CS 188: Artificial Intelligence

Adversarial Search

Prof. Yuke Zhu
The University of Texas at Austin

[These slides are based on those of Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]
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**: 2016: AlphaGo, created by Google DeepMind beat 9-dan professional Go player Lee Sedol 4-1 on a full sized 19 x 19 board. AlphaGo combined Monte Carlo Tree Search with deep neural networks, improving via reinforcement learning through self-play.

- **OpenAI Five (DOTA)**: getting close to world-class
How to consider behavior of ghosts?
Adversarial Games
Types of 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: \( S \times A \rightarrow S \)
  - 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)
  - Let's 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 a State

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

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

Minimax values: computed recursively

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

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

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

V(s') = \min_{s \in \text{successors}(s')} V(s)
Minimax 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 = -∞
    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 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?
Minimax vs Expectimax (Min)

End your misery!
Minimax vs Expectimax (Exp)

Hold on to hope, Pacman!
Game Tree Pruning
Minimax Example
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 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

\[\alpha: \text{MAX's best option on path to root}\]
\[\beta: \text{MIN's best option on path to root}\]

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

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

- **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)
Alpha-Beta Quiz

Diagram of a game tree with the following values:

- Root: 8
- Left child of root: 8
  - Left child of 8: 10
  - Right child of 8: 8
- Right child of root: 4
  - Left child of 4: 4
  - Right child of 4: 50
- Edge from root to left child: a
- Edge from root to right child: d
- Edge from left child of root to 10: b
- Edge from left child of root to 8: c
- Edge from right child of root to 4: e
- Edge from right child of root to 50: f

The diagram illustrates the alpha-beta pruning algorithm, where nodes are pruned early in the search to optimize the decision-making process.
Alpha-Beta Quiz 2

Diagram:

- a
- b
- c
- d
- e
- f
- g
- h
- i
- j
- k
- l
- m
- n

Values:
- 10
- 10
- 100
- 100
- 2
- 2
- 1
- 2
- 20
- 4
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 (10)
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.
Thrashing (d=2)

Evaluation function: Score
A danger of replanning agents!
- He knows his score will go up by eating the dot now (left, right)
- He knows his score will go up just as much by eating the dot later (right, right)
- 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!
Thrashing -- Fixed (d=2)

Evaluation function: Score + proximity to nearest dot
Smart ghosts — implicit coordination

Evaluation function: proximity to Pacman
Next Time: Uncertainty!