Minimax and $\alpha$–$\beta$ Pruning

16 September 2020
Adversarial search
Single-agent game trees
Single-agent game trees
Single-agent game trees
Single-agent game trees
Single-agent game trees
Single-agent game trees

2 0 ... 2 6 ... 4 6

8
Single-agent game 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

For non-terminal state $s$,\n\[ V(s) = \max_{s' \in \text{successors}(s)} V(s') \]

For terminal state $s$,\n\[ V(s) = \text{known} \]
Adversarial game trees
Adversarial game trees
Adversarial game trees
Adversarial game trees
\[ \min(-8, -5) = -8 \]
\[ \min(-10, +8) = -10 \]
\[
\begin{align*}
\min(-8, -5) &= -8 \\
\max(-8, -10) &= -8 \\
\min(-10, +8) &= -10
\end{align*}
\]
\[ \min(-8, -5) = -8 \]
\[ \min(-10, +8) = -10 \]
I want the +8, but if the ghost’s smart, it won’t let me get it. I can force the ghost to take −8 and avoid the −10!
$\min(-8, -5) = -8$

$\max(-8, -10) = -8$

$\min(-10, +8) = -10$

I want the +8, but if the ghost's smart, it won't let me get it. I can force the ghost to take -8 and avoid the -10!
State $s$ under agent's control:

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

State $s$ under opponent's control:

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

Terminal state $s$:

$$V(s) = \text{known}$$
Tic-tac-toe game tree
Terminal nodes
Which successor should Player X pick to maximize its utility?
Which successor should Player X pick to maximize its utility?

For these states, X has no choice!
What should be X’s initial move?
What should be X’s initial move?
Minimax is an adversarial search method for deterministic, zero-sum games, where players alternate turns, with one trying to maximize the result and the other trying to minimize it.

Compute each node’s *minimax value*: the best achievable utility against a rational (optimal) adversary.
Minimax is an adversarial search method for deterministic, zero-sum games, where players alternate turns, with one trying to maximize the result and the other trying to minimize it.

Compute each node’s *minimax value*: the best achievable utility against a rational (optimal) adversary.
Minimax is an adversarial search method for deterministic, zero-sum games, where players alternate turns, with one trying to maximize the result and the other trying to minimize it.

Compute each node’s *minimax value*: the best achievable utility against a rational (optimal) adversary.
Minimax is an adversarial search method for deterministic, zero-sum games, where players alternate turns, with one trying to maximize the result and the other trying to minimize it.

Compute each node’s *minimax value*: the best achievable utility against a rational (optimal) adversary.

![Diagram of a minimax tree](image)
Minimax is an adversarial search method for deterministic, zero-sum games, where players alternate turns, with one trying to maximize the result and the other trying to minimize it.

Compute each node’s *minimax value*: the best achievable utility against a rational (optimal) adversary.
Minimax implementation

```python
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):
    v = -\infty
    for each successor of state:
        v = \max(v, \text{value}(\text{successor}))
    return v

def min-value(state):
    v = +\infty
    for each successor of state:
        v = \min(v, \text{value}(\text{successor}))
    return v

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)
Minimax properties

Optimal against a perfect player. Otherwise?
Minimax properties

Optimal against a perfect player. Otherwise?
Minimax properties

Optimal against a perfect player. Otherwise?
SCORE: 0
Minimax properties

Optimal against a perfect player. Otherwise?
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?
Game tree pruning
Minimax example
Minimax example
Minimax example
Minimax example
Minimax example
Minimax example
Minimax example
Minimax example

```
    max
     /\n    /  \
min 3
    |  /  \
   | /    \
3 12 8 2
```
Minimax example

```
    max
     / \  
   min 3
  / | |
3 12 8 2 4
```
Minimax example
Minimax example
Minimax example
Minimax example

```
Minimax tree:

Max
  /  
/min
  /  
3   2

3 12 8 2 4 6 14
```
Minimax example
Minimax example
Minimax example
Minimax example
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
Minimax pruning
\(\alpha-\beta\) implementation

\(\alpha:\) MAX's best option on path to root
\(\beta:\) MIN's best option on path to root

```python
def max-value(state, α, β):
    v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
    if v ≥ β: return v
    α = max(α, v)
    return v

def min-value(state, α, β):
    v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
    if v ≤ α: return v
    β = min(β, v)
    return v
```
**α–β pruning properties**

This 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
- So the most naïve version won’t let you do action selection

Good child ordering improves effectiveness of pruning

With “perfect ordering”: 
\(\alpha-\beta\) pruning properties

This pruning has \textit{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

So the most naïve version won’t let you do action selection

Good child ordering improves effectiveness of pruning

With “perfect ordering”:

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

Doubles solvable depth!

Full search of, e.g. chess, is still hopeless…
Resource limits
Resource limits

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

**Solution:** Depth-limited minimax search
Resource limits

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

**Solution:** *Depth-limited minimax* search

Instead, 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
Resource limits

$\textit{Problem}$: In realistic games, cannot search to leaves!

$\textit{Solution}$: \textit{Depth-limited minimax} search

Instead, search only to a limited depth in the tree
Replace terminal utilities with an \textit{evaluation function} for non-terminal positions

$\textit{Example}$:

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

Guarantee of optimal play is gone

Use iterative deepening for an \textit{anytime algorithm}
Why Pac-Man 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 Pac-Man 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 Pac-Man 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 Pac-Man 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!
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:

\[ \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) \]

e.g., \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Evaluation for Pac-Man
Evaluation for Pac-Man
Evaluation for Pac-Man
Evaluation for Pac-Man
Evaluation for Pac-Man
SCORE: 0
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.

This is an important example of the tradeoff between complexity of features and complexity of computation.
Acknowledgments

The lecture incorporates material from:

Pieter Abbeel, Dan Klein, et al., University of California, Berkeley; ai.berkeley.edu
Hank Feild, Endicott College
Ketrina Yim (illustrations)