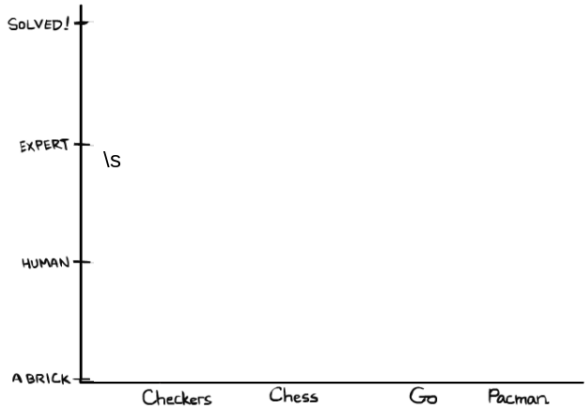
Game Playing State-of-the-Art

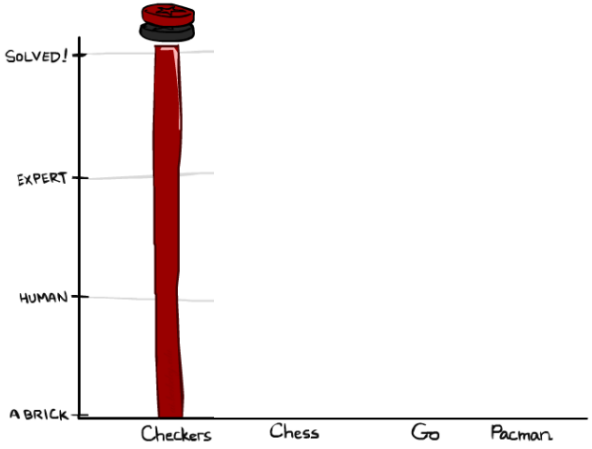
http://bit.ly/3GEMok9



Game Playing State-of-the-Art

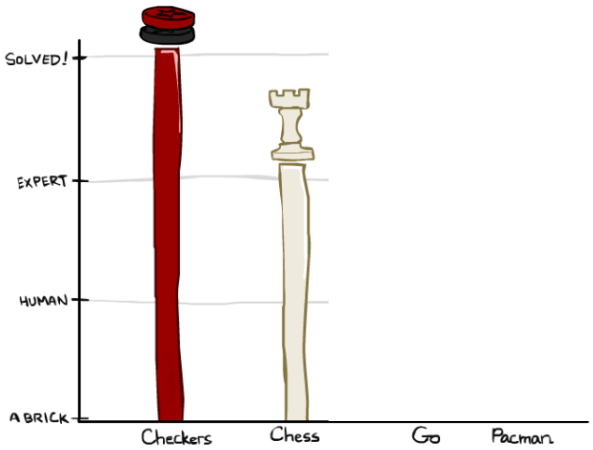
http://bit.ly/3GEMok9

Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame.
2007: Checkers solved!



Game Playing State-of-the-Art

http://bit.ly/3GEMok9

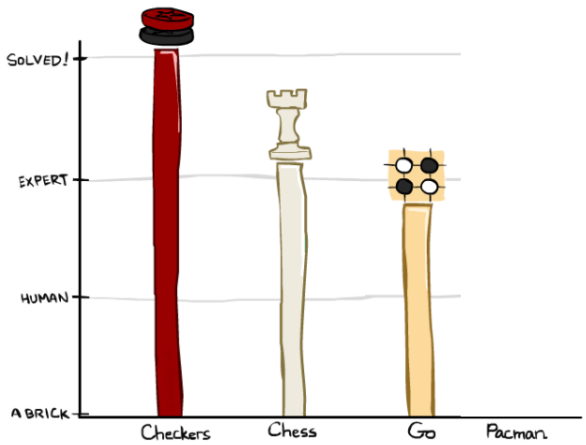


Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame.
2007: Checkers solved!

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Game Playing State-of-the-Art

http://bit.ly/3GEMok9



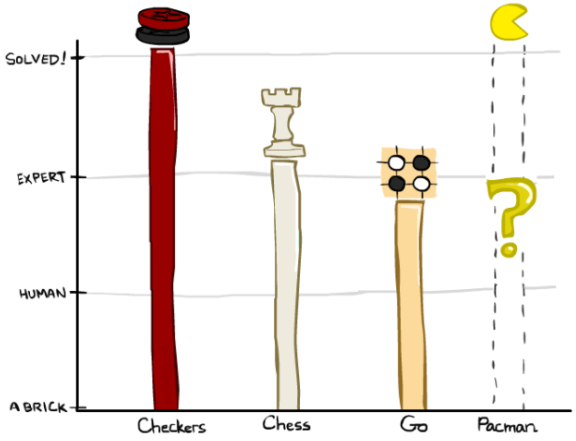
Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Go: Human champions are being beaten. In go, $b > 300$! Classically use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Game Playing State-of-the-Art

http://bit.ly/3GEMok9



Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

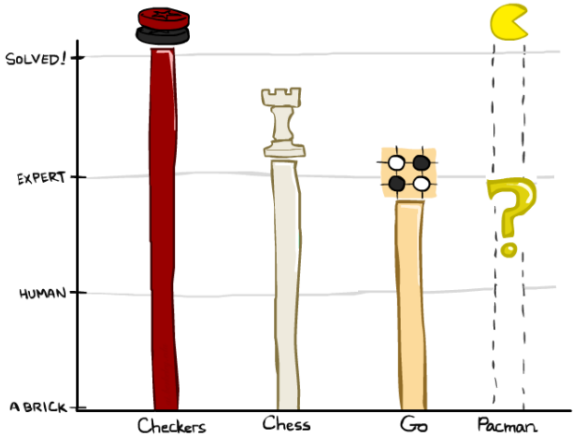
Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Go: Human champions are being beaten. In go, $b > 300$! Classically use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Pacman

Game Playing State-of-the-Art

http://bit.ly/3GEMok9



Chinook beat 40-year-reign of champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

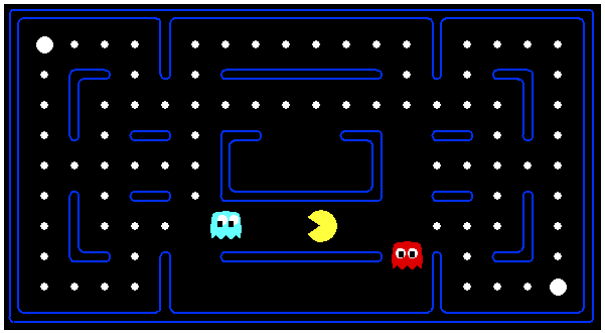
Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

Go: Human champions are being beaten. In go, $b > 300$! Classically use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Pacman

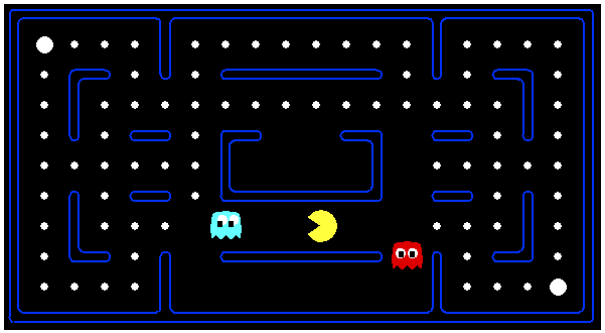
Behavior from Computation

http://bit.ly/3GEMok9



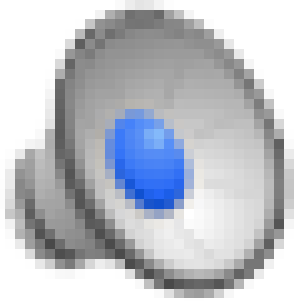
Behavior from Computation

http://bit.ly/3GEMok9



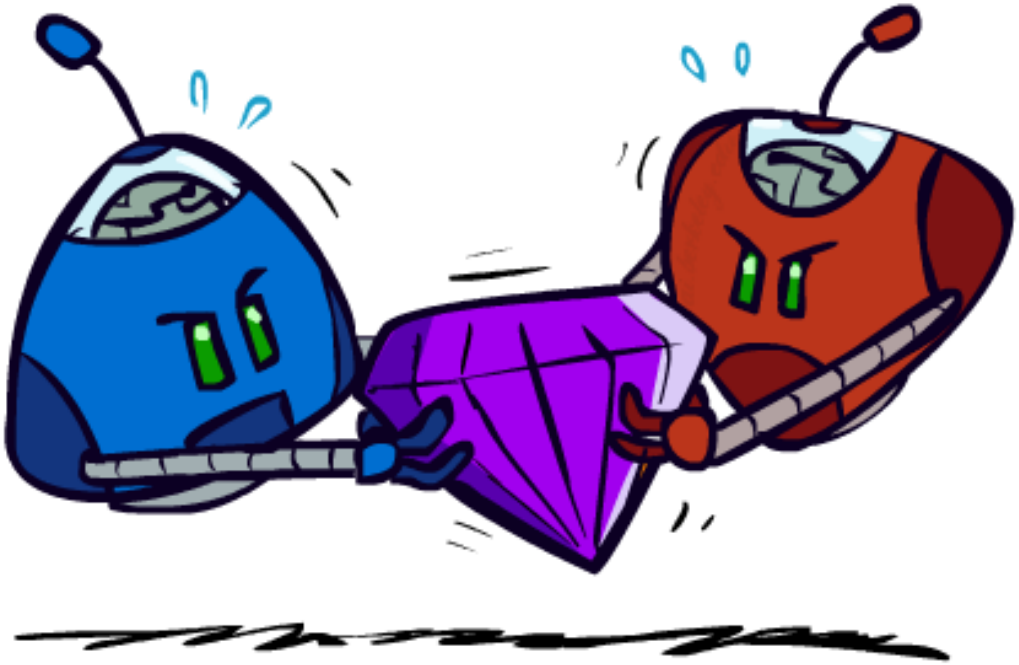
$Demo : mysterypacman(L6D1)$

Video of Demo Mystery Pacman



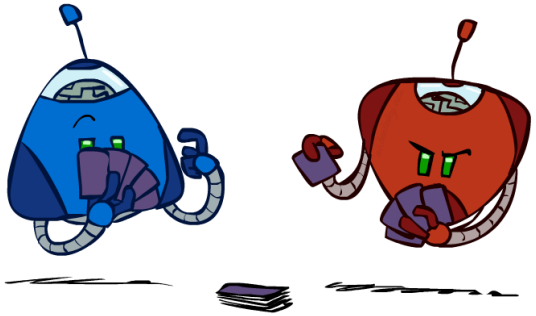
Adversarial Games

http://bit.ly/3GEMok9



Types of Games

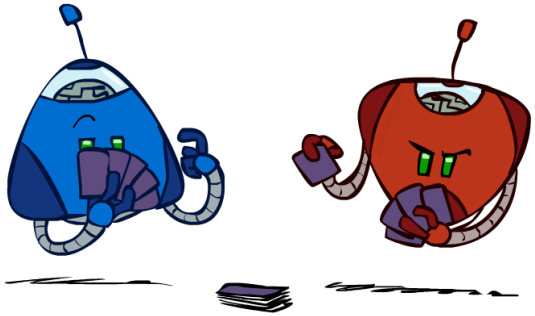
http://bit.ly/3GEMok9



Types of Games

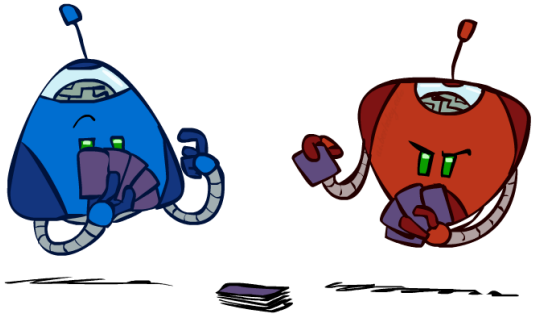
http://bit.ly/3GEMok9

Many different kinds of games!



Types of Games

http://bit.ly/3GEMok9

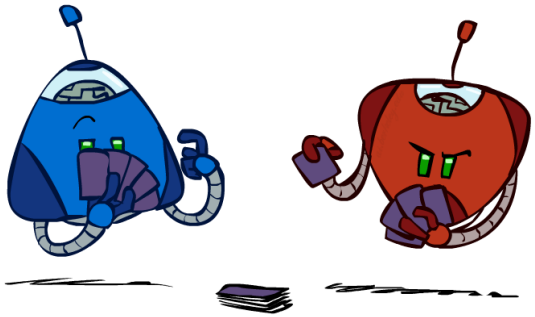


Many different kinds of games!

Axes:

Types of Games

http://bit.ly/3GEMok9



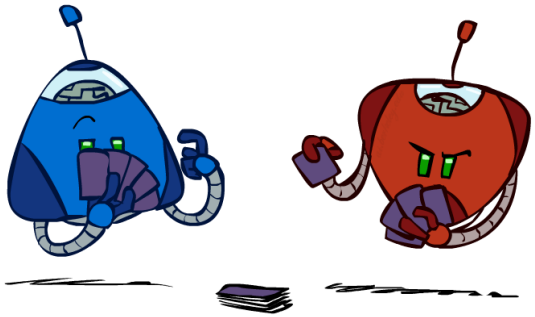
Many different kinds of games!

Axes:

- Deterministic or stochastic?

Types of Games

http://bit.ly/3GEMok9



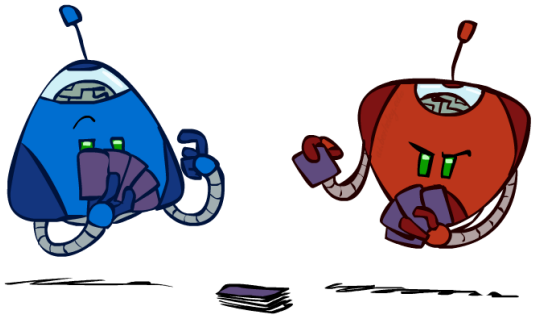
Many different kinds of games!

Axes:

- Deterministic or stochastic?
- One, two, or more players?

Types of Games

http://bit.ly/3GEMok9



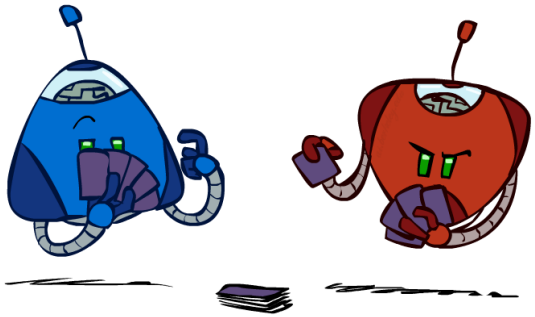
Many different kinds of games!

Axes:

- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?

Types of Games

http://bit.ly/3GEMok9



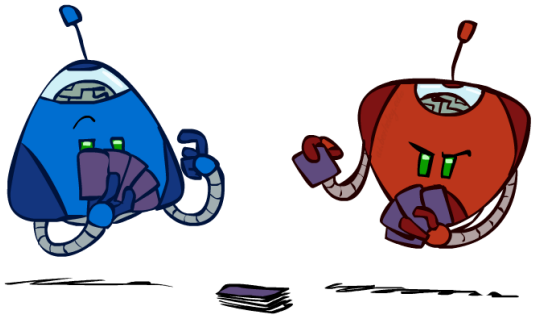
Many different kinds of games!

Axes:

- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?

Types of Games

http://bit.ly/3GEMok9



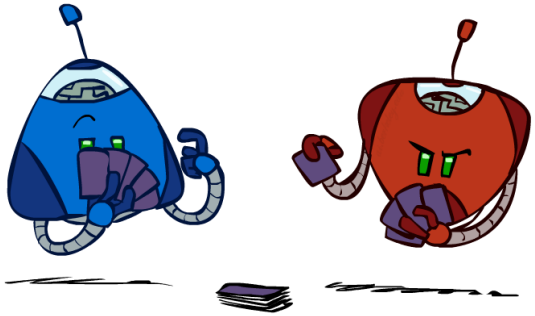
Many different kinds of games!

Axes:

- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?

Types of Games

http://bit.ly/3GEMok9



Many different kinds of games!

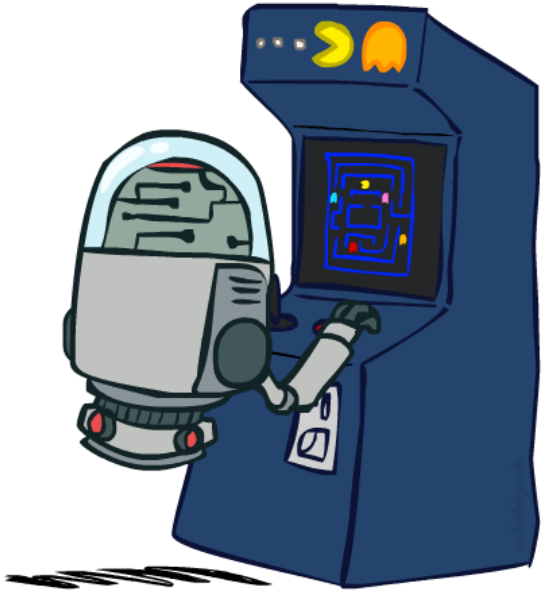
Axes:

- Deterministic or stochastic?
- One, two, or more players?
- Zero sum?
- Perfect information (can you see the state)?

Want algorithms for calculating a strategy (policy) which recommends a move from each state.

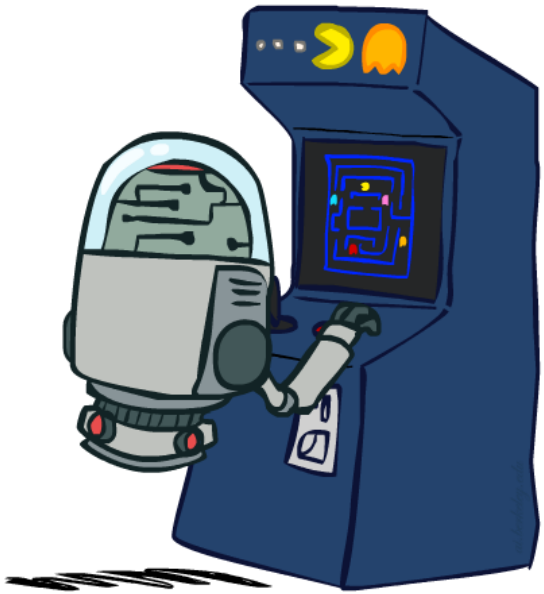
Deterministic Games

http://bit.ly/3GEMok9



Deterministic Games

http://bit.ly/3GEMok9

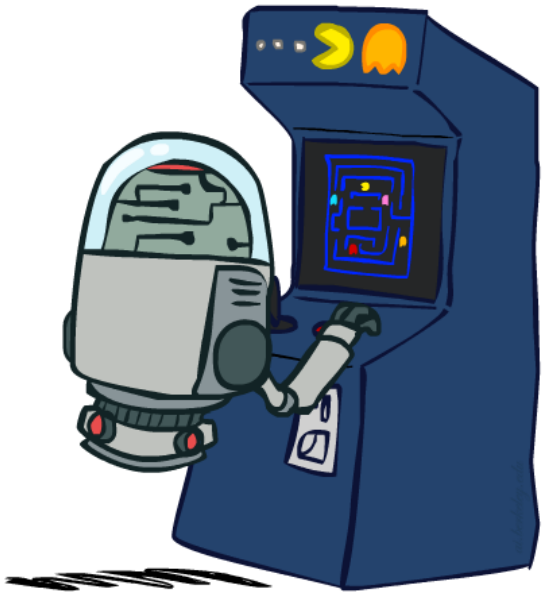


Many possible formalizations, one is:

- States: S (start at s0)

Deterministic Games

http://bit.ly/3GEMok9

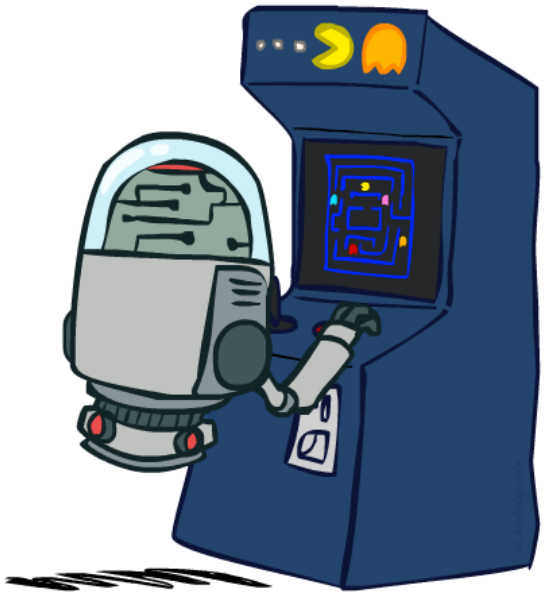


Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$

Deterministic Games

http://bit.ly/3GEMok9

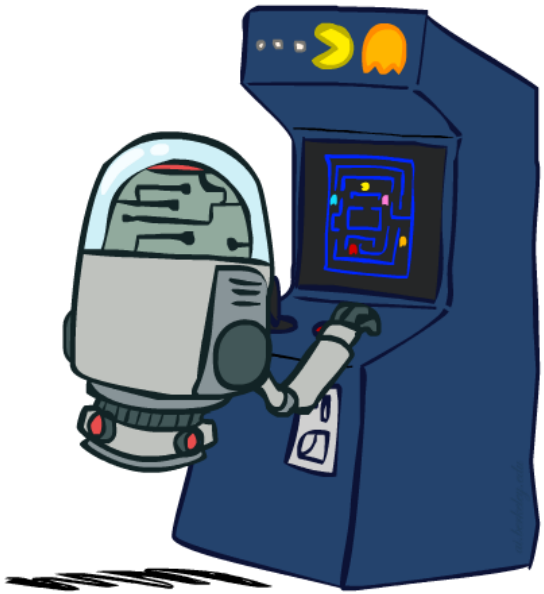


Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$ (usually take turns).

Deterministic Games

http://bit.ly/3GEMok9

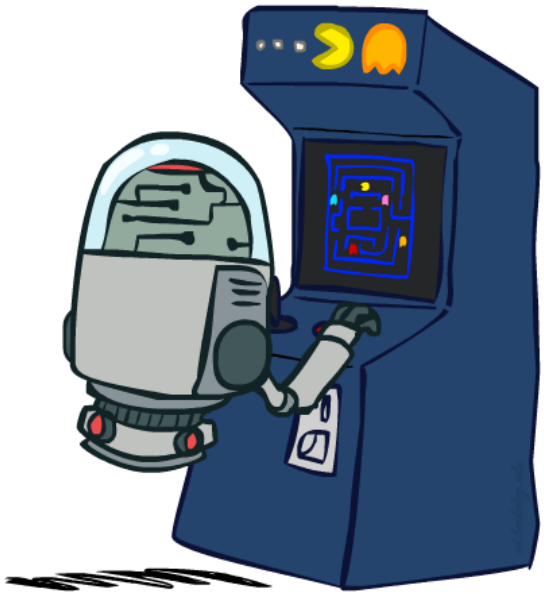


Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$ (usually take turns).
- Actions: A (may depend on player / state)

Deterministic Games

http://bit.ly/3GEMok9

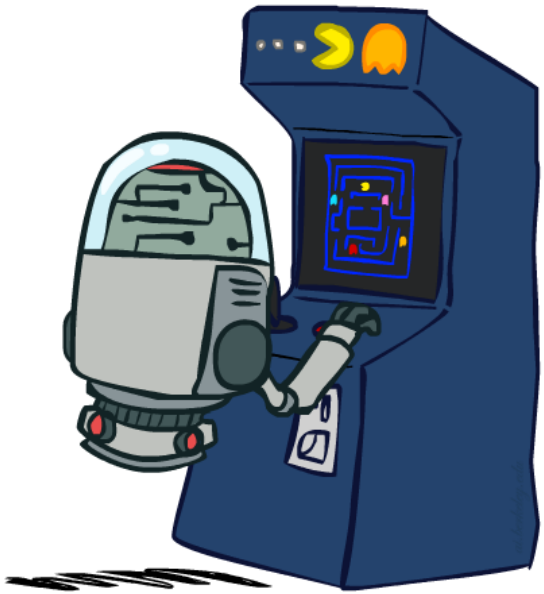


Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$ (usually take turns).
- Actions: A (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$.

Deterministic Games

http://bit.ly/3GEMok9

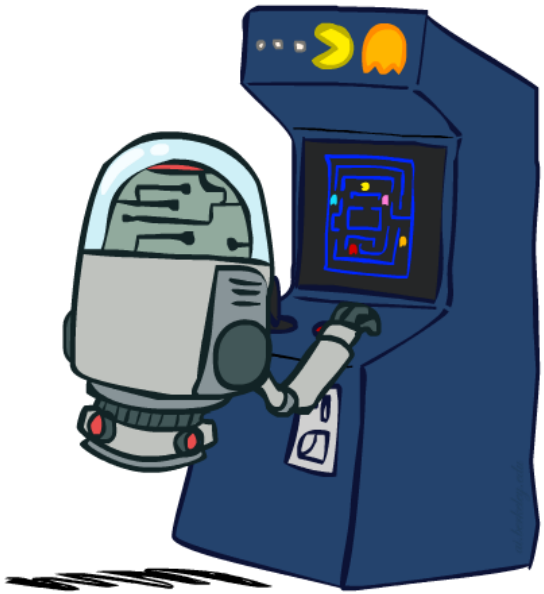


Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$ (usually take turns).
- Actions: A (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$.
- Terminal Test: $S \rightarrow \{t, f\}$

Deterministic Games

http://bit.ly/3GEMok9

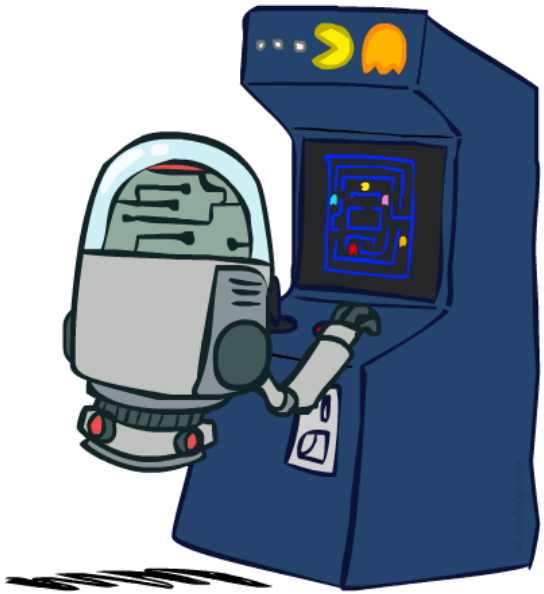


Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$ (usually take turns).
- Actions: A (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$.
- Terminal Test: $S \rightarrow \{t, f\}$
- Terminal Utilities: $S \times P \rightarrow R$.

Deterministic Games

http://bit.ly/3GEMok9

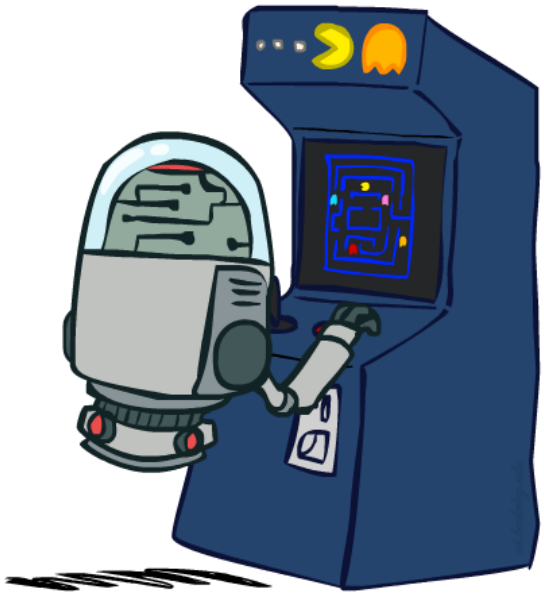


Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$ (usually take turns).
- Actions: A (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$.
- Terminal Test: $S \rightarrow \{t, f\}$
- Terminal Utilities: $S \times P \rightarrow R$.

Deterministic Games

http://bit.ly/3GEMok9



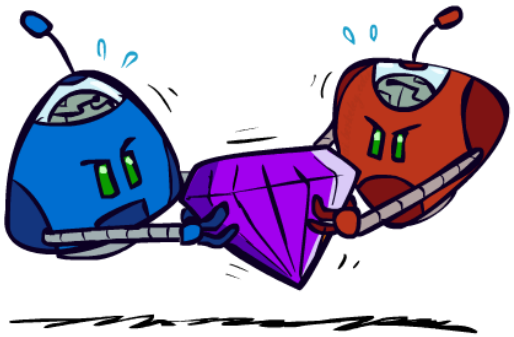
Many possible formalizations, one is:

- States: S (start at s0)
- Players: $P = \{1...N\}$ (usually take turns).
- Actions: A (may depend on player / state)
- Transition Function: $S \times A \rightarrow S$.
- Terminal Test: $S \rightarrow \{t, f\}$
- Terminal Utilities: $S \times P \rightarrow R$.

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

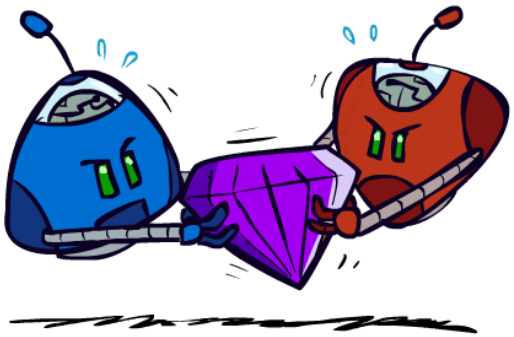
Zero-Sum Games

http://bit.ly/3GEMok9



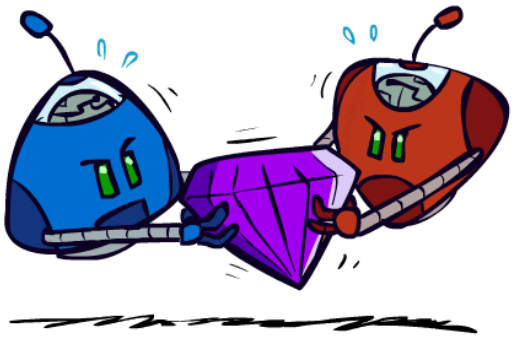
Zero-Sum Games

http://bit.ly/3GEMok9



Zero-Sum Games

http://bit.ly/3GEMok9

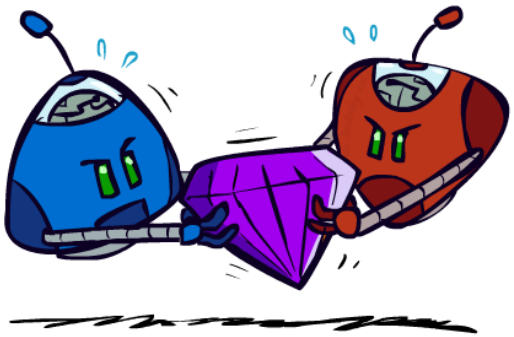


Zero-Sum Games

- Agents have opposite utilities (values on outcomes)

Zero-Sum Games

http://bit.ly/3GEMok9

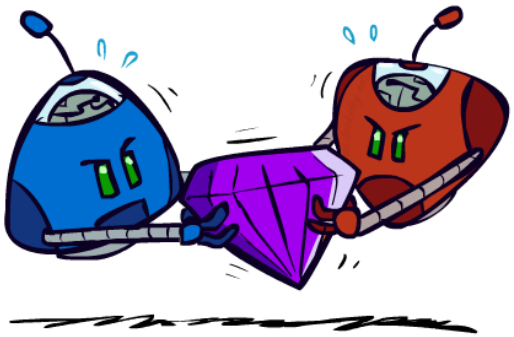


Zero-Sum Games

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

Zero-Sum Games

http://bit.ly/3GEMok9

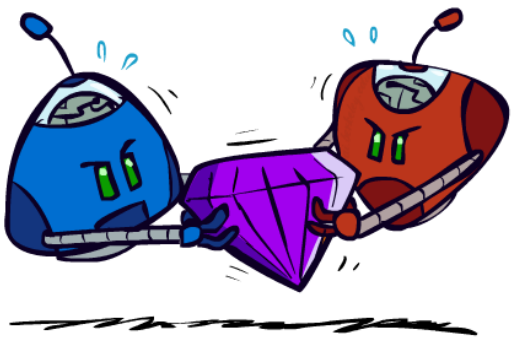


Zero-Sum Games

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

Zero-Sum Games

http://bit.ly/3GEMok9



Zero-Sum Games

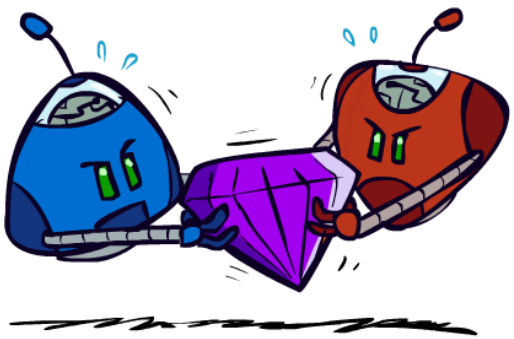
- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

- Agents have independent utilities (values on outcomes)

Zero-Sum Games

http://bit.ly/3GEMok9



Zero-Sum Games

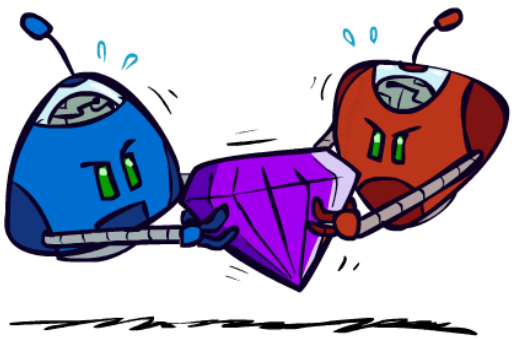
- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible

Zero-Sum Games

http://bit.ly/3GEMok9



Zero-Sum Games

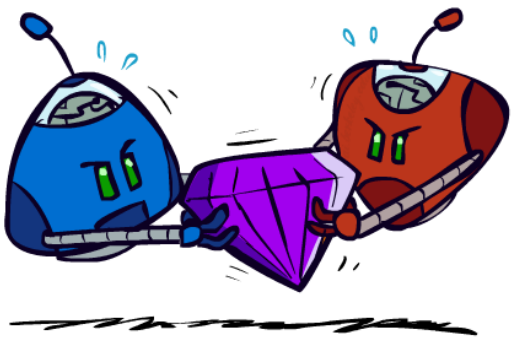
- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games

Zero-Sum Games

http://bit.ly/3GEMok9



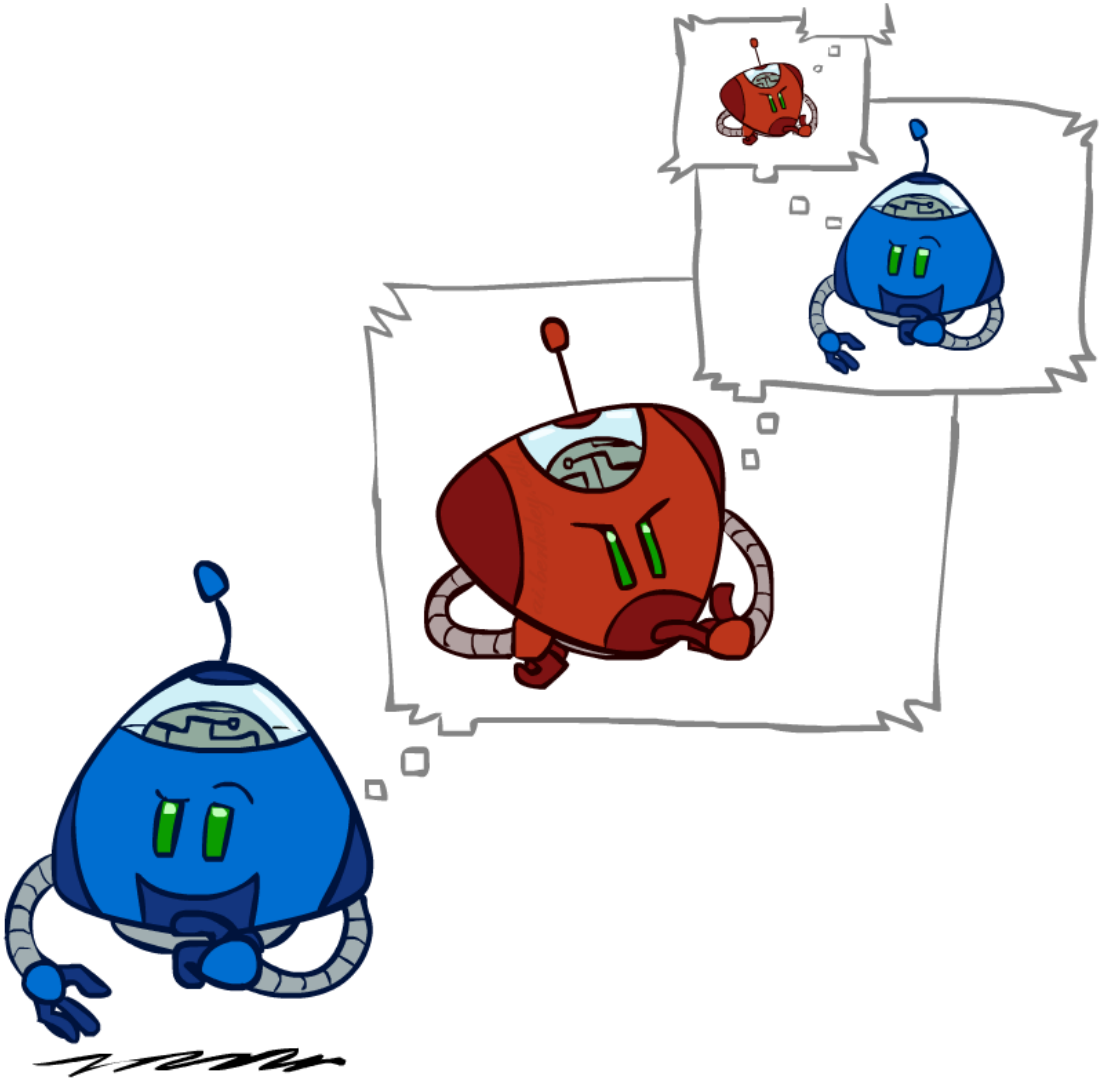
Zero-Sum Games

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

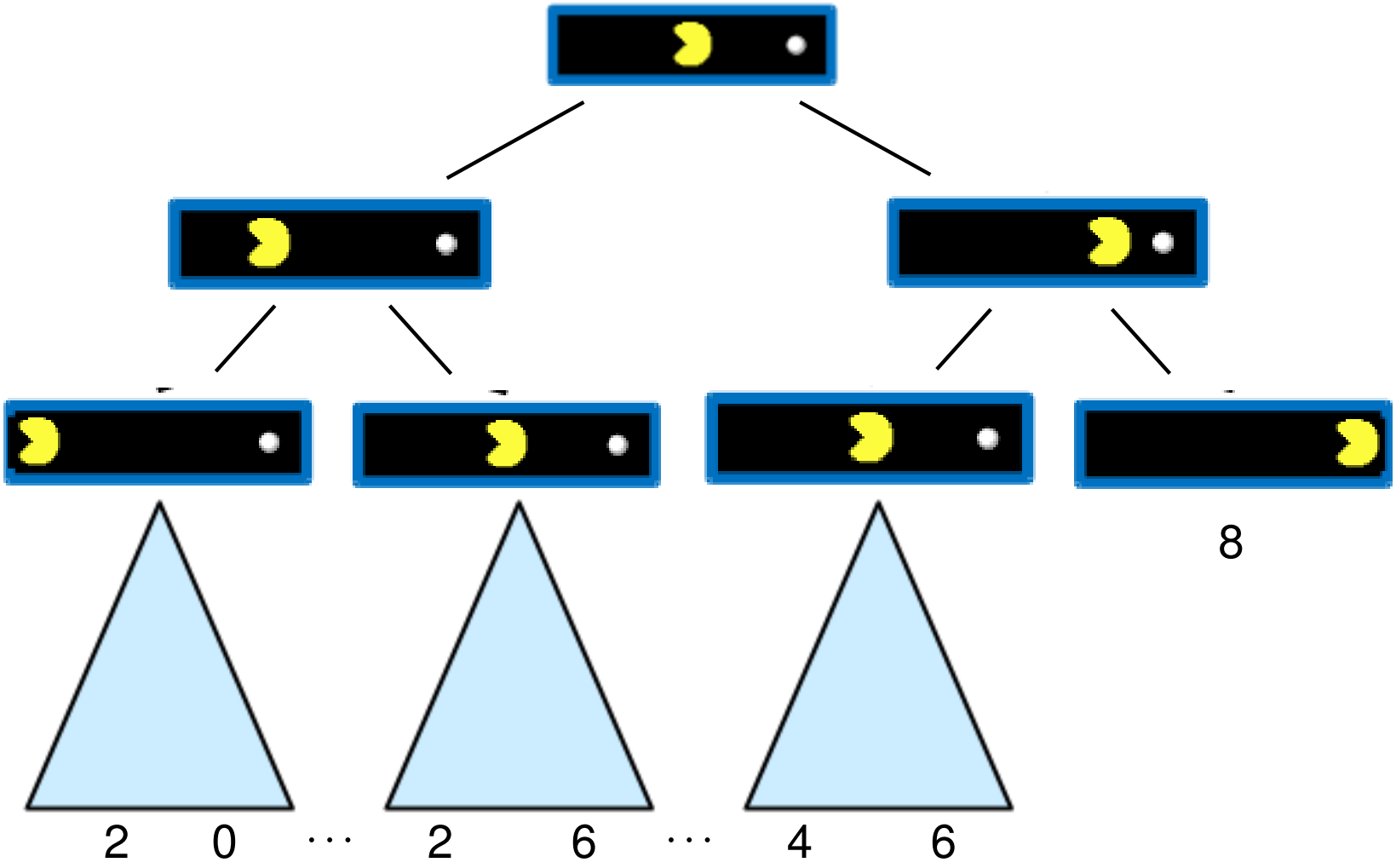
General Games

- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games

Adversarial Search



Single-Agent Trees



Value of Game Tree

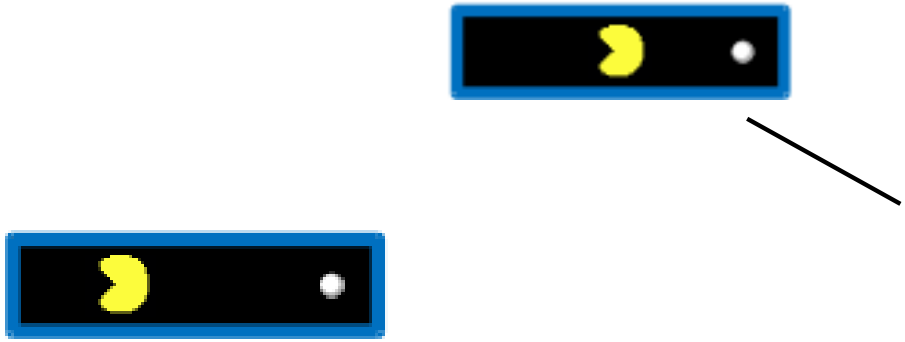
Value of a State: utility of best achievable outcome from state.

Value of Game Tree

Value of a State: utility of best achievable outcome from state.

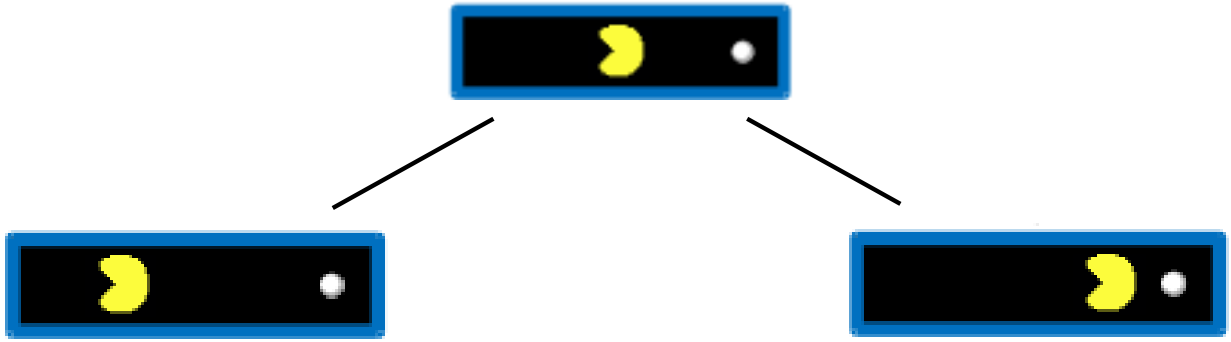
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



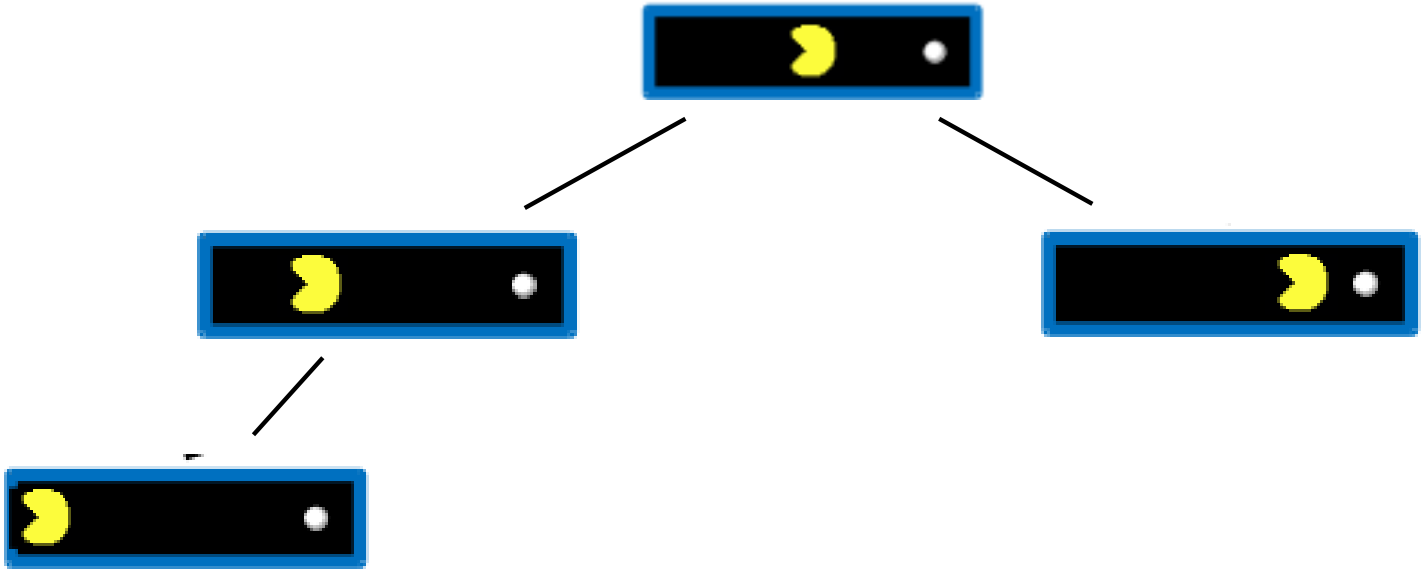
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



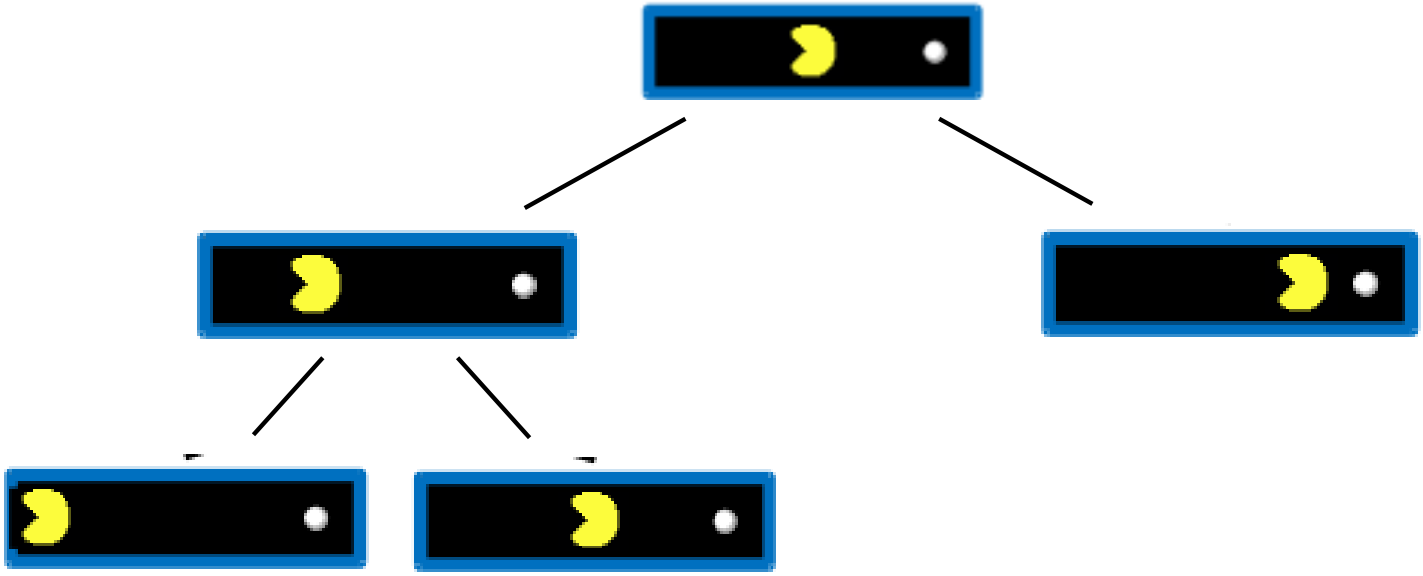
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



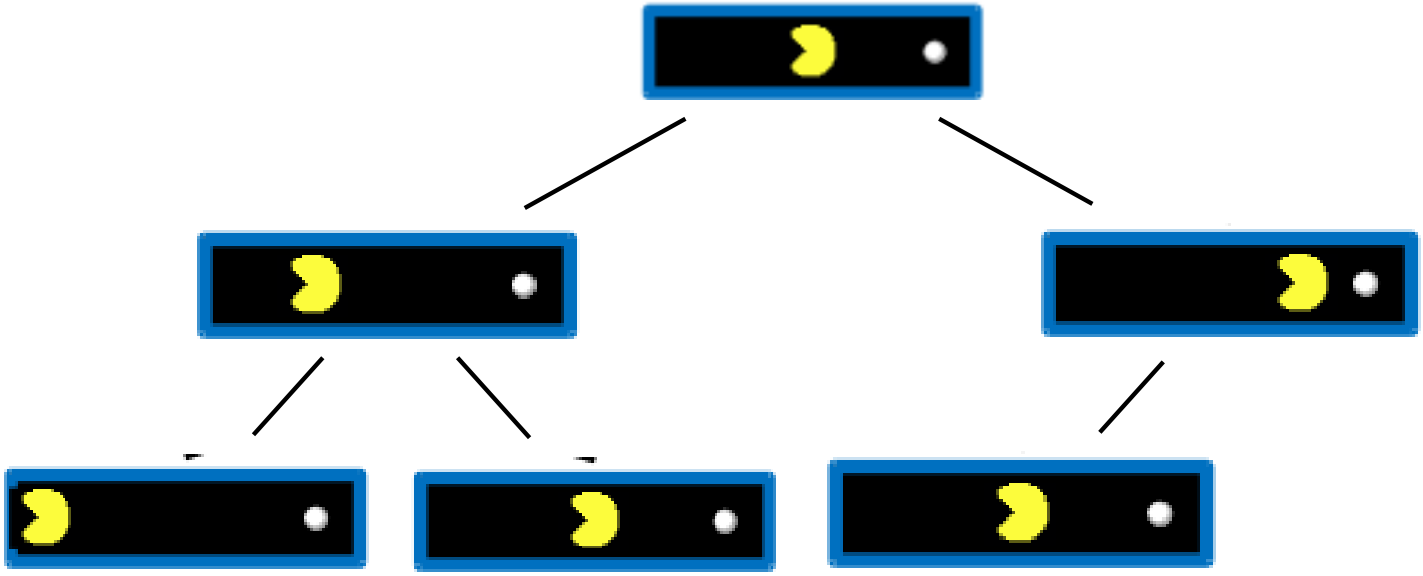
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



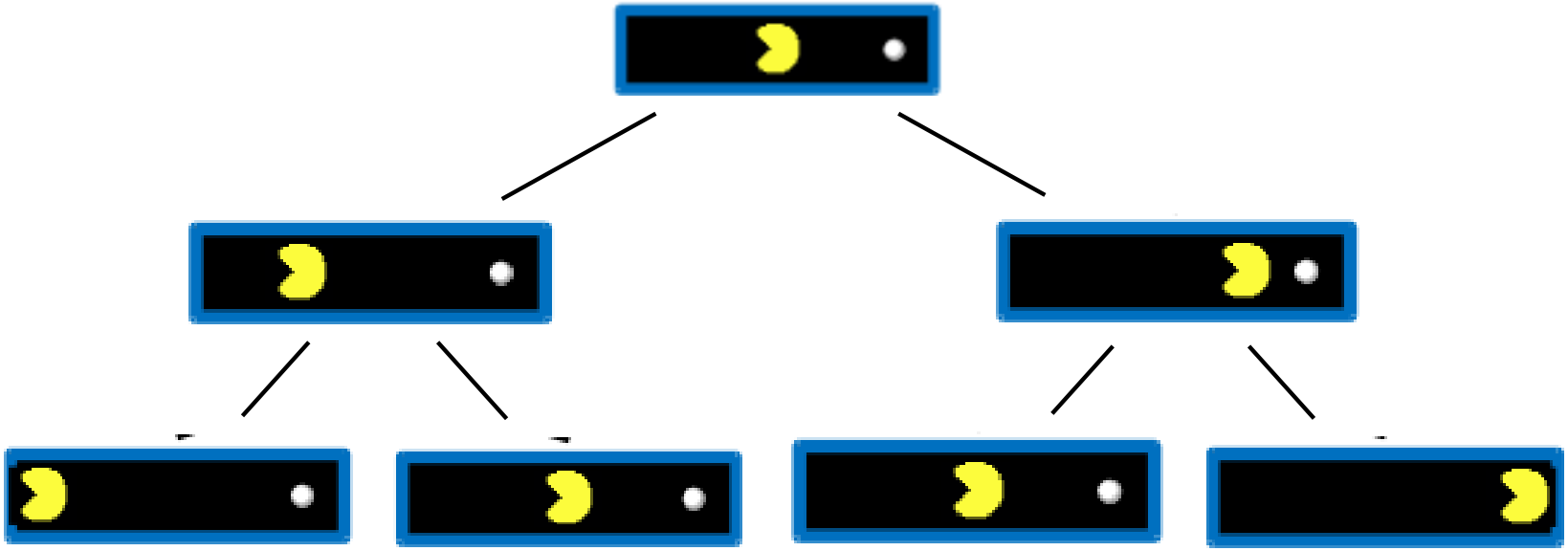
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



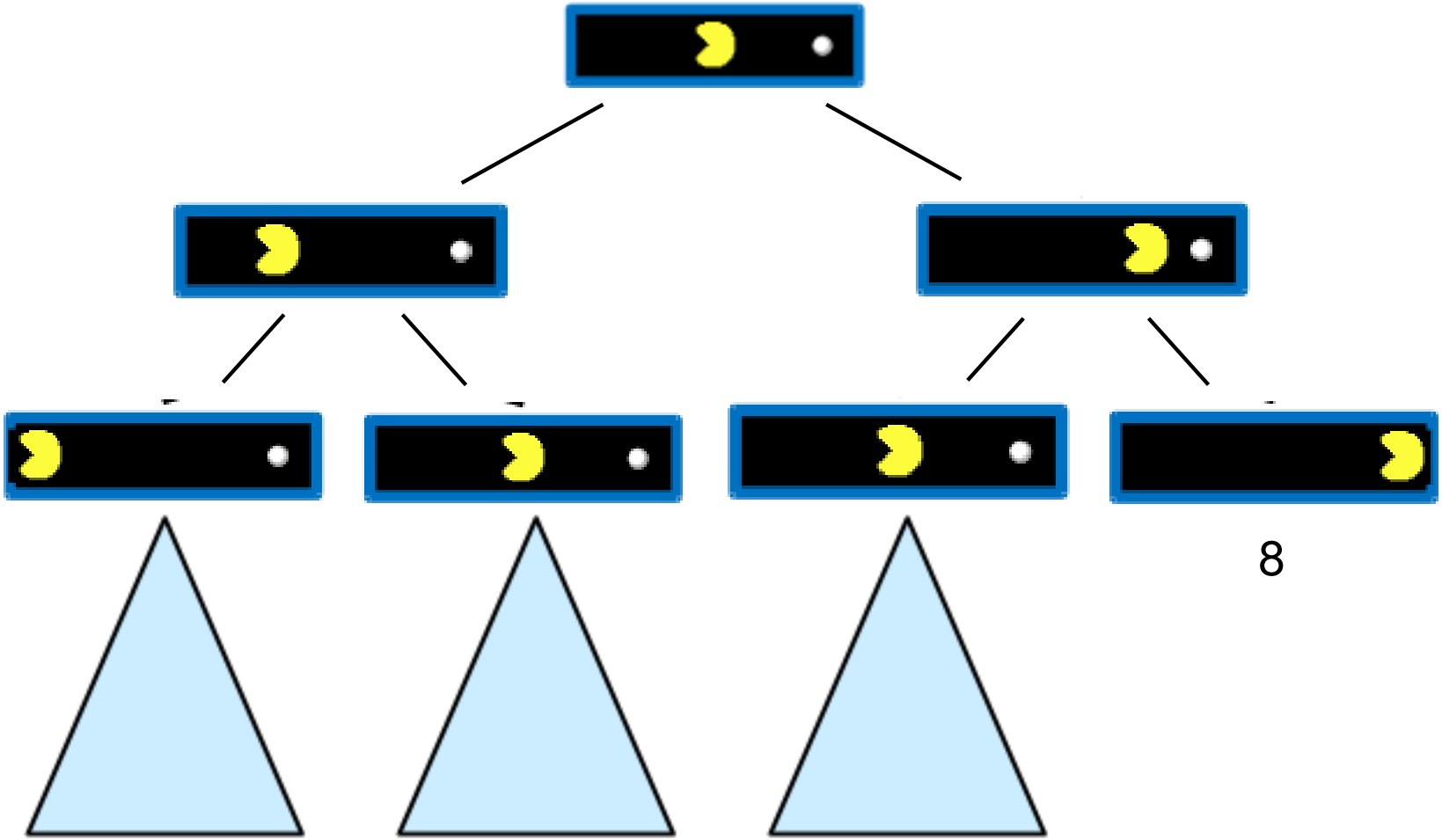
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



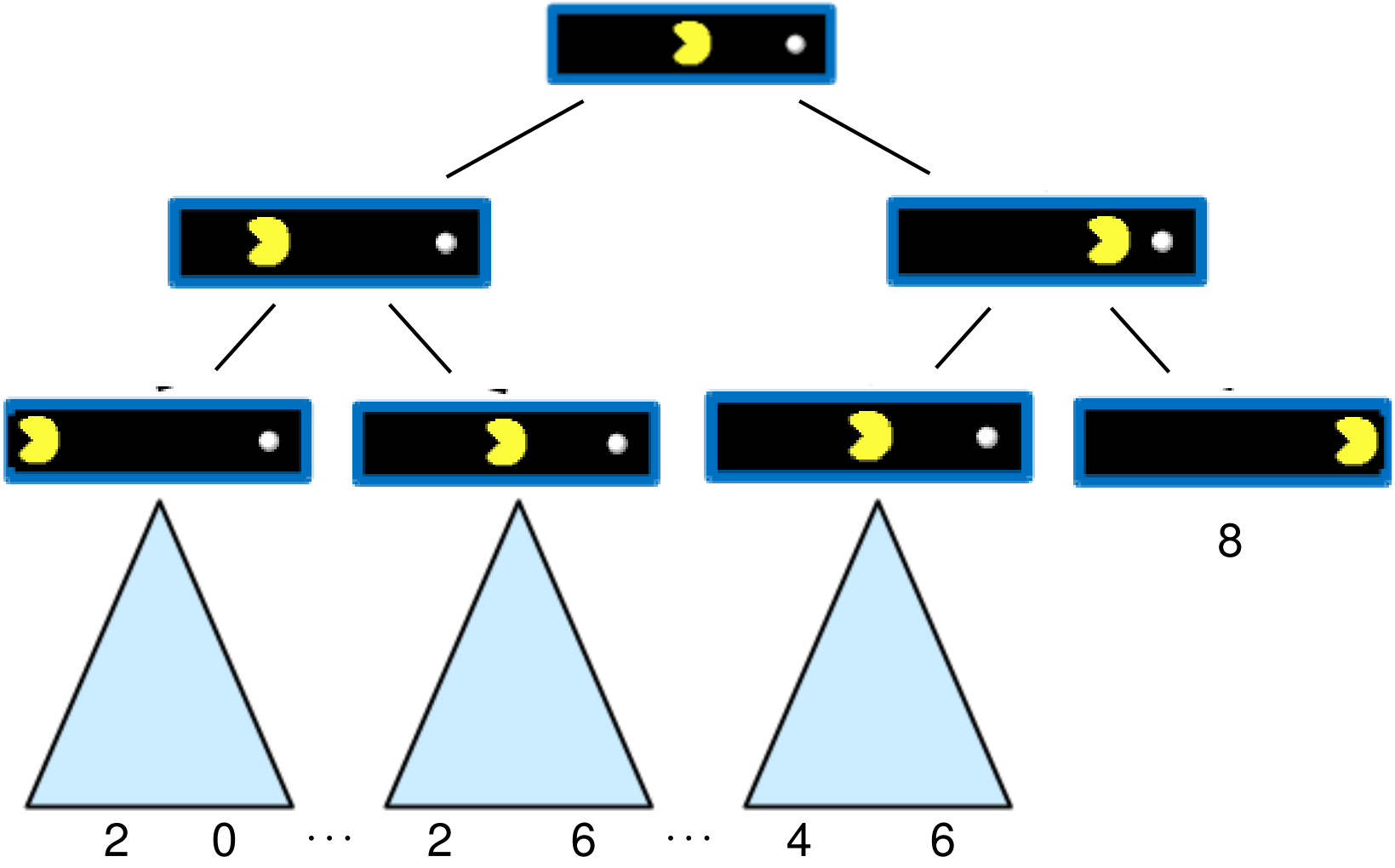
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



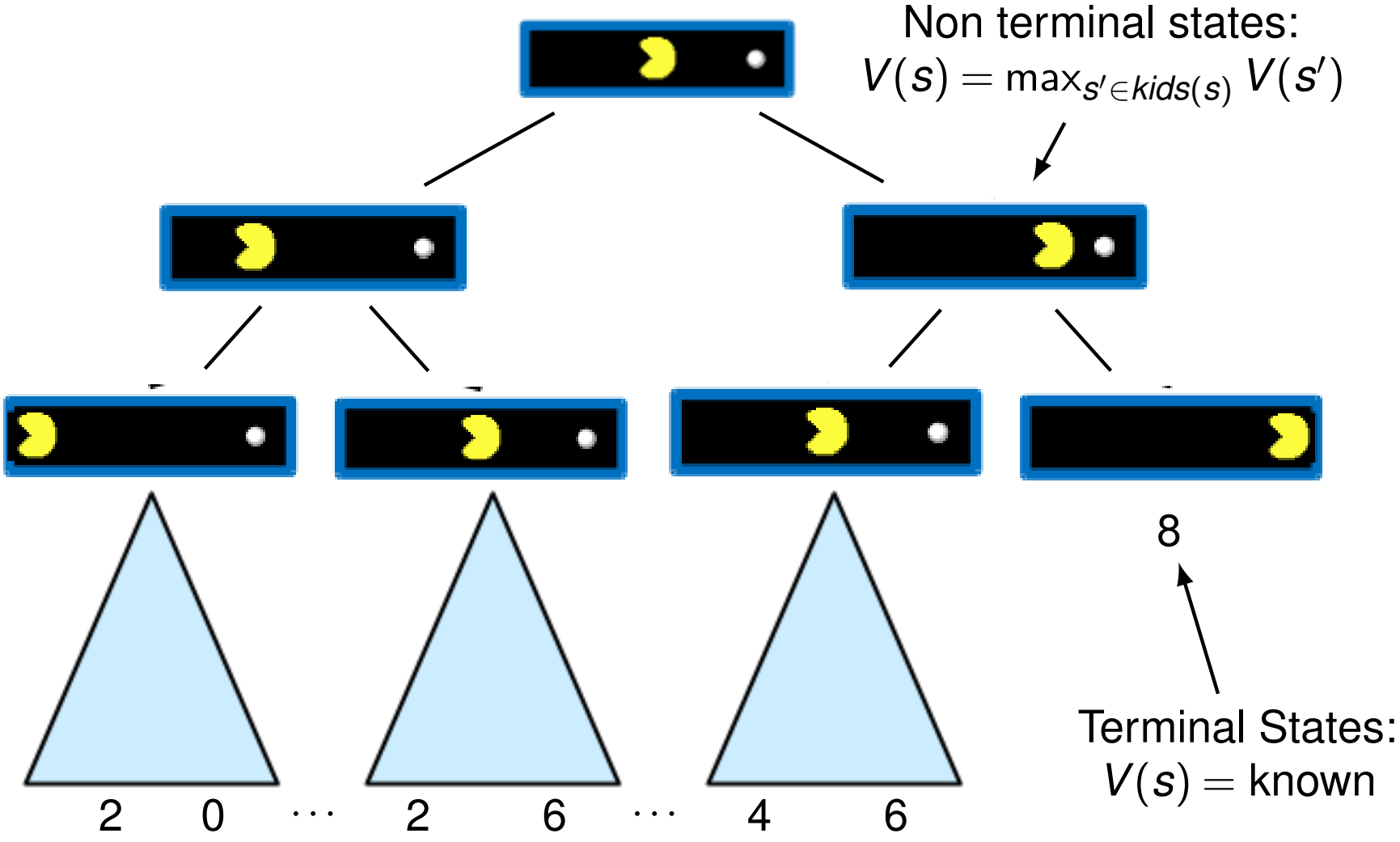
Value of Game Tree

Value of a State: utility of best achievable outcome from state.



Value of Game Tree

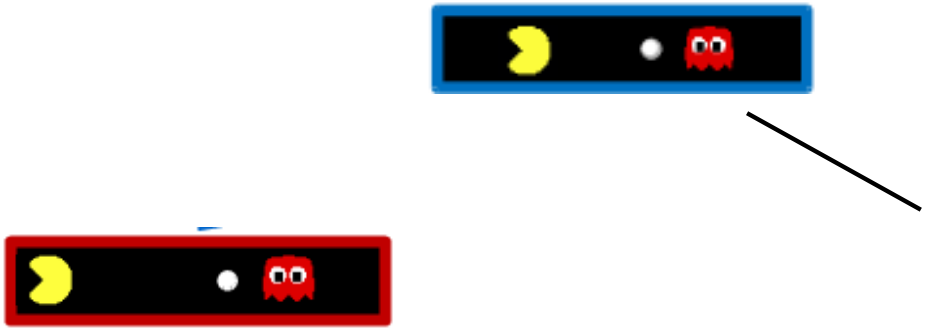
Value of a State: utility of best achievable outcome from state.



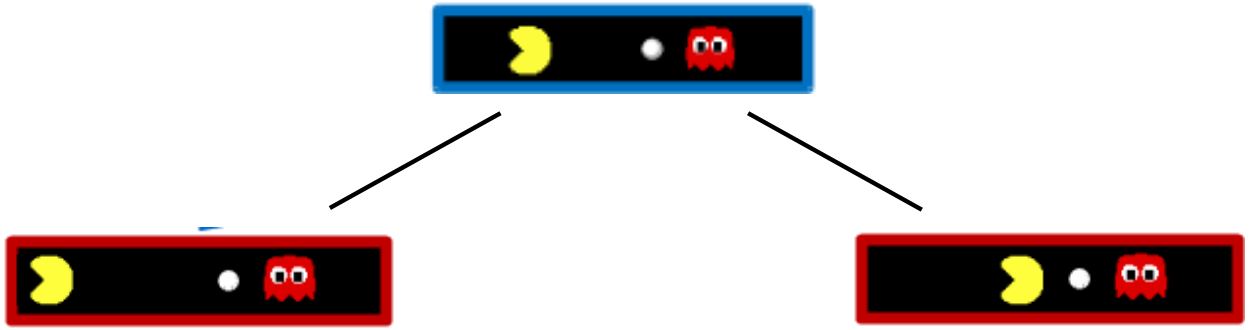
Adversarial Game Tree

Adversarial Game Tree

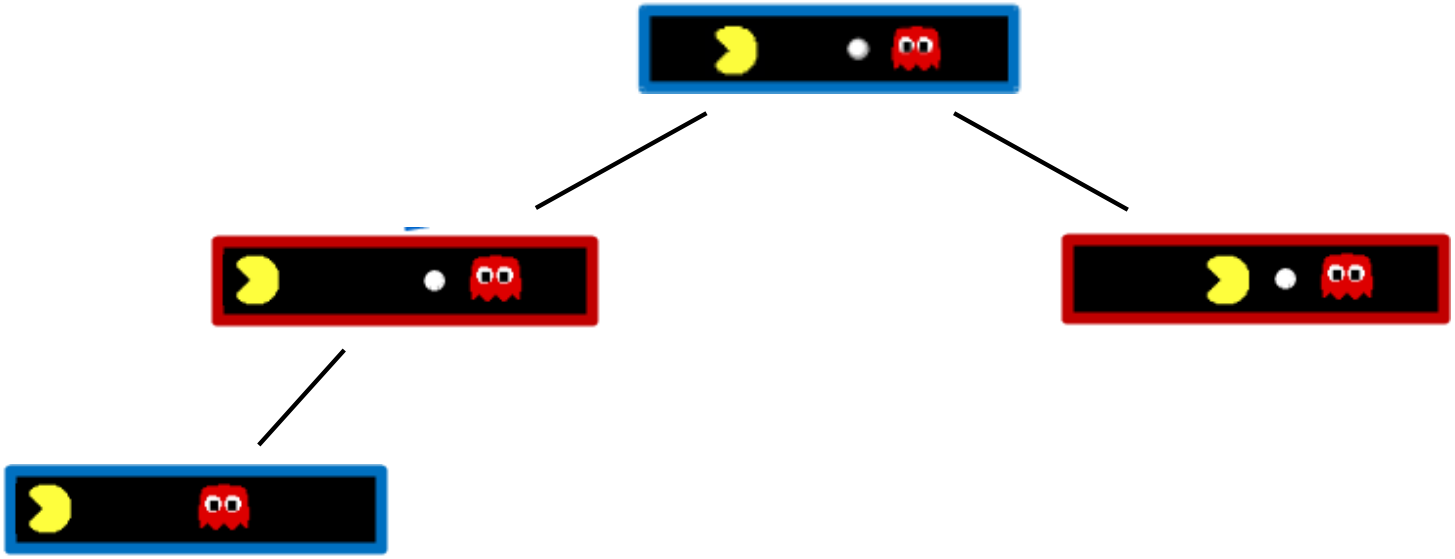
Adversarial Game Tree



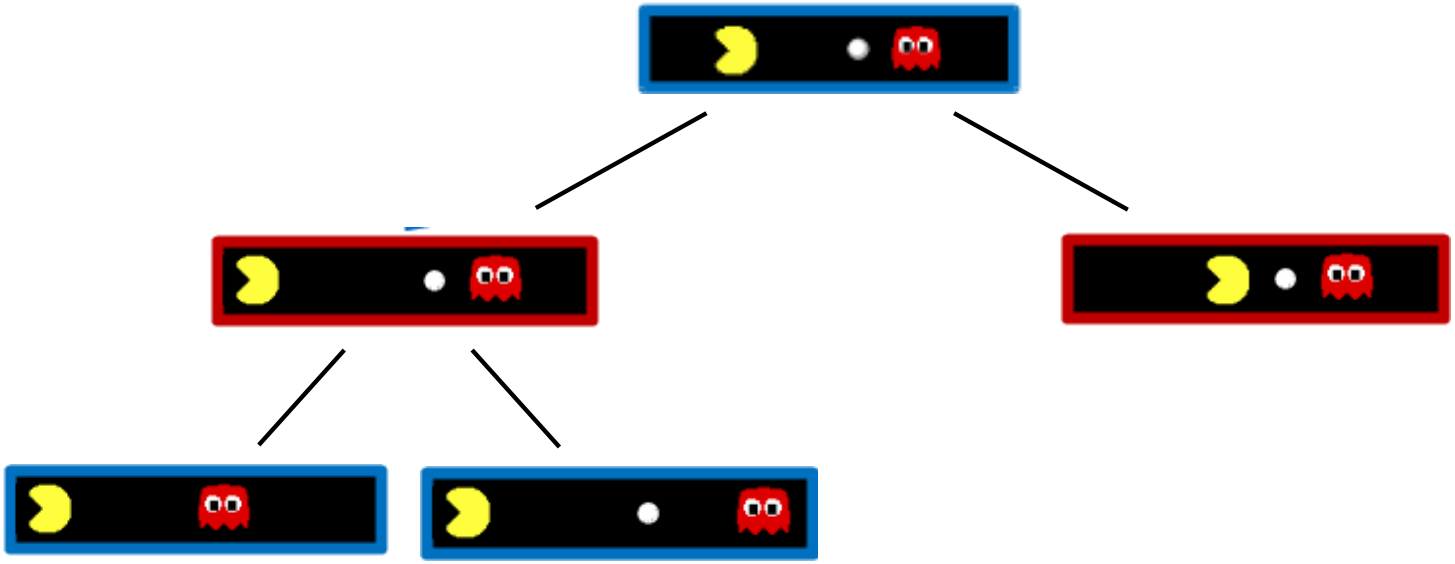
Adversarial Game Tree



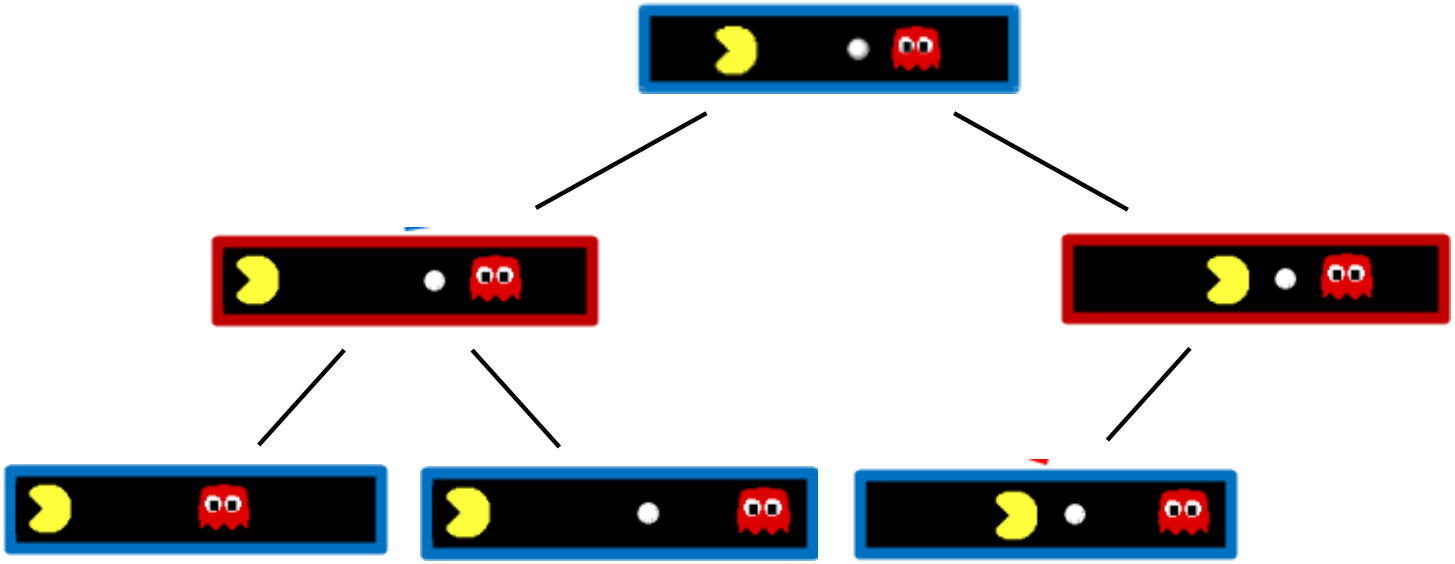
Adversarial Game Tree



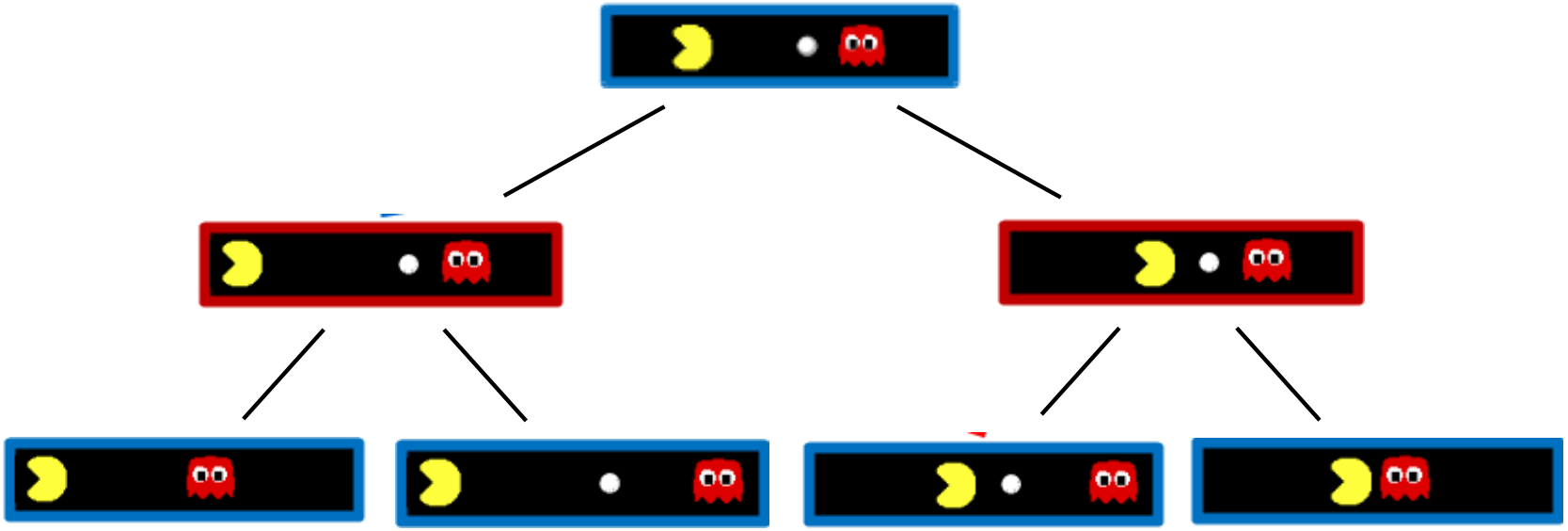
Adversarial Game Tree



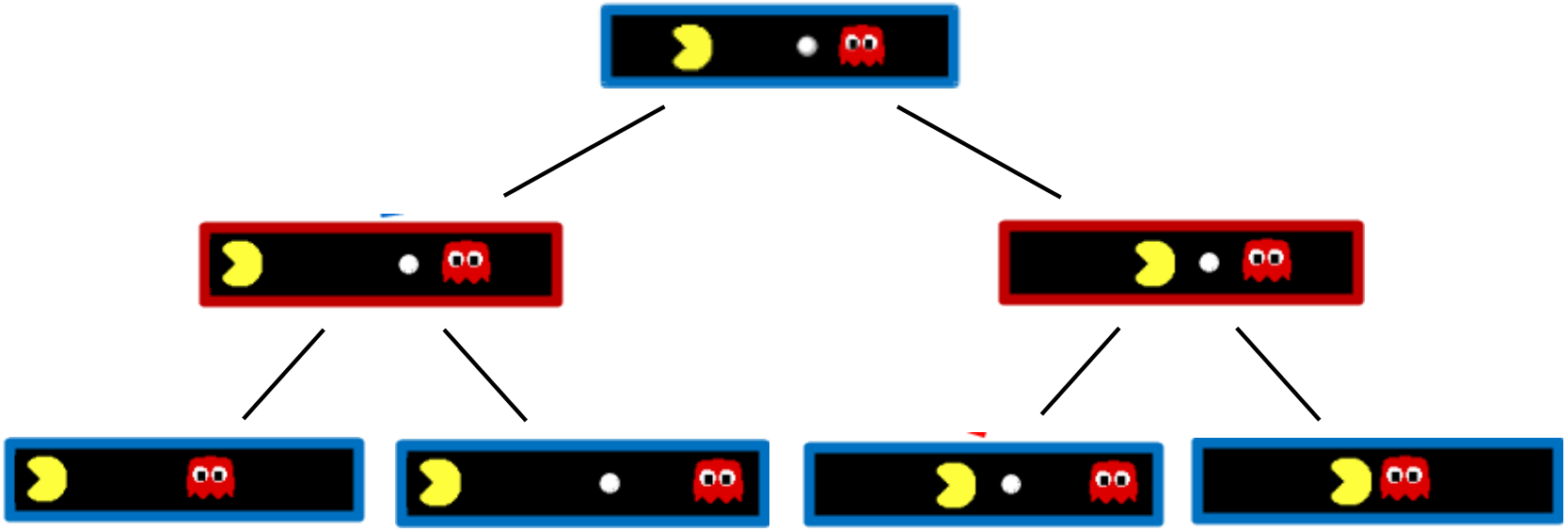
Adversarial Game Tree



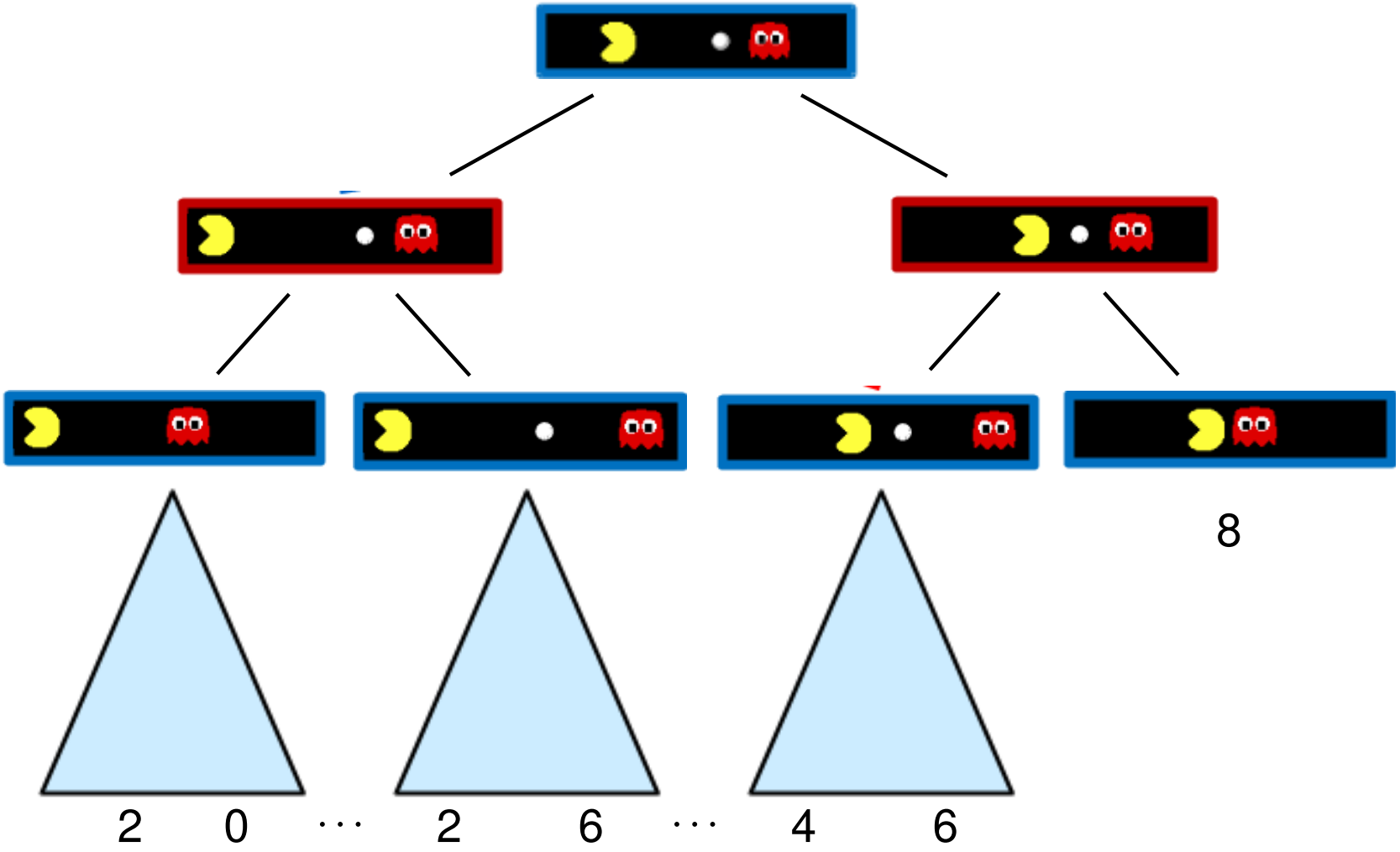
Adversarial Game Tree



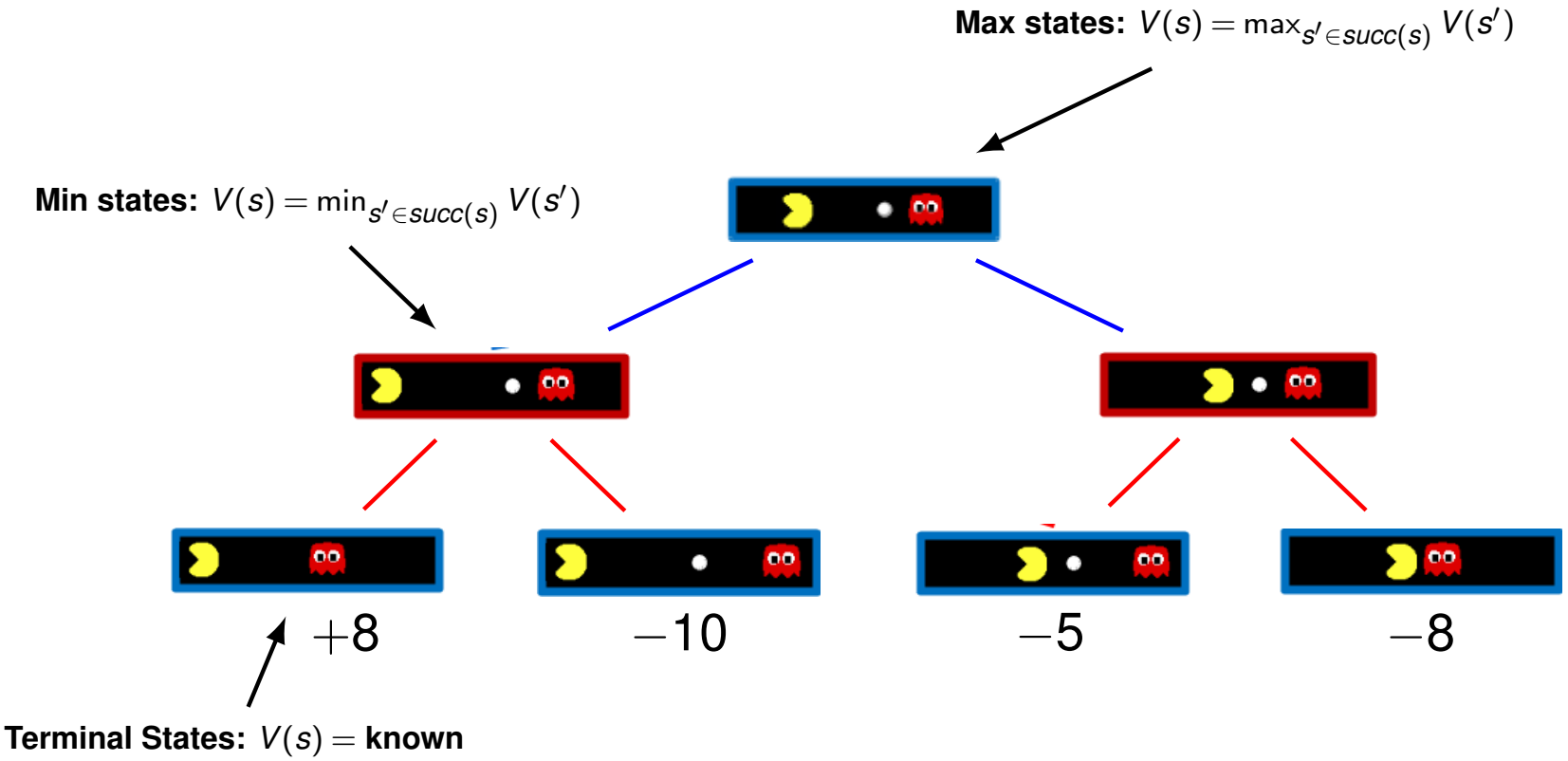
Adversarial Game Tree



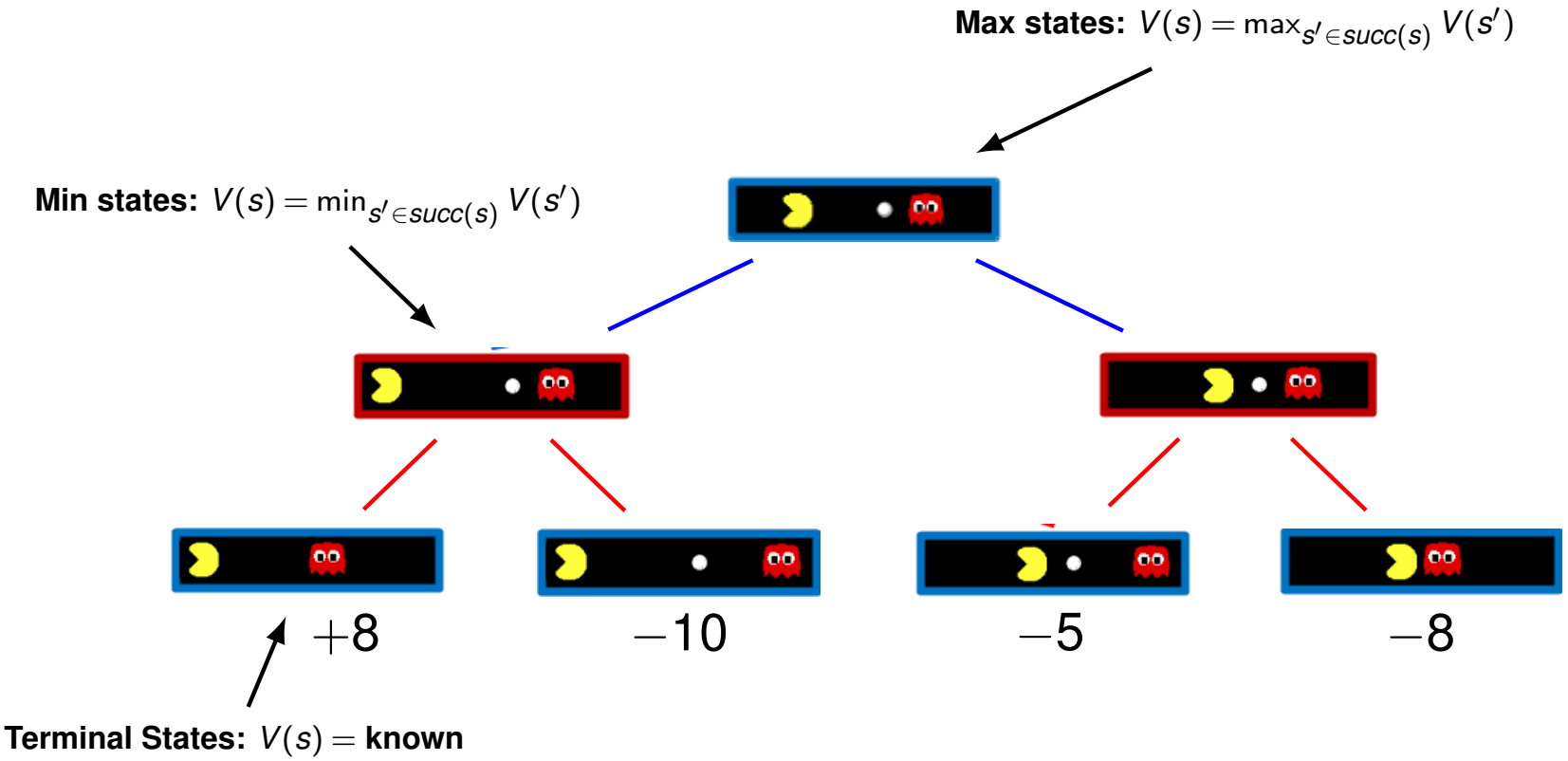
Adversarial Game Tree



Minimax Values

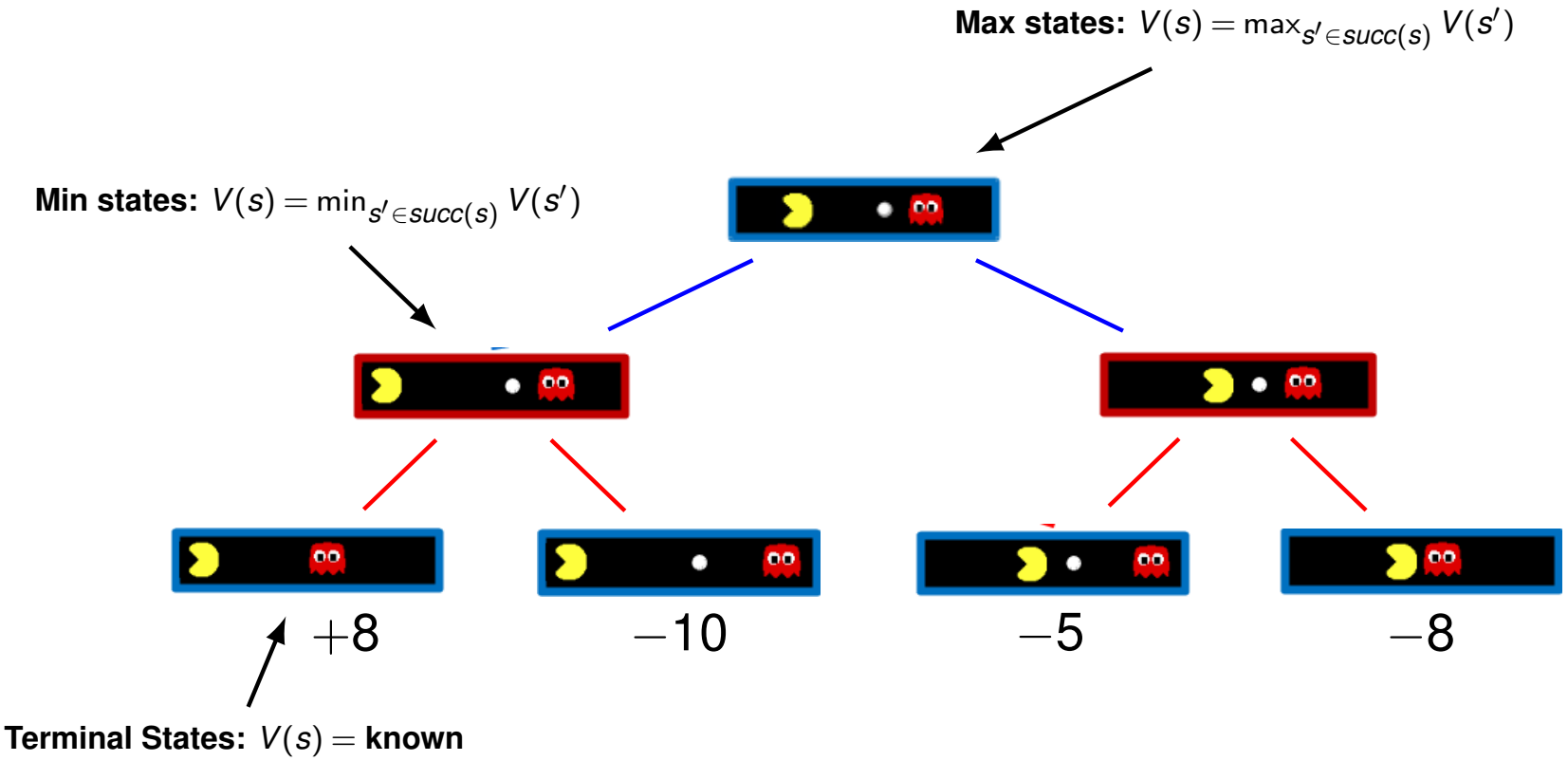


Minimax Values



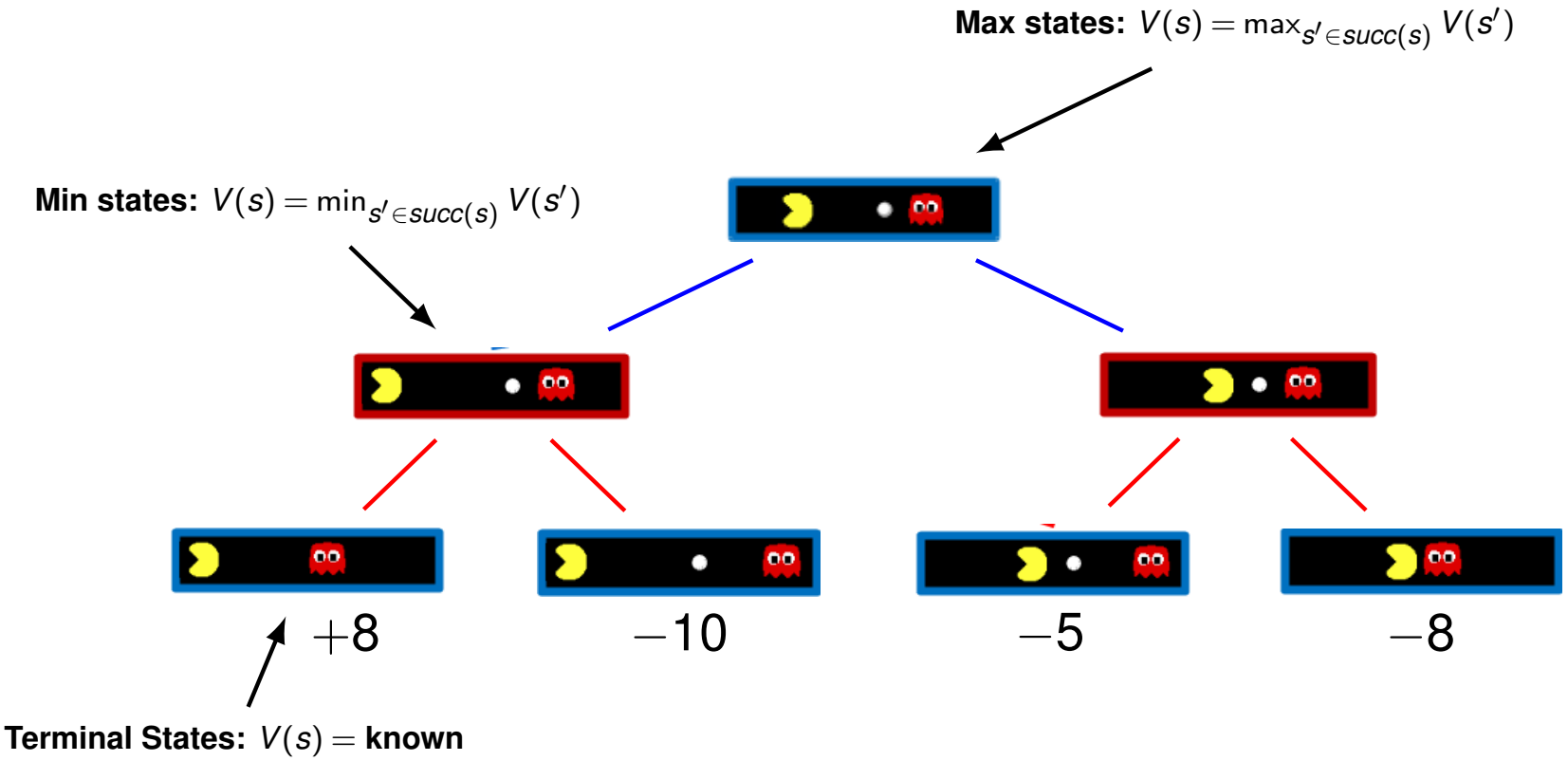
States Under Agent's Control.

Minimax Values



States Under Agent's Control. Max.

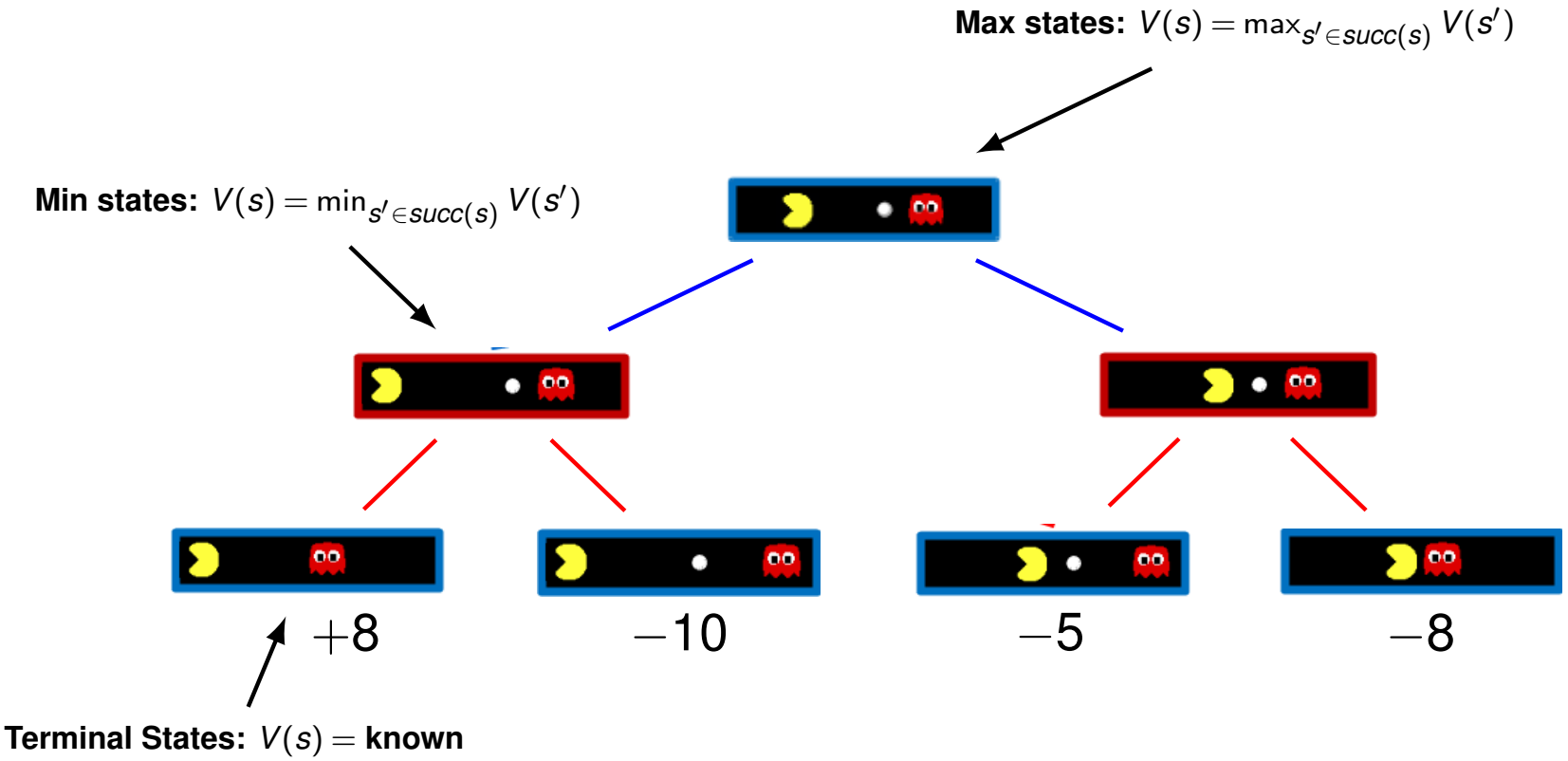
Minimax Values



States Under Agent's Control. Max.

Terminal States.

Minimax Values

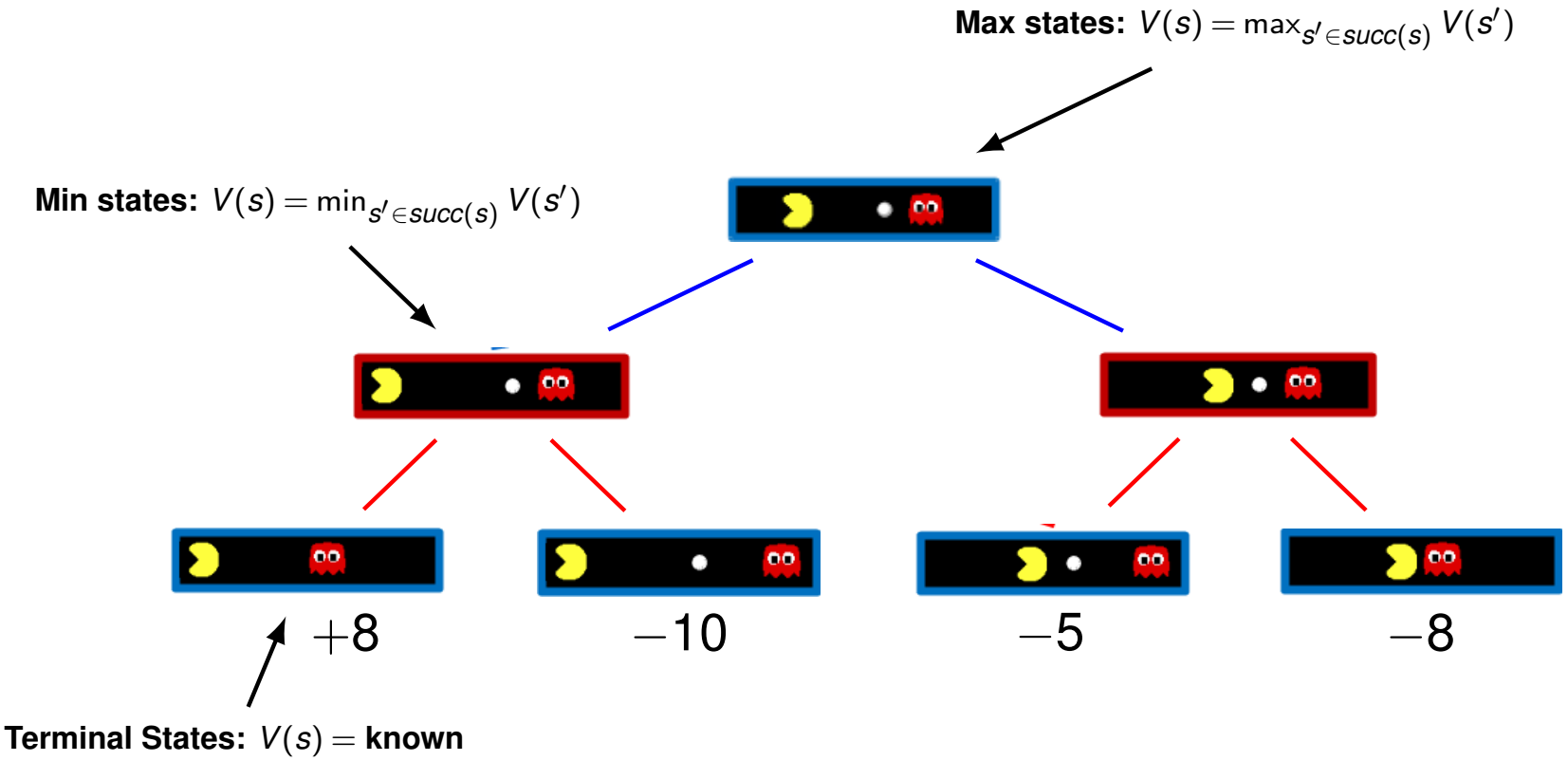


States Under Agent's Control. Max.

Terminal States.

States Under Opponent's Control.

Minimax Values

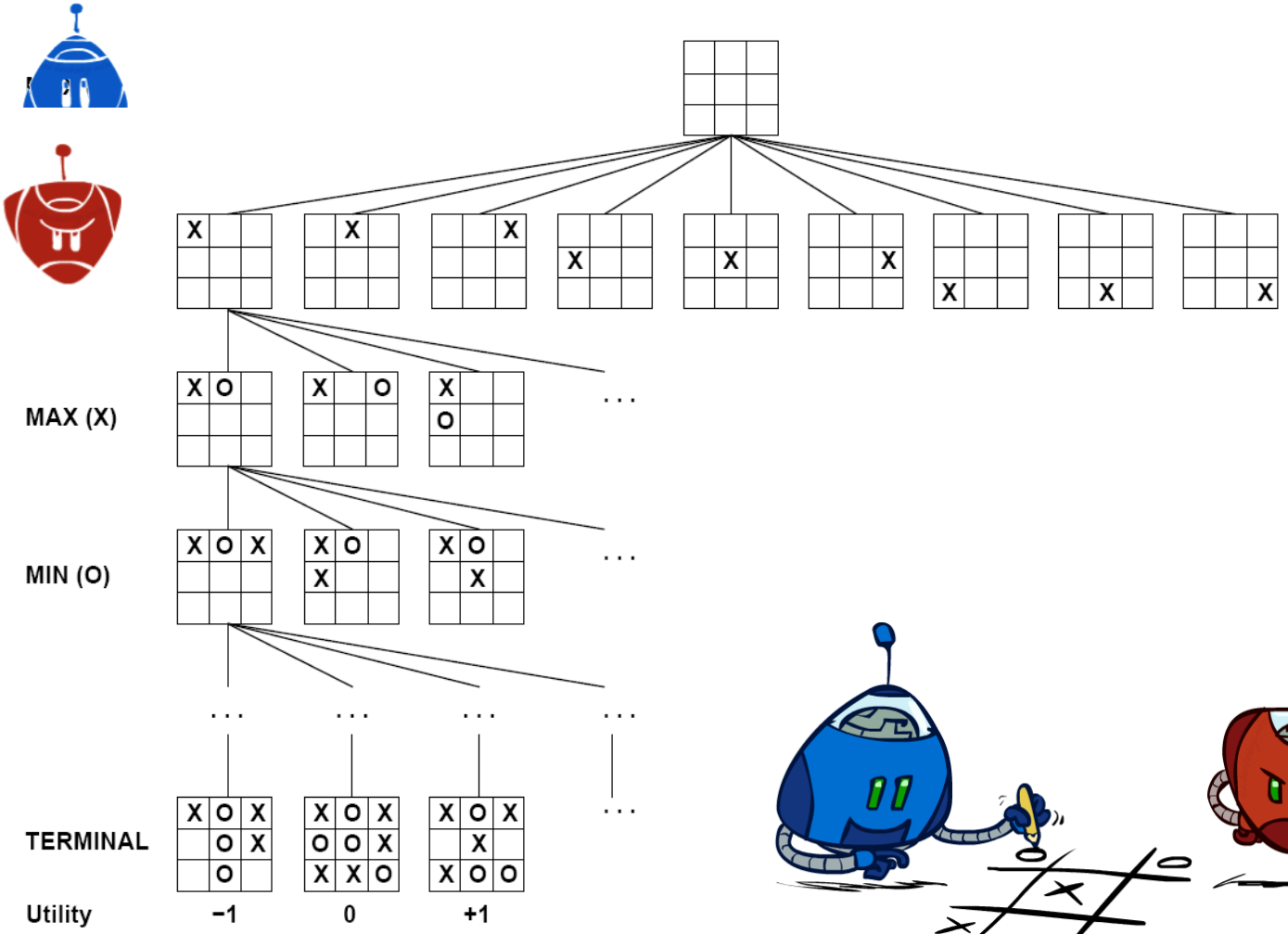


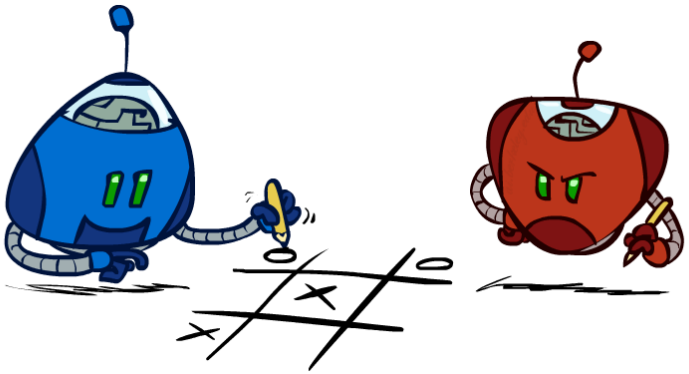
States Under Agent's Control. Max.

Terminal States.

States Under Opponent's Control. Min.

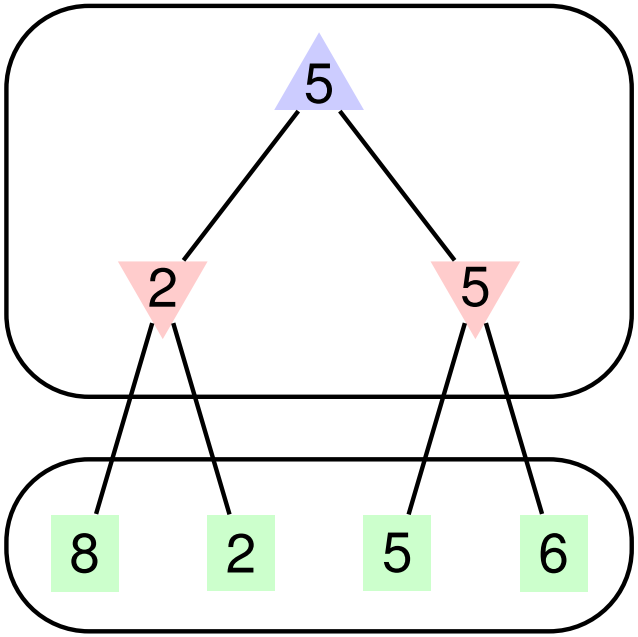
Tic-Tac-Toe Game Tree





Adversarial Search (Minimax)

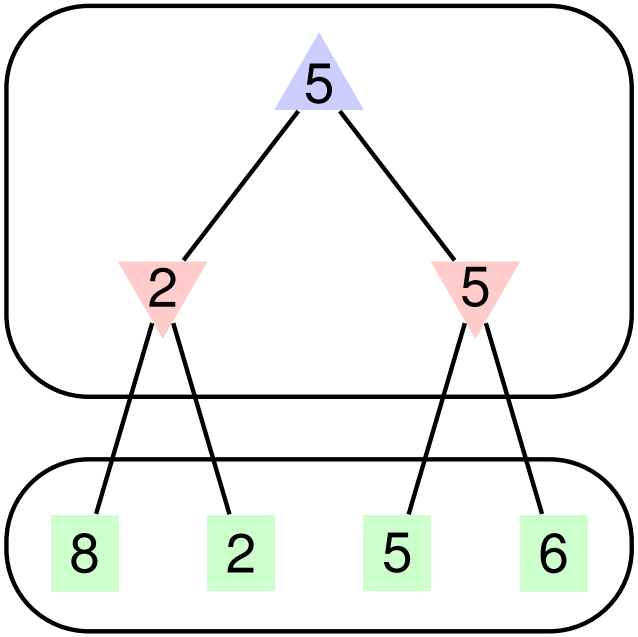
Minimax values:computed recursively.



Terminal values:part of the game.

Adversarial Search (Minimax)

Minimax values:computed recursively.



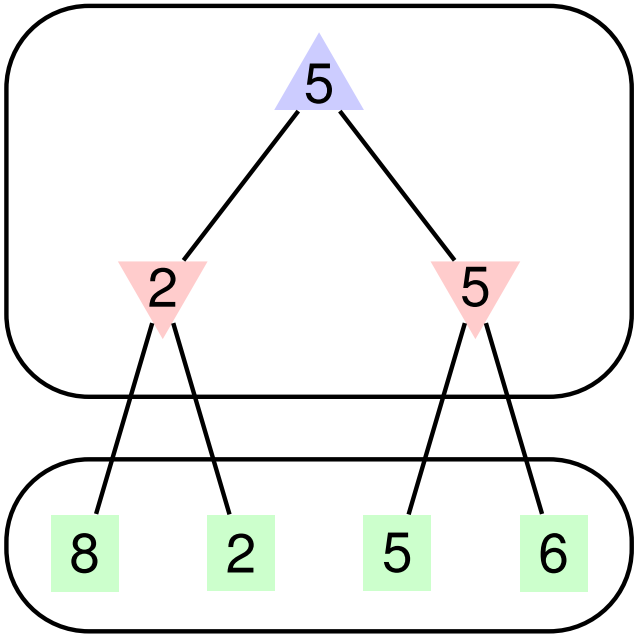
Terminal values:part of the game.

Deterministic, zero-sum games:

- Tic-tac-toe, chess, checkers

Adversarial Search (Minimax)

Minimax values:computed recursively.



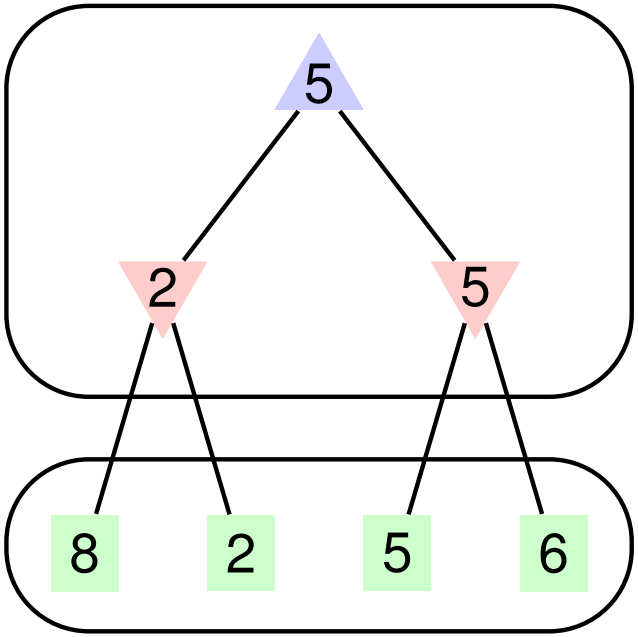
Terminal values:part of the game.

Deterministic, zero-sum games:

- Tic-tac-toe, chess, checkers
- One player maximizes result

Adversarial Search (Minimax)

Minimax values:computed recursively.



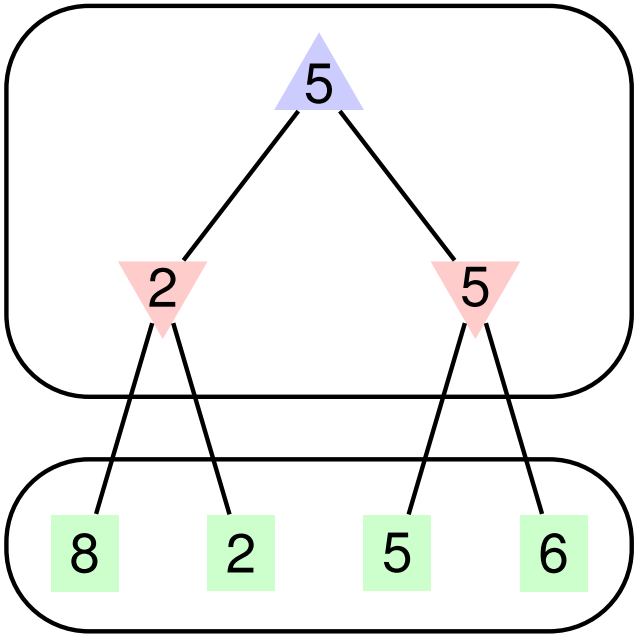
Terminal values:part of the game.

Deterministic, zero-sum games:

- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Adversarial Search (Minimax)

Minimax values:computed recursively.



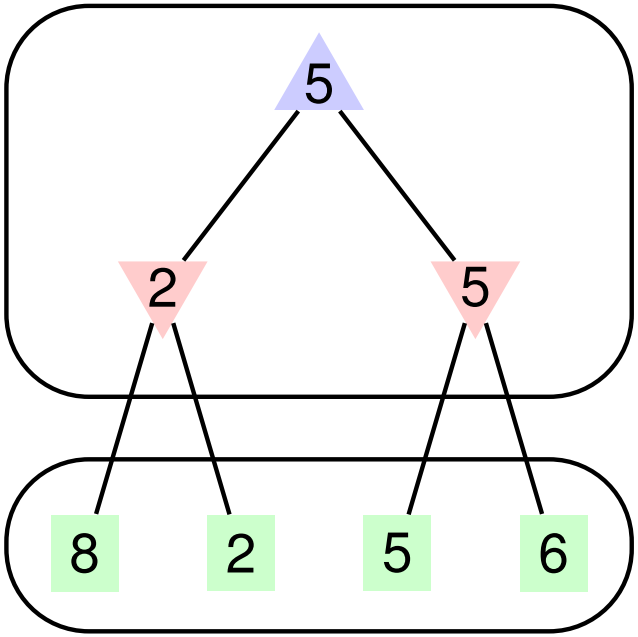
Terminal values:part of the game.

Deterministic, zero-sum games:

- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Adversarial Search (Minimax)

Minimax values:computed recursively.



Terminal values:part of the game.

Deterministic, zero-sum games:

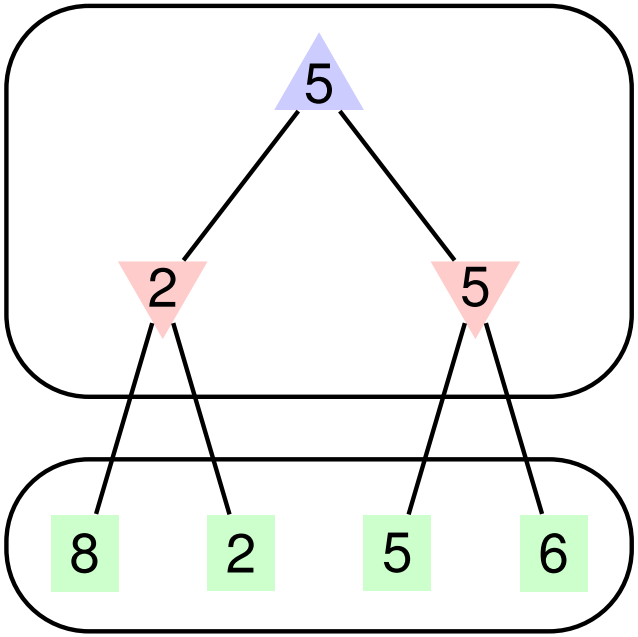
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Minimax search:

- A state-space search tree

Adversarial Search (Minimax)

Minimax values:computed recursively.



Terminal values:part of the game.

Deterministic, zero-sum games:

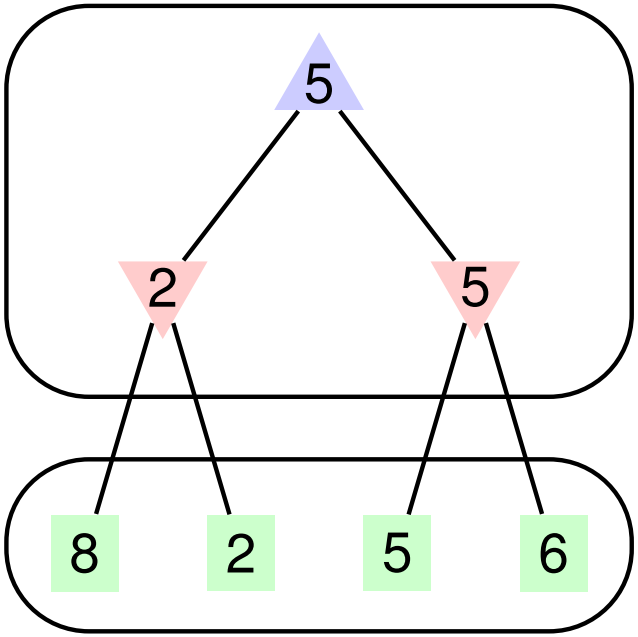
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

Minimax search:

- A state-space search tree
- Players alternate turns

Adversarial Search (Minimax)

Minimax values:computed recursively.



Terminal values:part of the game.

Deterministic, zero-sum games:

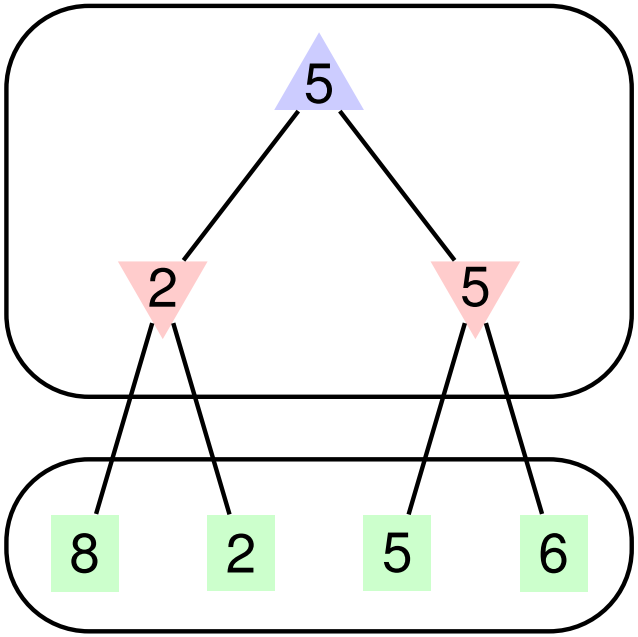
- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

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

Adversarial Search (Minimax)

Minimax values:computed recursively.



Terminal values:part of the game.

Deterministic, zero-sum games:

- Tic-tac-toe, chess, checkers
- One player maximizes result
- The other minimizes result

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

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

Minimax Implementation

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

def min-value(state):

Minimax Implementation

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

def min-value(state):
initialize v = $+\infty$

Minimax Implementation

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

def min-value(state):
initialize v = $+\infty$
for each successor s **of state:**

Minimax Implementation

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

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

Minimax Implementation

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

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

Minimax Implementation

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

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

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

Minimax Implementation

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

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

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

```
def max-value(state):
```

Minimax Implementation

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

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

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

```
def max-value(state):
    initialize v = −∞.
```

Minimax Implementation

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

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

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

```
def max-value(state):
    initialize v = −∞.
    for each successor s of state:
```

Minimax Implementation

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

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

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

```
def max-value(state):
  initialize v = −∞.
  for each successor s of state:
    v = max(v, min-value(s))
```

Minimax Implementation

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

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

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

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

Minimax Implementation

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

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

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

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

Minimax Implementation

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

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

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

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

```
def value(state):
```

Minimax Implementation

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

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

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

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

```
def value(state):
  if the state is terminal: return the state's utility
```

Minimax Implementation

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

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

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

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

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

Minimax Implementation

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

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

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

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

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

Minimax Implementation

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

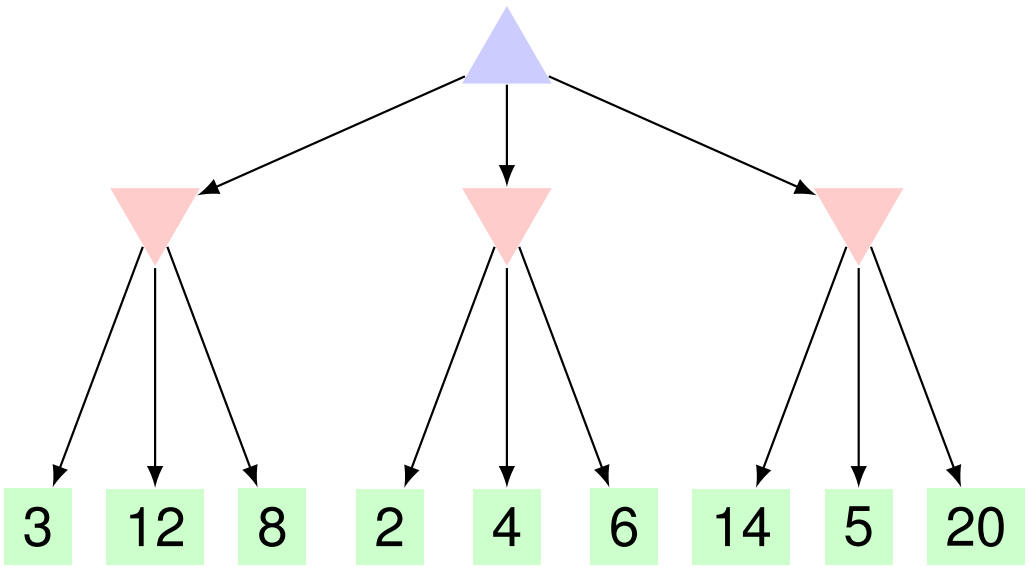
```
def min-value(state):
  initialize v = +∞
  for each successor s of state:
    v = min(v, max-value(s))
  return v
```

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

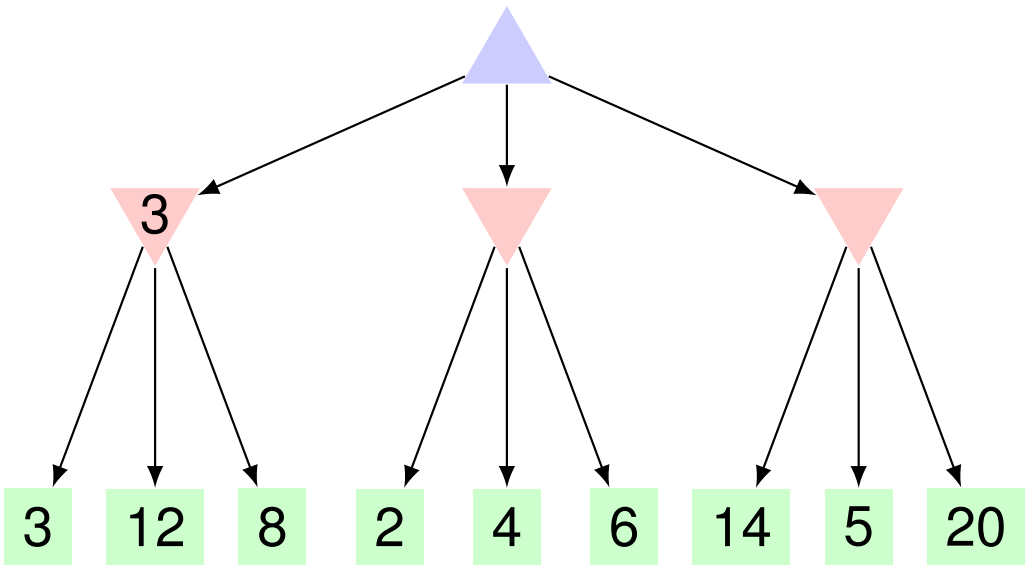
```
def max-value(state):
  initialize v = −∞.
  for each successor s of state:
    v = max(v, min-value(s))
  return v
```

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

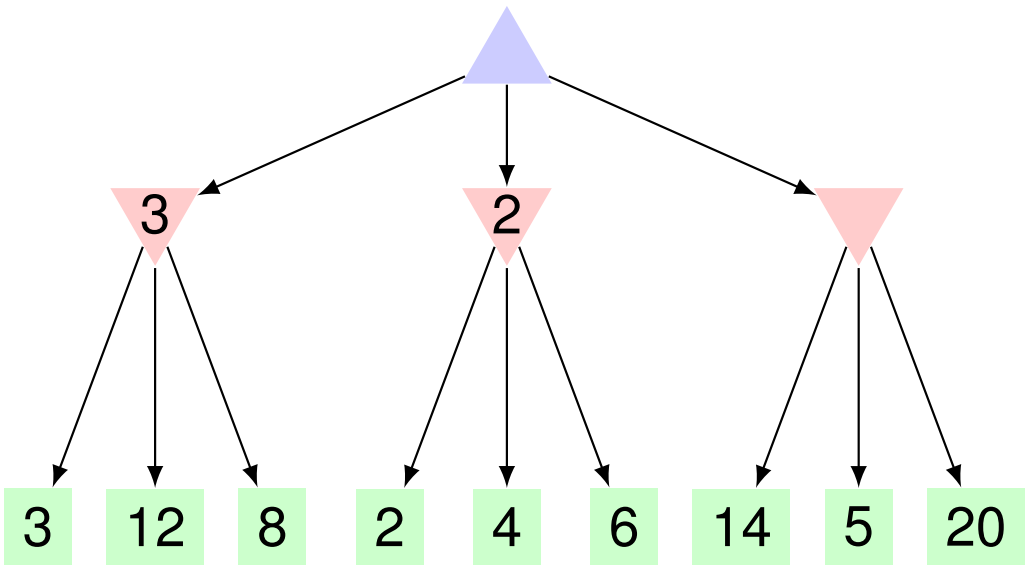
Minimax Example



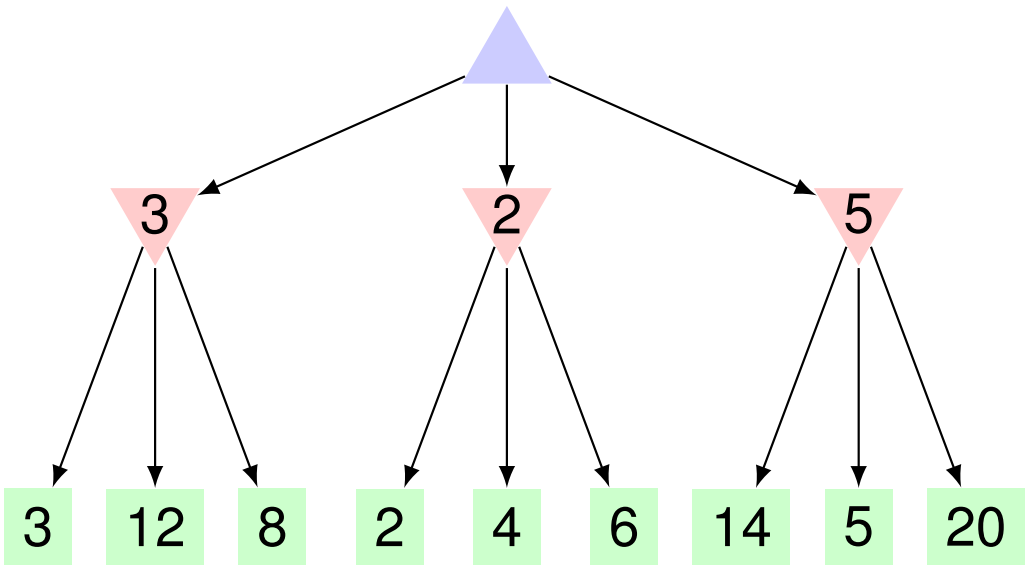
Minimax Example



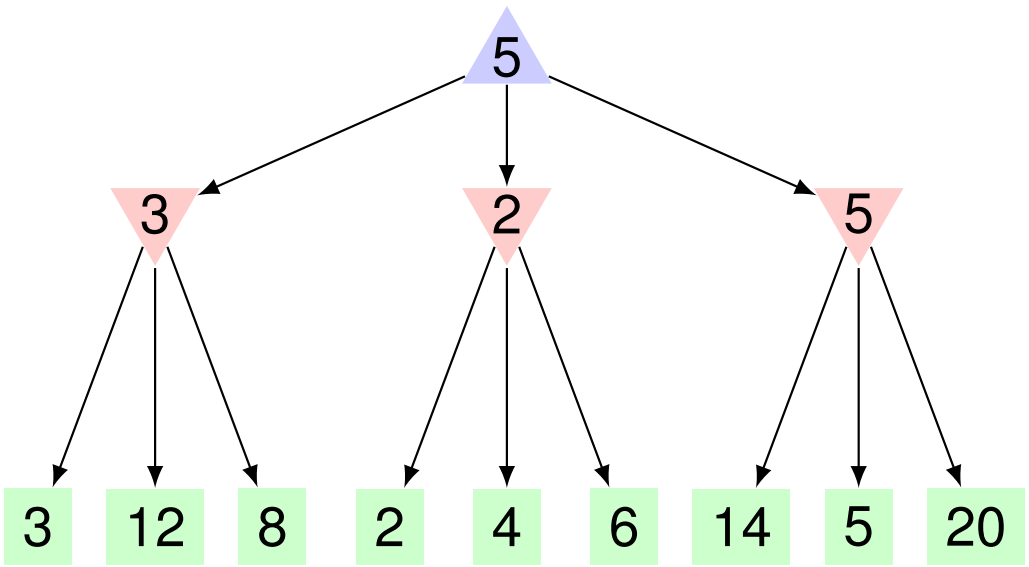
Minimax Example



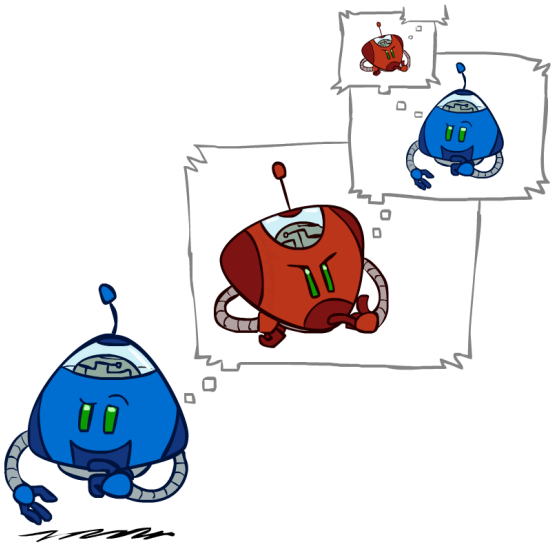
Minimax Example



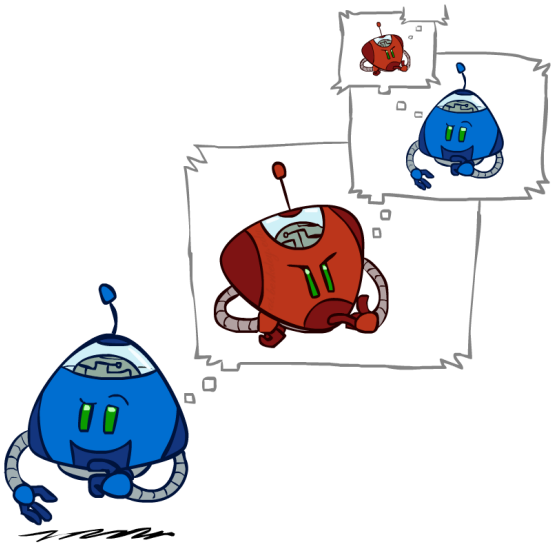
Minimax Example



Minimax Efficiency



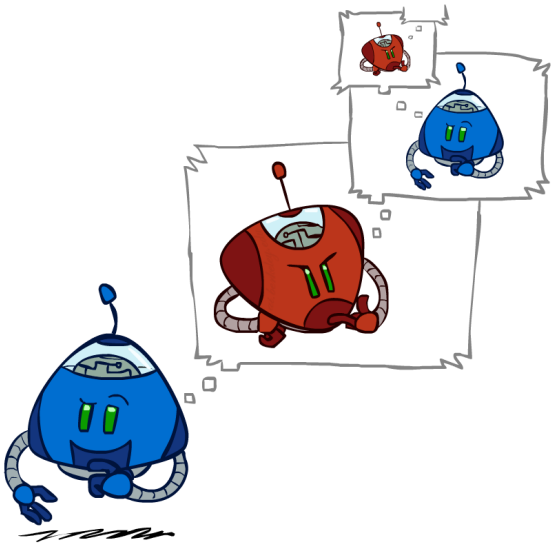
Minimax Efficiency



How efficient is minimax?

- Just like (exhaustive) DFS

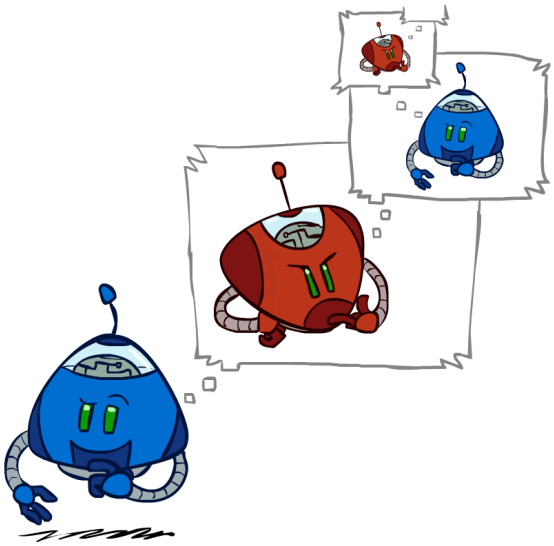
Minimax Efficiency



How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$

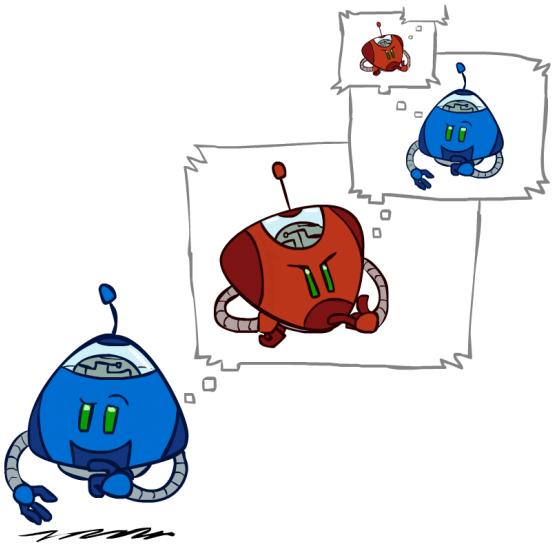
Minimax Efficiency



How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

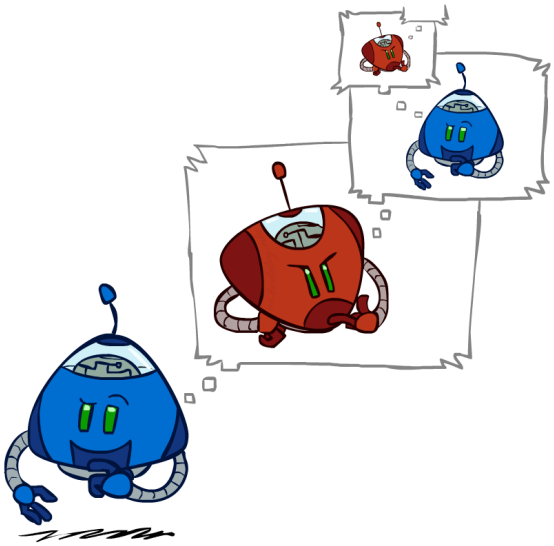
Minimax Efficiency



How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

Minimax Efficiency



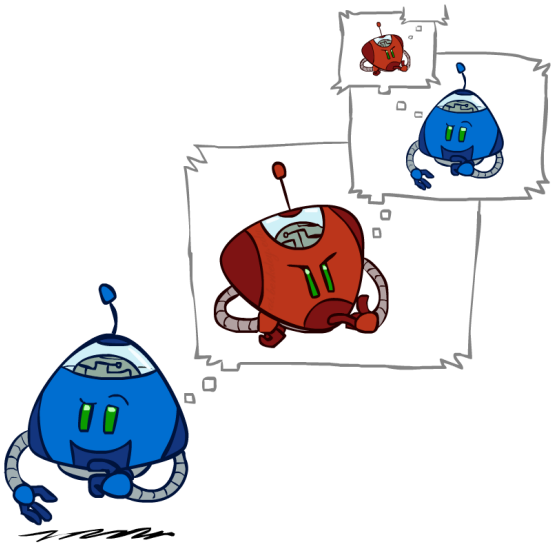
How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

Example: For chess, $b \approx 35$, $m \approx 100$

- Exact solution is completely infeasible.

Minimax Efficiency



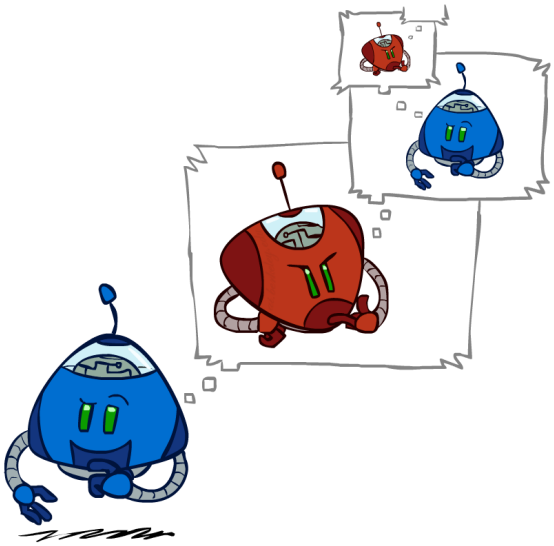
How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

Example: For chess, $b \approx 35$, $m \approx 100$

- Exact solution is completely infeasible.
- But, do we need to explore the whole tree?

Minimax Efficiency



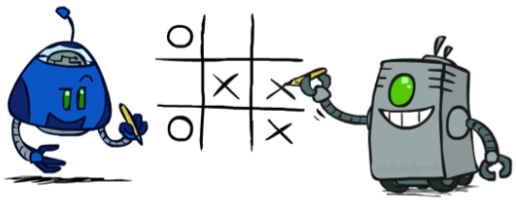
How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$

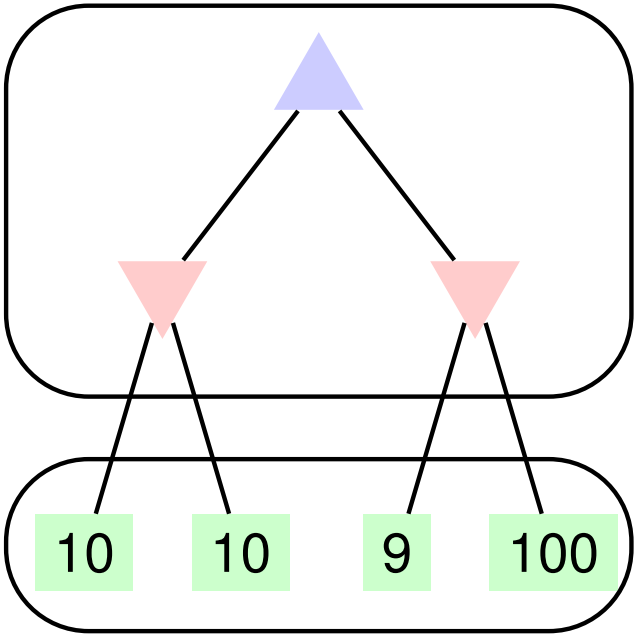
Example: For chess, $b \approx 35$, $m \approx 100$

- Exact solution is completely infeasible.
- But, do we need to explore the whole tree?

Minimax Properties

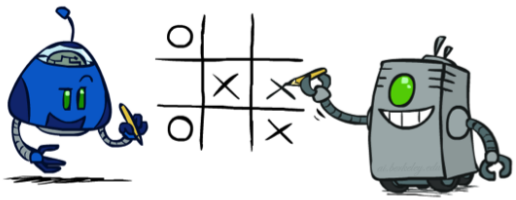


Minimax values:computed recursively.

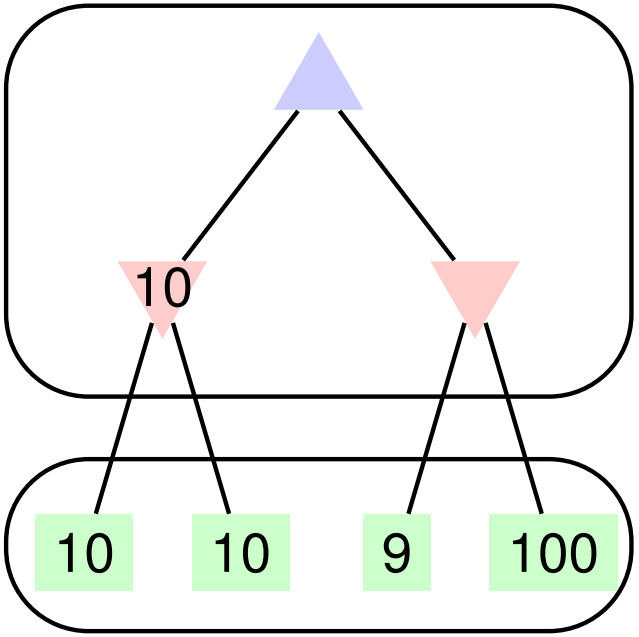


Terminal values:part of the game.

Minimax Properties

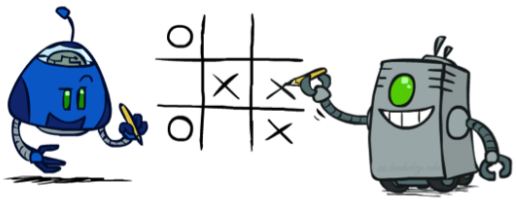


Minimax values:computed recursively.

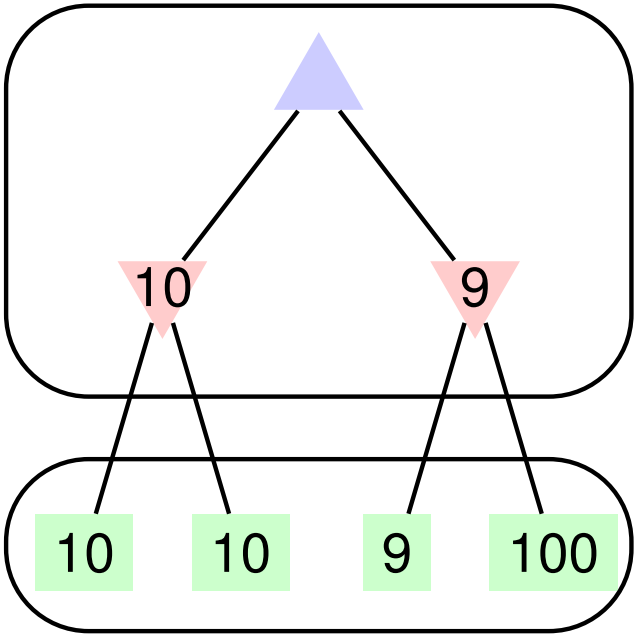


Terminal values:part of the game.

Minimax Properties

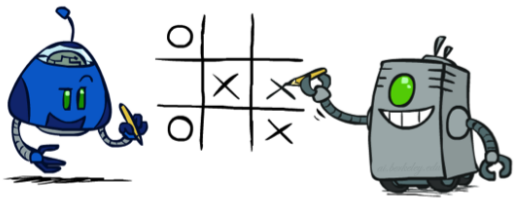


Minimax values:computed recursively.

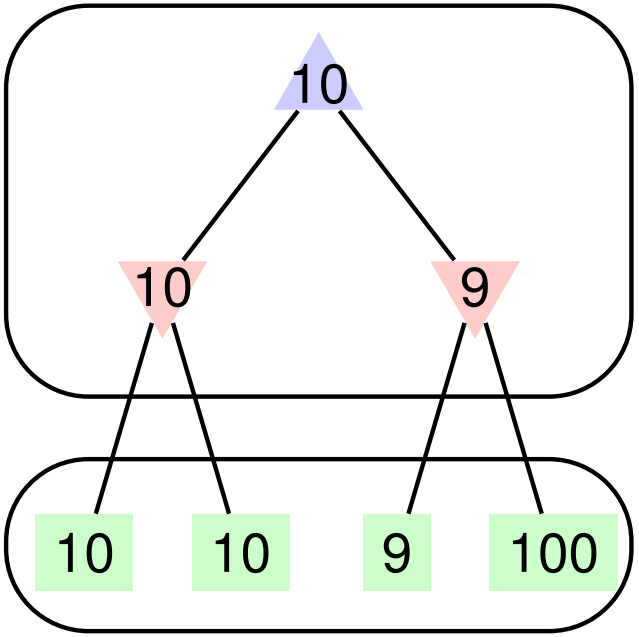


Terminal values:part of the game.

Minimax Properties

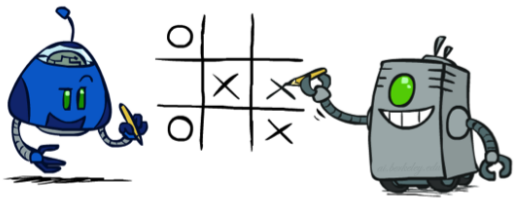


Minimax values:computed recursively.

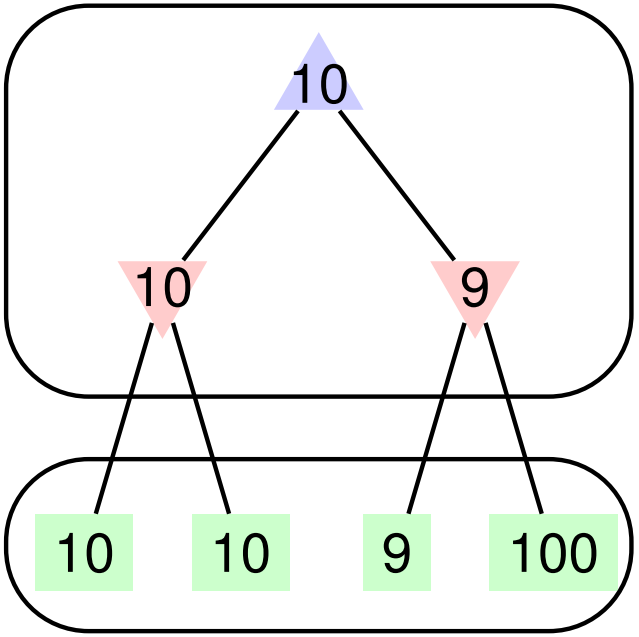


Terminal values:part of the game.

Minimax Properties



Minimax values:computed recursively.

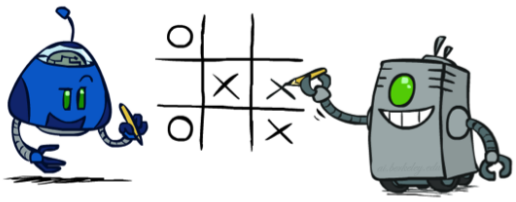


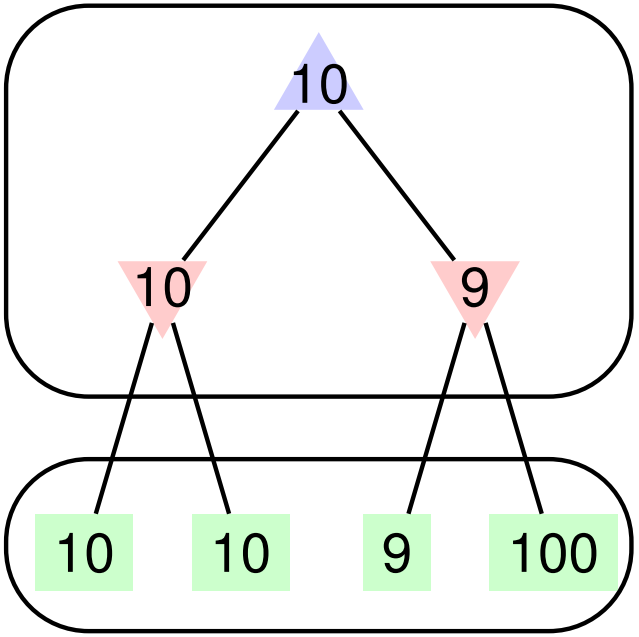
Terminal values:part of the game.

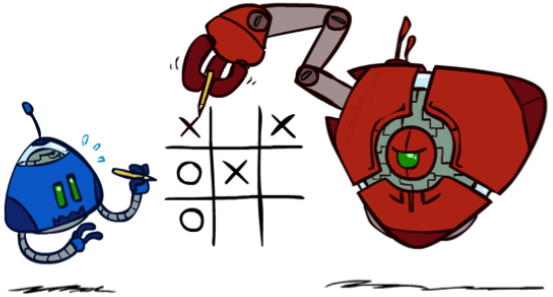
Optimal against a perfect player.

Minimax Properties

Minimax values:computed recursively.





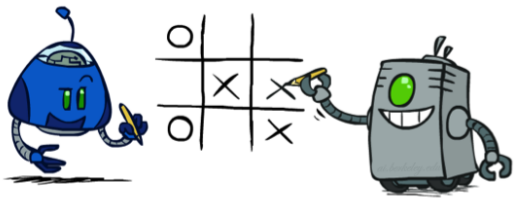


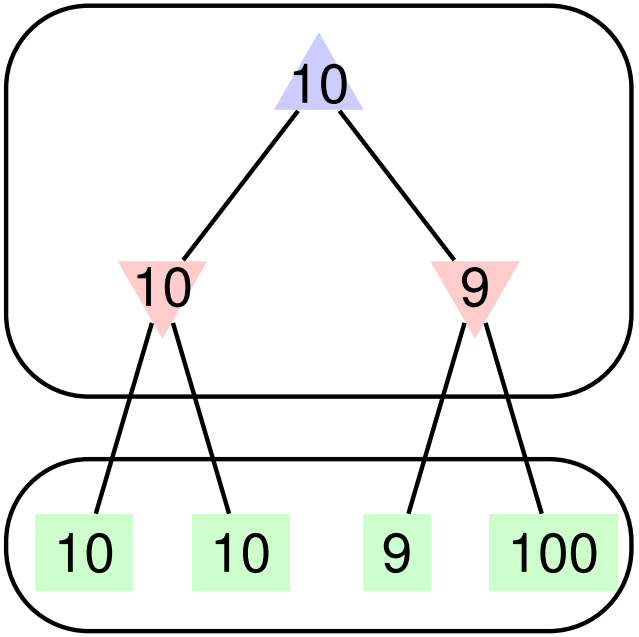
Terminal values:part of the game.

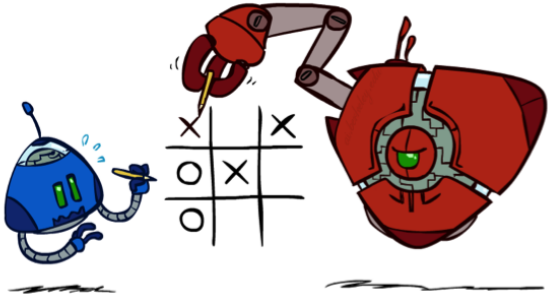
Optimal against a perfect player. Otherwise?

Minimax Properties

Minimax values:computed recursively.





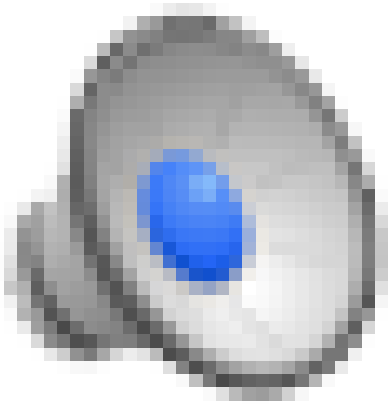


Terminal values:part of the game.

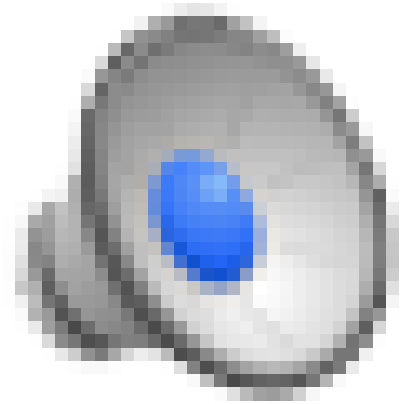
Optimal against a perfect player. Otherwise?

$$Demo : minvsexp(L6D2, L6D3)$$

Video of Demo Min vs. Exp (Min)



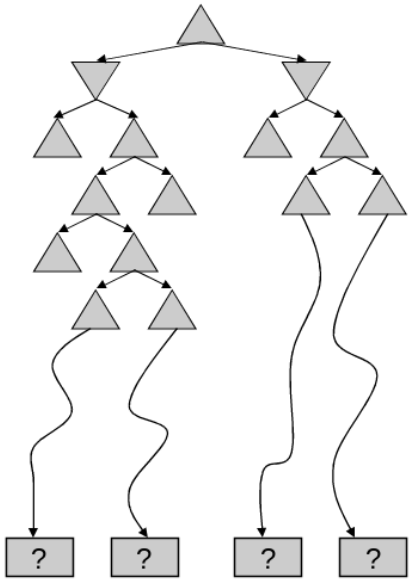
Video of Demo Min vs. Exp (Exp)



Resource Limits

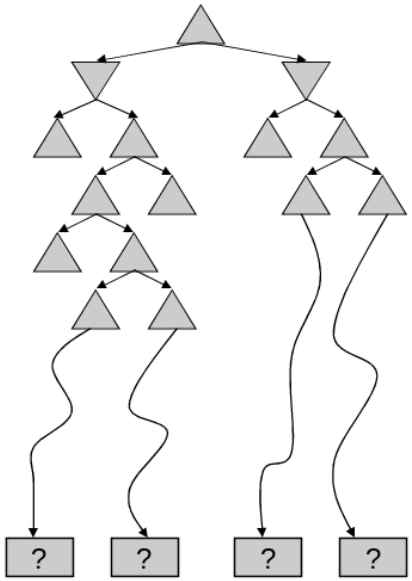
Resource Limits

Problem: In realistic games, cannot search to leaves!

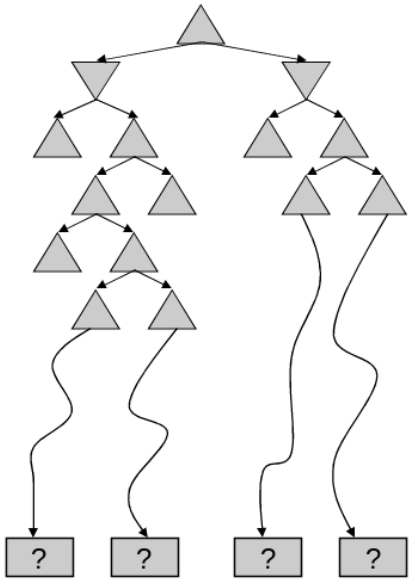


Resource Limits

Problem: In realistic games, cannot search to leaves!



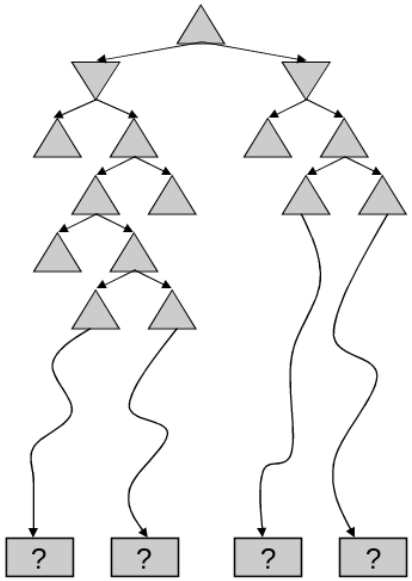
Resource Limits



Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

Resource Limits

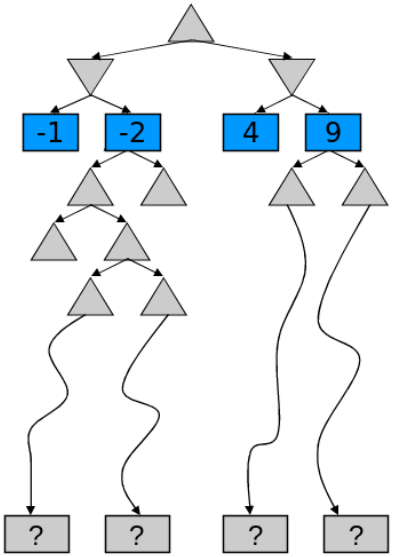


Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

- Instead, search to limited depth in the tree.

Resource Limits

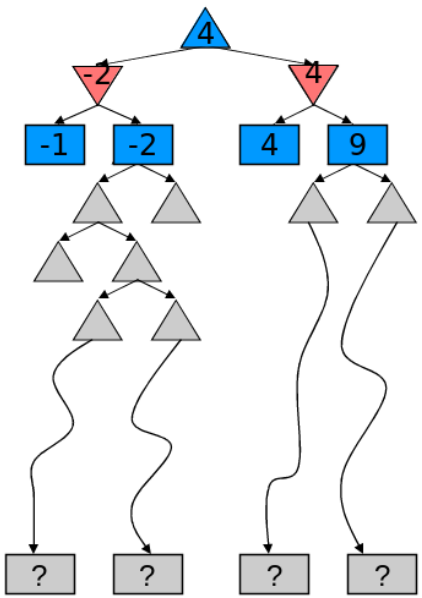


Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Resource Limits

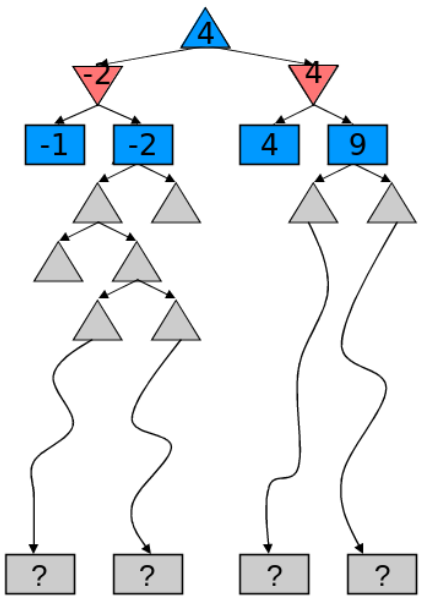


Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Resource Limits



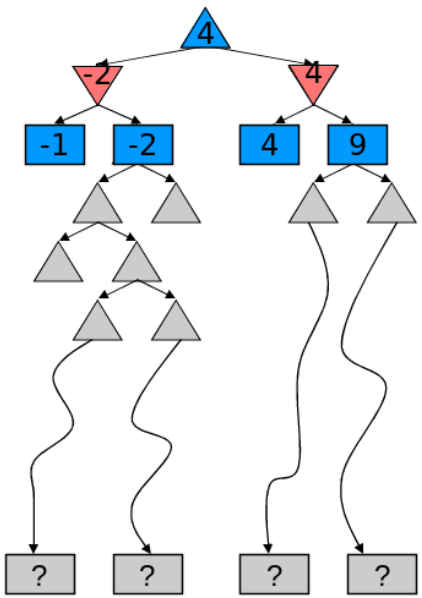
Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

Resource Limits



Problem: In realistic games, cannot search to leaves!

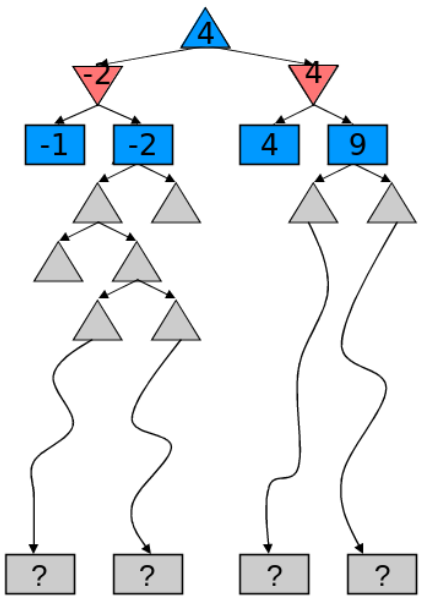
Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

- Suppose we have 100 seconds,
  can explore 10K nodes / sec

Resource Limits



Problem: In realistic games, cannot search to leaves!

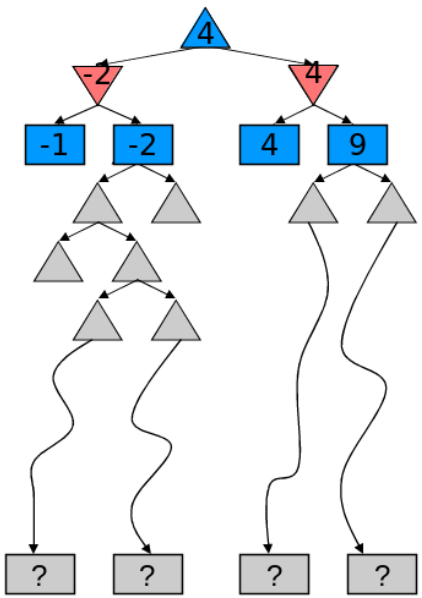
Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

- Suppose we have 100 seconds,
  can explore 10K nodes / sec
- So can check 1M nodes per move.

Resource Limits



Problem: In realistic games, cannot search to leaves!

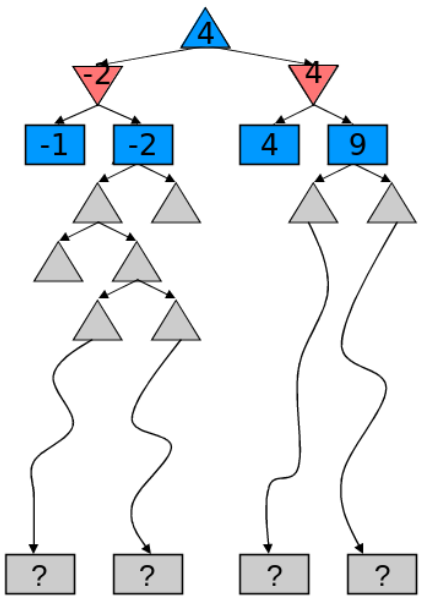
Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

- Suppose we have 100 seconds,
  can explore 10K nodes / sec
- So can check 1M nodes per move.
- $\downarrow, \uparrow$- reaches about depth 8
  – decent chess program

Resource Limits



Problem: In realistic games, cannot search to leaves!

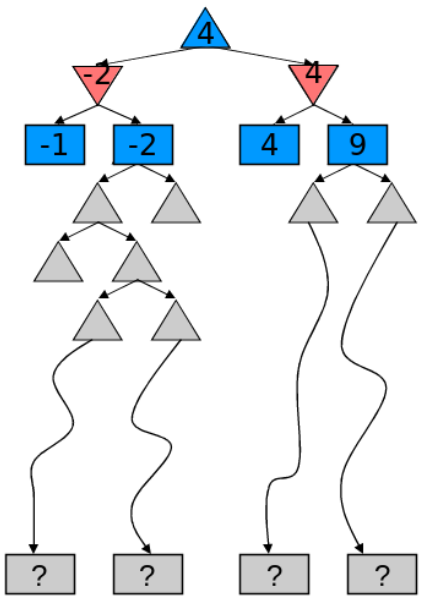
Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

- Suppose we have 100 seconds,
  can explore 10K nodes / sec
- So can check 1M nodes per move.
- $\downarrow, \uparrow$- reaches about depth 8
  – decent chess program

Resource Limits



Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

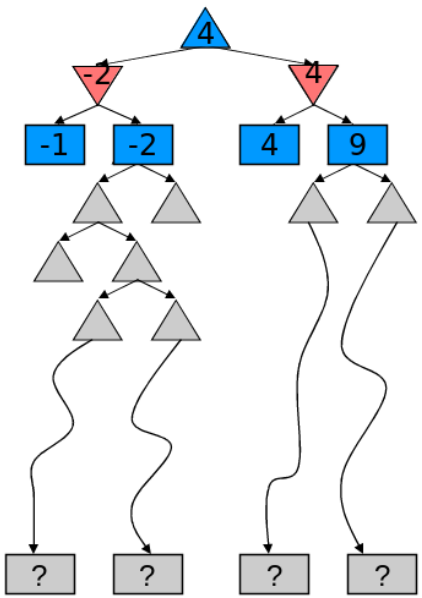
- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

- Suppose we have 100 seconds,
  can explore 10K nodes / sec
- So can check 1M nodes per move.
- $\downarrow, \uparrow$- reaches about depth 8
  – decent chess program

Guarantee of optimal play is gone.

Resource Limits



Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

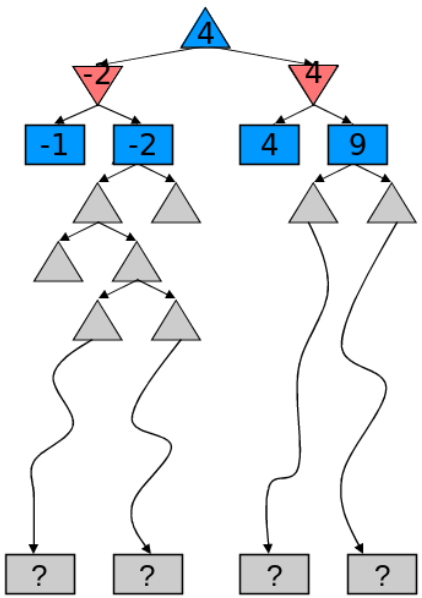
- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

- Suppose we have 100 seconds,
  can explore 10K nodes / sec
- So can check 1M nodes per move.
- $\downarrow, \uparrow$- reaches about depth 8
  – decent chess program

Guarantee of optimal play is gone.
More plies makes a BIG difference

Resource Limits



Problem: In realistic games, cannot search to leaves!

Solution: Depth-limited search

- Instead, search to limited depth in the tree.
- Use an evaluation function for non-terminal positions

Example:

- Suppose we have 100 seconds,
  can explore 10K nodes / sec
- So can check 1M nodes per move.
- $\downarrow, \uparrow$- reaches about depth 8
  – decent chess program

Guarantee of optimal play is gone.
More plies makes a BIG difference
Use iterative deepening for an anytime algorithm

Depth Matters

Depth Matters

Evaluation functions are always imperfect

Depth Matters

Evaluation functions are always imperfect

The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters

Depth Matters

Evaluation functions are always imperfect

The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters

An important example of the tradeoff between complexity of features and complexity of computation

Depth Matters

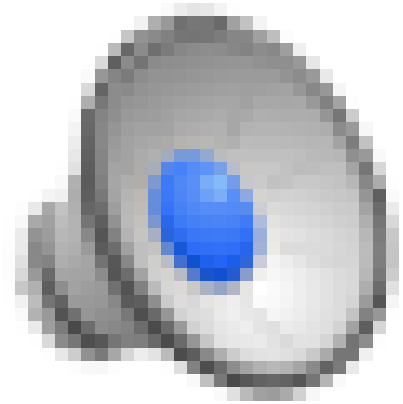
Evaluation functions are always imperfect

The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters

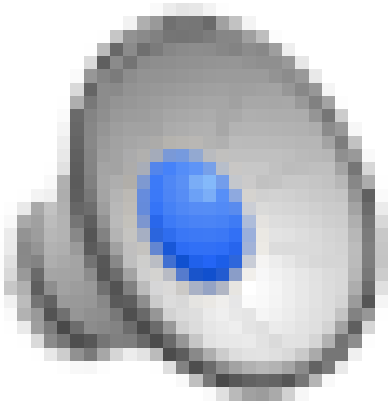
An important example of the tradeoff between complexity of features and complexity of computation

$$Demo : depthlimited(L6D4, L6D5)$$

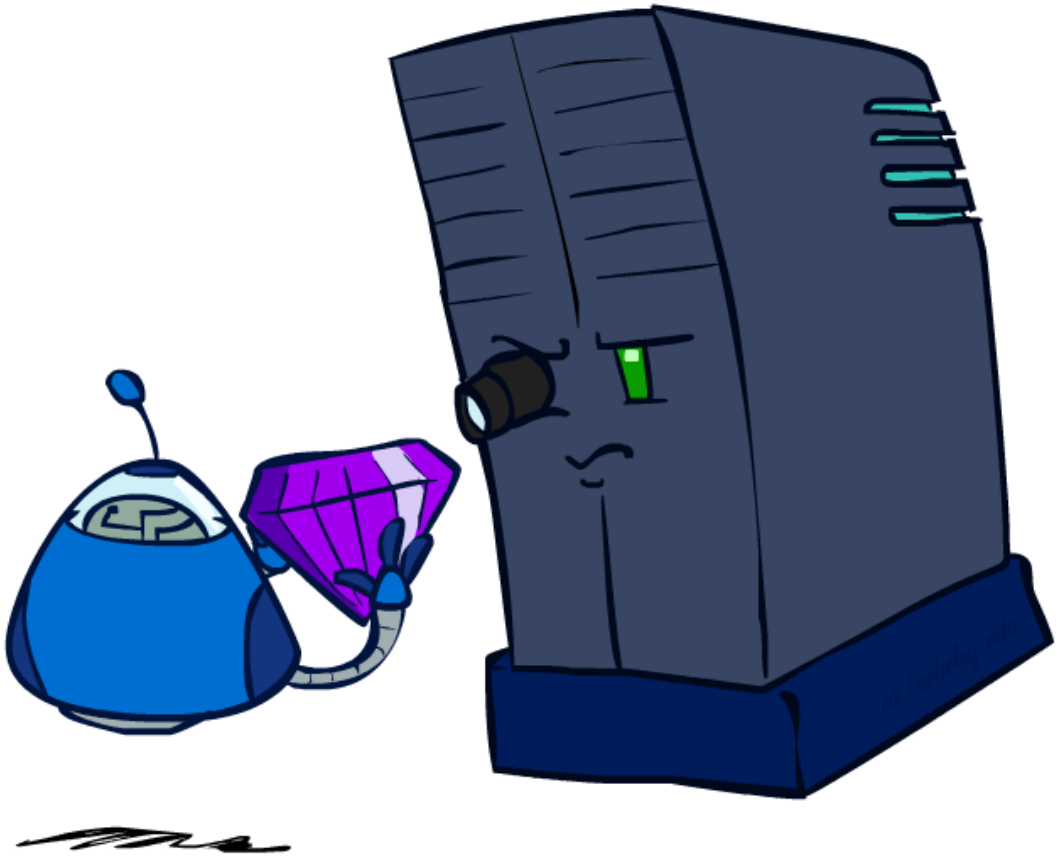
Video of Demo Limited Depth (2)



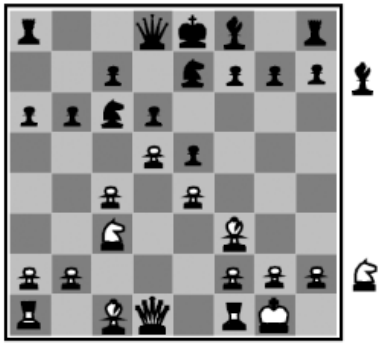
Video of Demo Limited Depth (10)



Evaluation Functions

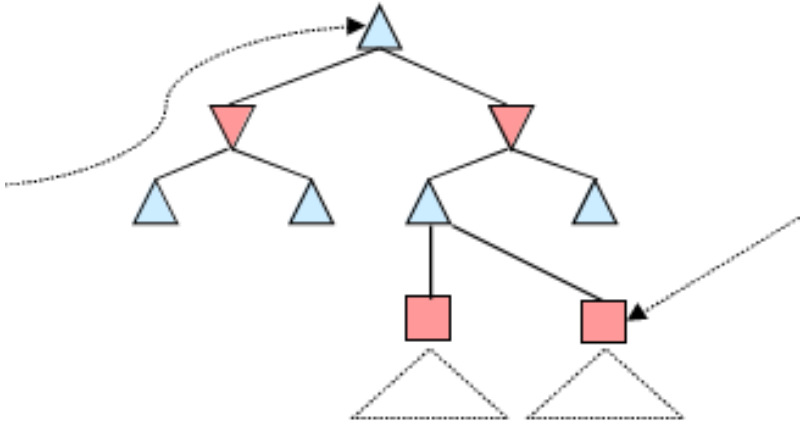


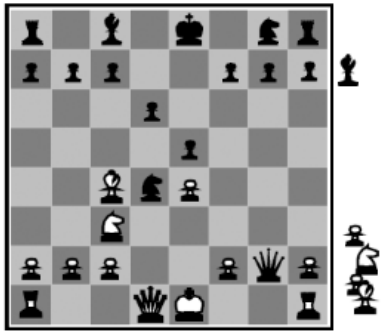
Evaluation Functions



Black to move

White slightly better

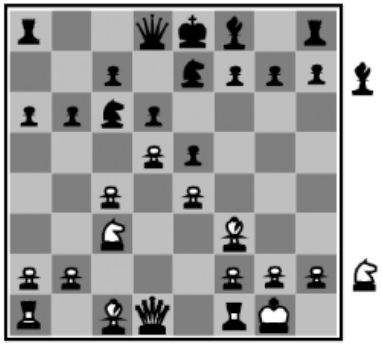




White to move

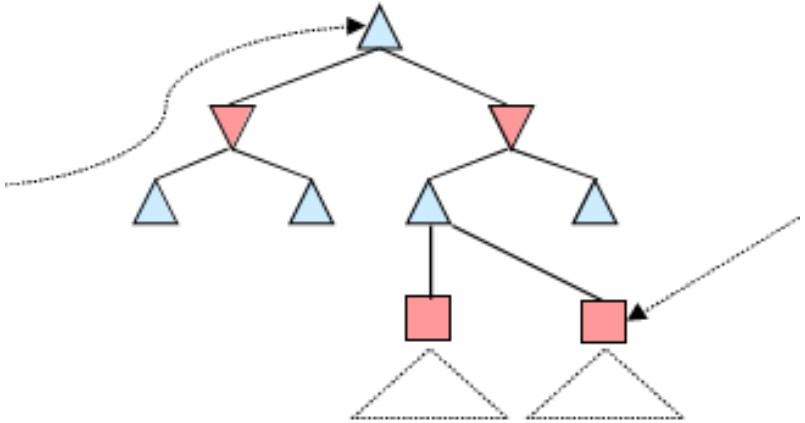
Black winning

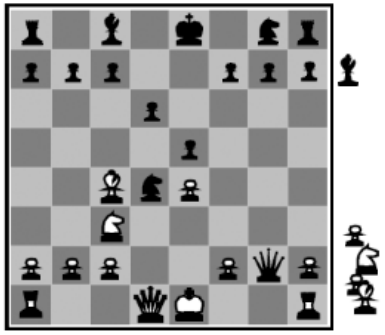
Evaluation Functions



Black to move

White slightly better



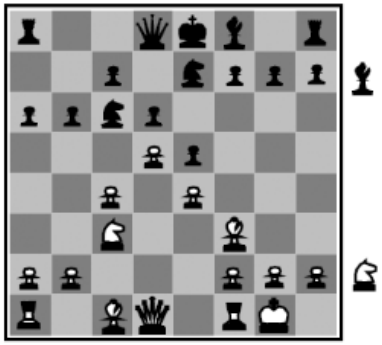


White to move

Black winning

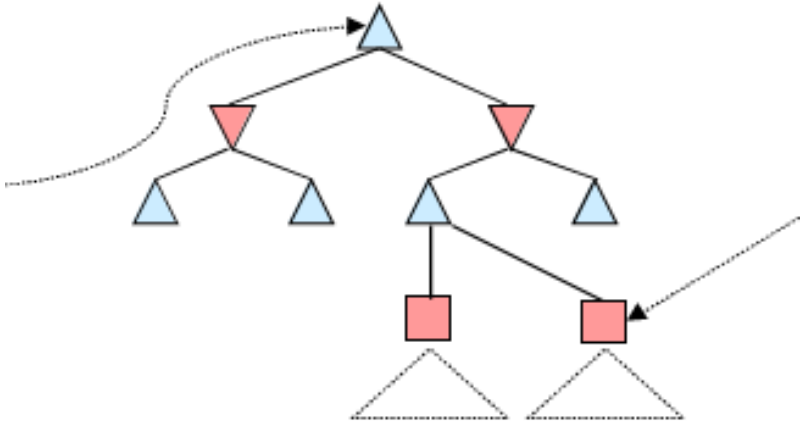
Evaluation functions score non-terminals in depth-limited search

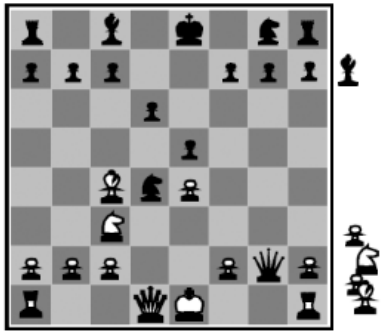
Evaluation Functions



Black to move

White slightly better





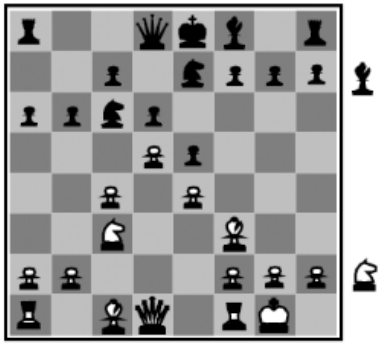
White to move

Black winning

Evaluation functions score non-terminals in depth-limited search

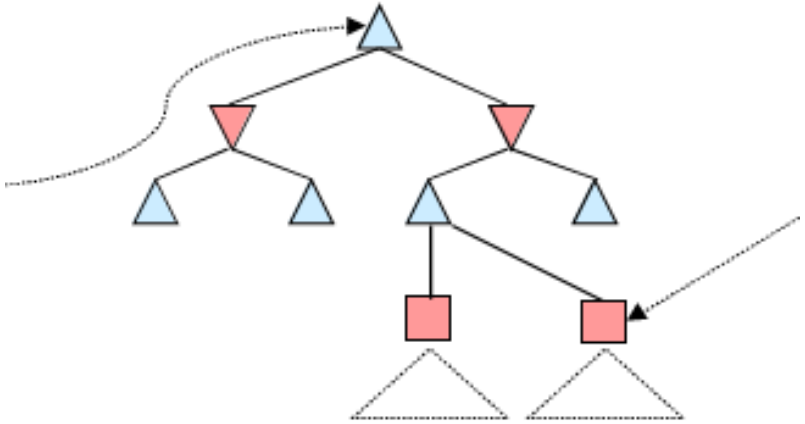
Ideal function: returns the actual minimax value of the position

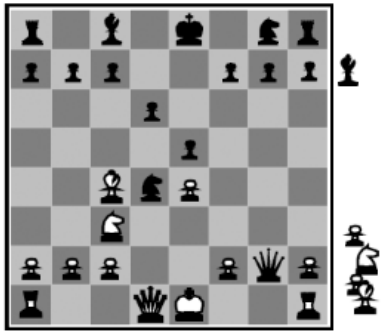
Evaluation Functions



Black to move

White slightly better





White to move

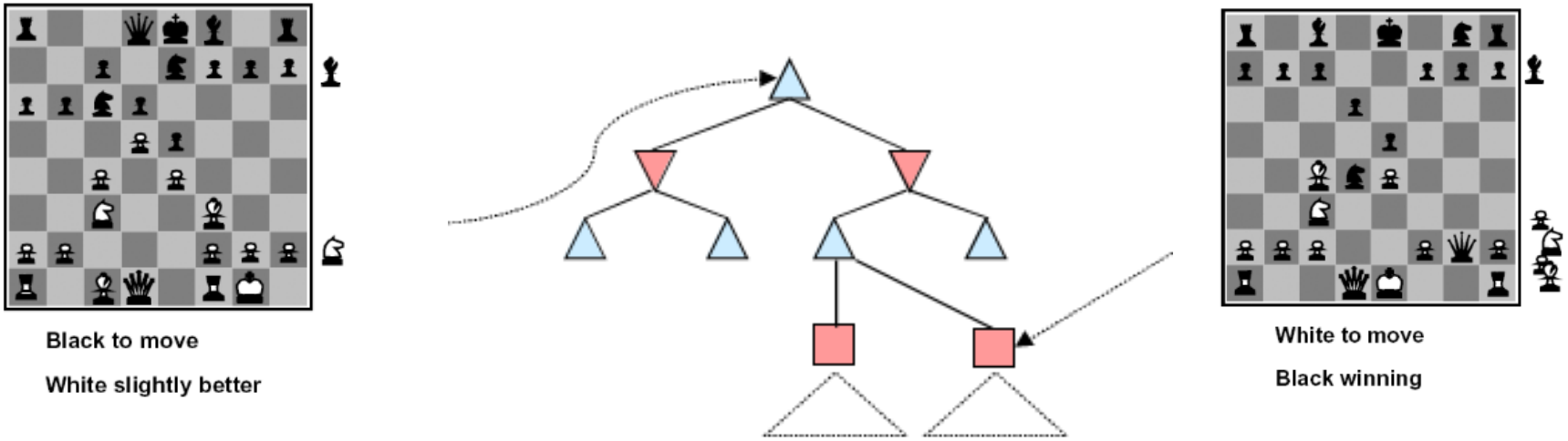
Black winning

Evaluation functions score non-terminals in depth-limited search

Ideal function: returns the actual minimax value of the position

In practice: typically weighted linear sum of features:

Evaluation Functions



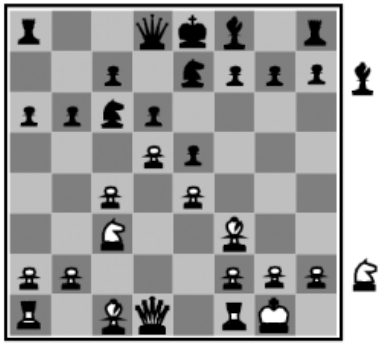
Evaluation functions score non-terminals in depth-limited search

Ideal function: returns the actual minimax value of the position

In practice: typically weighted linear sum of features:

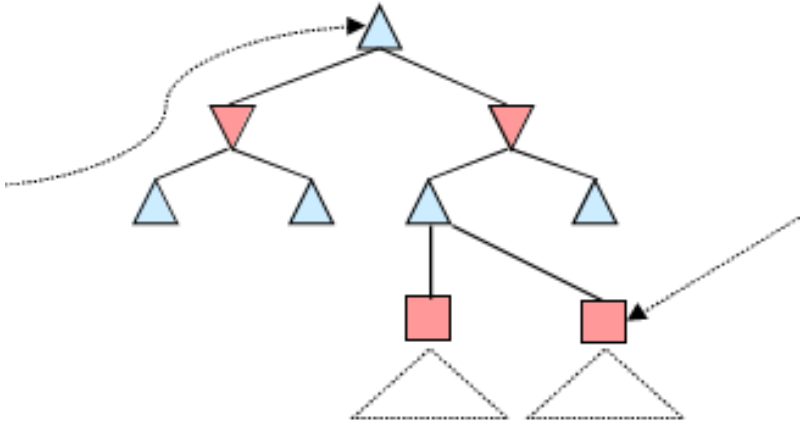
$$w_1 f_1(s) + w_2 f_2(s) + \cdots w_n f_n(s).$$

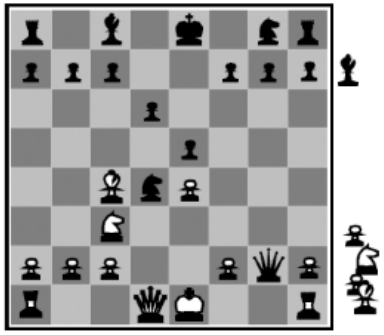
Evaluation Functions



Black to move

White slightly better





White to move

Black winning

Evaluation functions score non-terminals in depth-limited search

Ideal function: returns the actual minimax value of the position

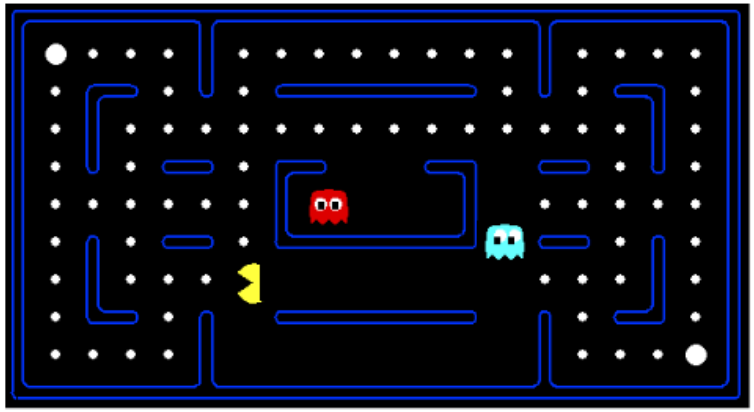
In practice: typically weighted linear sum of features:

$$w_1 f_1(s) + w_2 f_2(s) + \cdots w_n f_n(s).$$

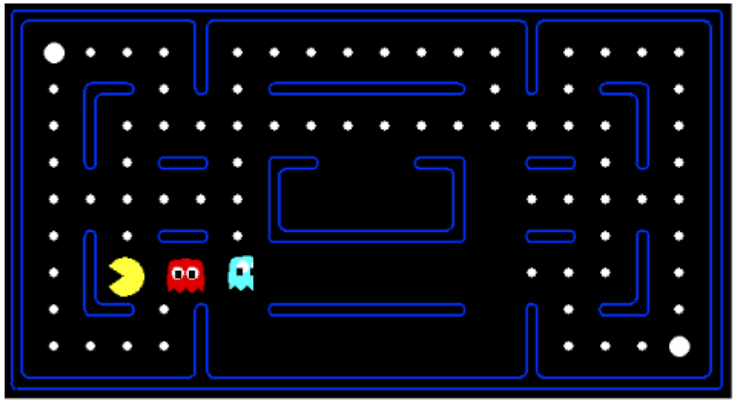
Example: $f1(s)$ = (num white queens – num black queens), etc.

Evaluation for Pacman

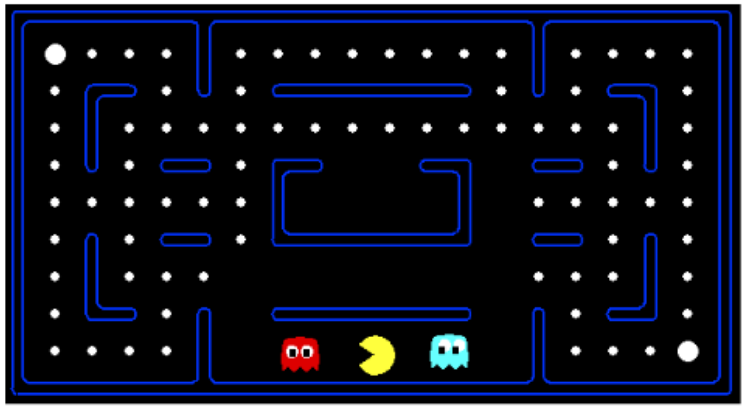
Evaluation for Pacman



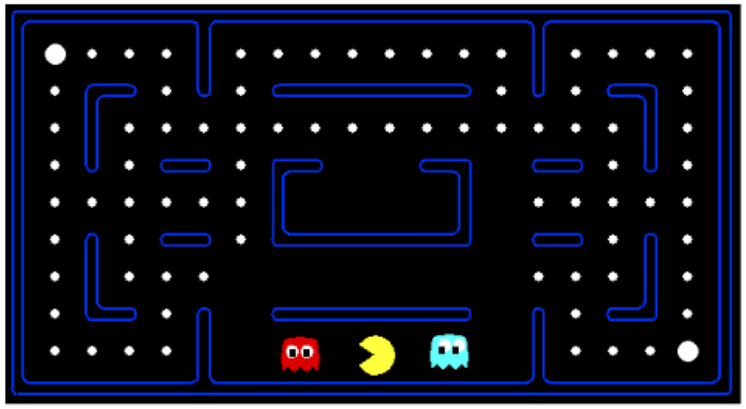
Evaluation for Pacman



Evaluation for Pacman

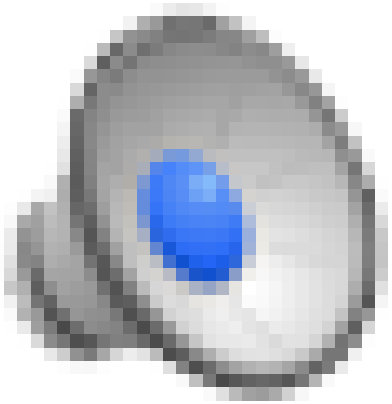


Evaluation for Pacman



$Demo: thrashingd = 2, thrashingd = 2(fixedevaluationfunction), smartghostsc$

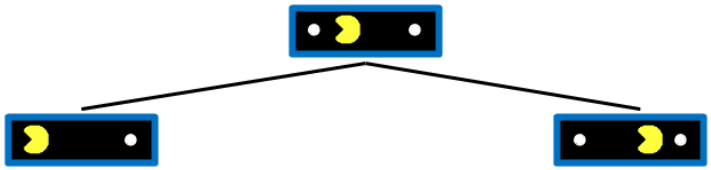
Video of Demo Thrashing (d=2)



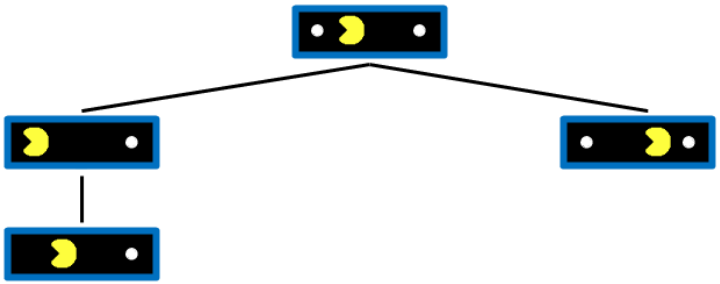
Why Pacman Starves

Why Pacman Starves

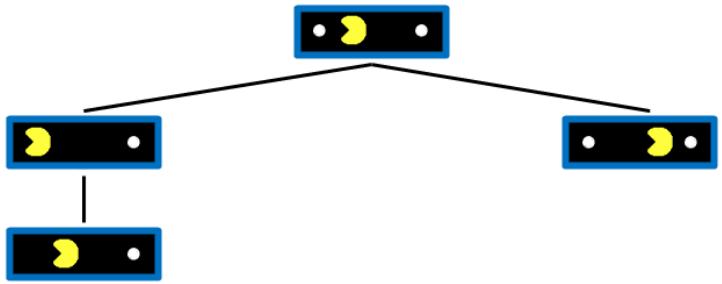
Why Pacman Starves



Why Pacman Starves



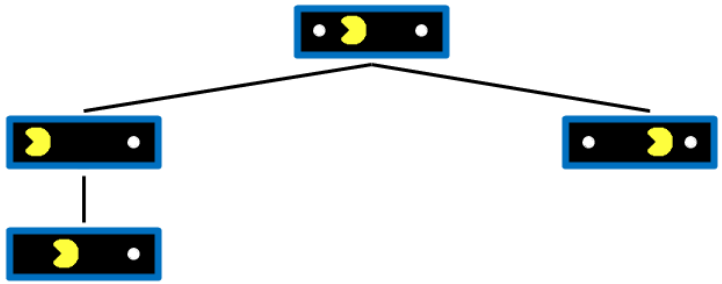
Why Pacman Starves



A danger of replanning agents!

- He knows his score will go up by eating the dot now (west, east)

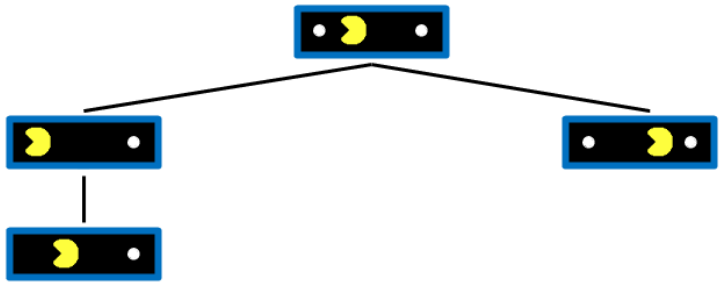
Why Pacman Starves



A danger of replanning agents!

- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)

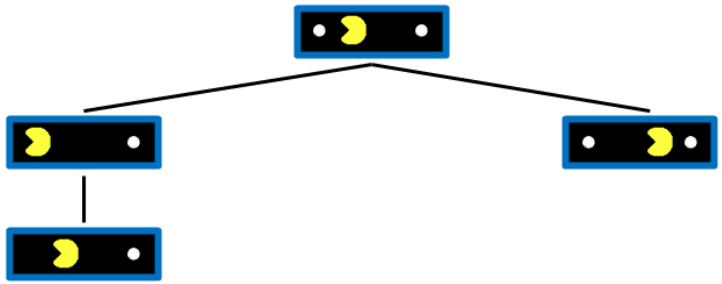
Why Pacman Starves



A danger of replanning agents!

- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)

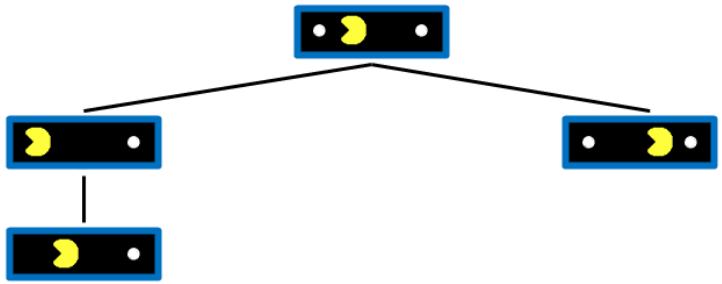
Why Pacman Starves



A danger of replanning agents!

- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

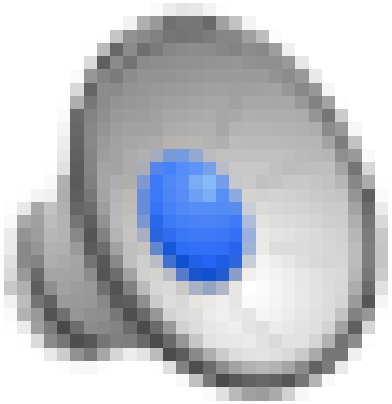
Why Pacman Starves



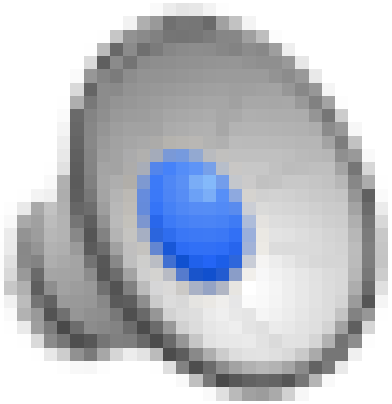
A danger of replanning agents!

- He knows his score will go up by eating the dot now (west, east)
- He knows his score will go up just as much by eating the dot later (east, west)
- There are no point-scoring opportunities after eating the dot (within the horizon, two here)
- Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

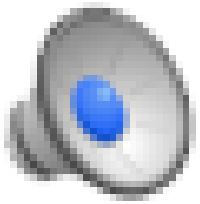
Video of Demo Thrashing – Fixed (d=2)



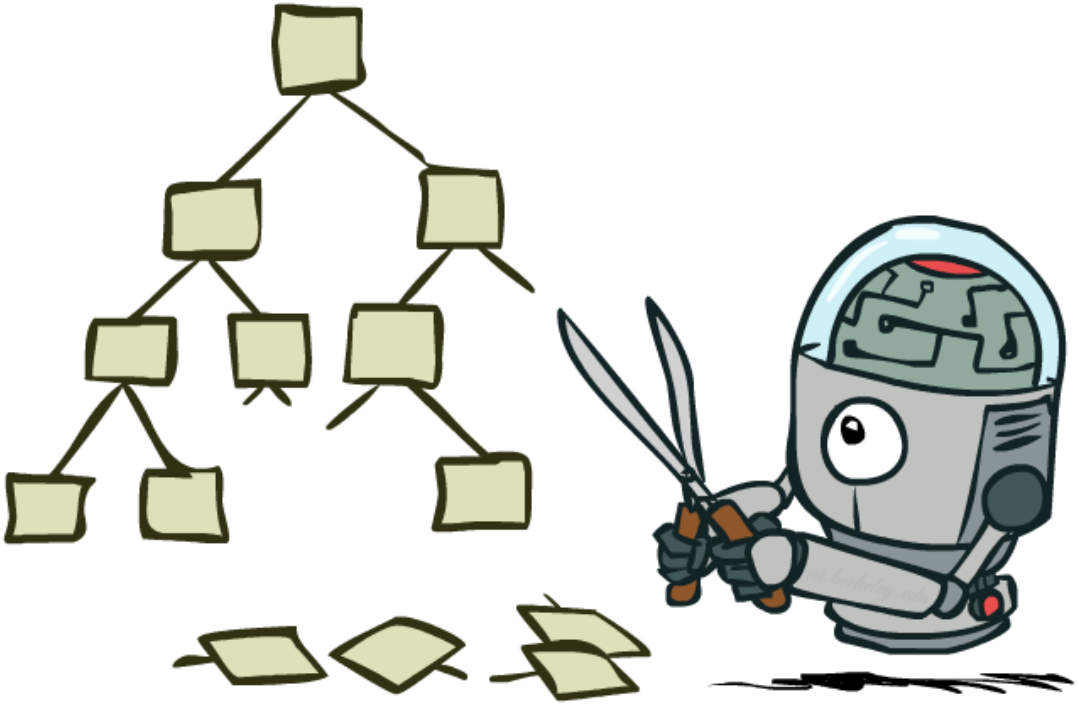
Video of Demo Smart Ghosts (Coordination)



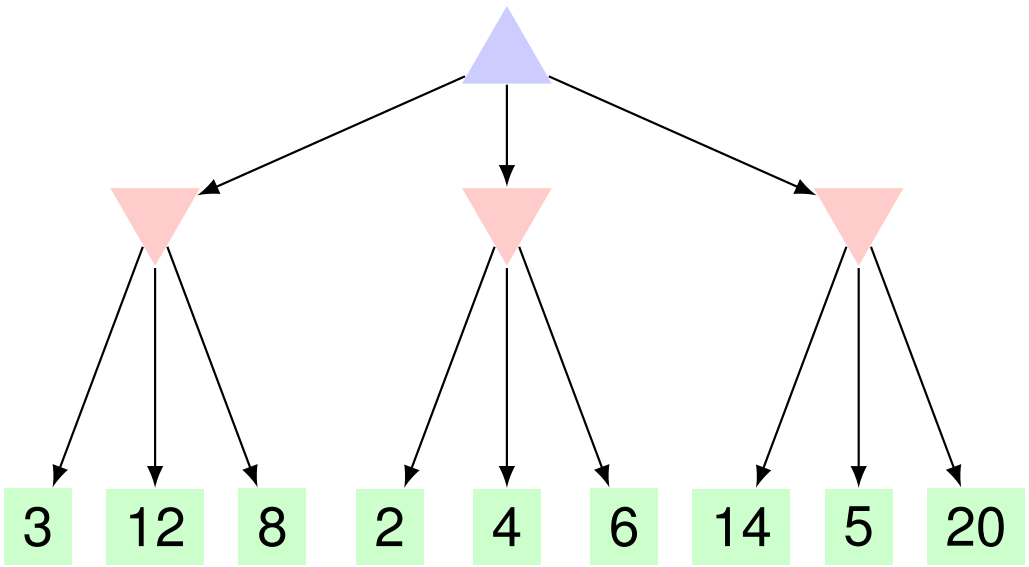
Video of Demo Smart Ghosts (Coordination) – Zoomed In



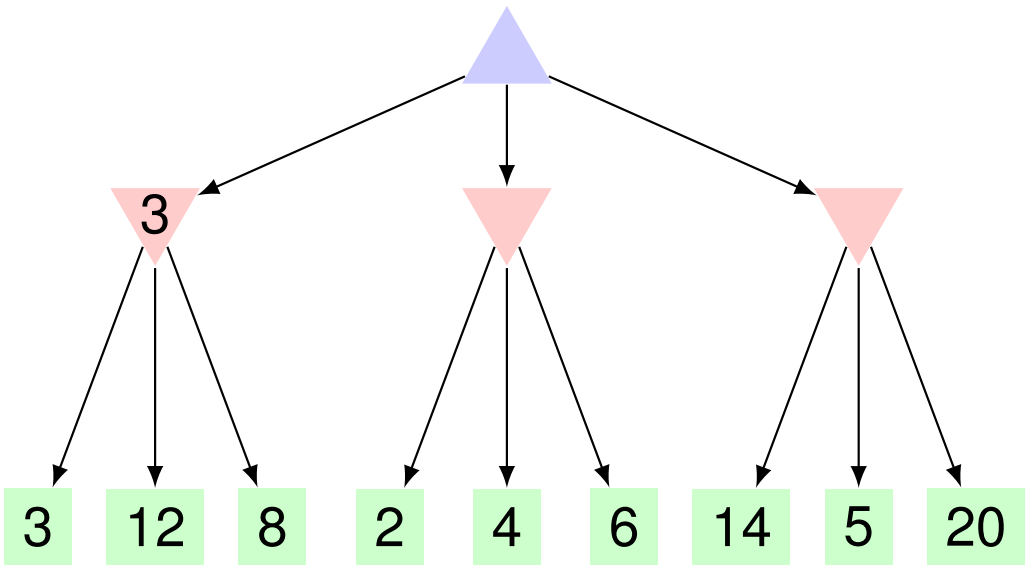
Game Tree Pruning



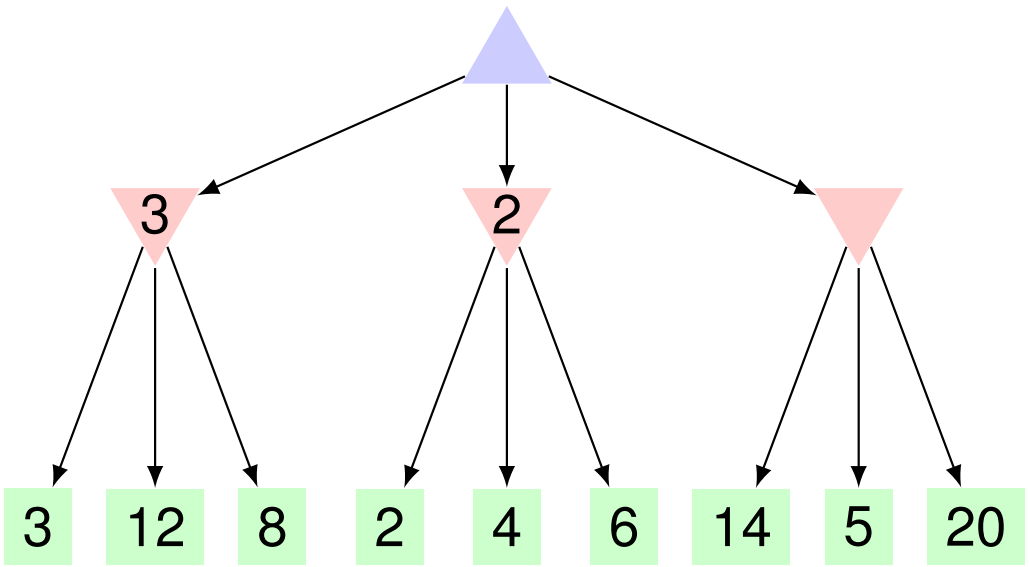
Minimax Example



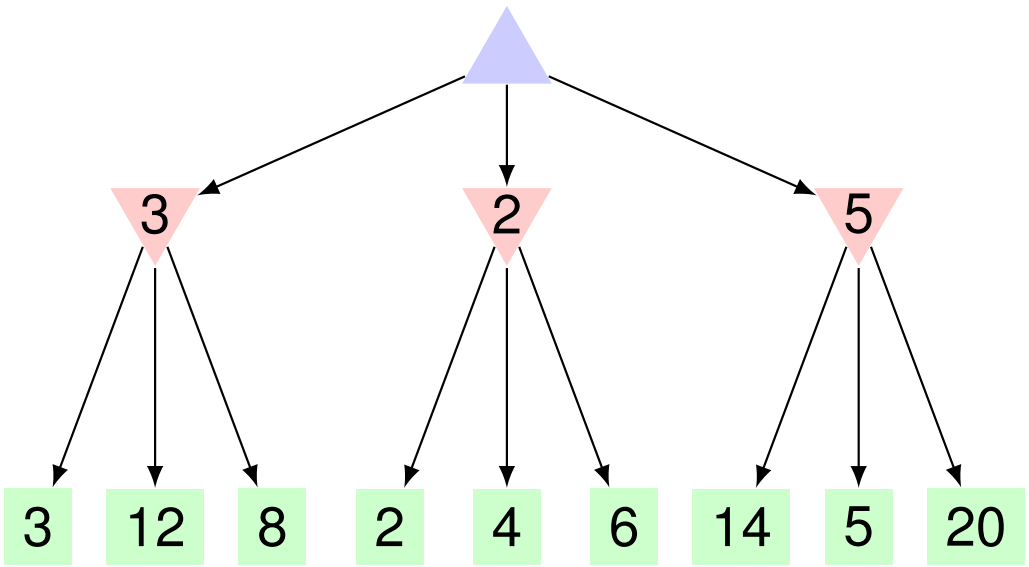
Minimax Example



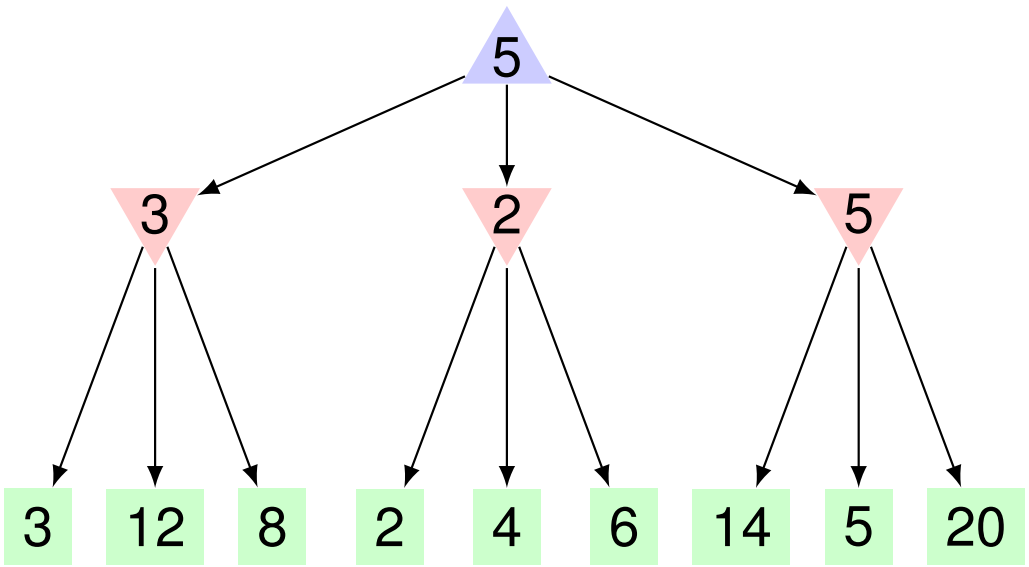
Minimax Example



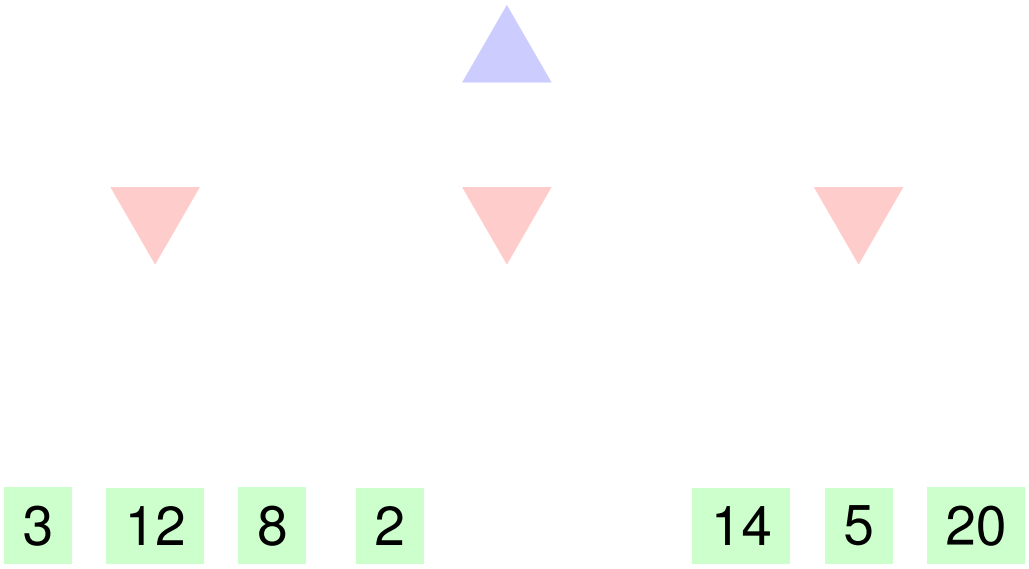
Minimax Example



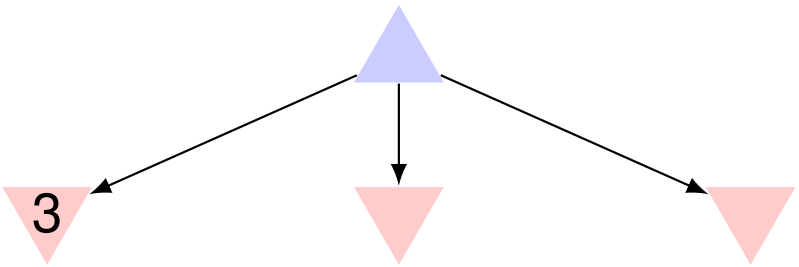
Minimax Example



Minimax Pruning

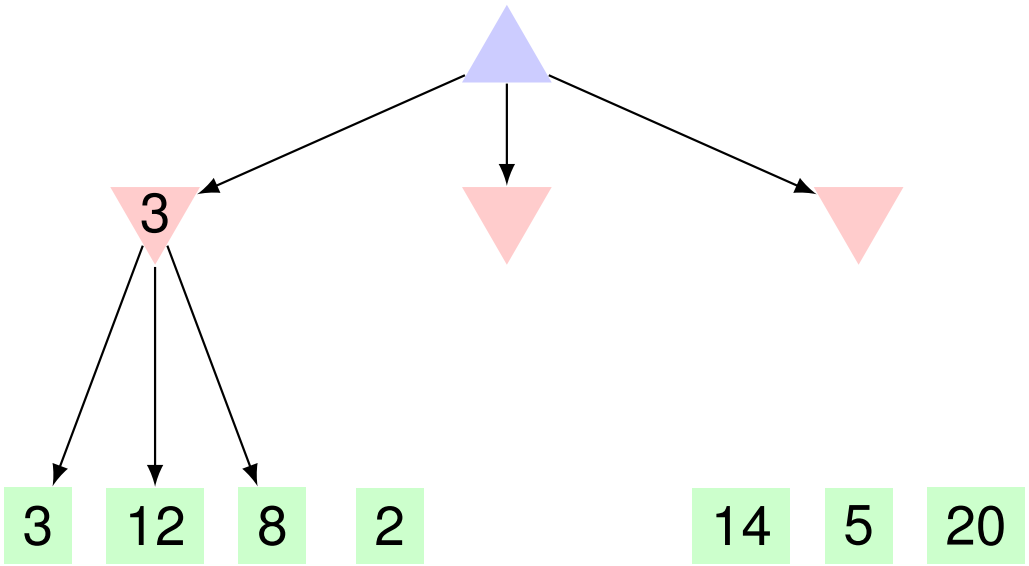


Minimax Pruning

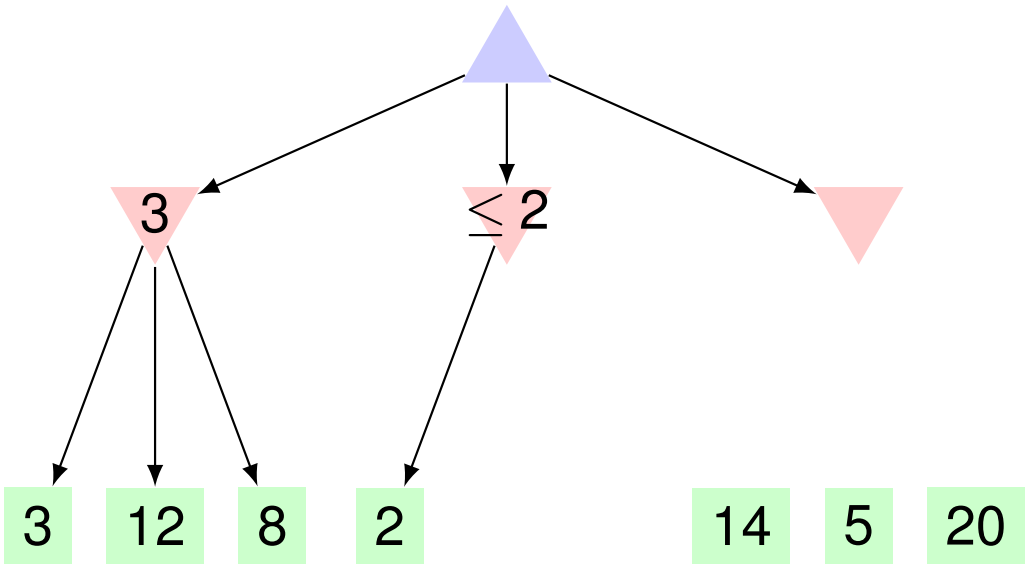


3 12 8 2 14 5 20

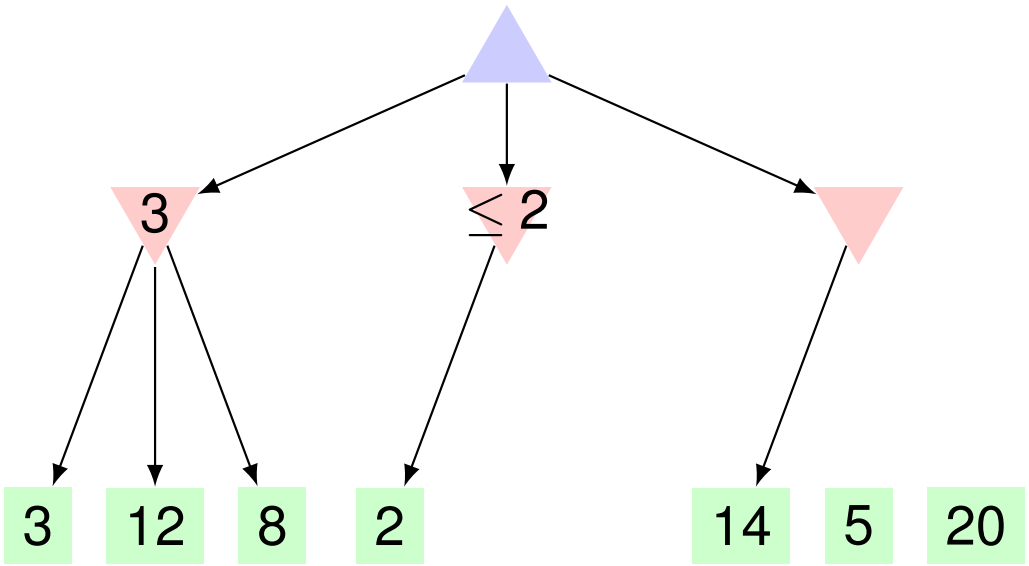
Minimax Pruning



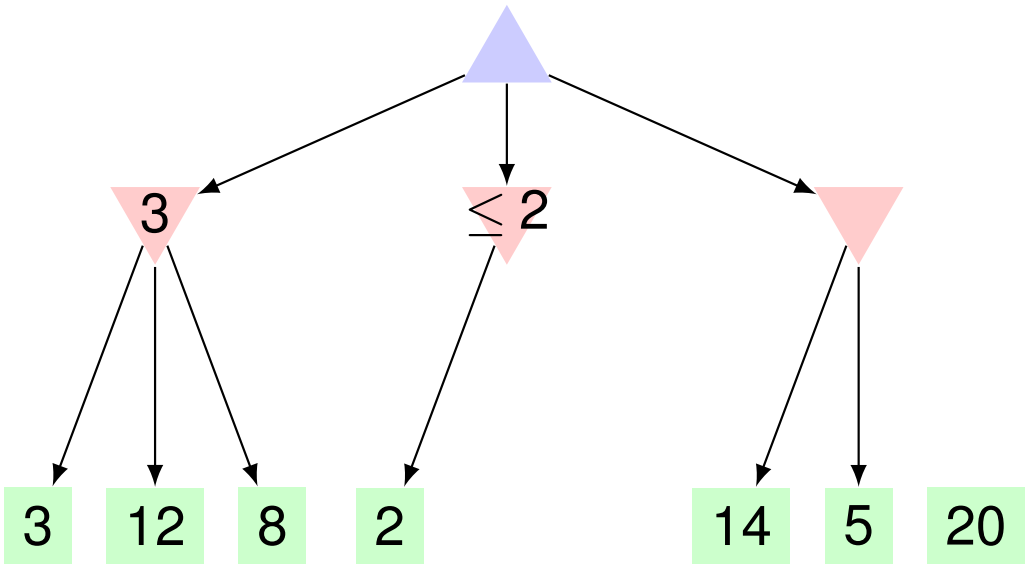
Minimax Pruning



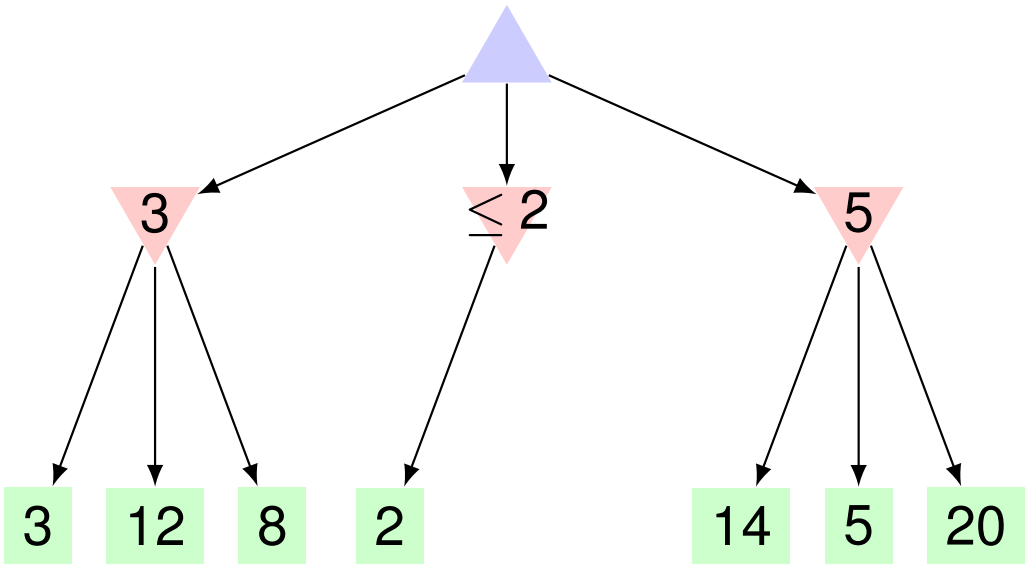
Minimax Pruning



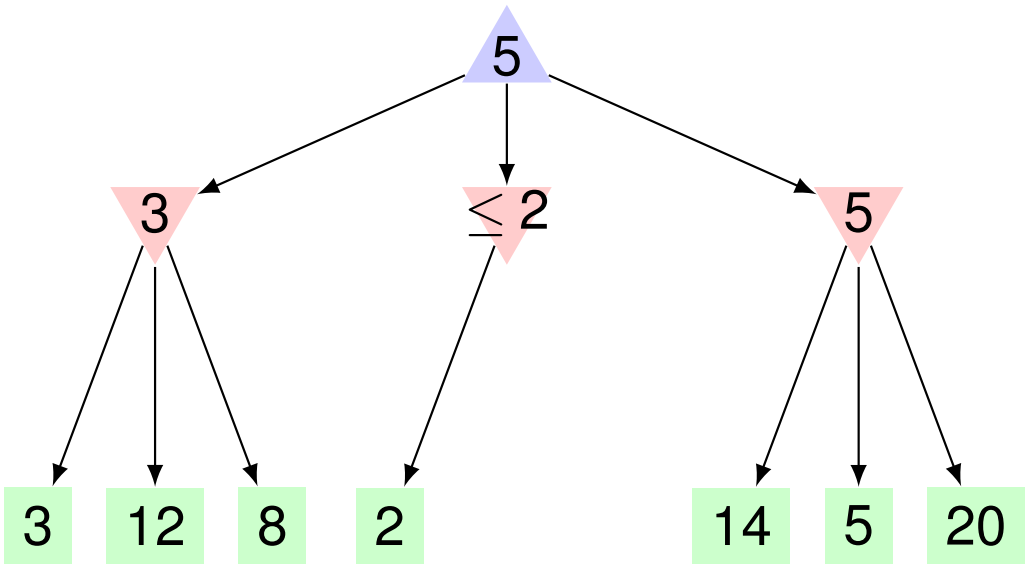
Minimax Pruning



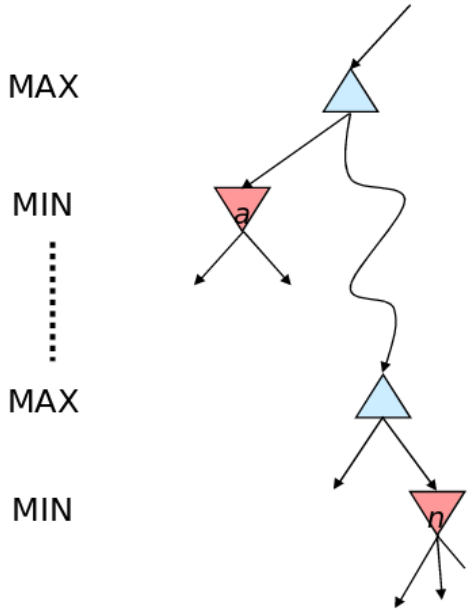
Minimax Pruning



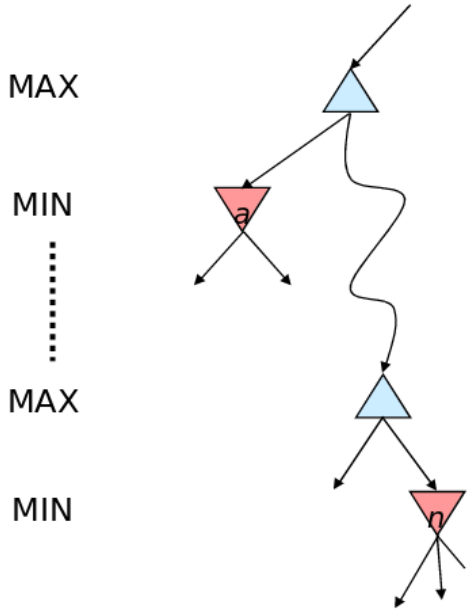
Minimax Pruning



Alpha-Beta Pruning



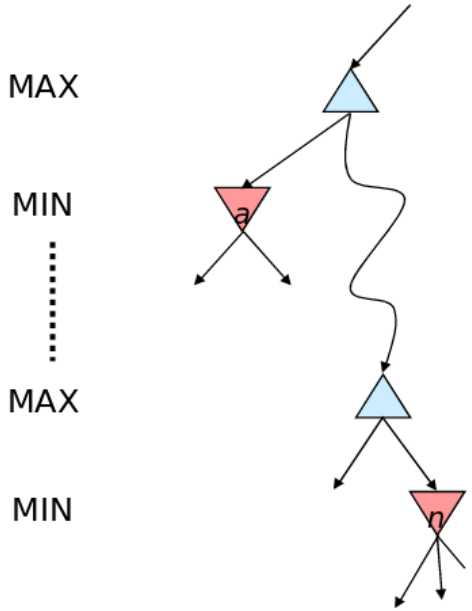
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node *n*.

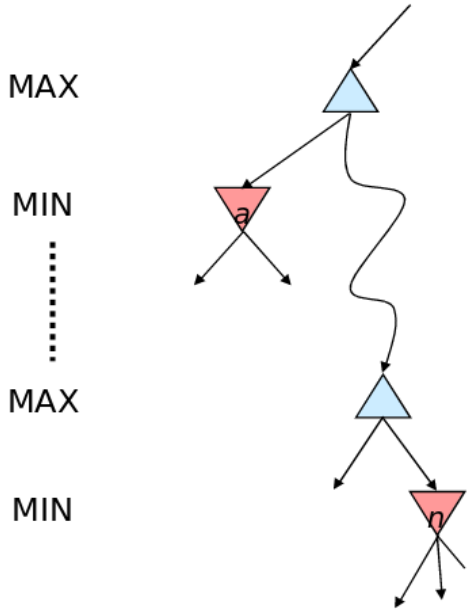
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node *n*.
- Looping over *n*'s children

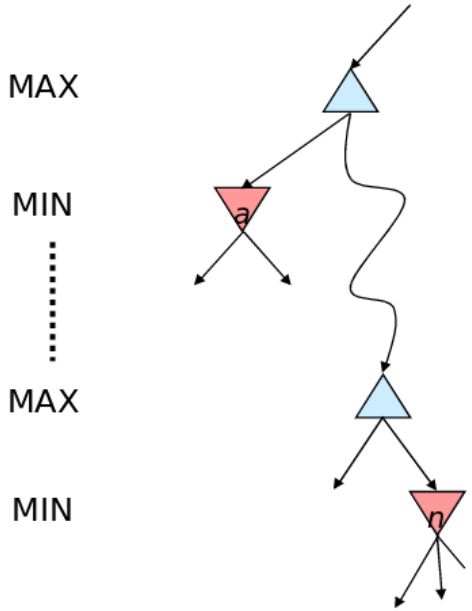
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node *n*.
- Looping over *n*'s children
- *n*'s estimate of min is dropping

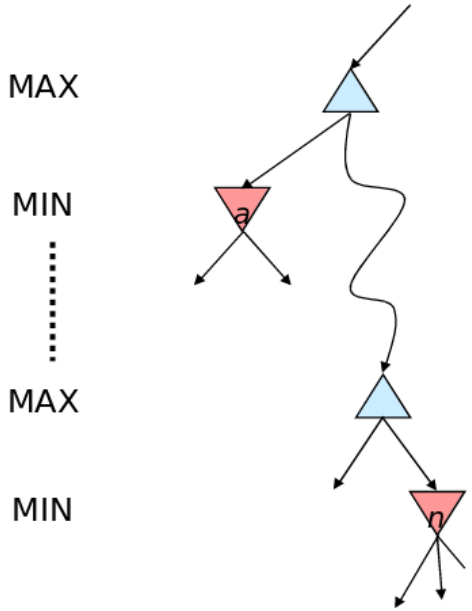
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node *n*.
- Looping over *n*'s children
- *n*'s estimate of min is dropping
- Who cares about *n*'s value? MAX

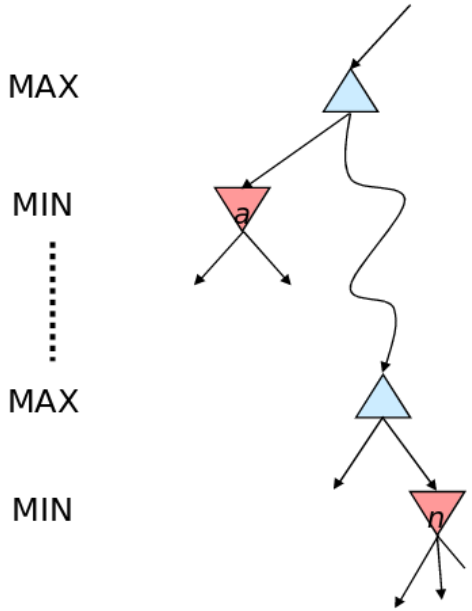
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node *n*.
- Looping over *n*'s children
- *n*'s estimate of min is dropping
- Who cares about *n*'s value? MAX
- $a =$ " best MAX value on path to root."

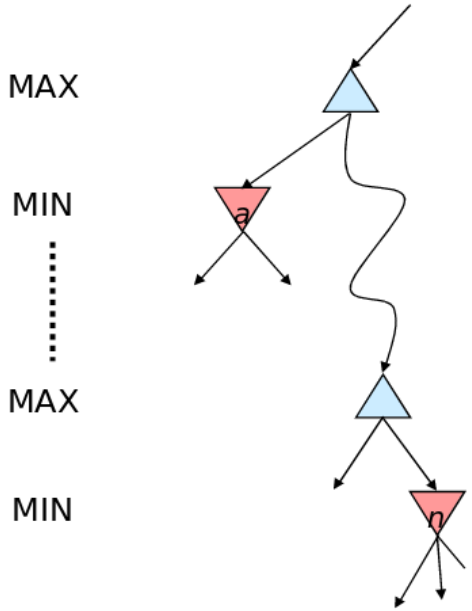
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node n.
- Looping over n's children
- n's estimate of min is dropping
- Who cares about n's value? MAX
- $a =$ " best MAX value on path to root."
- If $n < a$, than MAX will never choose it.

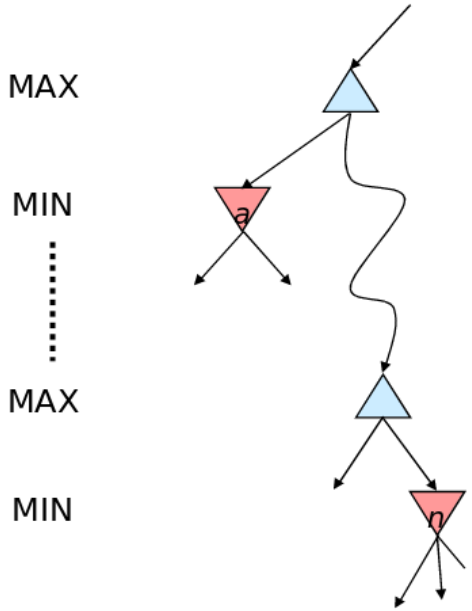
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node n.
- Looping over n's children
- n's estimate of min is dropping
- Who cares about n's value? MAX
- $a =$ " best MAX value on path to root."
- If $n < a$, than MAX will never choose it.
  So search "prunes" other children.

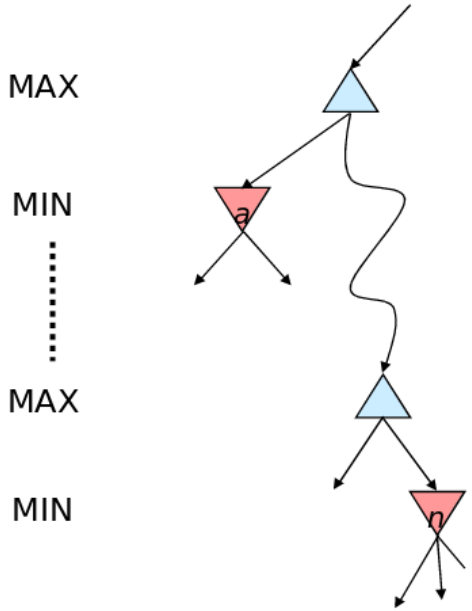
Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node n.
- Looping over n's children
- n's estimate of min is dropping
- Who cares about n's value? MAX
- $a =$ " best MAX value on path to root."
- If $n < a$, than MAX will never choose it.
  So search "prunes" other children.

Alpha-Beta Pruning



General configuration (MIN version)

- Computing MIN-VALUE at some node n.
- Looping over n's children
- n's estimate of min is dropping
- Who cares about n's value? MAX
- $a =$ " best MAX value on path to root."
- If $n < a$, than MAX will never choose it.
  So search "prunes" other children.

Symmetric for MAX version.

Alpha-Beta Implementation

α - MAX's best option on path to root.

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state , α, β):

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state , α, β):
initialize v = $+\infty$.

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state , α, β):
initialize v = $+\infty$.
for each successor s of state:

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state , α, β):
  initialize v = $+\infty$.
  for each successor *s* of state:
    v = min(v, value(*s*, α, β))

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state , α, β):
  initialize v = $+\infty$.
  for each successor *s* of state:
    v = min(v, value(*s*, α, β))
    if v $\leq \alpha$ return v

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state , α, β):
  initialize v = $+\infty$.
  for each successor s of state:
    v = min(v, value(s, α, β))
    if v $\leq \alpha$ return v
    $\beta = \min(\beta, v)$

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

```
def min-value(state , α,β):
  initialize v = +∞.
  for each successor s of state:
      v = min(v, value(s, α,β))
      if v ≤ α return v
      β = min(β, v)
 return v
```

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

def min-value(state , α, β):
  initialize v = $+\infty$.
  for each successor s of state:
    v = min(v, value(s, α, β))
    if v $\leq \alpha$ return v
    $\beta = \min(\beta, v)$
 return v

def max-value(state , α, β):

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

```
def min-value(state , α,β):
  initialize v = +∞.
  for each successor s of state:
      v = min(v, value(s, α,β))
      if v ≤ α return v
      β = min(β, v)
 return v
```

```
def max-value(state , α,β):
   initialize v = −∞.
```

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

```
def min-value(state , α,β):
  initialize v = +∞.
  for each successor s of state:
      v = min(v, value(s, α,β))
      if v ≤ α return v
      β = min(β, v)
 return v
```

```
def max-value(state , α,β):
   initialize v = −∞.
   for each successor s of state:
```

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

```
def min-value(state , α,β):
  initialize v = +∞.
  for each successor s of state:
     v = min(v, value(s, α,β))
     if v ≤ α return v
     β = min(β, v)
 return v
```

```
def max-value(state , α,β):
   initialize v = −∞.
   for each successor s of state:
      v = max(v, value(s, α,β))
```

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

```
def min-value(state , α,β):
  initialize v = +∞.
  for each successor s of state:
     v = min(v, value(s, α,β))
     if v ≤ α return v
     β = min(β, v)
 return v
```

```
def max-value(state , α,β):
   initialize v = −∞.
   for each successor s of state:
      v = max(v, value(s, α,β))
      if v ≤ α return v
```

Alpha-Beta Implementation

α - MAX's best option on path to root.

β - MIN's best option on path to root.

```
def min-value(state , α,β):
  initialize v = +∞.
  for each successor s of state:
     v = min(v, value(s, α,β))
     if v ≤ α return v
     β = min(β, v)
 return v
```

```
def max-value(state , α,β):
   initialize v = −∞.
   for each successor s of state:
      v = max(v, value(s, α,β))
      if v ≤ α return v
      α = min(α, v)
```

Alpha-Beta Implementation

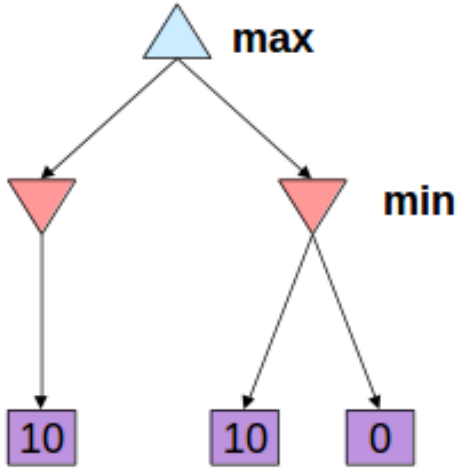
α - MAX's best option on path to root.

β - MIN's best option on path to root.

```
def min-value(state , α,β):
  initialize v = +∞.
  for each successor s of state:
     v = min(v, value(s, α,β))
     if v ≤ α return v
     β = min(β, v)
 return v
```

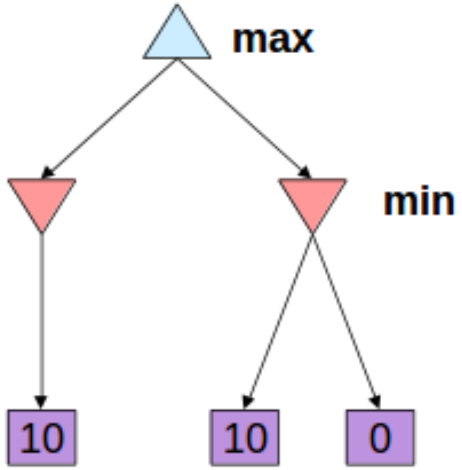
```
def max-value(state , α,β):
   initialize v = −∞.
   for each successor s of state:
      v = max(v, value(s, α,β))
      if v ≤ α return v
      α = min(α, v)
 return v
```

Alpha-Beta Pruning Properties

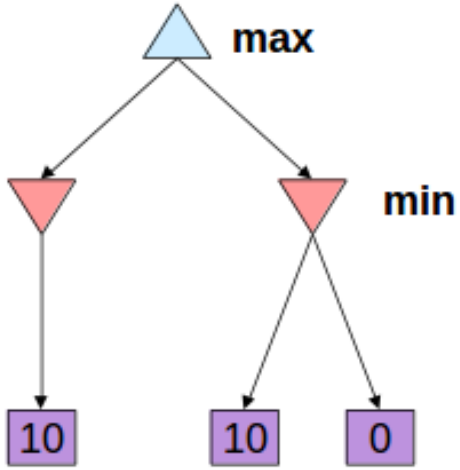


Alpha-Beta Pruning Properties

This pruning has no effect on minimax value computed for the root!



Alpha-Beta Pruning Properties

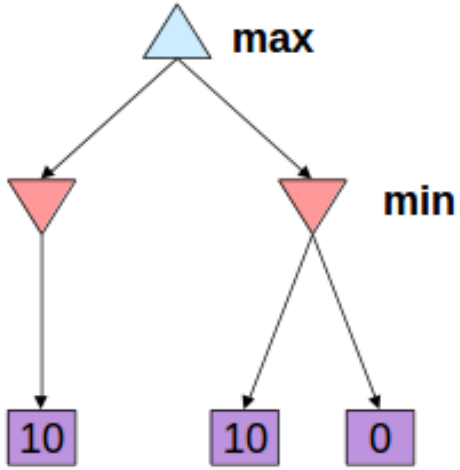


This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

- Important: root's children may be wrong

Alpha-Beta Pruning Properties

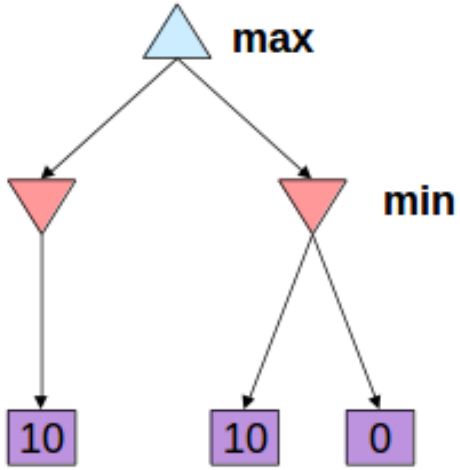


This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

Alpha-Beta Pruning Properties

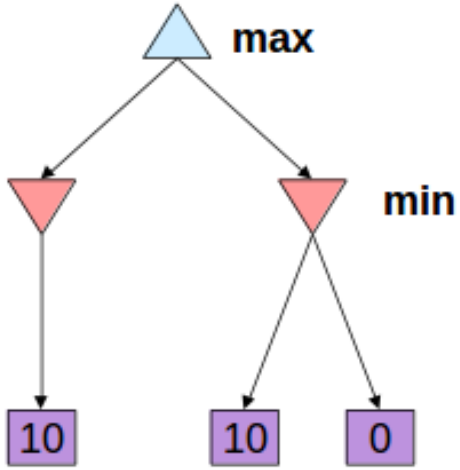


This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

Alpha-Beta Pruning Properties



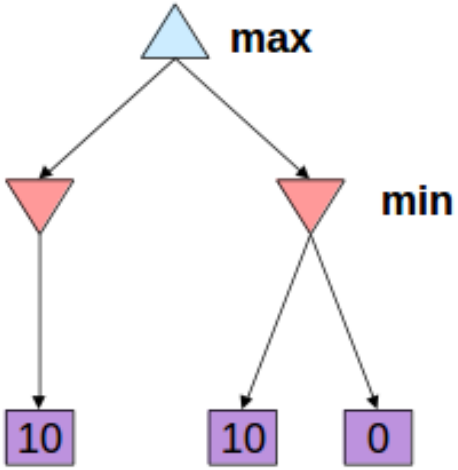
This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

Good child ordering improves effectiveness

Alpha-Beta Pruning Properties



This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

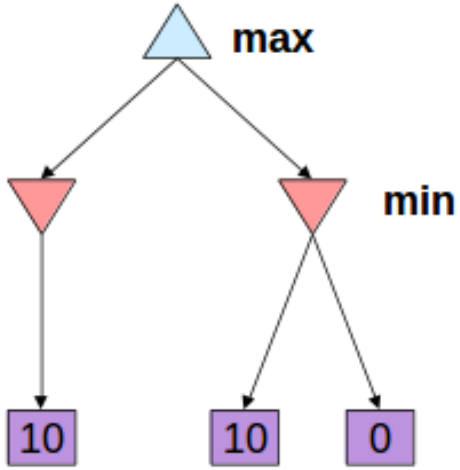
- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

Good child ordering improves effectiveness

With "perfect ordering":

- Time complexity drops to $O(b^{m/2})$

Alpha-Beta Pruning Properties



This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

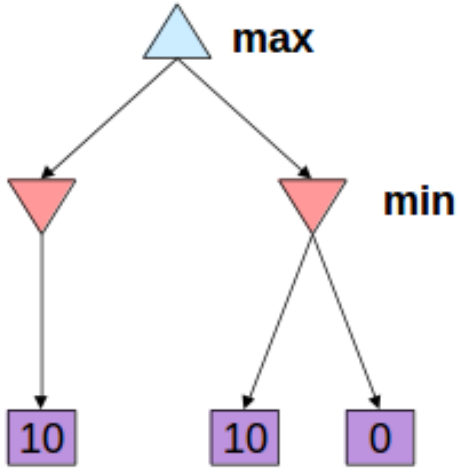
- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

Good child ordering improves effectiveness

With "perfect ordering":

- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!

Alpha-Beta Pruning Properties



This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

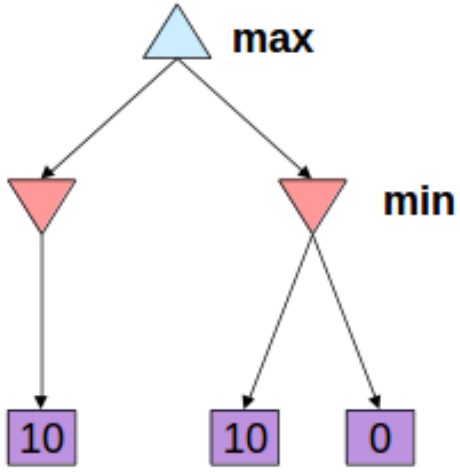
- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

Good child ordering improves effectiveness

With "perfect ordering":

- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!
- Full search of, e.g. chess, is still hopeless...

Alpha-Beta Pruning Properties



This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

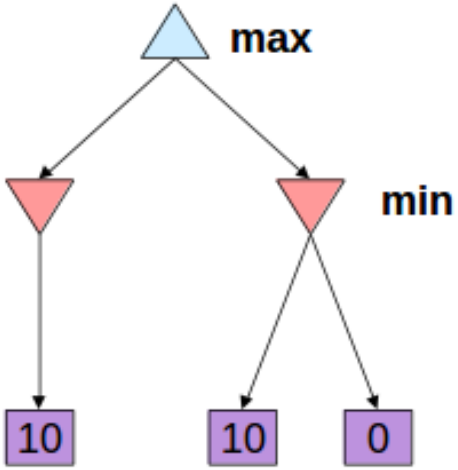
- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

Good child ordering improves effectiveness

With "perfect ordering":

- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!
- Full search of, e.g. chess, is still hopeless...

Alpha-Beta Pruning Properties



This pruning has no effect on minimax value computed for the root!

Values of intermediate nodes might be wrong

- Important: root's children may be wrong
- $\rightarrow$ most naive version not for action selection

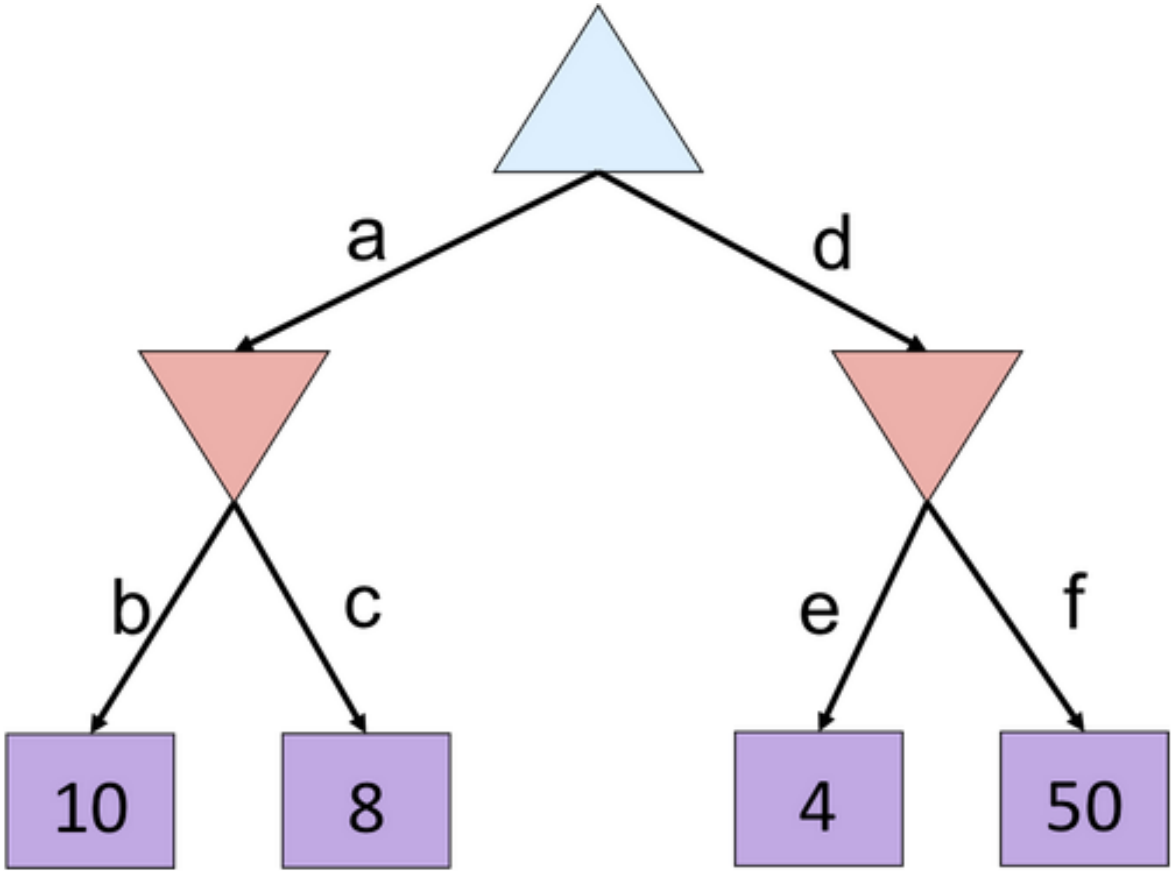
Good child ordering improves effectiveness

With "perfect ordering":

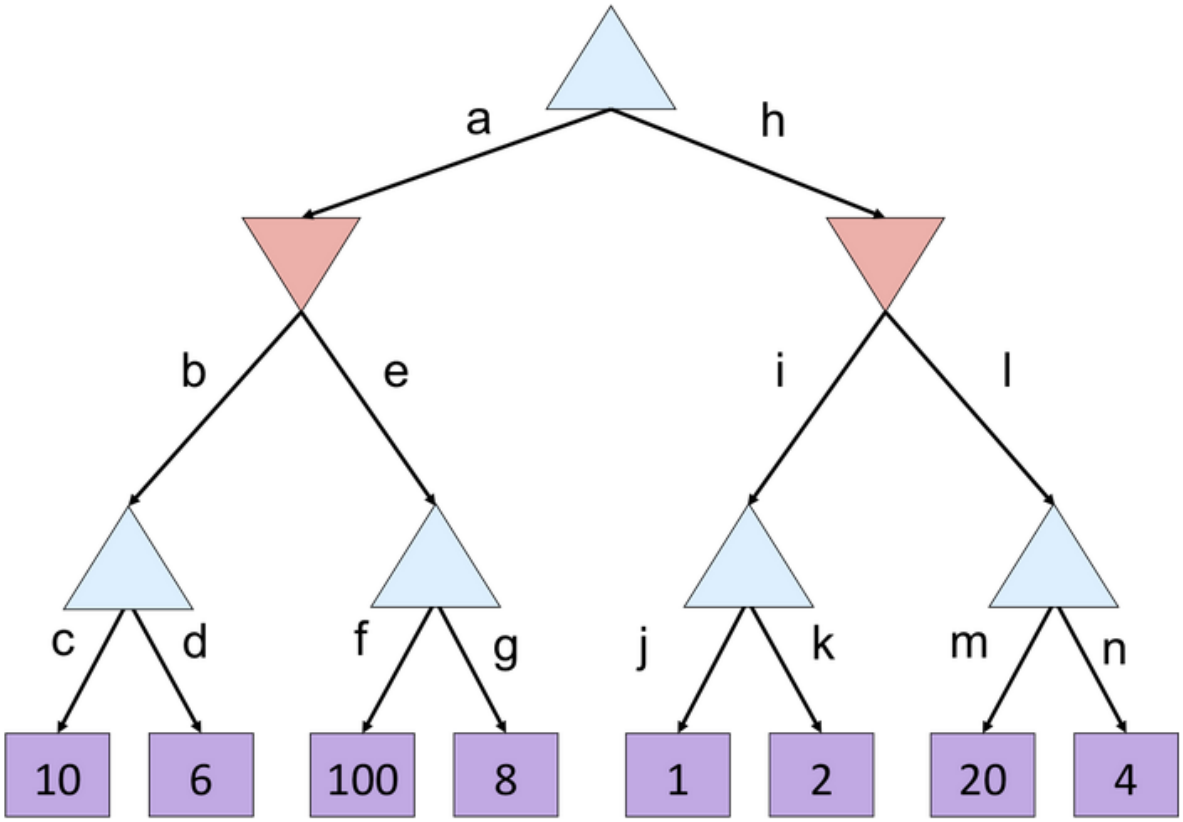
- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!
- Full search of, e.g. chess, is still hopeless...

This is a simple example of metareasoning (computing about what to compute)

Alpha-Beta Quiz



Alpha-Beta Quiz 2



Next Time: Uncertainty!