CSE 473: Artificial Intelligence

Adversarial Search
Dan Weld

Based on slides from
Dan Klein, Stuart Russell, Pieter Abbeel, Andrew Moore and Luke Zettlemoyer
(best illustrations from ai.berkeley.edu)

Outline

- Adversarial Search
  - Minimax search
  - α-β search
  - Evaluation functions
  - Expectimax

- Reminder:
  - Project 2 due in 7 days
**Game Playing State-of-the-Art**

**1994: Checkers.** Chinook ended 40-year-reign of human world champion Marion Tinsley. Used search plus an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions. Checkers is now solved!

![Checkers Game](image)

**Game Playing State-of-the-Art**

**1997: Chess.** Deep Blue defeated human world champion Gary Kasparov in a six-game match. Deep Blue examined 200 million 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 Game](image)
Game Playing State-of-the-Art

**Go:** $b > 300!$ Programs use Monte Carlo tree search + pattern KBs

2015: AlphaGo beats European Go champion Fan Hui (2 dan) 5-0
2016: AlphaGo beats Lee Sedol (9 dan) 4-1

Game Playing State-of-the-Art

**Othello:** Human champions refuse to compete against computers.
Game Playing State-of-the-Art

- Pacman: ... unknown ...

Types of Games

<table>
<thead>
<tr>
<th></th>
<th>deterministic</th>
<th>chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfect information</td>
<td>chess, checkers, go, othello</td>
<td>backgammon, monopoly</td>
</tr>
<tr>
<td>imperfect information</td>
<td>stratego</td>
<td>bridge, poker, scrabble, nuclear war</td>
</tr>
</tbody>
</table>

Number of Players? 1, 2, ...?
### 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*: $S \rightarrow A$

### Previously: Single-Agent Trees

Slide from Dan Klein & Pieter Abbeel - ai.berkeley.edu
Previously: 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

Value(interior) \leftarrow \text{diffrent}

Value(leaves) \leftarrow \text{same}
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:
\[ V(s) = \text{known} \]

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

def max-value(state):
    if leaf?(state), return U(state)
    initialize v = +∞
    for each c in children(state)
        v = max(v, min-value(c))
    return v

def min-value(state):
    if leaf?(state), return U(state)
    initialize v = -∞
    for each c in children(state)
        v = min(v, max-value(c))
    return v

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

Concrete Minimax Example

max

min
Minimax Example

Max:

Min:

Quiz
Minimax Properties

- Optimal?
  - Yes, against perfect player. Otherwise?

- Time complexity?
  - $O(b^m)$

- Space complexity?
  - $O(bm)$

- For chess, $b \sim 35$, $m \sim 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
Do We Need to Evaluate Every Node?

Max:

Min:

Progress of search...
Pruning Example

Progress of search...

- **α-β Pruning**
  - **General configuration**
    - **α** is MAX’s best choice on path to root
    - If $n$ becomes worse than **α**, MAX will avoid it, so can stop considering $n$’s other children
    - Define $β$ similarly for MIN

![Diagram of α-β Pruning example](image)
Min-Max Implementation

```python
def min_val(state):
    if leaf?(state), return U(state)
    initialize v = +\infty
    for each c in children(state):
        v = min(v, max_val(c))
    return v

def max_val(state):
    if leaf?(state), return U(state)
    initialize v = -\infty
    for each c in children(state):
        v = max(v, min_val(c))
    return v
```

Alpha-Beta Implementation

```python
def min_val(state, \alpha, \beta):
    if leaf?(state), return U(state)
    initialize v = -\infty
    for each c in children(state):
        v = max(v, min_val(c, \alpha, \beta))
    return v

def max_val(state, \alpha, \beta):
    if leaf?(state), return U(state)
    initialize v = +\infty
    for each c in children(state):
        v = min(v, max_val(c, \alpha, \beta))
    return v
```

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

Slide adapted from Dan Klein & Pieter Abbeel - ai.berkeley.edu
def min-val(state, α, β):
    if leaf?(state), return U(state)
    initialize v = $\infty$
    for each c in children(state):
        v = min(v, max-val(c, α, β))
    if v ≤ α return v
    β = min(β, v)
    return v

def max-val(state, α, β):
    if leaf?(state), return U(state)
    initialize v = $-\infty$
    for each c in children(state):
        v = max(v, min-val(c, α, β))
    if v ≥ β return v
    α = max(α, v)
    return v

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

Slide adapted from Dan Klein & Pieter Abbeel - ai.berkeley.edu