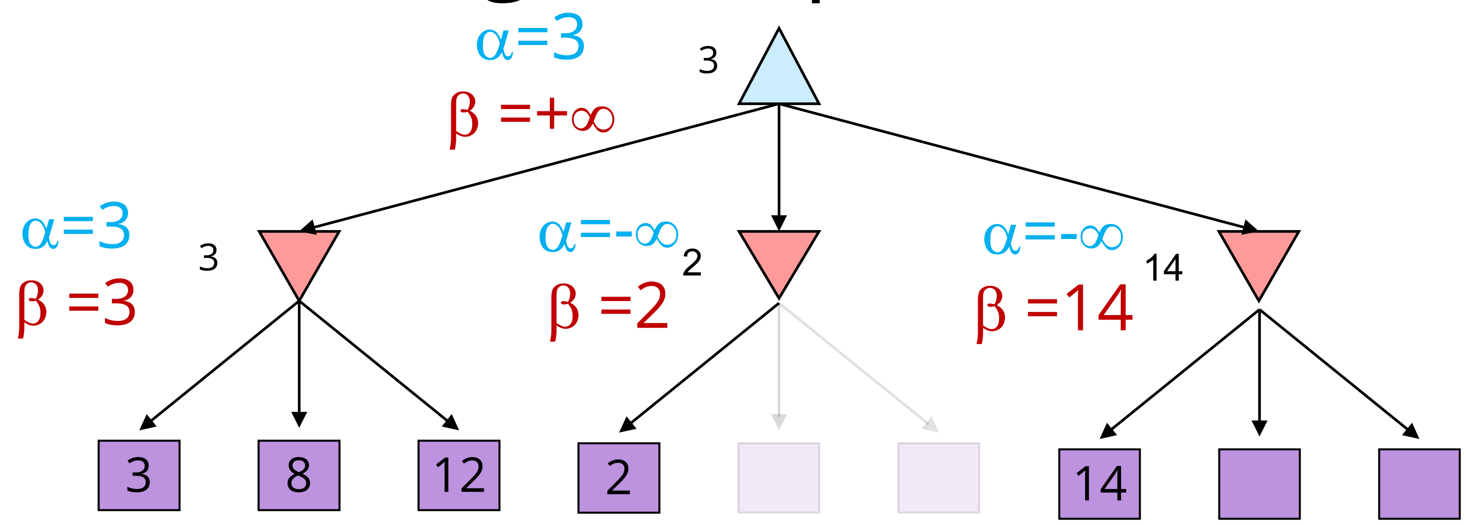
Alpha-Beta Pruning Example



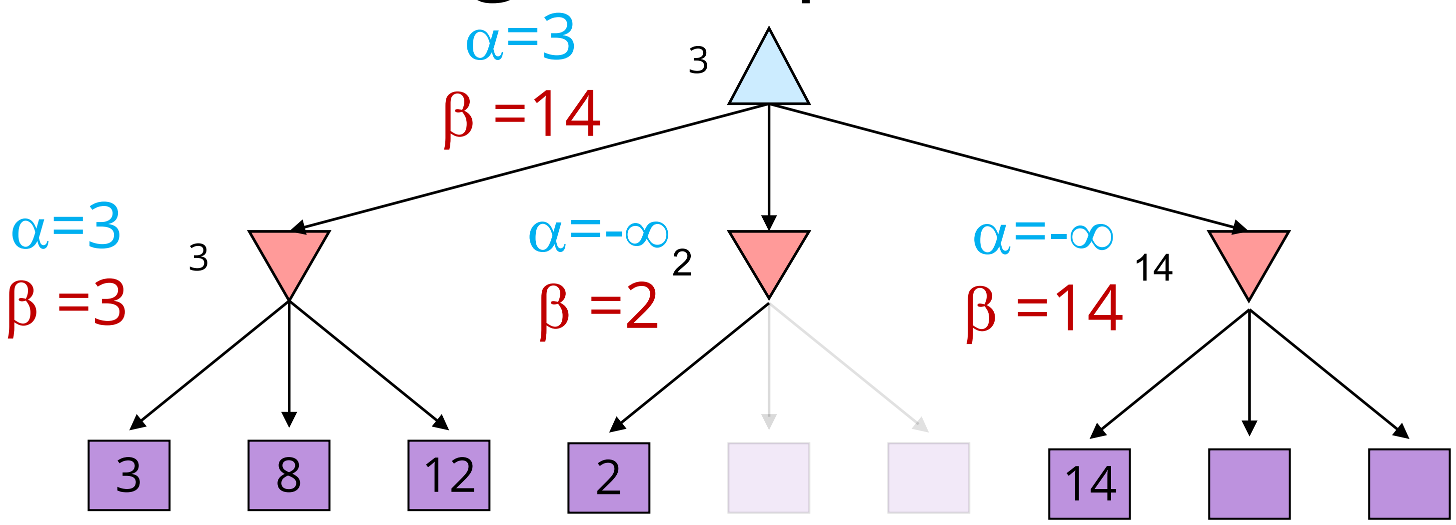
Alpha-Beta Pruning Example



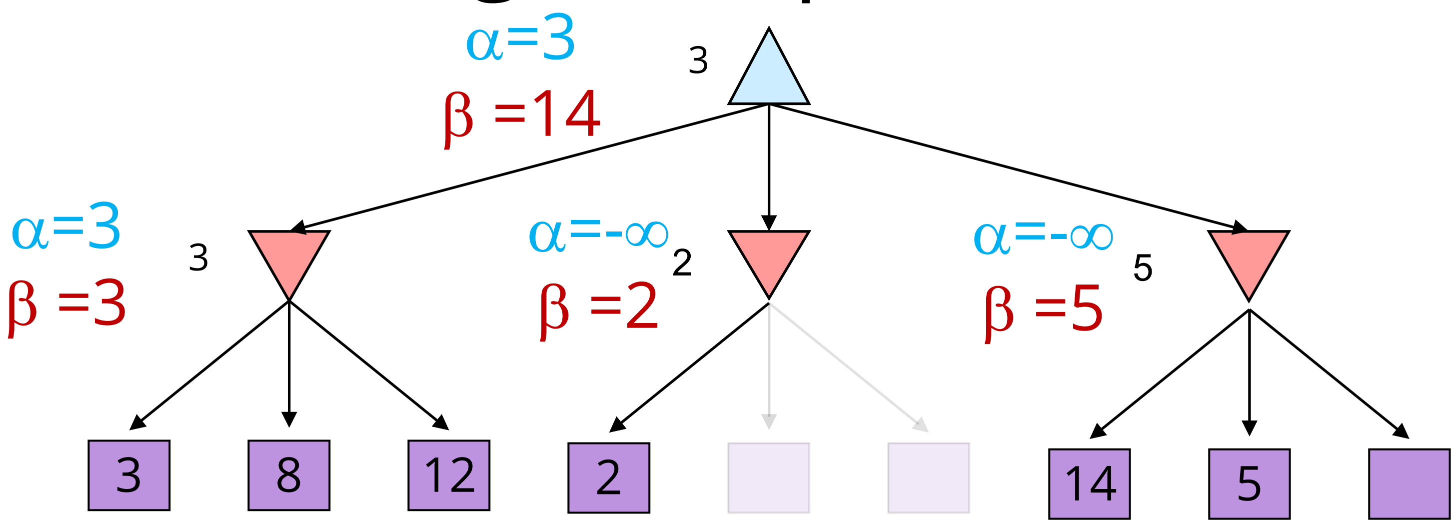
Alpha-Beta Pruning Example



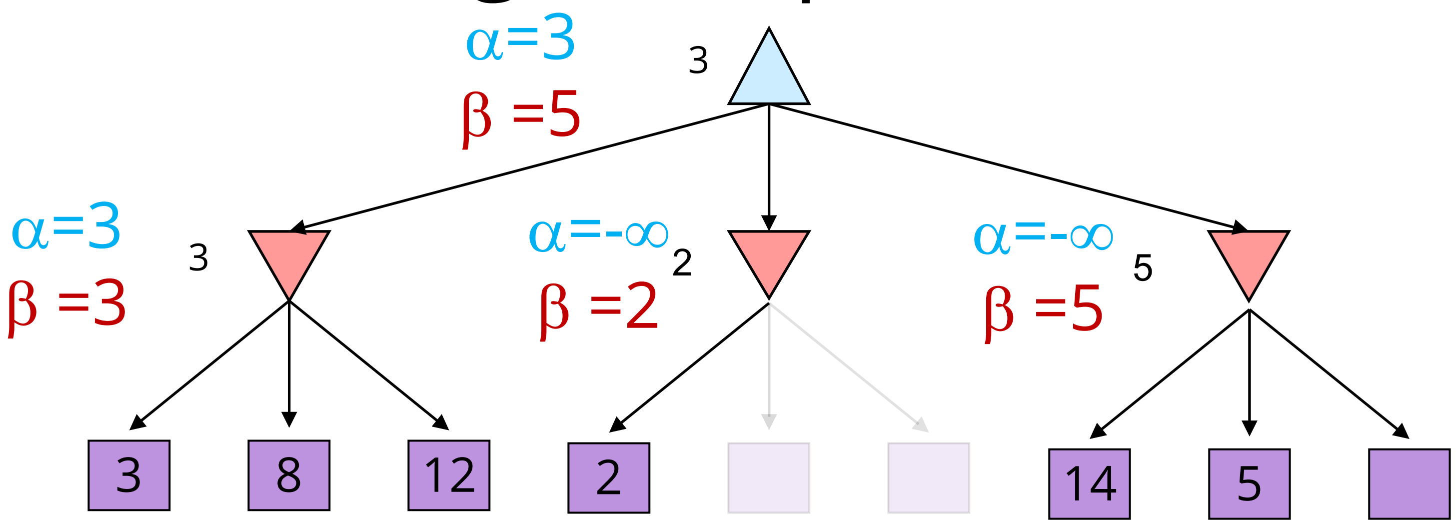
Alpha-Beta Pruning Example



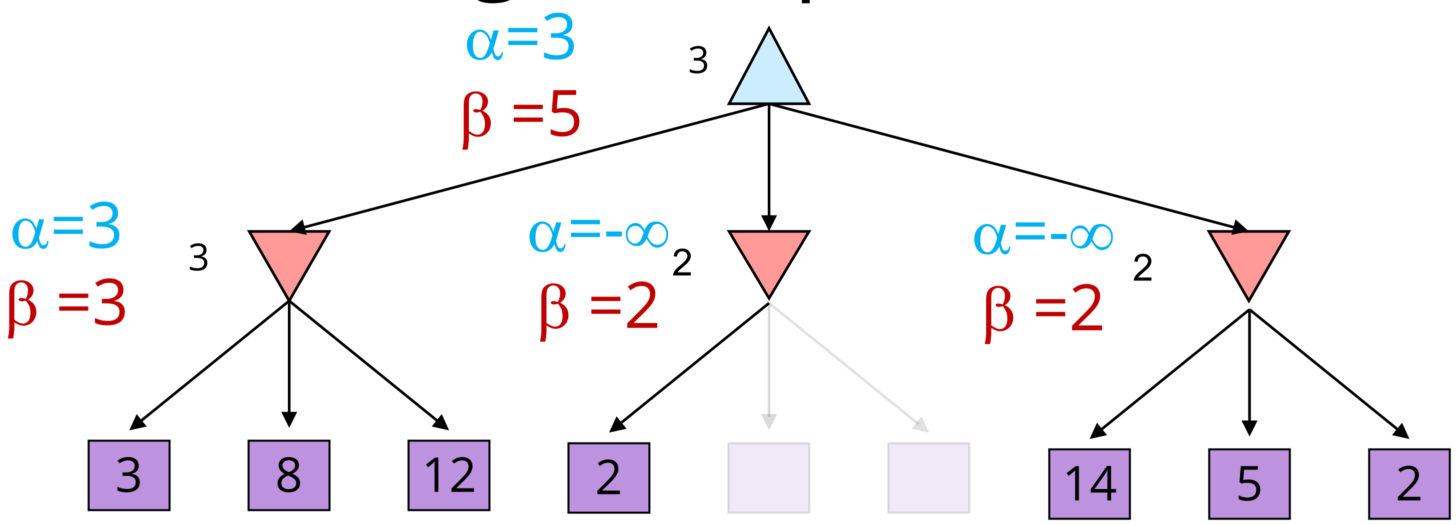
Alpha-Beta Pruning Example



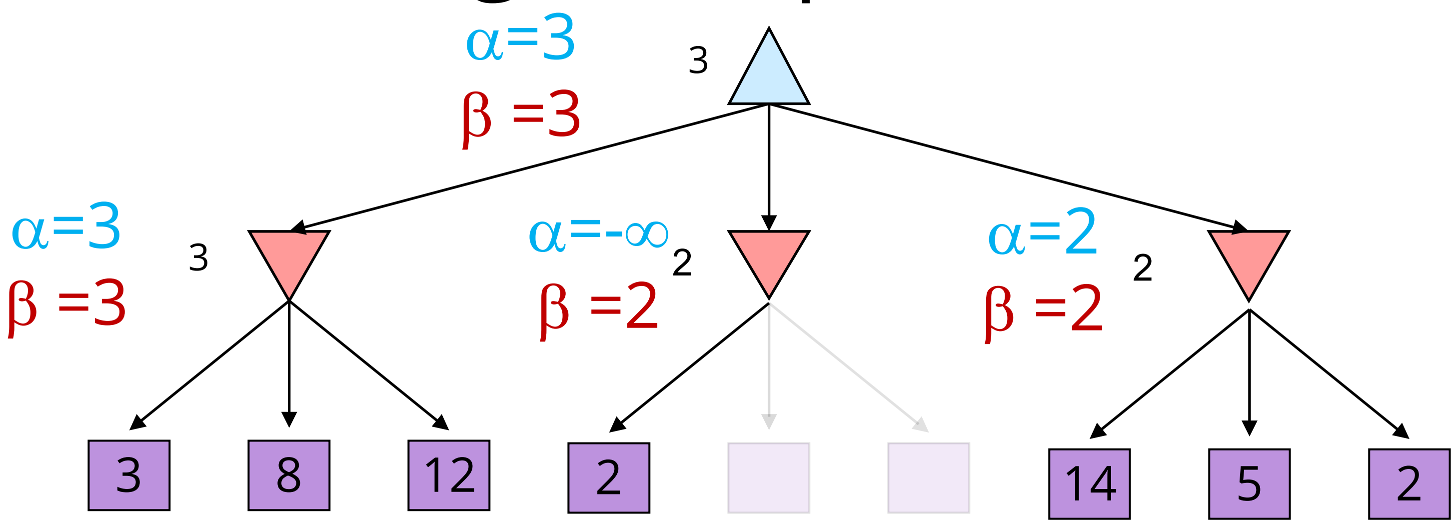
Alpha-Beta Pruning Example



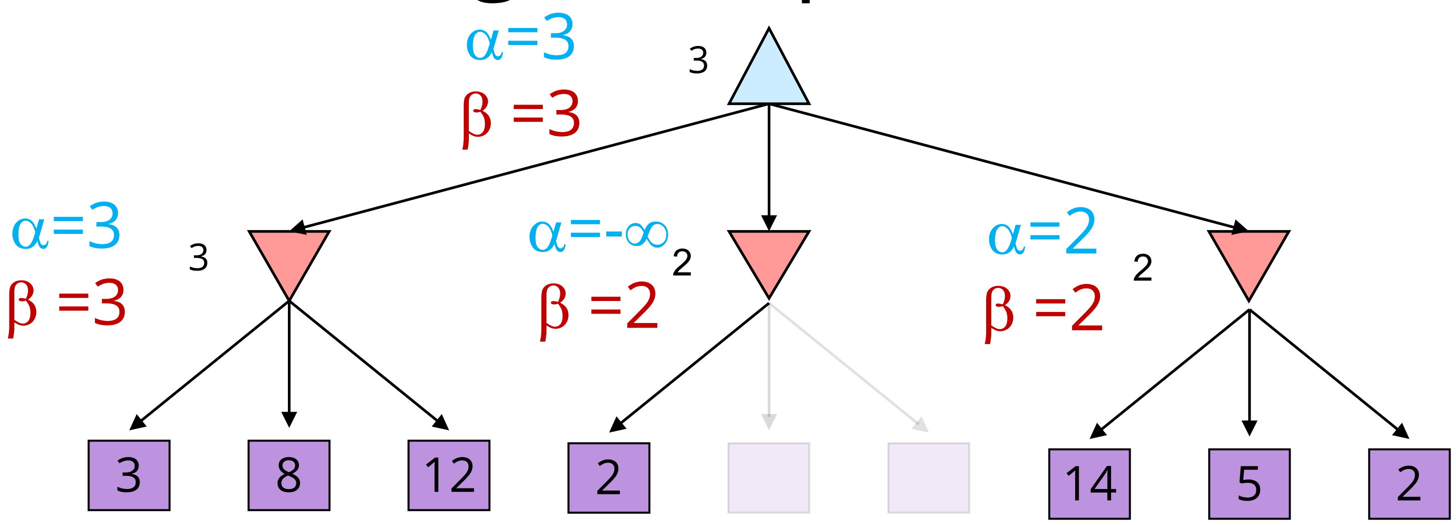
Alpha-Beta Pruning Example



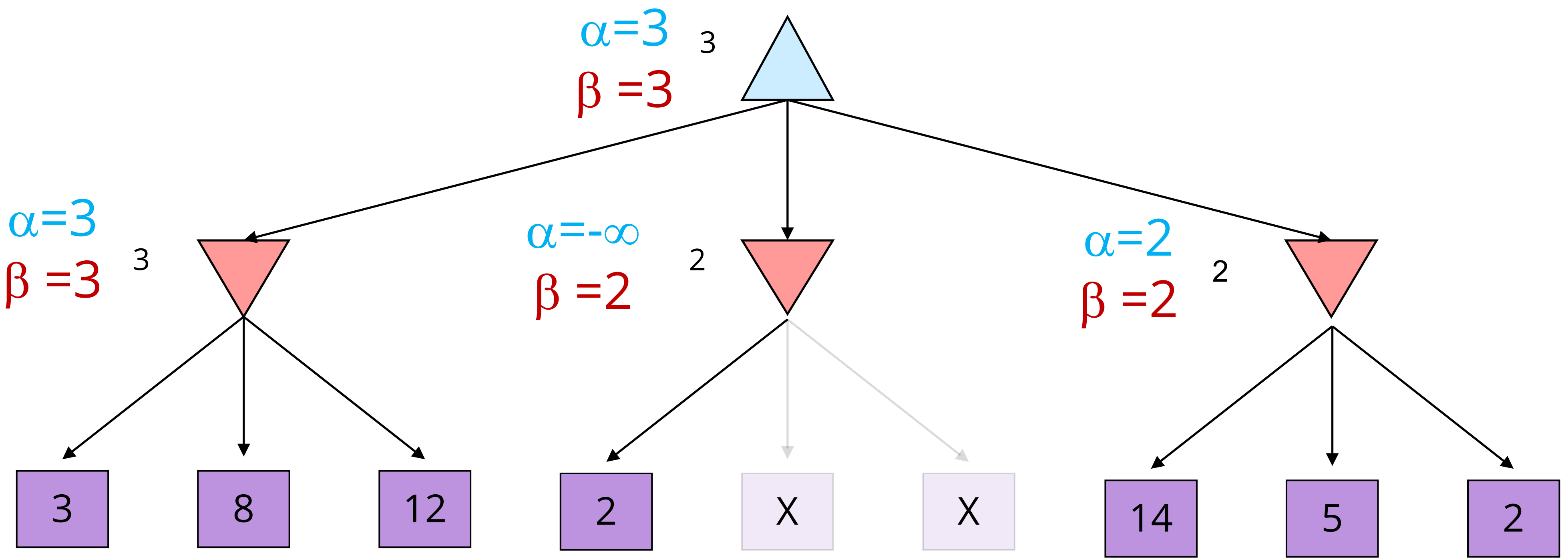
Alpha-Beta Pruning Example



Alpha-Beta Pruning Example



Alpha-Beta Pruning



α: MAX's current lower bound on MAX's outcome
β: MIN's current upper bound on MIN's outcome
α: MAX's best option on path to root
β: MIN's best option on path to root

Review: Evaluation functions

- Evaluates how good a 'board position' is
  - **Based on *static features* of that board alone**
- Zero-sum assumption lets us use one function to describe goodness for both players.
  - $\boldsymbol{f(n) > 0}$ if MAX is winning in position *n*
  - $\boldsymbol{f(n) = 0}$ if position *n* is tied
  - $\boldsymbol{f(n) < 0}$ if MIN is winning in position *n*
- Build using expert knowledge,
  - Tic-tac-toe: $\boldsymbol{f(n) = (\text{\# of 3 lengths open for MAX}) - (\text{\# open for MIN})}$

(AIMA 5..1)

Review: Chess Evaluation Functions

- Chess needs an evaluation function since it is impossible to search the game tree deeply enough to reach the terminal nodes
- $f(n) = (sum\ of\ A's\ piece\ values) - (sum\ of\ B's\ piece\ values)$
- More complex: weighted sum of positional features:

$$\sum w_i \cdot \text{feature}_i\ (n)$$

- $f(n)$ can be a **weighted linear function**

Pawn	**1.0**
Knight	**3.0**
Bishop	**3.25**
Rook	**5.0**
Queen	**9.0**

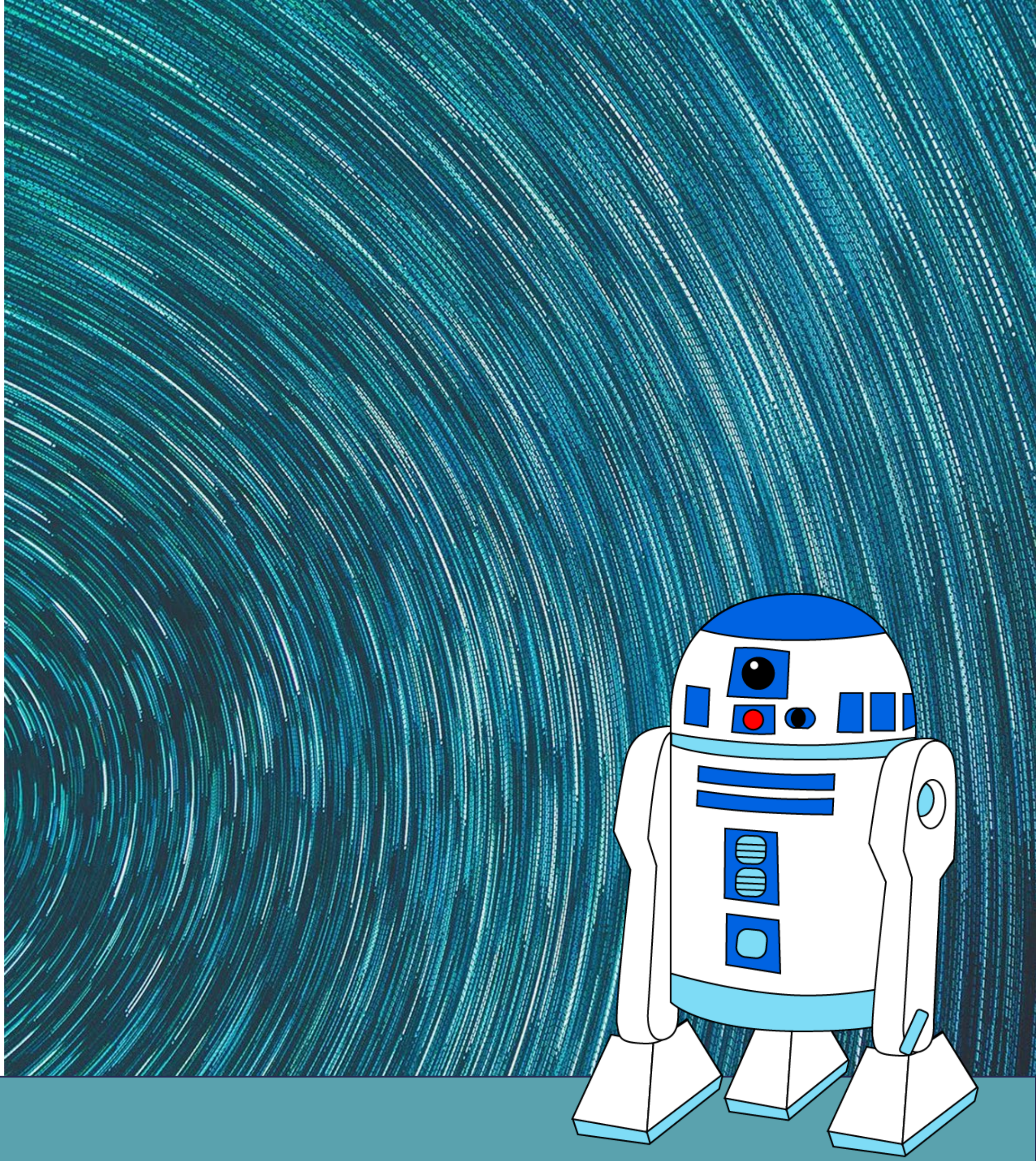
Pieces values for a simple evaluation function often taught to novice chess players

CIS 521:
ARTIFICIAL INTELLIGENCE

Expectimax and Utilities

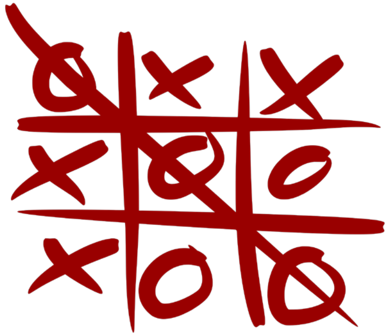
Harry Smith

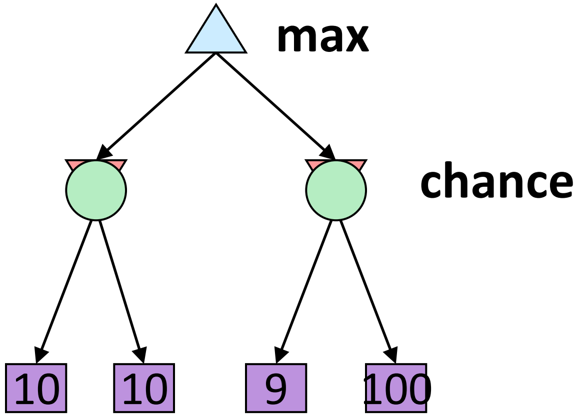
Many of today's slides are courtesy of Dan Klein and Pieter Abbeel of University of California, Berkeley



Penn Engineering
UNIVERSITY of PENNSYLVANIA

Uncertain Outcomes





Idea: Uncertain outcomes controlled by chance, not an adversary!

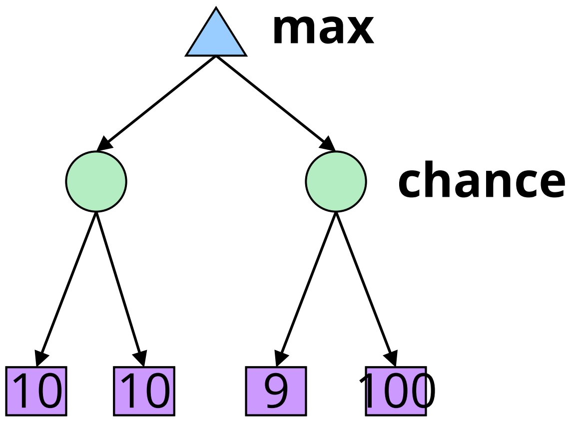
Expectimax Search

- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the opponent isn't optimal
  - Actions can fail: when moving a robot, wheels might slip

- Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes

- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - Chance nodes are like min nodes but the outcome is uncertain
  - Calculate their expected utilities
  - I.e. take weighted average (expectation) of children

- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes



Expectimax Pseudocode

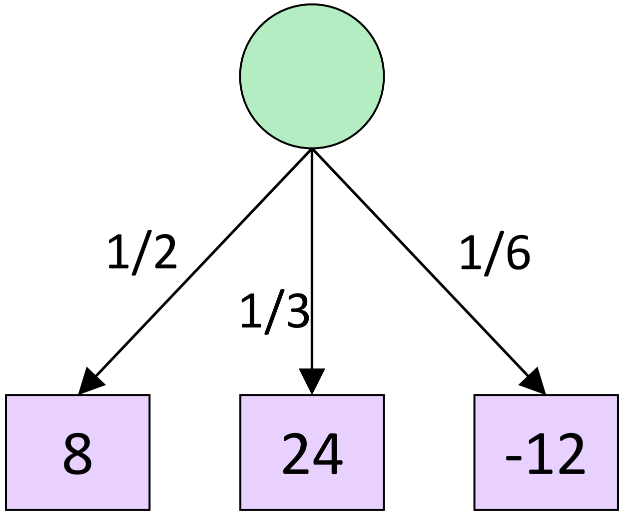
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor))
    return v
```

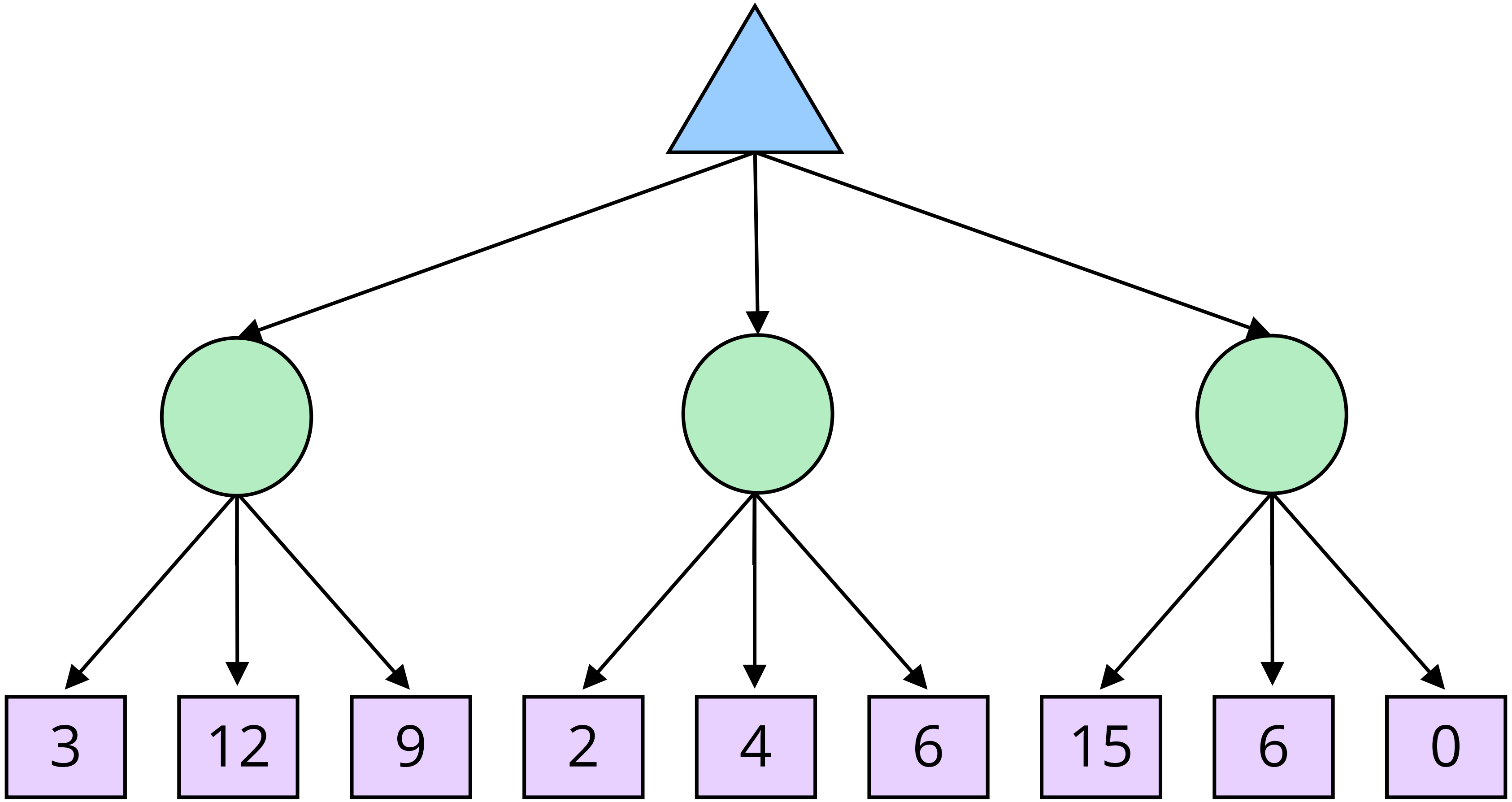
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

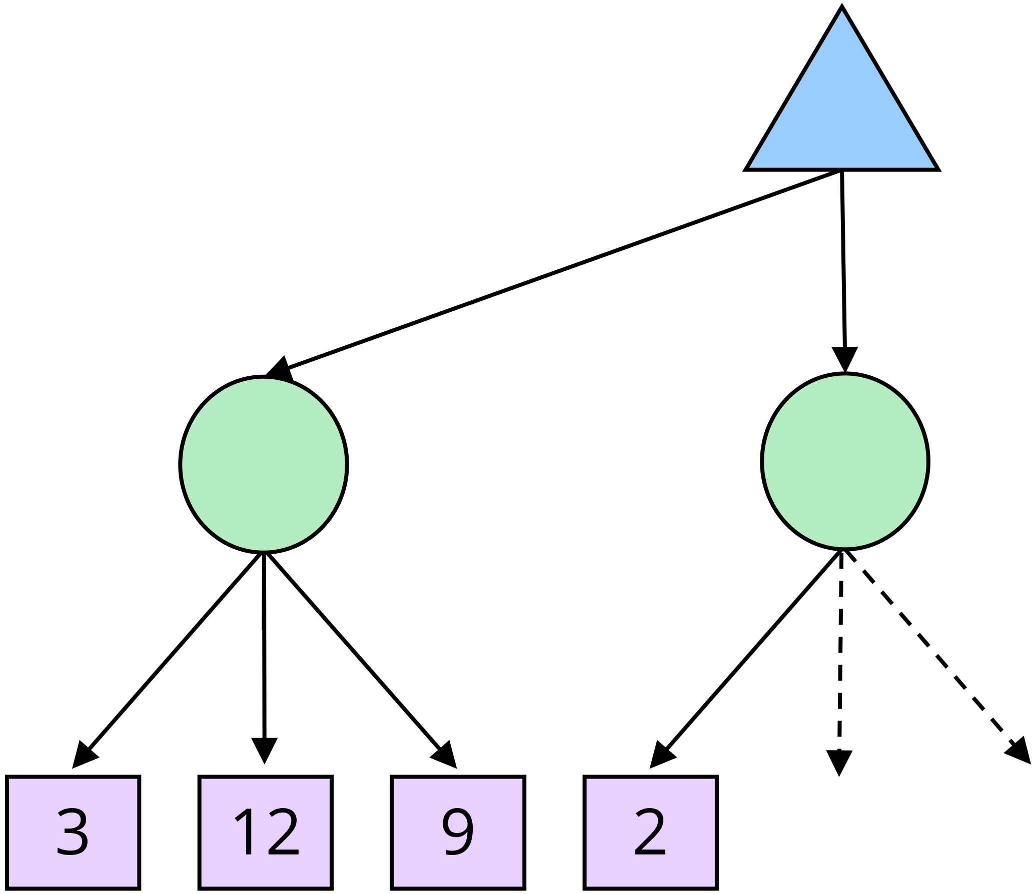
Expectimax Pseudocode

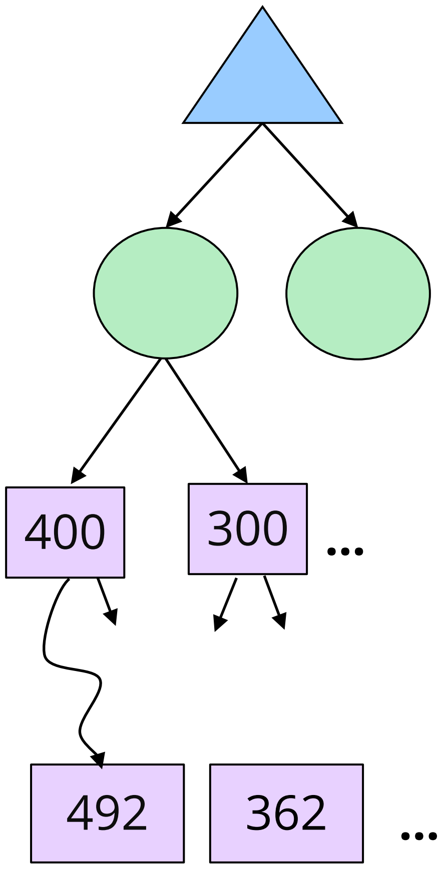
```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```



$$v = \frac{1}{2} \cdot (8) + \frac{1}{3} \cdot (24) + \frac{1}{6} \cdot (-12)$$

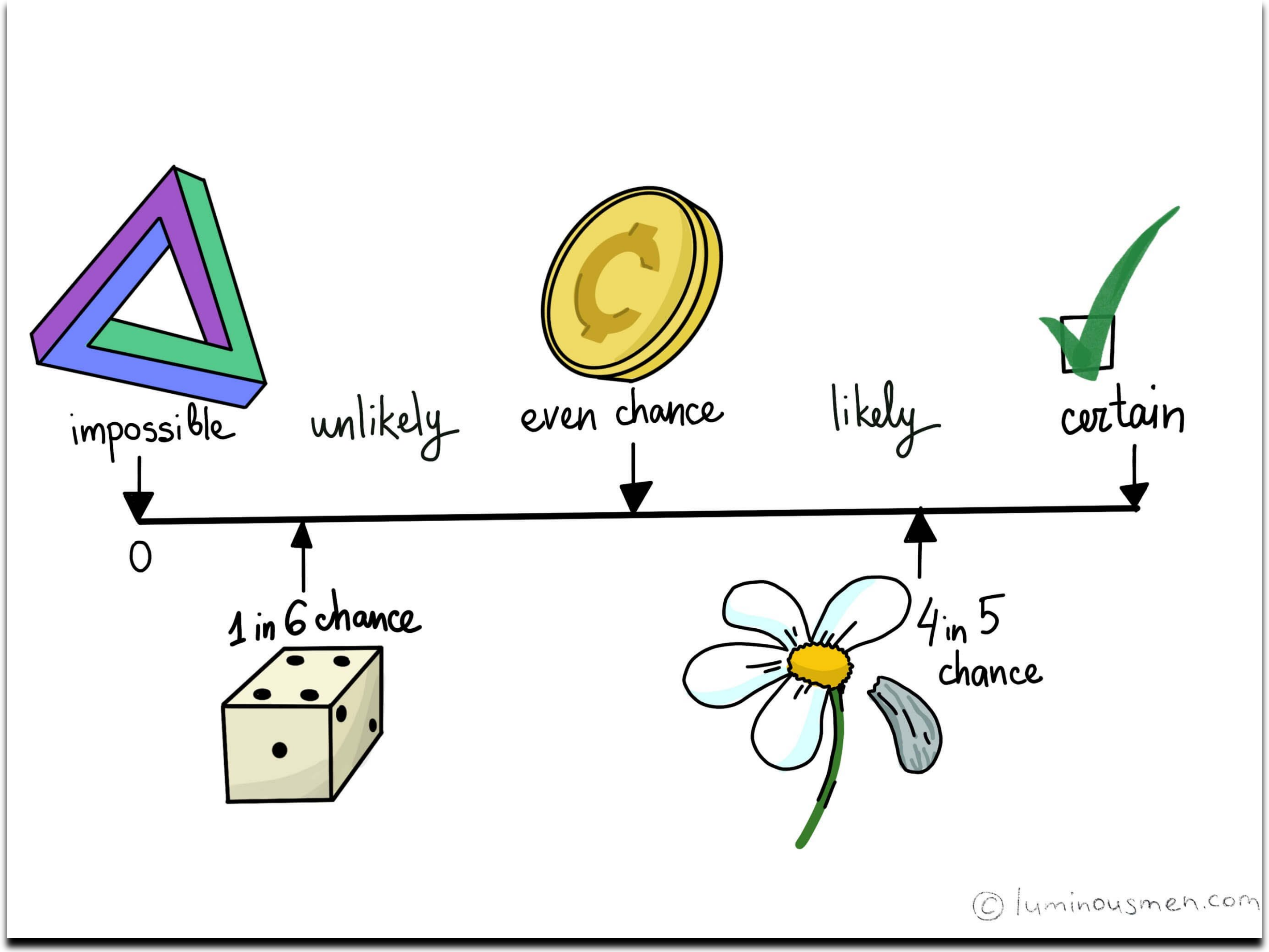

3
12
9
2
4
6
15
6
0


3
12
9
2



Estimate of true expectimax value (which would require a lot of work to compute)

Probabilities



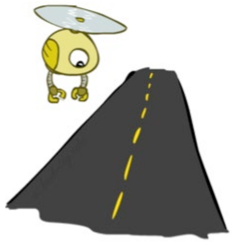
Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

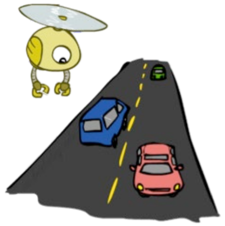
- Example: Traffic on freeway
  - Random variable: T = whether there's traffic
  - Outcomes: T in {none, light, heavy}
  - Distribution: $P(T = \text{none}) = 0.25$, $P(T = \text{light}) = 0.50$, $P(T = \text{heavy}) = 0.25$

- Some laws of probability (more later):
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one

- As we get more evidence, probabilities may change:
  - $P(T = \text{heavy}) = 0.25$, $P(T = \text{heavy} \mid \text{Hour} = \text{8am}) = 0.60$
  - We'll talk about methods for reasoning and updating probabilities later



0.25



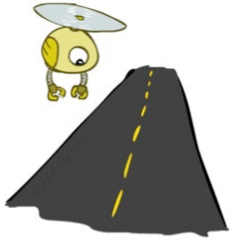
0.50

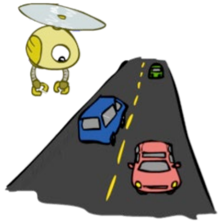
0.25

Probabilities

- The expected value of a function of a random variable is the average, weighted by the probability distribution over outcomes
- Example: How long to get to the airport?

Time:	20 min x	+	30 min x	+	60 min x	⇒	35 min
Probability:	0.25		0.50		0.25		

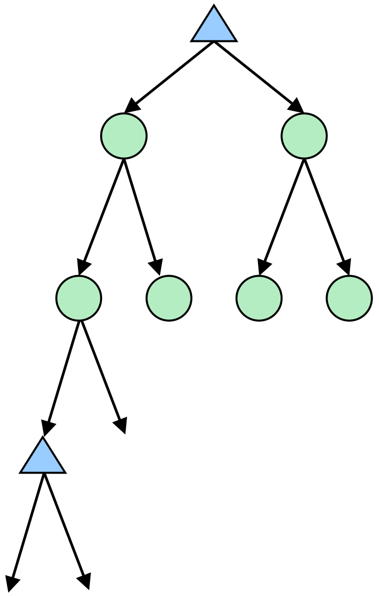




What Probabilities to Use?

- **In expectimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state**
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!

- **For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes**



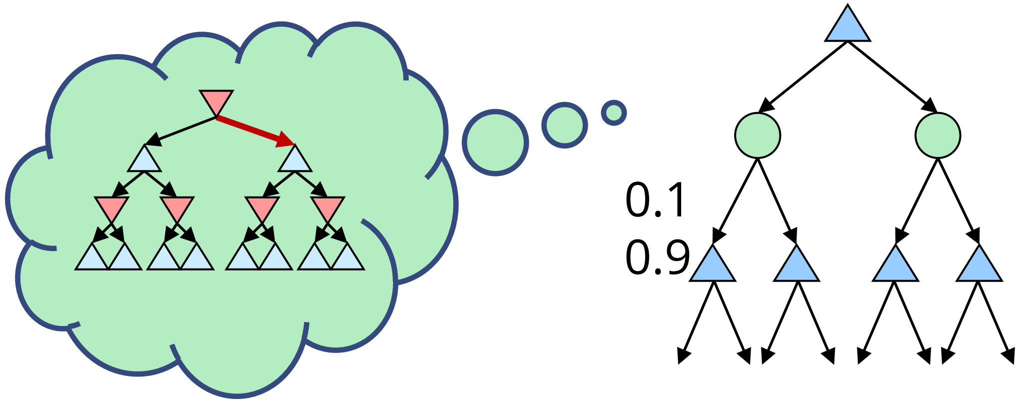
Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!

- **Objectivist / frequentist answer:**
  - Averages over repeated *experiments*
  - E.g. empirically estimating P(rain) from historical observation
  - Assertion about how future experiments will go (in the limit)
  - New evidence changes the *reference class*
  - Makes one think of *inherently random* events, like rolling dice

- **Subjectivist / Bayesian answer:**
  - Degrees of belief about unobserved variables
  - E.g. an agent's belief that it's raining, given the temperature
  - E.g. agent's belief how an opponent will behave, given the state
  - Often *learn* probabilities from past experiences (more later)
  - New evidence *updates beliefs* (more later)

Quiz: Informed Probabilities

- Let's say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise
- Question: What tree search should you use?



- Answer: Expectimax!
  - To figure out EACH chance node's probabilities, you have to run a simulation of your opponent
  - This kind of thing gets very slow very quickly
  - Even worse if you have to simulate your opponent simulating you…
  - … except for minimax, which has the nice property that it all collapses into one game tree

- **Dice rolls increase *b*: 21 possible rolls with 2 dice**
  - Backgammon ≈ 20 legal moves
  - Depth 2 → $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- **As depth increases, probability of reaching a given search node shrinks**
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier…
- **Historic AI: TDGammon uses depth-2 search + very good evaluation fuanction + reinforcement learning → world-champion level play**
- **1st AI world champion in any game!**

- **E.g. Backgammon**
- **Expectiminimax**
  - Environment is an extra "random agent" player that moves after each min/max agent
  - Each node computes the appropriate combination of its children