Game Engineering

CS420-2014F-24

Board / Strategy Games

David Galles

Department of Computer Science
University of San Francisco
Example games (board splitting, chess, Othello)
Min/Max trees
Alpha-Beta Pruning
Evaluation Functions
Stopping the Search
Playing with chance
Two player games

- Board-Splitting Game
  - Two players, $V$ & $H$
  - $V$ splits the board vertically, selects one half
  - $H$ splits the board horizontally, selects one half
  - $V$ tries to maximize the final value, $H$ tries to minimize the final value

<table>
<thead>
<tr>
<th></th>
<th>14</th>
<th>5</th>
<th>11</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>13</td>
<td>9</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
24-2: Two player games

- Board-Splitting Game
  - We assume that both players are rational (make the best possible move)
  - How can we determine who will win the game?
24-3: Two player games

- Board-Splitting Game
  - We assume that both players are rational (make the best possible move)
  - How can we determine who will win the game?
    - Examine all possible games!
24-4: Two player games
24-5: Two player games
Two player games
24-7: Two player games

- Game playing agent can do this to figure out which move to make
  - Examine all possible moves
  - Examine all possible responses to each move
  - ... all the way to the last move
  - Calculate the value of each move (assuming opponent plays perfectly)
Two-Player Games

- Initial state
- Successor Function
  - Just like other Searches
- Terminal Test
  - When is the game over?
- Utility Function
  - Only applies to terminal states
  - Chess: +1, 0, -1
  - Backgammon: 192 . . . -192
Max(node)
   if terminal(node)
       return utility(node)
   maxVal = MIN_VALUE
   children = successors(node)
   for child in children
       maxVal = max(maxVal, Min(child))
   return maxVal

Min(node)
   if terminal(node)
       return utility(node)
   minVal = MAX_VALUE
   children = successors(node)
   for child in children
       minVal = min(minVal, Max(child))
   return minVal
Branching factor of $b$, game length of $d$ moves, what are the time and space requirements for Minimax?
• Branching factor of $b$, game length of $d$ moves, what are the time and space requirements for Minimax?
  • Time: $O(b^d)$
  • Space: $O(d)$
• Not manageable for any real games – chess has an average $b$ of 35, can’t search the entire tree
• Need to make this more manageable
• What if there are $> 2$ players?
• We can use the same search tree:
  • Alternate between several players
  • Need a different evaluation function
Functions return a vector of utilities
- One value for each player
- Each player tries to maximize their utility
- May or may not be zero-sum
24-14: > 2 Player Games

to move
A
B
C
A

(1, 2, 6) 
(1, 2, 6) 
(1, 2, 6) 
(1, 2, 6) 
(1, 5, 2) 
(1, 5, 2) 
(1, 5, 2) 
(1, 5, 2) 
(5, 4, 5) 
(5, 4, 5) 
(5, 4, 5) 
(5, 4, 5) 

(4, 2, 3) 
(6, 1, 2) 
(7, 4, 1) 
(5, 1, 1) 
(1, 5, 2) 
(7, 7, 1) 
(5, 4, 5)
Non zero-sum games

- Even 2-player games don’t need to be zero-sum
  - Utility function returns a vector
  - Each player tries to maximize their utility
- If there is a state with maximal outcome for both players, rational players will cooperate to find it
- Minimax is rational, will find such a state
• Does it matter what value is in the yellow circle?
24-17: **Alpha-Beta Pruning**

- If the yellow leaf has a value $> 5$, parent won’t pick it.
- If the yellow leaf has a value $< 12$, grandparent won’t pick it.
- To affect the root, value must be $< 5$ and $> 12$. 

![Alpha-Beta Pruning Diagram]

---

**Diagram Notes:**
- Yellow leaves indicate nodes that are pruned.
- Min and Max nodes denote minimax decision points.
- Numbers represent node values.
Value of nodes in neither yellow circle matter. Are there more?
Value of nodes in none of the yellow circles matter.
• If $m$ is better than $n$ for Player, we will never reach $n$
  • (player would pick $m$ instead)
24-21: **Alpha-Beta Pruning**

- Maintain two bounds, lower bound $\alpha$, and an upper bound $\beta$
  - Bounds represent the values the node must have to possibly affect the root
- As you search the tree, update the bounds
  - Max nodes increase $\alpha$, min nodes decrease $\beta$
- If the bounds ever cross, this branch cannot affect the root, we can prune it.
24-22: **Alpha-Beta Pruning**

\[ \alpha = -\infty, \beta = \infty \]
24-23: Alpha-Beta Pruning

In the diagram, the Alpha-Beta Pruning algorithm is illustrated with a game tree. Each node in the tree represents a position in the game, and the values associated with each node represent the scores. The algorithm aims to prune branches of the tree that do not influence the final decision, thereby reducing the computational complexity. The initial values for \( \alpha \) (minimum) and \( \beta \) (maximum) are set to negative and positive infinity, respectively. The pruning occurs when \( \alpha \) becomes greater than or equal to \( \beta \), indicating that further exploration is unnecessary for that branch.
24-24: Alpha-Beta Pruning

\[
\begin{align*}
\alpha &= \text{-inf}, \quad \beta = \text{inf} \\
\alpha &= \text{-inf}, \quad \beta = \text{inf} \\
\alpha &= \text{-inf}, \quad \beta = \text{inf} \\
\alpha &= \text{-inf}, \quad \beta = 14 \\
\alpha &= \text{-inf}, \quad \beta = 14 \\
\end{align*}
\]
24-25: Alpha-Beta Pruning

\[ \alpha = -\infty, \ \beta = \infty \]

\[ \alpha = 12, \ \beta = \infty \]

\[ \alpha = 12, \ \beta = \infty \]
24-26: Alpha-Beta Pruning
24-27: Alpha-Beta Pruning

\[ \alpha = -\infty, \beta = \infty \]
24-28: **Alpha-Beta Pruning**

\[
\begin{align*}
\alpha &= -\infty, \quad \beta = \infty \\
\alpha &= -\infty, \quad \beta = 12 \\
\alpha &= -\infty, \quad \beta = 12 \\
\alpha &= -\infty, \quad \beta = 12 \\
\end{align*}
\]
24-29: Alpha-Beta Pruning
24-30: Alpha-Beta Pruning

\[
\alpha = -\infty, \quad \beta = \infty
\]

\[
\alpha = -\infty, \quad \beta = 12
\]

\[
\alpha = 15, \quad \beta = 12
\]
24-31: Alpha-Beta Pruning

\[ \alpha = 12, \beta = \text{inf} \]
24-32: Alpha-Beta Pruning

\[ \alpha = 12, \beta = \text{inf} \]
24-33: Alpha-Beta Pruning

$$\alpha = 12, \beta = \text{inf}$$
24-34: Alpha-Beta Pruning

\[ \alpha = 12, \ \beta = \text{inf} \]
**24-35: Alpha-Beta Pruning**

The diagram illustrates the Alpha-Beta pruning algorithm in a game tree. The values at the nodes are as follows:

- Max nodes are shown with an open circle and labeled with their values.
- Min nodes are shown with a closed circle and labeled with their values.
- Leaf nodes are shown with solid circles and labeled with their values.

The pruning process is demonstrated by the yellow lines, which indicate nodes that were pruned because they were determined to be inferior to the best value found during the search.

The algorithm is applied to determine the optimal move for the maximizing player, ensuring that only nodes with a potential benefit greater than the current best value are evaluated.
24-36: **Alpha-Beta Pruning**

```
α = 12, β = inf
```

Max

```
12
```

Min

```
12
```

Max

```
12
```

Min

```
5
```

Max

```
15
```

Min

```
15
```

Max

```
11
```

Min

```
11
```

Max

```
11
```

α = 12, β = 11
We can cut large branches from the search tree.

In the previous example, what would happen with similar values and a deeper tree?

If we choose the order that we evaluate nodes (more on this in a minute ...), we can dramatically cut down on how much we need to search.
Evaluation Functions

• We can’t search all the way to the bottom of the search tree
  • Trees are just too big

• Search a few levels down, use an evaluation function to see how good the board looks at the moment

• Back up the result of the evaluation function, as if it was the utility function for the end of the game
Evaluation Functions

- **Chess:**
  - Material - value for each piece (pawn = 1, bishop = 3, etc)
    - Sum of my material - sum of your material
  - Positional advantages
    - King protected
    - Pawn structure

- **Othello:**
  - Material – each piece has unit value
  - Positional advantages
    - Edges are good
    - Corners are better
    - “near” edges are bad
If we have an evaluation function that tells us how good a move is, why do we need to search at all?
  • Could just use the evaluation function

If we are only using the evaluation function, does search do us any good?
How can we use the evaluation function to maximize the pruning in alpha-beta pruning?
How can we use the evaluation function to maximize the pruning in alpha-beta pruning?

- Order children of max nodes, from highest to lowest
- Order children of min node, from lowest to highest
- (Other than for ordering, eval function is not used for interior nodes)

With perfect ordering, we need to search only $b^{d/2}$ (instead of $b^d$) to find the optimal move – can search up to twice as far
We can’t search all the way to the endgame
  • Not enough time

Search a set number of moves ahead
  • Problems?
24-44: Stopping the Search

- We can’t search all the way to the endgame
  - Not enough time
- Search a set number of moves ahead
  - What if we are in the middle of a piece trade?
  - In general, what if our opponent is about to capture one of our pieces
Stopping the Search

(a) White to move

(b) White to move
• **Quiescence Search**
  • Only apply the evaluation function to nodes that do not swing wildly in value
  • If the next move makes a large change to the evaluation function, look ahead a few more moves
  • Not increasing the search depth for the entire tree, just around where the action is
  • To prevent the search from going too deep, may restrict the kinds of moves (captures only, for instance)
Stopping the Search

- Horizon Problem
  - Sometimes, we can push a bad move past the horizon of our search
  - Not preventing the bad move, just delaying it
  - A position will look good, even though it is ultimately bad
24-48: Horizon Problem

Black to move
24-49: Horizon Problem

- Singular Extensions
  - When we are going to stop, see if there is one move that is clearly better than all of the others.
  - If so, do a quick “search”, looking only at the best move for each player.
  - Stop when there is no “clearly better” move.
  - Helps with the horizon problem, for a series of forced moves.
- Similar to quiescence search.
What about games that have an element of chance (backgammon, poker, etc)

We can add chance nodes to our search tree
  • Consider “chance” to be another player

How should we back up values from chance nodes?
24-51: Adding Chance

MAX

CHANCE

MIN

CHANCE

MAX

TERMINAL

2  -1  1  -1  1
24-52: Adding Chance

- For Max nodes, we backed up the largest value:
  \[
  \max_{s \in \text{Successors}(n)} Val(s)
  \]

- For Min nodes, we backed up the smallest
  \[
  \max_{s \in \text{Successors}(n)} Val(s)
  \]

- For chance nodes, we back up the expected value of the node
  \[
  \sum_{s \in \text{Successors}(n)} P(s) Val(s)
  \]
24-53: Adding Chance

- Adding chance dramatically increases the number of nodes to search
  - Braching factor $b$ (ignoring die rolls)
  - $n$ different dice outcomes per turn
  - Time to search to level $m$?
Adding chance dramatically increases the number of nodes to search

- Braching factor $b$ (ignoring die rolls)
- $n$ different dice outcomes per turn
- Time to search to level $m$: $b^m n^m$
Adding Chance

Because we are using expected value for chance nodes, need to be more careful about choosing the evaluation function
Summary

- Min/Max trees
- Alpha-Beta Pruning
- Evaluation Functions
- Stopping the Search
- Playing with chance