CS 5522: Artificial Intelligence II

Adversarial Search

Instructor: Alan Ritter
Ohio State University

[These slides were adapted from CS188 Intro to AI at UC Berkeley. All materials available at http://ai.berkeley.edu.]
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. 2007: Checkers solved!
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: Checkers solved!
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: Checkers solved!

- **Chess**: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very 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

- **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: Checkers solved!

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

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very 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 now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very 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 now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.
Game Playing State-of-the-Art

- **Checkers:**
  1950: First computer player.
  2007: Checkers solved!

- **Chess:**
  Deep Blue examined 200M positions per second, used very 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 now starting to be challenged by machines, though the best humans still beat the best machines.
  Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.
**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: Checkers solved!

- **Chess**: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very 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 now starting to be challenged by machines, though the best humans still beat the best machines. In go, $b > 300$! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

- **Pacman**

Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very 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 now starting to be challenged by machines, though the best humans still beat the best machines. In go, b > 300! Classic programs use pattern knowledge bases, but big recent advances use Monte Carlo (randomized) expansion methods.

Pacman
Behavior from Computation

[Demo: mystery pacman (L6D1)]
Adversarial Games
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

- Many possible formalizations, one is:
  - States: $S$ (start at $s_0$)
  - Players: $P=${1…$N$} (usually take turns)
  - Actions: $A$ (may depend on player / state)
  - Transition Function: $SxA \rightarrow S$
  - Terminal Test: $S \rightarrow \{t,f\}$
  - Terminal Utilities: $SxP \rightarrow R$

- 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

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

8
Value of a State
Value of a state:
The best achievable outcome (utility) from that state
Value of a State

The best achievable outcome (utility) from that state

Terminal States: $V(s) = \text{known}$
Value of a state:
The best achievable outcome (utility) from that state

Non-Terminal States:
\[ V(s) = \max_{s' \in \text{children}(s)} V(s') \]

Terminal States:
\[ V(s) = \text{known} \]
Adversarial Game Trees
Adversarial Game Trees
Adversarial Game Trees
Adversarial Game Trees
Adversarial Game Trees

-20  -8  ...  -18  -5  ...  -10  +4

-20  +8
Minimax Values

-8  -5  -10  +8
Minimax Values

Terminal States:

\[ V(s) = \text{known} \]
Minimax Values

States Under Opponent’s Control:

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

Terminal States:

\[ V(s) = \text{known} \]
Minimax Values

States Under Agent’s Control:

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

States Under Opponent’s Control:

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

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
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

Terminal values: part of the game
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 values: computed recursively

Terminal values: part of the game
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 values: computed recursively](image)

- Terminal values: part of the game
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 values: computed recursively

Terminal values: part of the game
def max-value(state):
    initialize $v = -\infty$
    for each successor of state:
        $v = \max(v, \text{min-value}(\text{successor}))$
    return $v$
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v

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

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

Min-Value Function:

```
def min-value(state):
    initialize v = +\infty
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```
Minimax Implementation

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

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

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

def min_value(state):
    initialize \( v = +\infty \)
    for each successor of state:
        \( v = \min(v, \max_value(\text{successor})) \)
    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 Implementation (Dispatch)

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

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

Optimal against a perfect player. Otherwise?

[Demo: min vs exp (L6D2, L6D3)]
Minimax Properties

Optimal against a perfect player. Otherwise?

[Demo: min vs exp (L6D2, L6D3)]
Minimax Properties

Optimal against a perfect player. Otherwise?

[Demo: min vs exp (L6D2, L6D3)]
Video of Demo Min vs. Exp (Min)
Video of Demo Min vs. Exp (Min)
Video of Demo Min vs. Exp (Exp)
Video of Demo Min vs. Exp (Exp)
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!
- **Solution:** Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
Resource Limits

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

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

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

- **Solution:** Depth-limited 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
  - So can check 1M nodes per move
  - α-β reaches about depth 8 - decent chess program
Resource Limits

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited 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
  - So 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
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 of Demo Limited Depth (2)
Video of Demo Limited Depth (2)
Video of Demo Limited Depth (10)
Video of Demo Limited Depth (10)
Video of Demo Limited Depth (10)

SCORE: -505
Evaluation Functions
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:

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

- e.g. \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Evaluation for Pacman

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate]
Evaluation for Pacman

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate]
Evaluation for Pacman

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate]
Evaluation for Pacman

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate]
Evaluation for Pacman

[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate]
Video of Demo Thrashing (d=2)
Video of Demo Thrashing (d=2)
Video of Demo Thrashing (d=2)
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!
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!
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 Smart Ghosts (Coordination)
Video of Demo Smart Ghosts (Coordination)
Video of Demo Smart Ghosts (Coordination)
Game Tree Pruning
Minimax Example
Minimax Example
Minimax Example
Minimax Example
Minimax Example

[Diagram of a minimax tree with node values 3, 12, 8]
Minimax Example
Minimax Example

```
3  12  8  2
```
Minimax Example
Minimax Example
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
Alpha-Beta Pruning

- General configuration (MIN version)
  - We’re computing the MIN-VALUE at some node $n$
  - We’re looping over $n$’s children
  - $n$’s estimate of the childrens’ 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

```python
def max_value(state, α, β):
    initialize 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):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
    β = min(β, v)
    return v
```
- 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
Alpha-Beta 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”:
  - 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)