Contents

• Search algorithms for games
  – MiniMax
  – Alpha-beta pruning
• Game trees
• Heuristics for games
• Assignment 1
Games vs. search problems

- “Unpredictable” opponent ⇒ solution is a strategy specifying a move for every possible opponent reply
- Time limits ⇒ unlikely to find goal, must approximate
Types of games

- There are different kinds of games:

<table>
<thead>
<tr>
<th></th>
<th>Deterministic</th>
<th>Stochastic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Perfect information</strong></td>
<td>chess, checkers, go, Othello</td>
<td>Backgammon, monopoly</td>
</tr>
<tr>
<td><strong>Imperfect information</strong></td>
<td>battleships, blind tic-tac-toe</td>
<td>bridge, poker, scrabble</td>
</tr>
</tbody>
</table>
Games can be hard

- Even “nicely behaved” games like chess can be very hard to solve.
  - Chess has an average branching factor of about 35
  - A game can last about 100 plies
  - A game tree can have $35^{100}$ leaves ($\sim 10^{154}$)

- Checkers has been completely solved (2007)
  - Alpha-Beta search
  - Endgame databases

- There are chess engines that play at or above elite level.
  - Note! This does not mean that chess has been solved.

- The best current go engines play at an advanced amateur level.
Game trees

• A deterministic, turn based, finite game with perfect information (e.g. chess) can be represented by a game tree:
  – The root is the initial position
  – The children of the root are possible positions after one ply
  – The grandchildren of the root are possible positions after two plies
  – Et cetera...
  – The leaves represent possible endings and contain information about the value (utility) of the ending to each of the players

• In a zero-sum game, what one player gains, the other player loses
Game tree: tic-tac-toe
MiniMax algorithm

- Two players; Min and Max. Assume that both players play perfectly
  - Therefore we cannot optimistically assume player will miss winning responses to our moves
- Min’s strategy:
  - Wants lowest possible score, ideally $-\infty$
  - But must account for Max aiming for $+\infty$
  - Min’s best strategy is:
    - Choose the move that minimizes the score that will result when Max chooses the maximizing move
    - hence the name MiniMax
- Max does the opposite
MiniMax: Game tree

Propagate upwards (depth first search)
MiniMax algorithm

```
function MINIMAX(search tree node s)
    if s is a leaf then
        return utility(s)
    end if
    if s is a max node then
        return max(MINIMAX(t)) : t is a child of s
    end if
    if s is a min node then
        return min(MINIMAX(t)) : t is a child of s
    end if
end function
```
Drawbacks of MiniMax

- MiniMax is very inefficient. With branching factor $b$ and depth $d$, we must explore $b^d$ nodes.
- We needlessly calculate the exact score at every node.
  - At many nodes we don’t need to know exact score.
- We can use alpha-beta pruning to remove unnecessary nodes.
Alpha-Beta Procedure

• Alpha: the value of the best (highest value) choice we have found so far along the path for Max.
  - i.e we want alpha to be as large as possible
  - $\alpha$ is a lower bound on the real outcome: $v \geq \alpha$

• Beta: the value of the best (lowest value) choice we have found so far along the path for Min.
  - i.e we want beta to be as small as possible
  - $\beta$ is an upper bound on the real outcome: $v \leq \beta$
\( \alpha - \beta \) pruning example

\[
\begin{array}{c}
\text{MAX} \\
\text{MIN} \\
3 \quad 12 \quad 8
\end{array}
\]

\( \alpha = 3 \)
α-β pruning example

α = 3

alpha cutoff
α-β pruning example

\[ \alpha = 3 \]
**α-β pruning example**

\[ \alpha = 3 \]
α-β pruning example

\[ \alpha = 3 \]
Beta cutoff

\[ \beta = 4 \]
# Alpha-Beta algorithm

```python
function ALPHA_BETA(search tree node s)
    return MAX.VALUE(s, -\infty, +\infty)
end function

function MAX.VALUE(search tree node s, \alpha, \beta))
    if s is a leaf then
        return utility(s)
    end if
    value \leftarrow -\infty
    for each child t of s do
        value \leftarrow max(value, MIN.VALUE(t, \alpha, \beta))
        if value \geq \beta then #\beta-cutoff
            return value
        end if
        \alpha \leftarrow max(\alpha, value)
    end for
    return value
end function
```
function MinValue(search tree node \( s \), \( \alpha \), \( \beta \))
    if \( s \) is a leaf then
        return utility(\( s \))
    end if
    value \( \leftarrow +\infty \)
    for each child \( t \) of \( s \) do
        value \( \leftarrow \min(\text{value}, \ \text{MAXVALUE}(t, \alpha, \beta)) \)
        if value \( \leq \alpha \) then \#\( \alpha \)-cutoff
            return value
        end if
        \( \beta \leftarrow \min(\beta, \text{value}) \)
    end for
    return value
end function
Another $\alpha \beta$ example
Another $\alpha\beta$ example

Max

Min

Max
Alpha-Beta vs. MiniMax

• Given branching factor $b$ and depth $d$, MiniMax have to visit $O(b^d)$ nodes.

• How many nodes Alpha-Beta visits depends on the order in which moves are examined:
  – Best case: just have to visit $O(b^{\frac{d}{2}})$ nodes.
  – With random move order: $O(b^{\frac{3d}{4}})$ nodes.
  – We can use domain-specific knowledge to achieve a reasonable move order.
  – For chess it is possible to come reasonably close to the best-case.
Numbers in perspective

- Let us say that we have a branching factor $b = 20$ and want to search to depth $d = 8$
- MiniMax will search $b^d = 20^8 = 25,600,000,000$ nodes
- A “perfect” Alpha-Beta would search $\frac{d}{b^2} = 20^4 = 160,000$ nodes.
- A “reasonable” Alpha-Beta would search $b^{3d/4} = 20^6 = 64,000,000$ nodes
Cutting off the search

Rather than letting Alpha-Beta search the whole game tree (which is unrealistic for, e.g., chess), we want to cut the search off and return a heuristic evaluation.

The simplest method is to simply set a fixed depth to search to, but there are also other useful approaches:

• Quiescence search
  – Only stop at “good” positions

• Forward pruning
  – Prune away some moves immediately

• Beam search (perhaps using statistical or probabilistic methods)
  – Consider only a “beam” of the $n$ best moves
Heuristics for game search

- Heuristic functions for game search often combine a number of **features**
- One way of coming up with a value is by using **statistics**
  - Of all positions in my database that have exactly the same features as the current position, how many was eventually won by white?
  - If this number is, say 75%, we can return the value 0.5 (where 1 is a white win and -1 a black win)
- This approach can also be useful for building opening databases.
Chess engines

- Alpha-Beta search
- Move ordering
  - Killer moves
  - Iterative deepening (also helps with cutoff)
- Transposition tables
  - See if a position already have been evaluated
- Advanced heuristic functions
- Quiescence search
- Opening databases
- Endgame databases
- Efficient board representation (move generation is a bottleneck)
Assignment 1: Othello

- Write a program that takes an Othello position and return a recommended move.
- In the Java helper code, `OthelloPosition` is provided to represent the positions.
  - The code is incomplete. You will have to fill in code at the places marked 'TODO' in the file.
- Use game search with alpha-beta pruning and heuristic evaluation.
  - Construct a class that implements the provided Java interface `OthelloAlgorithm`.
  - This interface has a method `evaluate` that takes a position and returns an `OthelloAction`.
- Must be able to beat the test code provided which uses a naïve heuristics (counting no. of markers on the board).
Assignment 1: Othello

- You should also provide an interface in form of a simple shell script
  - Input: description of the position (ascii) and a time limit
  - Output: print the move to stdout (e.g. (4,7) to place a marker at row 4, col 7), or ‘pass’ if unable to make a move
- **Tip:** do not store the pruned nodes in your game tree. You will probably run out of memory!
- **Due:** 2014-12-03, 12.00