Adversarial Search

CS151
David Kauchak
Fall 2010

Some material borrowed from:
Sara Owsley Sood and others
Admin

• Reading?

• Assignment 2
  – On the web page
  – 3 parts
  – Anyone looking for a partner?
  – Get started!

• Written assignments
  – Post solutions to W1 today
  – Post next written assignment soon
A quick review of search

• Rational thinking via search – determine a plan of actions by searching from starting state to goal state

• Uninformed search vs. informed search
  – what’s the difference?
  – what are the techniques we’ve seen?
  – pluses and minuses?

• Heuristic design
  – admissible?
  – dominant?
Why should we study games?

- Clear success criteria
- Important historically for AI
- Fun 😊
- Good application of search
  - hard problems (chess $35^{100}$ nodes in search tree, $10^{40}$ legal states)
- Some real-world problems fit this model
  - game theory (economics)
  - multi-agent problems
What are some of the games you’ve played?
Types of games: game properties

- single-player vs. 2-player vs. multiplayer
- Fully observable (perfect information) vs. partially observable
- Discrete vs. continuous
- real-time vs. turn-based
- deterministic vs. non-deterministic (chance)
Strategic thinking $\equiv$ intelligence

For reasons previously stated, two-player games have been a focus of AI since its inception…

Begs the question: Is strategic thinking the same as intelligence?
Strategic thinking ≠ intelligence

Humans and computers have different relative strengths in these games:

Humans:
good at evaluating the strength of a board for a player

Computers:
good at looking ahead in the game to find winning combinations of moves
Strategic thinking = intelligence

How could you figure out how humans approach playing chess?

humans

good at evaluating the strength of a board for a player
How humans play games…

An experiment (by deGroot) was performed in which chess positions were shown to novice and expert players…

- experts could reconstruct these perfectly
- novice players did far worse…
An experiment (by deGroot) was performed in which chess positions were shown to novice and expert players...

- experts could reconstruct these perfectly
- novice players did far worse...

Random chess positions (not legal ones) were then shown to the two groups

- experts and novices did just as badly at reconstructing them!
People are still working on this problem…

http://people.brunel.ac.uk/~hsstfg/FRG-research/chess_expertise/
Tic Tac Toe as search

How can we pose this as a search problem?
Tic Tac Toe as search
Tic Tac Toe as search
Tic Tac Toe as search

Eventually, we’ll get to a leaf

The **UTILITY** of a state tells us how good the states are.
Defining the problem

- INITIAL STATE – board position and the player whose turn it is
- SUCCESSOR FUNCTION – returns a list of (move, next state) pairs
- TERMINAL TEST – is game over? Are we in a terminal state?
- UTILITY FUNCTION – (objective or payoff func) gives a numeric value for terminal states (ie – chess – win/lose/draw +1/-1/0, backgammon +192 to -192)
Games’ Branching Factors

- On average, there are ~35 possible moves that a chess player can make from any board configuration…

```
<table>
<thead>
<tr>
<th>Game</th>
<th>Branching Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tic-tac-toe</td>
<td>4</td>
</tr>
<tr>
<td>Connect Four</td>
<td>7</td>
</tr>
<tr>
<td>Checkers</td>
<td>10</td>
</tr>
<tr>
<td>Othello</td>
<td>30</td>
</tr>
<tr>
<td>Chess</td>
<td>40</td>
</tr>
<tr>
<td>Go</td>
<td>300</td>
</tr>
</tbody>
</table>
```

Hydra at home in the United Arab Emirates…

18 Ply!!
Games’ Branching Factors

- On average, there are ~35 possible moves that a chess player can make from any board configuration...

<table>
<thead>
<tr>
<th>Branching Factor Estimates for different two-player games</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tic-tac-toe</td>
</tr>
<tr>
<td>Connect Four</td>
</tr>
<tr>
<td>Checkers</td>
</tr>
<tr>
<td>Othello</td>
</tr>
<tr>
<td>Chess</td>
</tr>
<tr>
<td>Go</td>
</tr>
</tbody>
</table>

Boundaries for qualitatively different games...
Games’ Branching Factors

- On average, there are ~35 possible moves that a chess player can make from any board configuration…

<table>
<thead>
<tr>
<th>Branching Factor Estimates for different two-player games</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tic-tac-toe</td>
</tr>
<tr>
<td>Connect Four</td>
</tr>
<tr>
<td>Checkers</td>
</tr>
<tr>
<td>Othello</td>
</tr>
<tr>
<td>Chess</td>
</tr>
<tr>
<td>Go</td>
</tr>
</tbody>
</table>

CHINOOK (2007)

“solved” games

computer-dominated

human-dominated

computer-dominated

human-dominated
Games vs. search problems?

- **Opponent!**
  - unpredictable/uncertainty
  - deal with opponent strategy

- **Time limitations**
  - must make a move in a reasonable amount of time
  - can’t always look to the end

- **Path costs**
  - not about moves, but about UTILITY of the resulting state/winning
Back to Tic Tac TOe

X’s turn

O’s turn

X’s turn

...
I’m X, what will ‘O’ do?

O’s turn
Minimizing risk

• The computer doesn’t know what move O (the opponent) will make
• It can assume, though, that it will try and make the best move possible
• Even if O actually makes a different move, we’re no worse off
Optimal Strategy

- An **Optimal Strategy** is one that is at least as good as any other, no matter what the opponent does
  - If there's a way to force the win, it will
  - Will only lose if there's no other option
How can X play optimally?
How can X play optimally?

- Start from the leaves and propagate the utility up:
  - if X’s turn, pick the move that maximizes the utility
  - if O’s turn, pick the move that minimizes the utility

Is this optimal?
Minimax Algorithm: An Optimal Strategy

minimax(state) =
    - if state is a terminal state
      Utility(state)
    - if MAX’s turn
      return the maximum of minimax(…)
on all successors of current state
    - if MIN’s turn
      return the minimum of minimax(…)
on all successors to current state

• Uses recursion to compute the “value” of each state
• Proceeds to the leaves, then the values are “backed up” through the tree as the recursion unwinds
• What type of search is this?
• What does this assume about how MIN will play? What if this isn’t true?
```python
def minimax(state):
    for all actions a in actions(state):
        return the a with the largest minValue(result(state,a))

def maxValue(state):
    if state is terminal:
        return utility(state)
    else:
        # return the a with the largest minValue(result(state,a))
        value = -\infty
        for all actions a in actions(state):
            value = max(value, minValue(result(state,a))
        return value

def minValue(state):
    if state is terminal:
        return utility(state)
    else:
        # return the a with the smallest maxValue(result(state,a))
        value = +\infty
        for all actions a in actions(state):
            value = min(value, maxValue(result(state,a))
        return value
```
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

= 1.0

MIN wins/
MAX loses

= -1.0
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

\[ \begin{align*}
\text{W} & = 1.0 \\
\text{MAX} & = -1.0
\end{align*} \]

MIN wins/
MAX loses

1.0
-1.0
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins
△ = 1.0

MIN wins/
MAX loses
▽ = -1.0

MAX
5

1

2

1

2

MIN
4

1

2

MAX
3

1

2

1

2

1

2

1

2

1

2

1

2

1

2

1

2

1

2

1

2

1

2
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

= 1.0

MIN wins/
MAX loses

= -1.0

MIN

MAX

WIN

1.0

2

1

MAX

MIN

WIN

1.0

2

1

WIN

2

1

WIN

1.0

2

1

WIN

1.0

1

WIN

1.0

2

1

WIN

1.0
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

\[ \rightarrow \]

\[ = 1.0 \]

MIN wins /
MAX loses

\[ \rightarrow \]

\[ = -1.0 \]
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins
Δ = 1.0

MIN wins/
MAX loses
Δ = -1.0

= 1.0

= -1.0
Take 1 or 2 at each turn
Goal: take the last match

MAX wins
= 1.0

MIN wins/
MAX loses
= -1.0
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

\[ \text{MAX wins} \]
\[ = 1.0 \]

MIN wins/
MAX loses

\[ \text{MIN wins/} \]
\[ \text{MAX loses} \]

\[ = -1.0 \]
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

\[ = 1.0 \]

MIN wins/
MAX loses

\[ = -1.0 \]
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins
= 1.0

MIN wins/
MAX loses
= -1.0

W
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

MIN wins/
MAX loses

MAX wins

MIN wins/
MAX loses
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

\[ \begin{align*}
\text{MIN wins/} & \quad \text{MAX loses} \\
\downarrow & \quad \downarrow \\
1.0 & = -1.0
\end{align*} \]
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

\[ W = 1.0 \]

MIN wins

\[ \text{MAX loses} \]

\[ W = -1.0 \]
Baby Nim

Take 1 or 2 at each turn
Goal: take the last match

MAX wins

\[ = 1.0 \]

MIN wins/
MAX loses

\[ = -1.0 \]

could still win, but not optimal!!!
Minimax example 2

Which move should be made: A₁, A₂ or A₃?
Minimax example 2
Properties of minimax

• Minimax is optimal!
• Are we done?
  – For chess, $b \approx 35$, $d \approx 100$ for reasonable games $\rightarrow$ exact solution completely infeasible
  – Is minimax feasible for Mancala or Tic Tac Toe?
    • Mancala: 6 possible moves. average depth of 40, so $6^{40}$ which is on the edge
    • Tic Tac Toe: branching factor of 4 (on average) and depth of 9… yes!
• Ideas?
  – pruning!
  – improved state utility/evaluation functions
Pruning: do we have to traverse the whole tree?

MAX

MIN

A_{11} A_{12} A_{13} A_{21} A_{22} A_{23} A_{31} A_{32} A_{33}

4 12 7 10 3 16 2 4 1
Pruning: do we have to traverse the whole tree?
Minimax example 2

MAX

MIN

A_{11}  A_{12}  A_{13}  \quad 4  \quad 12  \quad 7

A_{21}  A_{22}  A_{23}  \quad 10  \quad 3  \quad 16

A_{31}  A_{32}  A_{33}  \quad 2  \quad 4  \quad 1
Minimax example 2

Any others if we continue?
Minimax example 2

MAX

MIN

A1

A2

A3

A11

A12

A13

A21

A22

4

12

7

10

3

2

4

1

MIN

MAX
Minimax example 2

MAX

MIN

A_1

A_2

A_3

A_{11} A_{12} A_{13}

A_{21} A_{22}

A_{31}

4 12 7

3?

10 3

2?

prune!
Alpha-Beta pruning

• An optimal pruning strategy
  – only prunes paths that are suboptimal (i.e. wouldn’t be chosen by an optimal playing player)
  – returns the *same* result as minimax, but faster

• As we go, keep track of the best and worse along a path
  – alpha = best choice we’ve found so far for MAX
  – beta = best choice we’ve found so far for MIN
Alpha-Beta pruning

• alpha = best choice we’ve found so far for MAX
• Using alpha and beta to prune:
  – We’re examining MIN’s options for a ply. To do this, we’re examining all possible moves for MAX. If we find a value for one of MAX’s moves that is less than alpha, return. (MIN could do better down this path)
Alpha-Beta pruning

• beta = best choice we’ve found so far for MIN
• Using alpha and beta to prune:
  – We’re examining MAX’s options for a ply. To do this, we’re examining all possible moves for MIN. If we find a value for one of MIN’s possible moves that is greater than beta, return. (MIN won’t end up down here)

```
MAX

MIN
```

return if any > beta
Alpha-Beta pruning

Do DF-search until first leaf
alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN
alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN
alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN

[3, +\infty]
[3, 3]
Alpha-Beta Example (continued)

alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN
alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN
Alpha-Beta Example (continued)

alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN

MAX

MIN
alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN
alpha = best choice we’ve found so far for MAX
beta = best choice we’ve found so far for MIN
def maxValue(state, alpha, beta):
    if state is terminal:
        return utility(state)
    else:
        value = -∞
        for all actions a in actions(state):
            value = max(value, minValue(result(state,a), alpha, beta))
            if value >= beta:
                return value # prune!
        alpha = max(alpha, value) # update alpha
    return value

We’re making a decision for MAX.
• When considering the MIN’s choices, if we find a value that is greater than beta, stop, because MIN won’t make this choice
• if we find a better path than alpha, update alpha
def minValue(state, alpha, beta):
    if state is terminal:
        return utility(state)
    else:
        value = +∞
        for all actions a in actions(state):
            value = min(value, maxValue(result(state,a), alpha, beta))
        if value <= alpha:
            return value # prune!
        beta = min(beta, value) # update alpha
        return value

We’re making a decision for MIN.
• When considering the MAX’s choices, if we find a value that is less than alpha, stop, because MAX won’t make this choice
• if we find a better path than beta for MIN, update beta
Baby NIM2: take 1, 2 or 3 sticks
Effectiveness of pruning

• Notice that as we gain more information about the state of things, we’re more likely to prune

• What affects the performance of pruning?
  – key: which order we visit the states
  – can try and order them so as to improve pruning
Effectiveness of pruning

• If perfect state ordering:
  – $O(b^m)$ becomes $O(b^{m/2})$
  – We can solve a tree twice as deep!

• Random order:
  – $O(b^m)$ becomes $O(b^{3m/4})$
  – still pretty good

• For chess using a basic ordering
  – Within a factor of 2 of $O(b^{m/2})$