Games

• Ref: Chapter 6
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.
Issues

• Optimal Decisions
  – typically require searching

• Search space is too large
  – pruning techniques
  – heuristics

• Imperfect games
  – opponent is not predictable

• Games that include an element of chance.
  – dice games, etc.
Types of games

<table>
<thead>
<tr>
<th>Perfect Information</th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>chess, checkers, go, othello</td>
<td>backgammon</td>
<td></td>
</tr>
<tr>
<td></td>
<td>monopoly</td>
<td></td>
</tr>
<tr>
<td>battleship, blind</td>
<td>bridge</td>
<td></td>
</tr>
<tr>
<td>tic-tac-toe</td>
<td>poker</td>
<td></td>
</tr>
<tr>
<td></td>
<td>scrabble</td>
<td></td>
</tr>
<tr>
<td></td>
<td>nuclear war</td>
<td></td>
</tr>
</tbody>
</table>
Games and Search

• Adversarial search.
• Solution is strategy (strategy specifies move for every possible opponent reply).
  – Time limits force an *approximate* solution
  – Evaluation function: evaluate “goodness” of game position
  – Examples: chess, checkers, Othello, backgammon
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 trade-off 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

E 9
F -6
G 0
H 0
I -2
J -4
K -3
Optimal Strategy

• Assumption: Both players play optimally
• Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

$$\text{MINIMAX-VALUE}(node) =$$

- $\text{total}(node)$ if $n$ is a terminal node
- $\max_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$ if $n$ is a max node
- $\min_{s \in \text{successors}(n)} \text{MINIMAX-VALUE}(s)$ if $n$ is a min node
Two Ply Game Tree

MAX

MIN

MAX's turn
possible moves: $a_1 \ a_2 \ a_3$
Two Ply Game Tree

MIN's turn. MAX has picked move \( a_1 \)
possible moves: \( b_1 \ b_2 \ b_3 \)
Two Ply Game Tree

MAX

MIN

MIN will pick the minimum move
Two Ply Game Tree

MAX will pick the maximum move
(the maximum of the minimums)
Maximizes the worst case outcome.
What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally: maximizes worst-case outcome for MAX.
- But if MIN does not play optimally, MAX will do even better.
Minimax Algorithm

• Utility(x) is a the static evaluation function used to evaluate leaf nodes.
• Non-leaf nodes are evaluated by
  – first evaluating each child node (recursively)
  – either taking the maximum of it's children or the minimum of it's children (depending on whose turn the node represents)
Minimax algorithm

**function** MINIMAX-DECISION(state) **returns** an action

**inputs:** state, current state in game

ν ← MAX-VALUE(state)

**return** the action in SUCCESSORS(state) with value ν

**function** MAX-VALUE(state) **returns** a utility value

if TERMINAL-TEST(state) **then return** UTILITY(state)

ν ← -∞

**for** s in SUCCESSORS(state) **do**

ν ← MAX(ν, MIN-VALUE(s))

return ν

**function** MIN-VALUE(state) **returns** a utility value

if TERMINAL-TEST(state) **then return** UTILITY(state)

ν ← ∞

**for** s in SUCCESSORS(state) **do**

ν ← MIN(ν, MAX-VALUE(s))

return ν
Properties of Minimax

\( b \) is average branching factor.
\( m \) is search depth.

- Minimax needs \( O(bm) \) memory (space).
  - this is great.
- Minimax looks at \( O(b^m) \) nodes (time).
  - this is not so great... But we can do better.
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 Maximizing Nodes

After evaluating B, we know the worst we can do (A) is 3.
When evaluating C, after we get a value of 2 for $c_1$, we know that C will not be greater than 2 (C is a minimizing node).
No need to evaluate $c_2$ or $c_3$.
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.
A

[\alpha, \beta]

B

D  E

3  5

F

I  J

5  7

K  L

0  7

C

G

H

4

N  M

8  7
\[ (-\infty, 3] \rightarrow 3 \rightarrow \alpha, \beta \rightarrow [3, \infty] \]

\[ 3 \rightarrow 5 \rightarrow \]
\[ \alpha > \beta, \text{ so stop and return } \beta \]
\( \alpha, \beta \) 
\([3, \infty]\)

no change to \( \alpha \)
(which can only get higher)

\([3, \infty]\)

\([3, 0]\)

0

no change to \( \alpha \)
now change $\alpha$
\[\alpha, \beta \in [3, \infty]\]

update \(\beta\)

\[\beta \]
\[ \alpha, \beta \in [3, \infty] \]
now change $\beta$
The end result:
- picking B can lead to at least a 3
- picking C will lead to at least 4.

Choose move to C

We will get the same result whether or not we use $\alpha$ and $\beta$.

With $\alpha$ and $\beta$ the search can ignore some trees (nodes) in the search tree.
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 $\sqrt{b}$.
  – reduces time by 1/2: $O(m^{b/2})$
• 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