Overview

- Midterm exam date survey
- LISP and emacs tips (read it yourself)
- Game playing
- Minimax
- $\alpha$-$\beta$ pruning

LISP and Emacs Tips (read it yourself)

- return value in Lisp: the last item in the function
  
  \[
  > (\text{defun fun () } '1 '2 '3 '4) \backslash \text{FUN} \\
  > (\text{fun}) \backslash 4
  \]

- forcefully returning a value from a function in Lisp:
  
  \[
  (\text{return-from <function-name>} <value>)
  \]

- multiple windows in emacs: \text{C-x 2}
- navigation between windows in emacs: \text{C-x o}
- increasing height of window in emacs: \text{C-x k}

Game Playing

- attractive AI problem because it is \textbf{abstract}
- one of the oldest domains in AI
- in most cases, the world state is fully accessible
- computer representation of the situation can be clear and exact
- challenging: uncertainty introduced by the opponent and the complexity of the problem (full search is impossible)
- hard: in chess, branching factor is about 35, and 50 moves by each player $= 35^{100}$ nodes to search
  - compare to $10^{40}$ possible legal board states
- \textit{game playing is more like real life than mechanical search}
Games vs. Search Problems

“Unpredictable” opponent → solution is a contingency plan

Time limits → unlikely to find goal, must approximate

Plan of attack:

- algorithm for perfect play (Von Neumann, 1944)
- finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952–57)
- pruning to reduce costs (McCarthy, 1956)

Types of Games

<table>
<thead>
<tr>
<th>perfect info</th>
<th>deterministic</th>
<th>chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>chess, checkers, go, othello</td>
<td>backgammon, monopoly</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>imperfect info</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>bridge, poker, scrabble</td>
</tr>
</tbody>
</table>

Two-Person Perfect Information Game

- initial state: initial position and who goes first
- operators: legal moves
- terminal test: game over?
- utility function: outcome (win:+1, lose:-1, draw:0, etc.)

- two players (MIN and MAX) taking turns to maximize their chances of winning (each turn generates one ply)
- one player’s victory is another’s defeat
- need a strategy to win no matter what the opponent does

Minimax: Strategy for Two-Person Perfect Info

- generate the whole tree, and apply util function to the leaves
- go back upward assigning utility value to each node
- at MIN node, assign min(successors’ utility)
- at MAX node, assign max(successors’ utility)
- assumption: the opponent acts optimally
Minimax Decision

```plaintext
function Minimax-Decision (game) returns operator
    return operator that leads to a child state with the
    max(Minimax-Value(child state, game))

function Minimax-Value(state, game) returns utility value
    if Goal(state), return Utility(state)
    else if Max's move then
        return max of successors' Minimax-Value
    else
        return min of successors' Minimax-Value
```

Minimax: Evaluation

Branching factor $b$, max depth $m$:

- **complete**: if the game tree is finite
- **optimal**: if opponent is optimal
- **time**: $b^m$
- **space**: $b^m$ – depth-first (only when utility function values of all
  nodes are known!)

Minimax Exercise

Resource Limits

- **Time limit**: as in Chess → can only evaluate a fixed number of
  paths
- **Approaches**:
  - **evaluation function**: how desirable is a given state?
  - **cutoff test**: depth limit
  - **pruning**

Depth limit can result in the **horizon effect**: interesting or devastating
events can be just over the horizon!
Evaluation Functions

For chess, usually a linear weighted sum of feature values:

- \( \text{Eval}(s) = \sum_i w_i f_i(s) \)
- \( f_i(s) = (\text{number of white piece } X) - (\text{number of black piece } X) \)
- other features: degree of control over the center area
- exact values do not matter: the order of Minimax-Value of the successors matter.

\[ \alpha \text{ Cuts} \]

When the current max value is greater than the successor’s min value, don’t look further on that min subtree:

\[ \beta \text{ Cuts} \]

When the current min value is less than the successor’s max value, don’t look further on that max subtree:

\[ \alpha - \beta \text{ Pruning} \]

- memory of best MAX value \( \alpha \) and best MIN value \( \beta \)
- do not go further on any one that does worse than the remembered \( \alpha \) and \( \beta \)

Right subtree can be at most 2, so MAX will always choose the left path regardless of what appears next.

Right subtree can be at least 5, so MIN will always choose the left path regardless of what appears next.
Key Points

- Game playing: what are the types of games?
- Minimax: definition, and how to get minmax values
- Minimax: evaluation
- \( \alpha - \beta \) pruning: why it saves time

Next Time

- formal \( \alpha - \beta \) pruning algorithm
- games with an element of chance
- state-of-the-art game playing with AI
- more complex games
- project #1: full description
  - core routines for 8-puzzle

\( \alpha - \beta \) Pruning Properties

Cut off nodes that are known to be suboptimal.

Properties:

- pruning does not affect final result
- good move ordering improves effectiveness of pruning
- with perfect ordering, time complexity = \( b^{m/2} \)
  → doubles depth of search
  → can easily reach 8-ply in chess
- \( b^{m/2} = (\sqrt{b})^m \), thus \( b = 35 \) in chess reduces to \( b = \sqrt{35} \approx 6 \) !!!