Game-playing AIs: Games and Adversarial Search I

AIMA 5.1-5.2
Games: Outline of Unit

Part I: Games as Search
- Motivation
- Game-playing AI successes
- Game Trees
- Evaluation Functions

Part II: Adversarial Search
- The Minimax Rule
- Alpha-Beta Pruning
Why study games?

- **Multi-agent environments**: environments with other agents, whose actions affect our success
  - Two general categories: *Cooperative* vs. *competitive*
  - Competitive multi-agent environments give rise to *adversarial search* a.k.a. *games*

- **Huge** state spaces – Games are **hard**!

- Historical role in AI

- Games are fun!
May 11, 1997

Deep Blue Wins 3.5 to 2.5

With a dramatic victory in Game 6, Deep Blue won its six-game rematch with Champion Garry Kasparov.

Commentary
George Plumpton on chess, Kasparov, and the limitations of computers
Read the article

Commentary
Vishwanathan Anand on the legacy of Kasparov vs. Deep Blue
Read the article

Club Kasparov
Visit the virtual home of the world’s greatest chess player.

Guest essays
Thoughts on chess, computers, and what it all means
Read the essays...

Community
During the rematch, more than 20,000 people from 120 countries joined the community to talk about the match.

Clips from the rematch
Video footage from the games
Highlights from the games
State of the art

• How good are computer game players?

  Chess:
  • 1997 - Deep Blue beat Gary Kasparov
  • 2006 - Vladmir Kramnik, the undisputed world champion, defeated 4–2 by Deep Fritz ($72 on Amazon!)

  Checkers: Chinook (an AI program with a very large endgame database) is the world champion. Checkers has been solved exactly - it's a draw!

  Go: 2013 – Two 9-dan professional Go players were defeated by two different programs using probabilistic Monte Carlo methods, albeit with a 3- and 4-stone handicap.

  Bridge: "Expert" computer players exist (but no world champions yet!)

Good place to learn more: http://www.cs.ualberta.ca/~games/
Ratings of human & computer chess champions
A **cooperative multi-agent environment: Pragbot**

- Two players, Commander and Junior, must coordinate to:
- Tasks:
  - Defuse bombs that can kill Commander
  - Defeat badguys before they flip Junior and/or escape
  - Rescue hostages
The Simplest Game Environment

- **Multiagent**
- **Static:** No change while an agent is deliberating
- **Discrete:** A finite set of percepts and actions
- **Fully observable:** An agent's sensors give it the complete state of the environment.
- **Strategic:** The next state is determined by the current state and the action executed by the agent and the actions of one other agent.
- **Episodic:** The game can be viewed as many atomic "episodes" during which the agent perceives and then performs a single action, which depends only on the episode itself.
Key properties of our sample games

1. Two players alternate moves
2. Zero-sum: one player’s loss is another’s gain
3. Clear set of legal moves
4. Well-defined outcomes (e.g. win, lose, draw)

• Examples:
  • Chess, Checkers, Go,
  • Mancala, Tic-Tac-Toe, Othello,
  • Nim, …
More complicated games

- Most card games (e.g. Hearts, Bridge, etc.) and Scrabble
  - non-deterministic
  - lacking in perfect information
- Cooperative games
- Real-time strategy games (lack alternating moves). e.g. Warcraft
Formalizing the Game setup

1. Two players: \textit{MAX} and \textit{MIN}; \textit{MAX} moves first.
2. \textit{MAX} and \textit{MIN} take turns until the game is over.
3. Winner gets award, loser gets penalty.

- Games as \textit{search}:
  - \textit{Initial state}: e.g. board configuration of chess
  - \textit{Successor function}: list of (move,state) pairs specifying legal moves.
  - \textit{Terminal test}: Is the game finished?
  - \textit{Utility function}: Gives numerical value of terminal states. E.g. win (+1), lose (-1) and draw (0) in tic-tac-toe
  - \textit{MAX} uses search tree to determine next move.
How to Play a Game by Searching

- General Scheme
  1. Consider all legal successors to the current state (‘board position’)
  2. Evaluate each successor board position
  3. Pick the move which leads to the best board position.
  4. After your opponent moves, repeat.

- Design issues
  1. Representing the ‘board’
  2. Representing legal next boards
  3. Evaluating positions
  4. Looking ahead
Hexapawn: A very Simple Game

- Hexapawn is played on a 3x3 chessboard

- Only standard pawn moves:
  1. A pawn moves forward one square onto an empty square
  2. A pawn “captures” an opponent pawn by moving diagonally forward one square, if that square contains an opposing pawn. The opposing pawn is removed from the board.
Hexapawn: A very Simple Game

- Hexapawn is played on a 3x3 chessboard

- Player $P_1$ wins the game against $P_2$ when:
  - One of $P_1$’s pawns reaches the far side of the board.
  - $P_2$ cannot move because no legal move is possible.
  - $P_2$ has no pawns left.

(Invented by Martin Gardner; gives learning “program” with match boxes. Reprinted in “The Unexpected Hanging..”)
Hexapawn: Three Possible First Moves

White moves
Game Trees

- Represent the game problem space by a tree:
  - Nodes represent ‘board positions’; edges represent legal moves.
  - Root node is the first position in which a decision must be made.
Hexapawn: Simplified Game Tree for 2 Moves

White to move

Black to move

White to move
MAX & MIN Nodes: An egocentric view

- Two players: MAX, MAX’s opponent MIN
- All play is computed from MAX’s vantage point.
- When MAX moves, MAX attempts to MAXimize MAX’s outcome.
- When MAX’s opponent moves, they attempt to MINimize MAX’s outcome.

WE TYPICALLY ASSUME MAX MOVES FIRST:

- Label the root (level 0) MAX
- Alternate MAX/MIN labels at each successive tree level (ply).
- Even levels represent turns for MAX
- Odd levels represent turns for MIN

Max (von Sydow) plays chess....
Game Trees

- Represent the game problem space by a tree:
  - Nodes represent ‘board positions’; edges represent legal moves.
  - Root node is the first position in which a decision must be made.

- Evaluation function $f$ assigns real-number scores to `board positions’ *without reference to path*

- Terminal nodes represent ways the game could end, labeled with the desirability of that ending (e.g. win/lose/draw or a numerical score)
Evaluation functions: $f(n)$

- Evaluates how good a ‘board position’ is
  - Based on static features of that board alone
- Zero-sum assumption lets us use one function to describe goodness for both players.
  - $f(n)>0$ if MAX is winning in position $n$
  - $f(n)=0$ if position $n$ is tied
  - $f(n)<0$ if MIN is winning in position $n$
- Build using expert knowledge,
  - Tic-tac-toe: $f(n)=\text{(number of 3 lengths open for MAX)} - \text{(number open for MIN)}$

(AIMA 5.4.1)
A Partial Game Tree for Tic-Tac-Toe

\[ \text{MAX (X)} \]

\[ \text{MIN (O)} \]

\[ f(n) = 8 - 5 = 3 \]

\[ f(n) = 6 - 5 = 1 \]

\[ f(n) = 6 - 3 = 3 \]

\[ f(n) = 6 - 4 = 2 \]

\[ f(n) = 6 - 2 = 4 \]

\[ f(n) = \# \text{ of potential three-lines for X} - \# \text{ of potential three-line for Y if } n \text{ is not terminal} \]

\[ f(n) = 0 \text{ if } n \text{ is a terminal tie} \]

\[ f(n) = + \infty \text{ if } n \text{ is a terminal win} \]

\[ f(n) = - \infty \text{ if } n \text{ is a terminal loss} \]
Chess Evaluation Functions

- Alan Turing’s
  \[ f(n) = (\text{sum of } A's \text{ piece values}) - (\text{sum of } B's \text{ piece values}) \]

- More complex: weighted sum of positional features:
  \[ \sum w_i \cdot \text{feature}_i(n) \]

- Deep Blue has > 8000 features

<table>
<thead>
<tr>
<th>Piece</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pawn</td>
<td>1.0</td>
</tr>
<tr>
<td>Knight</td>
<td>3.0</td>
</tr>
<tr>
<td>Bishop</td>
<td>3.25</td>
</tr>
<tr>
<td>Rook</td>
<td>5.0</td>
</tr>
<tr>
<td>Queen</td>
<td>9.0</td>
</tr>
</tbody>
</table>

Pieces values for a simple Turing-style evaluation function often taught to novice chess players

**Positive:** rooks on open files, knights in closed positions, control of the center, developed pieces

**Negative:** doubled pawns, wrong-colored bishops in closed positions, isolated pawns, pinned pieces

Examples of more complex features
Some Chess Positions and their Evaluations

White to move
\[ f(n) = (9+3) - (5+5+3.25) \]
\[ = -1.25 \]

So, considering our opponent’s possible responses would be wise.
The Minimax Rule (AIMA 5.2)
The Minimax Rule: `Don’t play hope chess’

Idea: Make the best move for MAX assuming that MIN always replies with the best move for MIN

Easily computed by a recursive process

- The backed-up value of each node in the tree is determined by the values of its children:
  - For a MAX node, the backed-up value is the maximum of the values of its children (i.e. the best for MAX)
  - For a MIN node, the backed-up value is the minimum of the values of its children (i.e. the best for MIN)
The Minimax Procedure

Until game is over:

1. Start with the current position as a MAX node.
2. Expand the game tree a fixed number of ply.
3. Apply the evaluation function to the leaf positions.
4. Calculate back-up values bottom-up.
5. Pick the move assigned to MAX at the root.
6. Wait for MIN to respond.
2-ply Example: Backing up values

Evaluation function value

This is the move selected by minimax
What if MIN does not play optimally?

- Definition of optimal play for MAX assumes MIN plays optimally:
  - *Maximizes worst-case outcome* for MAX.
  - (Classic game theoretic strategy)

- But if MIN does not play optimally, MAX will do even better. [Theorem-not hard to prove]
Comments on Minimax Search

- **Depth-first search with fixed number of ply \( m \) as the limit.**
  - \( O(b^m) \) time complexity – *As usual!*
  - \( O(bm) \) space complexity

- **Performance will depend on**
  - the quality of the static evaluation function (expert knowledge)
  - depth of search (computing power and search algorithm)

- **Differences from normal state space search**
  - Looking to make *one* move only, despite deeper search
  - No cost on arcs – costs from backed-up static evaluation
  - MAX can’t be sure how MIN will respond to his moves

- **Minimax forms the basis for other game tree search algorithms.**
Alpha-Beta Pruning (AIMA 5.3)

Many slides adapted from Richard Lathrop, USC/ISI, CS 271
Alpha-Beta Pruning

- A way to improve the performance of the Minimax Procedure
- Basic idea: "If you have an idea which is surely bad, don’t take the time to see how truly awful it is" ~ Pat Winston

We don’t need to compute the value at this node.

No matter what it is it can’t effect the value of the root node.
General alpha-beta pruning idea

- Consider a node \( n \) in the tree

- If player has a better choice TO WHAT???? at:
  - Parent node of \( n \)
  - Or any choice point further up

- Then \( n \) will never be reached in play.

- Hence, when that much is known about \( n \), it can be pruned.
Alpha-Beta Pruning II

- During Minimax, keep track of two additional values:
  - $\alpha$: MAX’s current *lower* bound on MAX’s outcome
  - $\beta$: MIN’s current *upper* bound on MIN’s outcome

- MAX will never allow a move that could lead to a worse score (for MAX) than $\alpha$
- MIN will never allow a move that could lead to a better score (for MAX) than $\beta$

- Therefore, stop evaluating a branch whenever:
  - When evaluating a MAX node: a value $v \geq \beta$ is backed-up
    — MIN will never select that MAX node
  - When evaluating a MIN node: a value $v \leq \alpha$ is found
    — MAX will never select that MIN node