Adversary Search

• Ref: Chapter 5
Games & A.I.

• Easy to measure success
• Easy to represent states
• Small number of operators
• Comparison against humans is possible.
• Many games can be modeled very easily, although game playing turns out to be very hard.
2 Player Games

• Requires reasoning under uncertainty.

• Two general approaches:
  – Assume nothing more than the rules of the game are important - reduces to a search problem.
  – Try to encode strategies using some type of pattern-directed system (perhaps one that can learn).
Search and Games

• Each node in the search tree corresponds to a possible state of the game.
• Making a move corresponds to moving from the current state (node) to a child state (node).
• Figuring out which child is best is the hard part.
• The branching factor is the number of possible moves (children).
Search Tree Size

• For most interesting games it is impossible to look at the entire search tree.

• Chess:
  – branching factor is about 35
  – typical match includes about 100 moves.
  – Search tree for a complete game: $35^{100}$
Heuristic Search

• Must evaluate each choice with less than complete information.

• For games we often evaluate the game tree rooted at each choice.

• There is a tradeoff between the number of choices analyzed and the accuracy of each analysis.
Game Trees
Plausible Move Generator

- Sometimes it is possible to develop a move generator that will (with high probability) generate only those moves worth consideration.

- This reduces the branching factor, which means we can spend more time analyzing each of the plausible moves.
Recursive State Evaluation

• We want to rank the plausible moves (assign a value to each resulting state).

• For each plausible move, we want to know what kind of game states could follow the move (Wins? Loses?).

• We can evaluate each plausible move by taking the value of the best of the moves that could follow it.
Assume the adversary is good.

- To evaluate an adversary’s move, we should assume they pick a move that is good for them.
- To evaluate how good their moves are, we should assume we will do the best we can after their move (and so on…)
Static Evaluation Function

- At some point we must stop evaluating states recursively.
- At each leaf node we apply a static evaluation function to come up with an estimate of how good the node is from our perspective.
- We assume this function is not good enough to directly evaluate each choice, so we instead use it deeper in the tree.
Example evaluation functions

• Tic-Tac-Toe: number of rows, columns or diagonals with 2 of our pieces.
• Checkers: number of pieces we have - the number of pieces the opponent has.
• Chess: weighted sum of pieces:
  – king=1000, queen=10, bishop=5, knight=5, ...
Minimax

• Depth-first search with limited depth.

• Use a static evaluation function for all leaf states.

• Assume the opponent will make the best move possible.
Minimax Search Tree

Our Move
Maximizing Ply

Opponent’s Move
Minimizing Ply
Minimax Algorithm

Minimax(curstate, depth, player):

If (depth==max)
    Return static(curstate,player)

generate successor states s[1..n]

If (player==ME)
    Return max of Minimax(s[i],depth+1,OPPONENT)

Else
    Return min of Minimax(s[i],depth+1,ME)
The Game of MinMax

-3  2  -1
-2  3
4  1  -4

• Start in the center square.
• Player MAX picks any number in the current row.
• Player MIN picks any number in the resulting column.
• The game ends when a player cannot move.
• MAX wins if the sum of numbers picked is > 0.
Max's Turn

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>-4</td>
</tr>
</tbody>
</table>

Max

\[ -3 \]

Min

\[ 2 \]  \[ -1 \]

Max

\[ 1 \]  \[ 3 \]  \[ -4 \]

Min

\[ 4 \]  \[ -4 \]

Max

\[ -1 \]  \[ 3 \]  \[ 2 \]  \[ 1 \]  \[ 4 \]
Pruning

• We can use a branch-and-bound technique to reduce the number of states that must be examined to determine the value of a tree.

• We keep track of a lower bound on the value of a maximizing node, and don’t bother evaluating any trees that cannot improve this bound.
Pruning in MinMax
Pruning Minimizing Nodes

- Keep track of an upper bound on the value of a minimizing node.
- Don’t bother with any subtrees that cannot improve (lower) the bound.
Minimax with Alpha-Beta Cutoffs

• Alpha is the lower bound on maximizing nodes.
• Beta is the upper bound on minimizing nodes.

• Both alpha and beta get passed down the tree during the Minimax search.
Usage of Alpha & Beta

- At minimizing nodes, we stop evaluating children if we get a child whose value is less than the current lower bound (alpha).

- At maximizing nodes, we stop evaluating children as soon as we get a child whose value is greater than the current upper bound (beta).
Alpha & Beta

• At the root of the search tree, alpha is set to $-\infty$ and beta is set to $+\infty$.
• Maximizing nodes update alpha from the values of children.
• Minimizing nodes update beta from the value of children.
• If alpha > beta, stop evaluating children.
Movement of Alpha and Beta

• Each node passes the current value of alpha and beta to each child node evaluated.
• Children nodes update their copy of alpha and beta, but do not pass alpha or beta back up the tree.
• Minimizing nodes return beta as the value of the node.
• Maximizing nodes return alpha as the value of the node.
The Effectiveness of Alpha-Beta

• The effectiveness depends on the order in which children are visited.
• In the best case, the effective branching factor will be reduced from $b$ to $\sqrt{b}$.
• In an average case (random values of leaves) the branching factor is reduced to $b/\log b$. 
The Horizon Effect

• Using a fixed depth search can lead to the following:
  – A bad event is inevitable.
  – The event is postponed by selecting only those moves in which the event is not visible (it is over the horizon).
  – Extending the depth only moves the horizon, it doesn’t eliminate the problem.
Quiescence

• Using a fixed depth search can lead to other problems:
  – it’s not fair to evaluate a board in the middle of an exchange of Chess pieces.
  – What if we choose an odd number for the search depth on the game of MinMax?

• The evaluation function should only be applied to states that are *quiescent* (relatively stable).
Pattern-Directed Play

• Encode a bunch of patterns and some information that indicates what move should be selected if the game state ever matches the pattern.

• *Book play*: often used in Chess programs for the beginning and ending of games.
Iterative Deepening

• Many games have time constraints.
• It is hard to estimate how long the search to a fixed depth will take (due to pruning).
• Ideally we would like to provide the best answer we can, knowing that time could run out at any point in the search.
• One solution is to evaluate the choices with increasing depths.
Iterative Deepening

- There is lots of repetition!
- The repeated computation is small compared to the new computation.
- Example: branching factor 10
  - depth 3: 1,000 leaf nodes
  - depth 4: 10,000 leaf nodes
  - depth 5: 100,000 leaf nodes
A* Iterative Deepening

- Iterative deepening can also be used with A*.

1. Set THRESHOLD to be \( f(\text{start\_state}) \).
2. Depth-first search, don’t explore any nodes whose \( f \) value is greater than THRESHOLD.
3. If no solution is found, increase THRESHOLD and go back to step 2.