Minimax and Alpha-Beta Reduction

Borrows from Spring 2006 CS 440 Lecture Slides
Motivation

- Want to create programs to play games
- Want to play optimally
- Want to be able to do this in a reasonable amount of time
# Types of Games

<table>
<thead>
<tr>
<th>Deterministic</th>
<th>Nondeterministic (Chance)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess, Checkers, Go, Othello</td>
<td>Backgammon, Monopoly</td>
</tr>
<tr>
<td>Battleship</td>
<td>Card Games</td>
</tr>
</tbody>
</table>

Minimax is for deterministic, fully observable games.
Basic Idea

• Search problem
  – Searching a tree of the possible moves in order to find the move that produces the best result
  – Depth First Search algorithm
• Assume the opponent is also playing optimally
  – Try to guarantee a win anyway!
Required Pieces for Minimax

• An initial state
  – The positions of all the pieces
  – Whose turn it is

• Operators
  – Legal moves the player can make

• Terminal Test
  – Determines if a state is a final state

• Utility Function
Utility Function

• Gives the utility of a game state
  – $\text{utility(State)}$

• Examples
  – -1, 0, and +1, for Player 1 loses, draw, Player 1 wins, respectively
  – Difference between the point totals for the two players
  – Weighted sum of factors (e.g. Chess)
    • $\text{utility}(S) = w_1 f_1(S) + w_2 f_2(S) + \ldots + w_n f_n(S)$
      – $f_1(S) = (\text{Number of white queens}) - (\text{Number of black queens})$, $w_1 = 9$
      – $f_2(S) = (\text{Number of white rooks}) - (\text{Number of black rooks})$, $w_2 = 5$
      – ...
Two Agents

• MAX
  – Wants to maximize the result of the utility function
  – Winning strategy if, on MIN's turn, a win is obtainable for MAX for all moves that MIN can make

• MIN
  – Wants to minimize the result of the evaluation function
  – Winning strategy if, on MAX's turn, a win is obtainable for MIN for all moves that MAX can make
Basic Algorithm

function **MINIMAX-DECISION**(*state*) returns an action
inputs: *state*, current state in game
return the *a* in **ACTIONS**(*state*) maximizing **MIN-VALUE**(**RESULT**( *a*, *state*) )

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 \( v \)
Example

- Coins game
  - There is a stack of N coins
  - In turn, players take 1, 2, or 3 coins from the stack
  - The player who takes the last coin loses
Coins Game: Formal Definition

- **Initial State:** The number of coins in the stack
- **Operators:**
  1. Remove one coin
  2. Remove two coins
  3. Remove three coins
- **Terminal Test:** There are no coins left on the stack
- **Utility Function:** $F(S)$
  - $F(S) = 1$ if MAX wins, $0$ if MIN wins
\[ \text{Solution} \]

\[
\begin{array}{c|c}
\text{MAX} & \text{MIN} \\
\hline
N = 4 & K = 1 \\
N = 3 & K = 0 \\
N = 2 & K = 0 \\
N = 1 & K = 1 \\
N = 0 & K = 1 \\
\end{array}
\]
Analysis

- Max Depth: 5
- Branch factor: 3
- Number of nodes: 15
- Even with this trivial example, you can see that these trees can get very big
  - Generally, there are \( O(b^d) \) nodes to search for
    - Branch factor \( b \): maximum number of moves from each node
    - Depth \( d \): maximum depth of the tree
- Exponential time to run the algorithm!
- How can we make it faster?
Alpha-Beta Pruning

• Main idea: Avoid processing subtrees that have no effect on the result
• Two new parameters
  – $\alpha$: The best value for MAX seen so far
  – $\beta$: The best value for MIN seen so far
• $\alpha$ is used in MIN nodes, and is assigned in MAX nodes
• $\beta$ is used in MAX nodes, and is assigned in MIN nodes
Alpha-Beta Pruning

• MAX (Not at level 0)
  - If a subtree is found with a value $k$ greater than the value of $\beta$, then we do not need to continue searching subtrees
    • MAX can do at least as good as $k$ in this node, so MIN would never choose to go here!

• MIN
  - If a subtree is found with a value $k$ less than the value of $\alpha$, then we do not need to continue searching subtrees
    • MIN can do at least as good as $k$ in this node, so MAX would never choose to go here!
Algorithm

function **Alpha-Beta-Decision**(*state*) returns an action
    return the *a* in \text{Actions}(*state*) maximizing \text{Min-Value}(*Result(a, state)*)

function **Max-Value**(*state*, α, β) returns a utility value
    \textbf{inputs}: *state*, current state in game
    \hspace{1em} α, the value of the best alternative for \text{Max} along the path to *state*
    \hspace{1em} β, the value of the best alternative for \text{Min} along the path to *state*
    \textbf{if} Terminal-Test(*state*) \textbf{then} return Utility(*state*)
    \hspace{1em} \( v \leftarrow -\infty \)
    \textbf{for} *a*, s in Successors(*state*) \textbf{do}
        \hspace{1em} \( v \leftarrow \text{Max}(v, \text{Min-Value}(s, \alpha, \beta)) \)
        \hspace{1em} \textbf{if} \( v \geq \beta \) \textbf{then} return \( v \)
        \hspace{1em} \alpha \leftarrow \text{Max}(\alpha, v)
    \hspace{1em} return \( v \)

function **Min-Value**(*state*, α, β) returns a utility value
    \text{same as Max-Value but with roles of α, β reversed}
\[
\begin{align*}
\text{MAX} & \quad \text{MIN} \\
N = 4 & \quad \alpha = \\
K = & \quad \beta = \\
N = 3 & \quad \alpha = \\
K = & \quad \beta = \\
N = 2 & \quad \alpha = \\
K = & \quad \beta = \\
N = 1 & \quad \alpha = \\
K = & \quad \beta = \\
N = 0 & \quad \alpha = \\
K = & \quad \beta = \\
F(S) & = 0 \\
\end{align*}
\]
Nondeterministic Games

• Minimax can also be used for nondeterministic games (those that have an element of chance)
• There is an additional node added (Random node)
• Random node is between MIN and MAX (and vice versa)
• Make subtrees over all of the possibilities, and average the results
Example

N = 2
K = 8.6

Random Node

K = 0.4*5 + 0.6*11 = 8.6

K = 0.4*2 + 0.6*7 = 5

Weighted coin
0.6 Heads (1)
0.4 Tails (0)
Our Project

• We will focus on deterministic, two-player, fully observable games
• We will be trying to learn the evaluator function, in order to save time when playing the game
  – Training on data from Minimax runs (Neural Network)
  – Having the program play against itself (Genetic Algorithms)
Conclusion

- Minimax finds optimal play for deterministic, fully observable, two-player games
- Alpha-Beta reduction makes it faster