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 1 due Today

Types of Games

<table>
<thead>
<tr>
<th>deterministic</th>
<th>chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfect information</td>
<td>chess, checkers, go, othello</td>
</tr>
<tr>
<td>imperfect information</td>
<td>backgammon, monopoly</td>
</tr>
<tr>
<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 s₀)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: S x A → S
  - Terminal Test: S → {t,f}
  - Terminal Utilities: S x P → R

- Solution for a player is a policy: S → A

Previously: Single-Agent Trees

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

\[ V(s) = \max_{a \in A(s)} V(s') \]

Previously: Value of a State

Non-Terminal States:

Terminal States:

8

Slide from Dan Klein & Pieter Abbeel - ai.berkeley.edu
Adversarial Game Trees

Minimax Values

States Under Agent’s Control:
\[ V(s) = \max_{s’(\text{outcome}(s))} V(s') \]

States Under Opponent’s Control:
\[ V(s') = \min_{s''(\text{outcome}(s'))} V(s) \]

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

Do We Need to Evaluate Every Node?

α-β Pruning Example
**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

---

**α-β Implementation**

- **α**: MAX’s best option on path to root
- **β**: MIN’s best option on path to root

**Function Definitions**

```python
def max_val(state, α, β):
    initialize v = -∞
    for each c in children(state):
        v = max(v, min_val(child, α, β))
    if v ≥ β return v
    α = max(α, v)
    return v

def min_val(state, α, β):
    initialize v = +∞
    for each c in children(state):
        v = min(v, max_val(state, α, β))
    if v ≤ α return v
    β = min(β, v)
    return v
```

---

**Alpha-Beta Pruning Example**

At max node:
- Prune if v ≥ β
- Update α

At min node:
- Prune if v ≤ α
- Update β

---

**Alpha-Beta Pruning Properties**

- This pruning has no effect on final result at the root
- Values of intermediate nodes might be wrong!
  - but, they are bounds
- 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...

---

**Alpha-Beta Quiz**

- ![Diagram of a tree with nodes labeled with numbers 10, 8, 4, 50 and nodes a, b, c, d, e, f]

---

**Alpha-Beta Quiz 2**

- ![Diagram of a tree with nodes labeled with numbers 10, 8, 4, 50 and nodes a, b, c, d, e, f, g, h, i, j, k, l, m, n]
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

Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.
   ....and so on.

Why do we want to do this for multiplayer games?

Heuristic Evaluation Function

- Function which scores non-terminals
- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:
  - e.g., \( f(s) \) = (num white queens - num black queens), etc.

Evaluation for Pacman

What features would be good for Pacman?

\[ \text{Eval}(s) = w_1f_1(s) + w_2f_2(s) + \ldots + w_nf_n(s) \]

Which algorithm?

\( \alpha - \beta \), depth 4, simple eval fun
Which algorithm?

$\alpha$-$\beta$, depth 4, better eval fun

Why Pacman Starves

- He knows his score will go up by eating the dot now
- He knows his score will go up just as much by eating the dot later on
- There are no point-scoring opportunities after eating the dot
- Therefore, waiting seems just as good as eating