Adversarial Search

Hanna Hajishirzi

Based on slides from Dan Klein, Luke Zettlemoyer
Many slides over the course adapted from either Stuart Russell or Andrew Moore
Announcements

- PS2 is due Jan 30
- Final poster session date
  - New poll posted
  - Best time so far: Thu, March 14 12-2pm.
Outline

- Games
  - Review: Minimax search
  - $\alpha$-$\beta$ search
  - Evaluation functions
  - Expectimax search
  - Complex Games
Game Playing

- Many different kinds of games!
- Want algorithms for calculating a strategy (policy) which recommends a move in each state

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

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

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

- States: S (start at \(s_0\))
- 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
Deterministic Single-Player

- Deterministic, single player, perfect information:
  - Know the rules, action effects, winning states
  - E.g. Freecell, 8-Puzzle, Rubik’s cube
- … it’s just search!

- Slight reinterpretation:
  - Each node stores a value: the best outcome it can reach
  - This is the maximal outcome of its children (the max value)
  - Note that we don’t have path sums as before (utilities at end)
- After search, can pick move that leads to best node
Adversarial Game Trees

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:

\[ V(s) = \text{known} \]
Deterministic Two-Player

- E.g. tic-tac-toe, chess, checkers
- Zero-sum games
  - Agents have opposite utilities
  - One player maximizes result
  - The other minimizes result

**Minimax search**
- A state-space search tree
- Players alternate
- Choose move to position with highest **minimax value** = best achievable utility against best play
Tic-tac-toe Game Tree

MAX (X)

MIN (O)

MAX (X)

MIN (O)

TERMINAL

Utility

-1
0
+1
Minimax Example
Minimax Search

function **Max-Value**(*state*) returns a utility value
  if **Terminal-Test**(*state*) then return **Utility**(*state*)
  \( v \leftarrow -\infty \)
  for \( a, s \) in **Successors**(*state*) do \( v \leftarrow \text{Max}(v, \text{Min-Value}(s)) \)
  return \( v \)

function **Min-Value**(*state*) returns a utility value
  if **Terminal-Test**(*state*) then return **Utility**(*state*)
  \( v \leftarrow \infty \)
  for \( a, s \) in **Successors**(*state*) do \( v \leftarrow \text{Min}(v, \text{Max-Value}(s)) \)
  return \( v \)
Minimax Properties

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

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

- Space complexity?
  - $O(bm)$

- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
Can we do better?
$\alpha$-$\beta$ Pruning Example

![Pruning Example Diagram]
α-β Pruning

- General configuration
  - α is the best value that MAX can get at any choice point along the current path
  - If \( n \) becomes worse than \( \alpha \), MAX will avoid it, so can stop considering \( n \)'s other children
  - Define \( \beta \) similarly for MIN: value of the best (lowest value) choice along the current path for MIN
Alpha-Beta Pruning Example

α is MAX’s best alternative here or above
β is MIN’s best alternative here or above
Min-Max Implementation

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

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

Slide adapted from Dan Klein & Pieter Abbeel - ai.berkeley.edu
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-val(state, \( \alpha \), \( \beta \)):**
- if leaf?(state), return \( U(state) \)
- initialize \( v = -\infty \)
- for each \( c \) in children(state):
  - \( v = \max(v, \text{min-val}(c, \alpha, \beta)) \)
- return \( v \)

**def min-val(state , \( \alpha \), \( \beta \)):**
- if leaf?(state), return \( U(state) \)
- initialize \( v = +\infty \)
- for each \( c \) in children(state):
  - \( v = \min(v, \text{max-val}(c, \alpha, \beta)) \)
- return \( v \)
def min-val(state, α, β):
    if leaf?(state), return U(state)
    initialize v = -∞
    for each c in children(state):
        v = min(v, min-val(c, α, β))
        if v ≤ β return v
    α = max(α, v)
    β = min(β, v)
    return v

def max-val(state, α, β):
    if leaf?(state), return U(state)
    initialize v = +∞
    for each c in children(state):
        v = max(v, max-val(c, α, β))
        if v ≥ β return v
    α = max(α, v)
    return v
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…
Resource Limits

- Cannot search to leaves
- Depth-limited search
  - Instead, search a limited depth of tree
  - Replace terminal utilities with an eval function for non-terminal positions
    - e.g., \( \alpha-\beta \) reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- Evaluation function matters
  - It works better when we have a greater depth look ahead
Depth Matters
Depth Matters

SCORE: 0

depth 10
Evaluation Functions

- Function which scores non-terminals

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

- Ideal function: returns the utility of the position
- In practice: typically weighted linear sum of features:
  - e.g. \( f_1(s) = (\text{num white queens} - \text{num black queens}) \), etc.
Bad Evaluation Function
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.
What features would be good for Pacman?

\[ \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]
Evaluation Function
Which algorithm?

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

$\alpha$-$\beta$, depth 4, better eval fun
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
Synergies between alpha-beta and evaluation function

- **Alpha-Beta:** amount of pruning depends on expansion ordering
  - Evaluation function can provide guidance to expand most promising nodes first

- **Alpha-beta:**
  - Similar for roles of mini-max swapped
  - Value at a min-node will only keep going down
  - Once value of min-node lower than better option for max along path to root, can prune
  - Hence, IF evaluation function provides upper-bound on value at min-node, and upper-bound already lower than better option for max along path to root THEN can prune