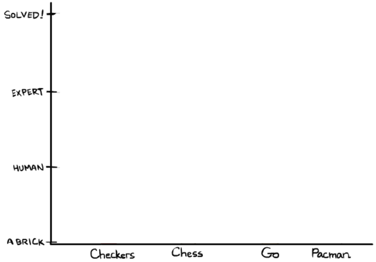
Adversarial Search

CSE 4711: Artificial Intelligence

Md. Bakhtiar Hasan

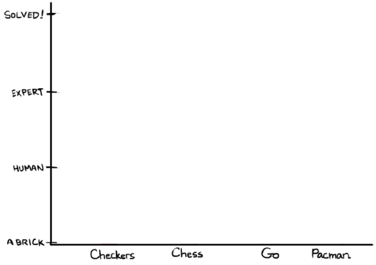
Assistant Professor
Department of Computer Science and Engineering
Islamic University of Technology

Game Playing (State-of-the-Art)



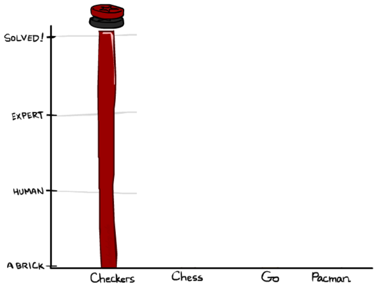
Game Playing (State-of-the-Art)

- Checkers
  - 1950: First computer player
  - 1994: First computer champion
    - Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame



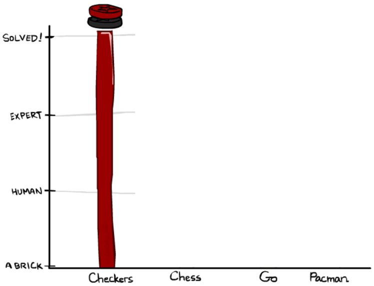
Game Playing (State-of-the-Art)

- Checkers
  - 1950: First computer player
  - 1994: First computer champion
    - Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame
  - 2007: Solved
    - If both players play optimally, you can at least force a draw



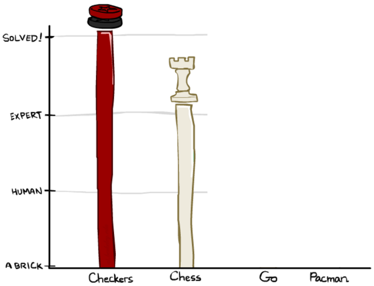
Game Playing (State-of-the-Art)

- Checkers
  - 1950: First computer player
  - 1994: First computer champion
    - Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame
  - 2007: Solved
    - If both players play optimally, you can at least force a draw
- Chess
  - 1997: Deep Blue defeats human champion Gary Kasparove in a six-game match
    - Examined 200M positions per second
    - Used very sophisticated evaluation function
    - Undisclosed methods for searching up to 40 ply



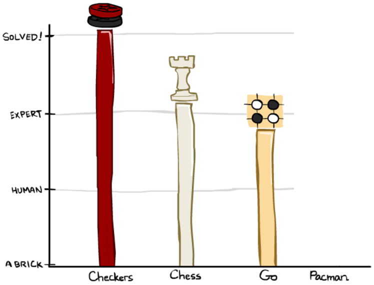
Game Playing (State-of-the-Art)

- Checkers
  - 1950: First computer player
  - 1994: First computer champion
    - Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame
  - 2007: Solved
    - If both players play optimally, you can at least force a draw
- Chess
  - 1997: Deep Blue defeats human champion Gary Kasparove in a six-game match
    - Examined 200M positions per second
    - Used very sophisticated evaluation function
    - Undisclosed methods for searching up to 40 ply
  - Current programs are even better, if less historic



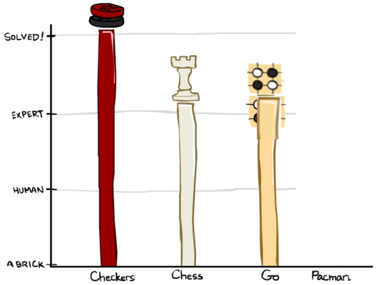
Game Playing (State-of-the-Art)

- Go
  - Human champions are now starting to be challenged by machines
    - Branching Factor $b > 300$
    - Classic programs $\rightarrow$ Pattern knowledge bases
    - Recent programs $\rightarrow$ Monte Carlo (randomized) expansion methods



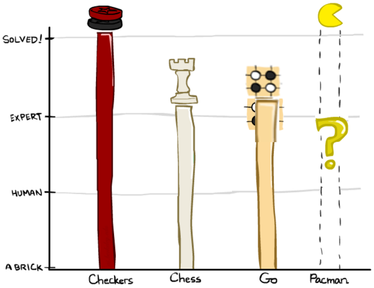
Game Playing (State-of-the-Art)

- Go
  - Human champions are now starting to be challenged by machines
    - Branching Factor $b > 300$
    - Classic programs $\rightarrow$ Pattern knowledge bases
    - Recent programs $\rightarrow$ Monte Carlo (randomized) expansion methods
  - 2016: Alpha GO defeats human champiom
    - Uses Monte Carlo Tree Search
    - Learned evaluation function: Odd-Even Function

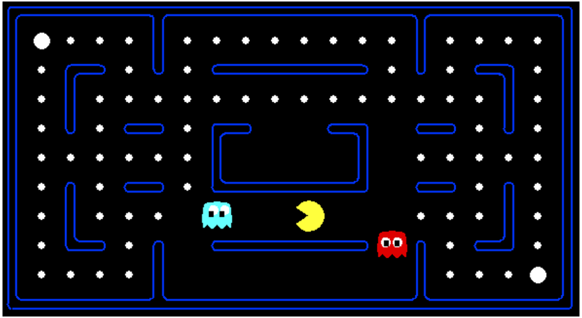


Game Playing (State-of-the-Art)

- Go
  - Human champions are now starting to be challenged by machines
    - Branching Factor $b > 300$
    - Classic programs $\rightarrow$ Pattern knowledge bases
    - Recent programs $\rightarrow$ Monte Carlo (randomized) expansion methods
  - 2016: Alpha GO defeats human champiom
    - Uses Monte Carlo Tree Search
    - Learned evaluation function: Odd-Even Function
- Pacman

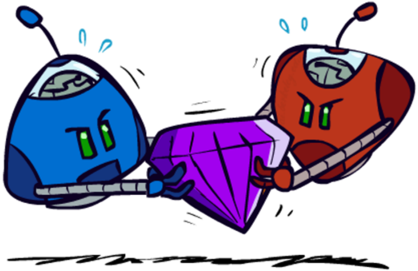


Behavior from Computation



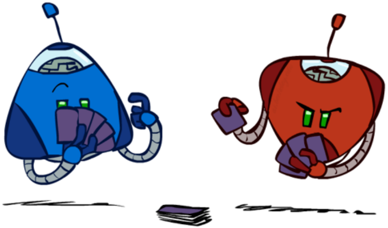
Video: mystery-pacman

Adversarial Games



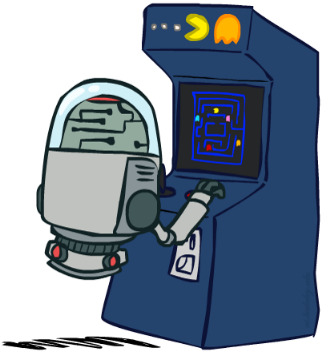
Types of Games

- Many different kinds of games!
- Criteria/Axes:
  - Deterministic or stochastic?
    - e.g., Chess vs Monopoly
  - One, two, or more players?
    - e.g., Solitaire vs Checkers vs D&D, etc.
  - Zero sum?
    - e.g., Football vs Nuclear war
  - Perfect information?
    - e.g., Tic-Tac-Toe vs Poker
- Want algorithms for calculating a **strategy (policy)** which recommends a move from each state

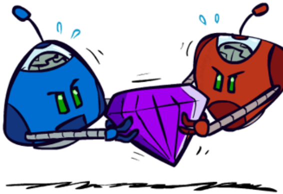


Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: $P = \{1 \ldots N\}$ (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:** $S \rightarrow A$

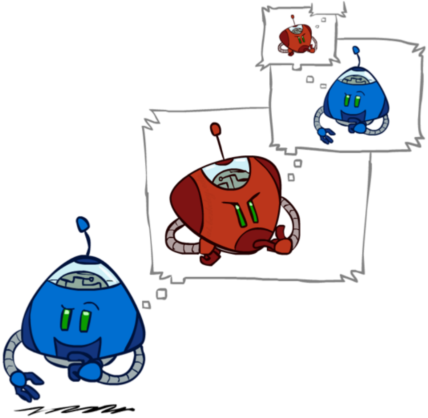


Zero-Sum Games



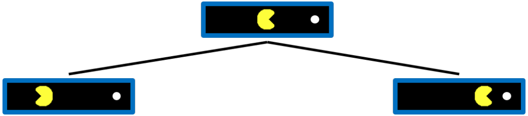
- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition
- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

Adversarial Search

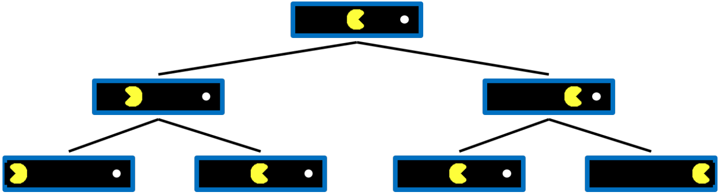


Single-Agent Trees

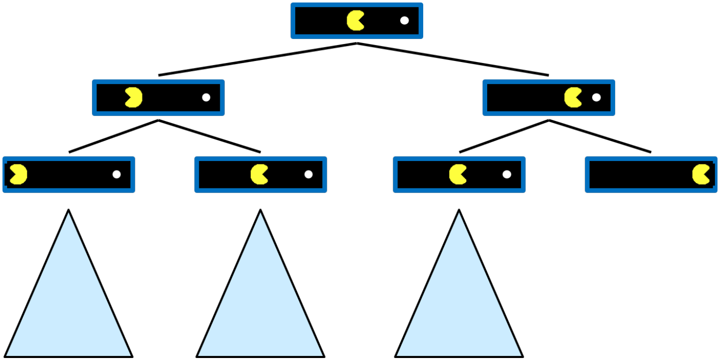
Single-Agent Trees



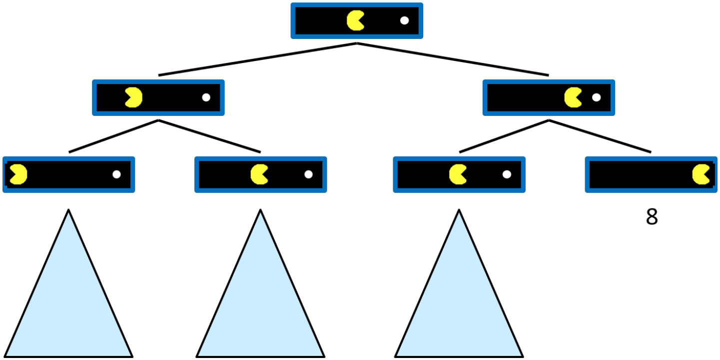
Single-Agent Trees



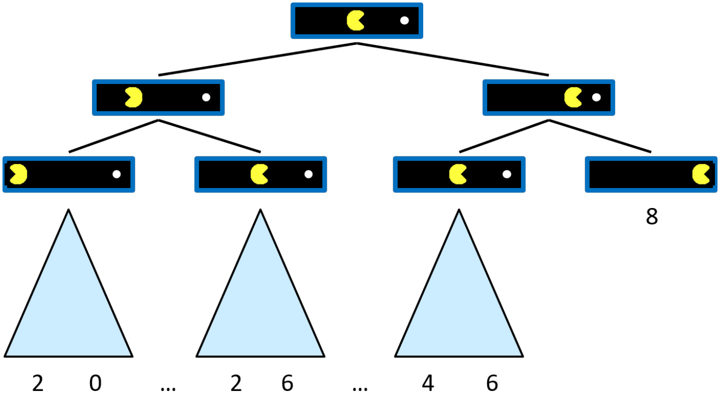
Single-Agent Trees



Single-Agent Trees

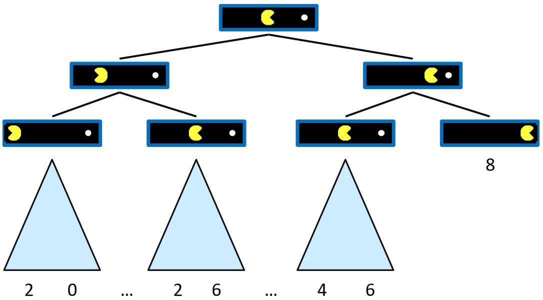


Single-Agent Trees



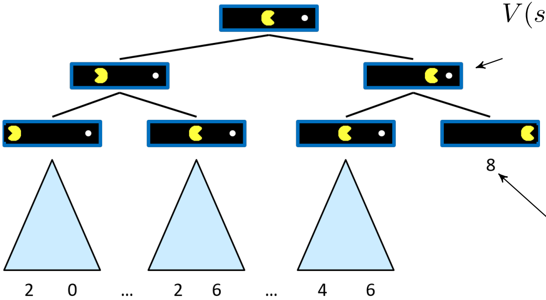
Value of a State

- The best achievable outcome (utility) from that state



Value of a State

- The best achievable outcome (utility) from that state



Non-Terminal States:

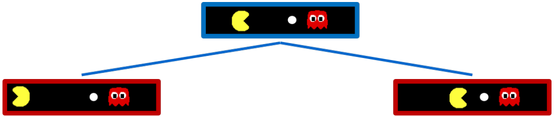
$$V(s) = \max_{s' \in children(s)} V(s')$$

Terminal States:

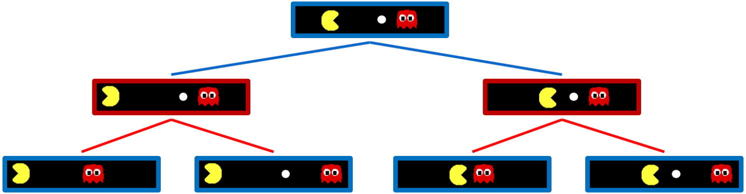
$V(s) = $ known

Adversarial Game Trees

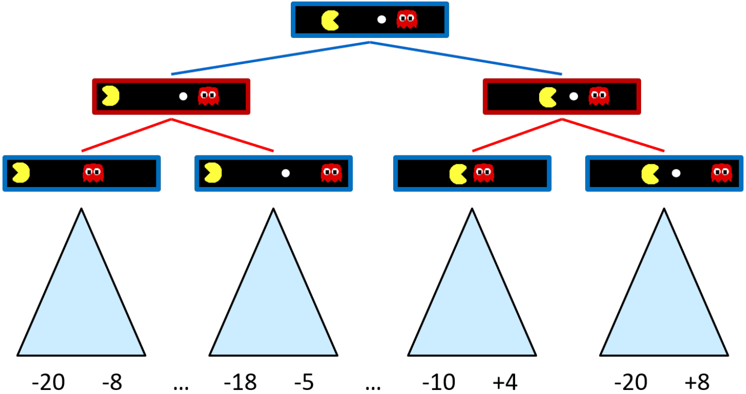
Adversarial Game Trees



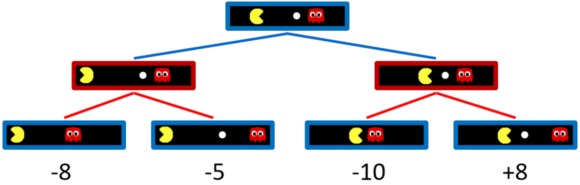
Adversarial Game Trees



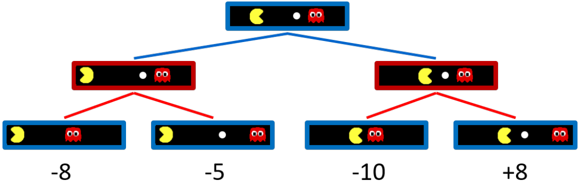
Adversarial Game Trees



Minimax Values



Minimax Values

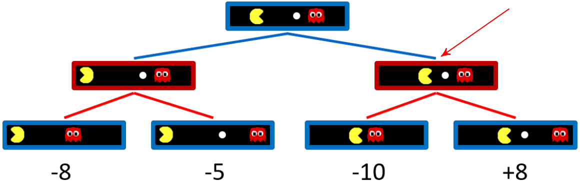


Terminal States:
$V(s) = \text{known}$

Minimax Values

States Under Opponent's Control:

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



Terminal States:

$$V(s) = \text{known}$$

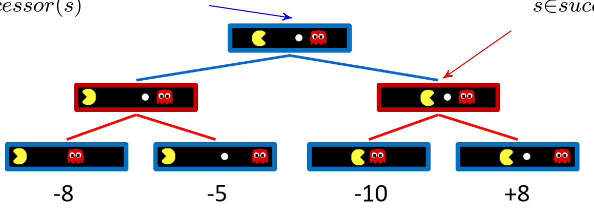
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

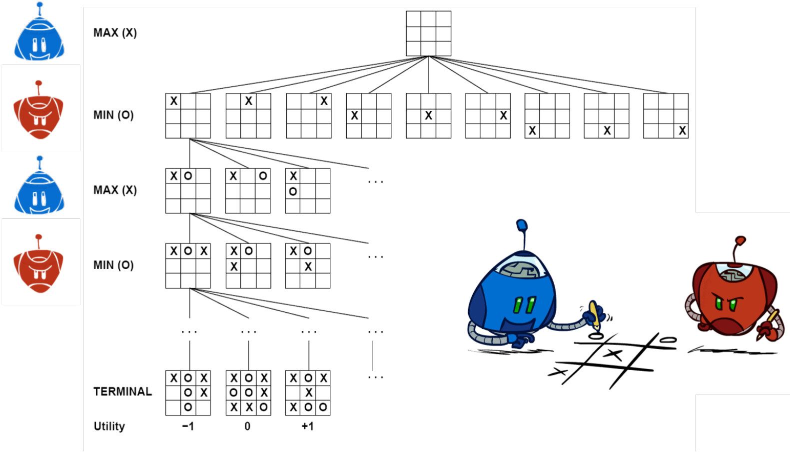
$$V(s') = \min_{s \in successors(s')} V(s)$$



Terminal States:

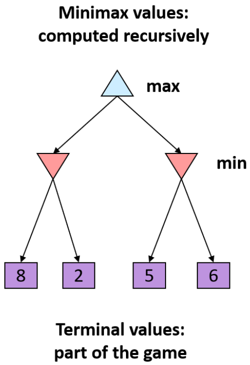
$$V(s) = \text{known}$$

Tic-Tac-Toe Game Tree



Adversarial Search (Minimax)

- 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

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

def max-value(*state*):

initialize $v = -\infty$

for each successor of *state*:

$v = \max(v,$ min-value$(successor))$

return v

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

def min-value(*state*):

initialize $v = +\infty$

for each successor of *state*:

$v = \min(v,$ max-value$(successor))$

return v

Minimax Implementation (Dispatch)

```
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 MIN: return min-value(state)
```

def max-value(*state*):

  initialize $v = -\infty$

  for each successor of *state*:

    $v = \max(v, \text{value}(successor))$

  return v

def min-value(*state*):

  initialize $v = +\infty$

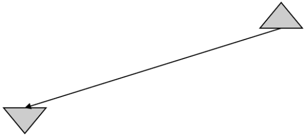
  for each successor of *state*:

    $v = \min(v, \text{value}(successor))$

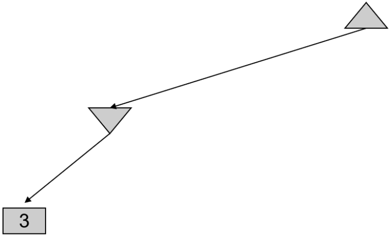
  return v

Minimax Example

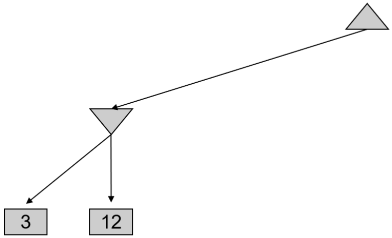
Minimax Example



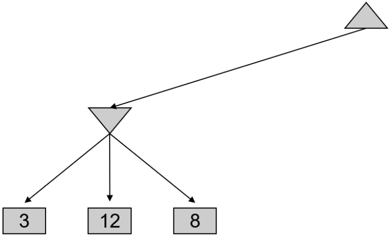
Minimax Example



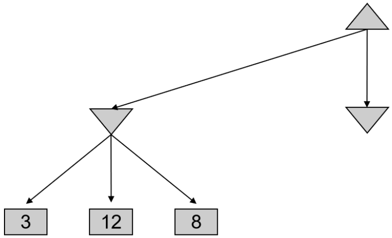
Minimax Example



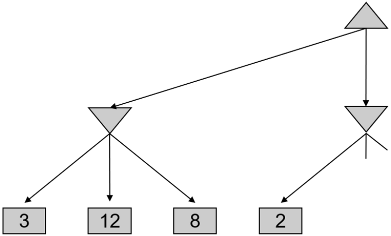
Minimax Example



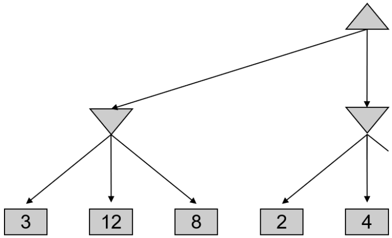
Minimax Example



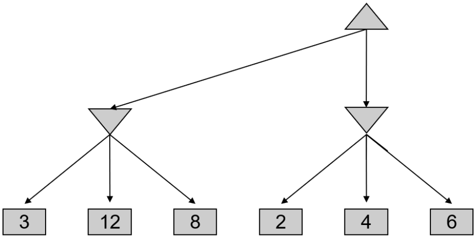
Minimax Example



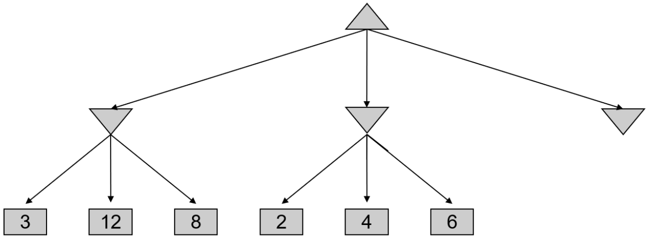
Minimax Example



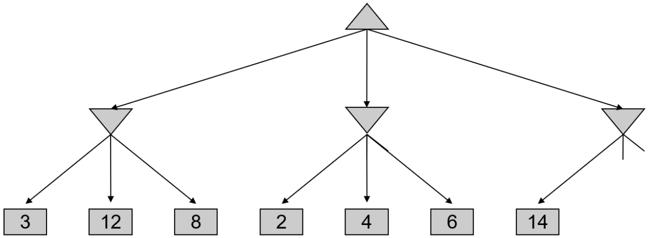
Minimax Example



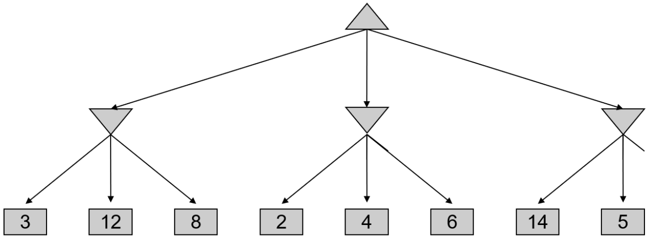
Minimax Example



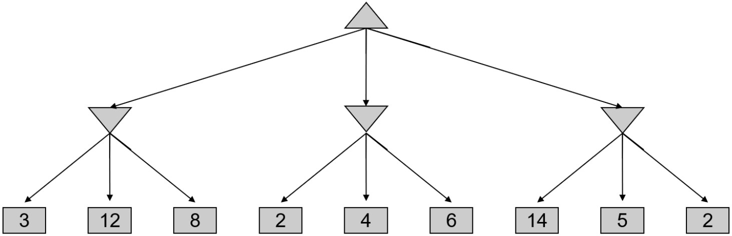
Minimax Example



Minimax Example

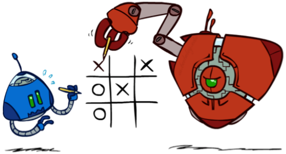


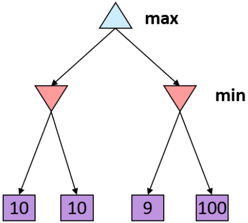
Minimax Example



Minimax Properties

Optimal against a perfect player.

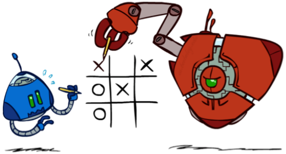


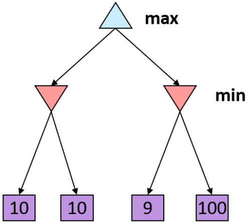


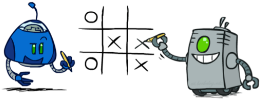
Video: min, exp

Minimax Properties

Optimal against a perfect player. Otherwise?



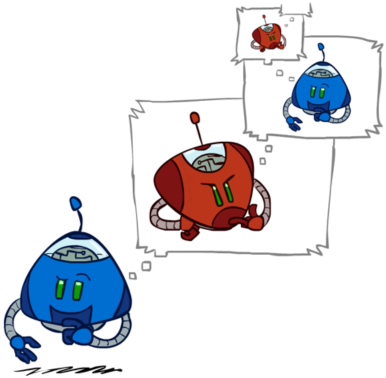




Video: min, exp

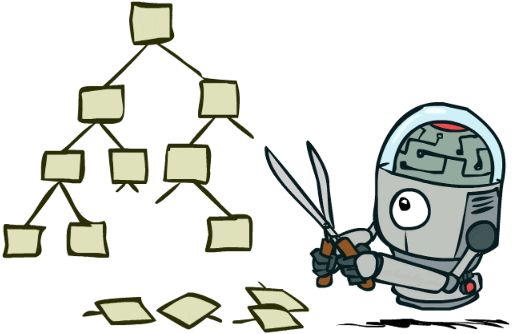
Minimax Efficiency

- 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?

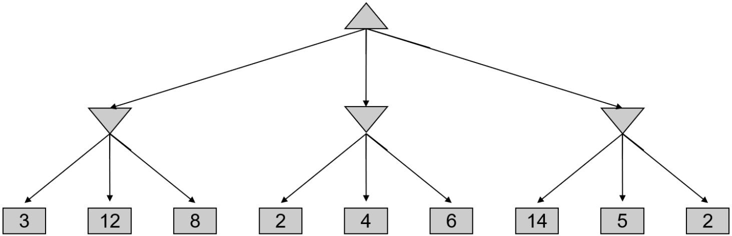


Resource Limits

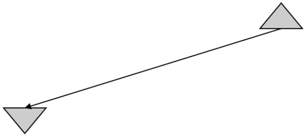
Game Tree Pruning



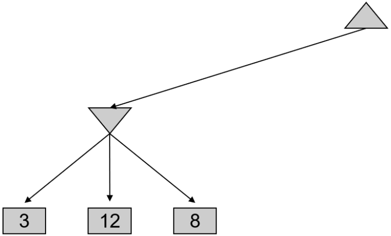
Minimax Example (Revisited)



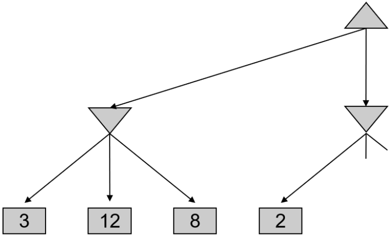
Minimax Pruning



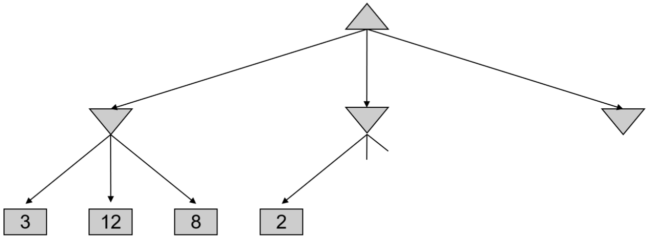
Minimax Pruning



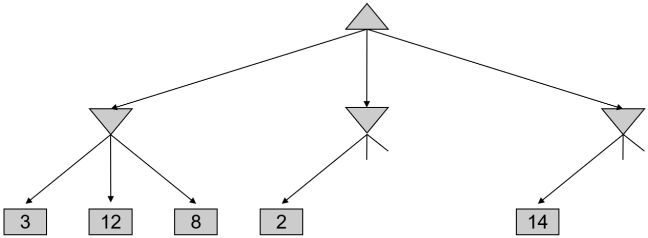
Minimax Pruning



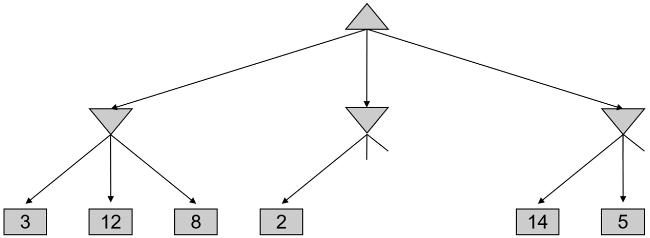
Minimax Pruning



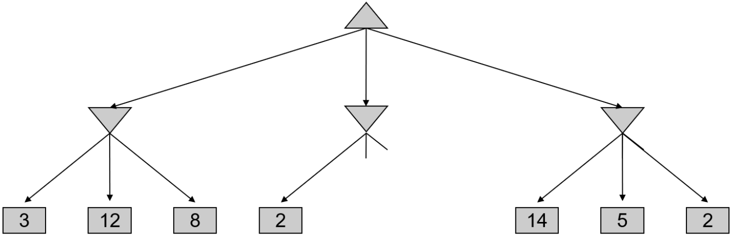
Minimax Pruning



Minimax Pruning

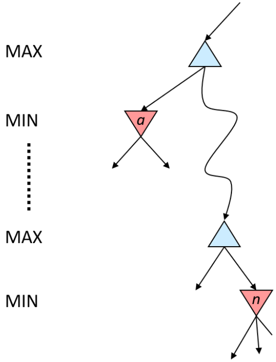


Minimax Pruning



Alpha-Beta Pruning

- General configuration (MIN version)
  - Computing the MIN-VALUE at some node n
  - Looping over n's children
  - n's estimate of the children's min is dropping
  - Who cares about n's value? MAX
  - Let a be the best value that MAX can get at any choice point along the current path from the root
  - If n becomes worse than a, MAX will avoid it, so we can stop considering n's other children (it's already bad enough that it won't be played)
- MAX version is symmetric



Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

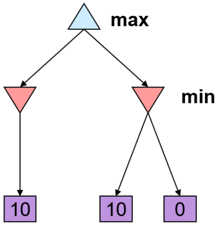
def max-value(*state*, α, β):
    initialize $v = -\infty$
    for each successor of *state*:
$v = \max(v, value(successor, \alpha, \beta))$
        if $v \geq \beta$: return v
        $\alpha = \max(\alpha, v)$
    return v

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

Alpha-Beta Pruning Properties

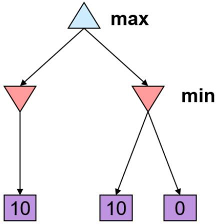
- The pruning has **no effect** on minimax value computed for the root!

- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - The most naïve version won't let you do action selection



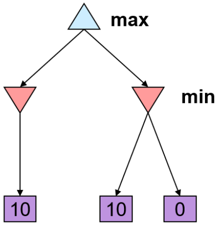
Alpha-Beta Pruning Properties

- The pruning has **no effect** on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - The most naïve version won't let you do action selection
    - Solution 1: Prune only on inequality
    - Solution 2: Keep track of which one was first



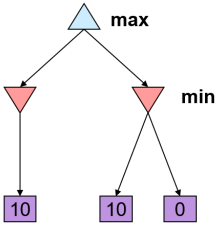
Alpha-Beta Pruning Properties

- The pruning has **no effect** on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - The most naïve version won't let you do action selection
    - Solution 1: Prune only on inequality
    - Solution 2: Keep track of which one was first
- Good child ordering improves effectiveness



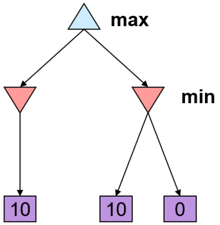
Alpha-Beta Pruning Properties

- The pruning has **no effect** on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - The most naïve version won't let you do action selection
    - Solution 1: Prune only on inequality
    - Solution 2: Keep track of which one was first
- 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...

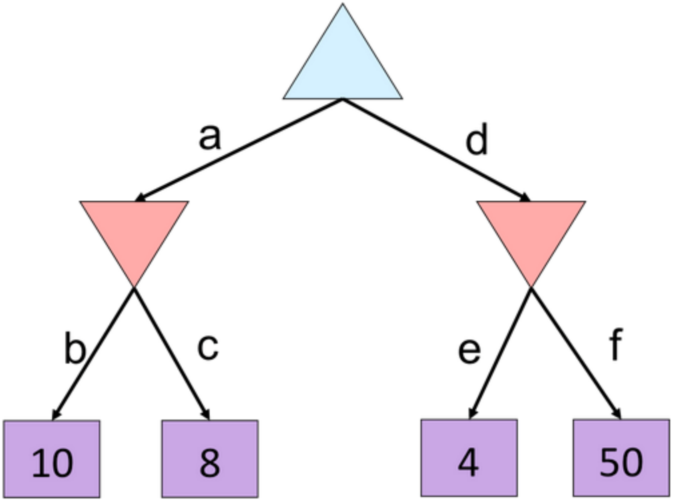


Alpha-Beta Pruning Properties

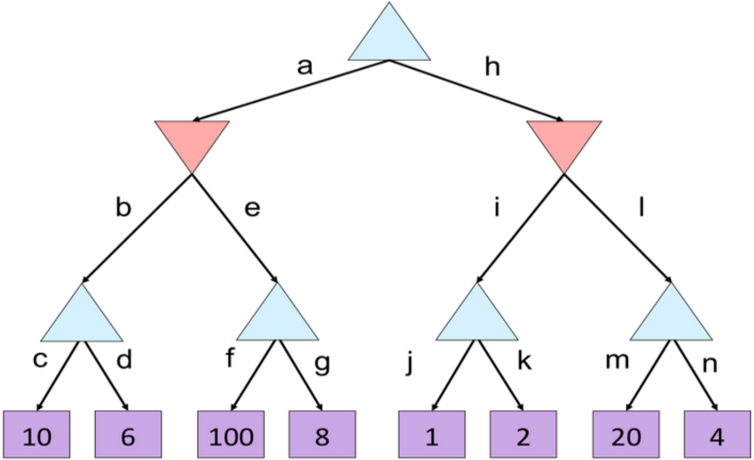
- The pruning has **no effect** on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - The most naïve version won't let you do action selection
    - Solution 1: Prune only on inequality
    - Solution 2: Keep track of which one was first
- 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

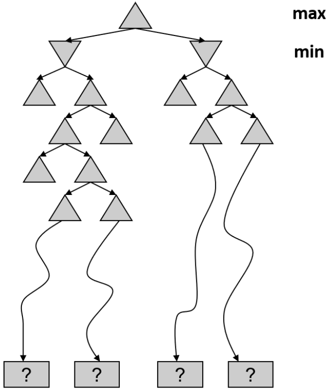


Alpha-Beta Quiz



Resource Limits

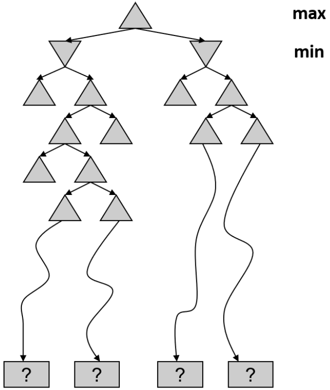
- Problem: In realistic games, cannot search to leaves!



Video: demo-thrashing

Resource Limits

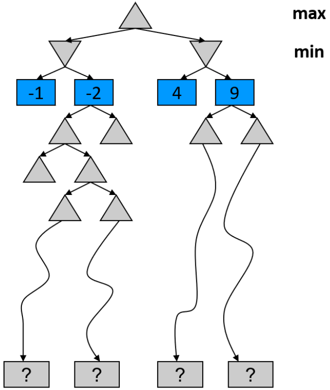
- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions



Video: demo-thrashing

Resource Limits

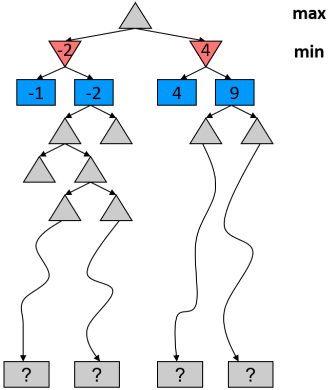
- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions



Video: demo-thrashing

Resource Limits

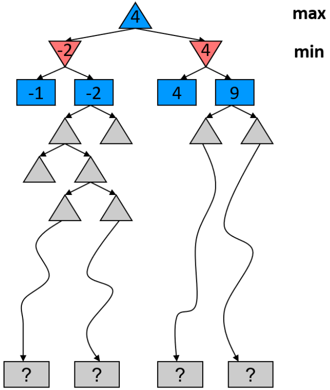
- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions



Video: demo-thrashing

Resource Limits

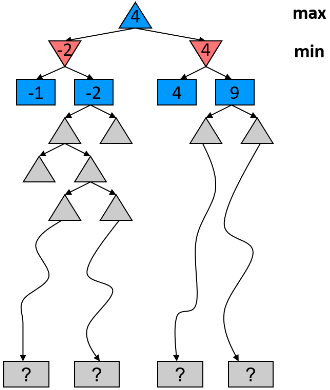
- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions



Video: demo-thrashing

Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
    - Search only to a limited depth in the tree
    - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
    - Suppose we have 100 seconds, can explore 10K nodes/sec
    - Can check 1M nodes per move
    - $\alpha - \beta$ reaches about depth 8 - decent chess program

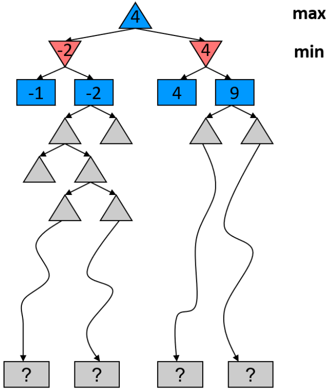


Video: demo-thrashing

Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes/sec
  - Can check 1M nodes per move
  - $\alpha - \beta$ 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

Video: demo-thrashing

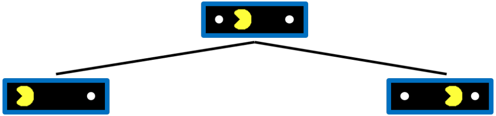


Why Pacman Starves

- 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: demo-thrashing-fixed

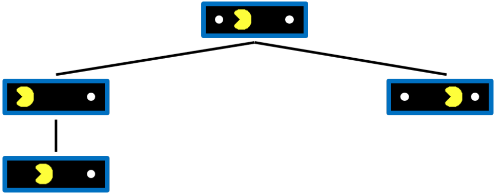
Why Pacman Starves



- 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: demo-thrashing-fixed

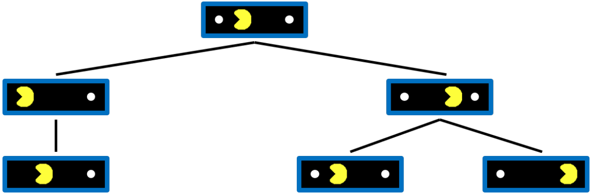
Why Pacman Starves



- 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: demo-thrashing-fixed

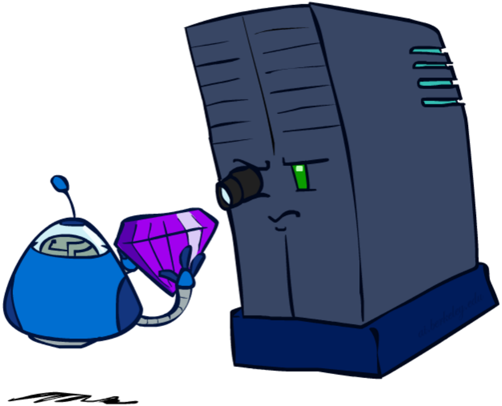
Why Pacman Starves



- 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!

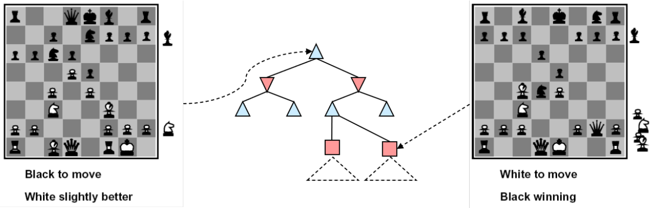
Video: demo-thrashing-fixed

Evaluation Functions



Evaluation Functions

- Used to score non-terminals in depth-limited search

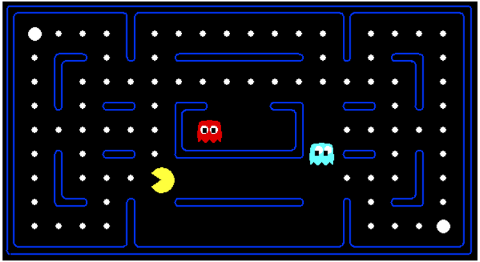


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:
  $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$
- e.g.: $f_1(s)$ = (# of white queens - # of black queens), etc.

Evaluation Function for Pacman

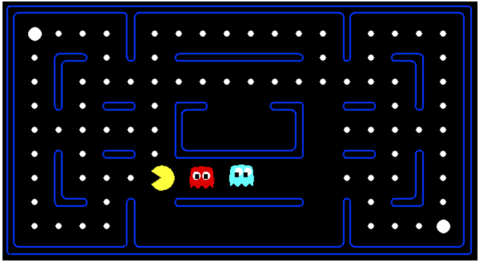
Video: smart-ghosts, smart-ghosts-zoomed

Evaluation Function for Pacman



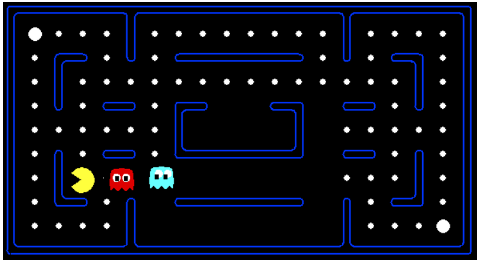
Video: smart-ghosts, smart-ghosts-zoomed

Evaluation Function for Pacman



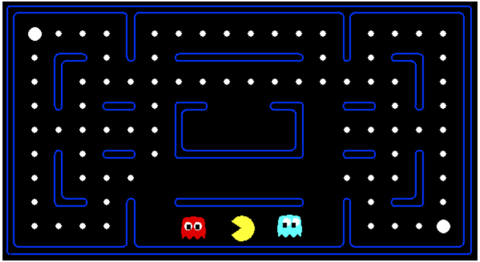
Video: smart-ghosts, smart-ghosts-zoomed

Evaluation Function for Pacman



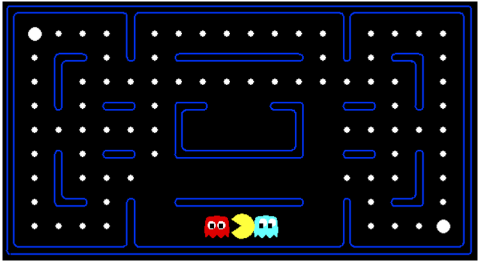
Video: smart-ghosts, smart-ghosts-zoomed

Evaluation Function for Pacman



Video: smart-ghosts, smart-ghosts-zoomed

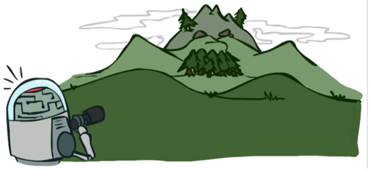
Evaluation Function for Pacman



Video: smart-ghosts, smart-ghosts-zoomed

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



Video: depth-limited-2, depth-limited-10

Suggested Reading

- Russell & Norvig: Chapter 5.2-5.5