Games and adversarial search
(Chapter 5)

World Champion chess player Garry Kasparov is defeated by IBM’s Deep Blue chess-playing computer in a six-game match in May, 1997 (link)
Why study games?

• Games are a traditional hallmark of intelligence
• Games are easy to formalize
• Games can be a good model of real-world competitive or cooperative activities
  – Military confrontations, negotiation, auctions, etc.
# Types of game environments

<table>
<thead>
<tr>
<th></th>
<th>Deterministic</th>
<th>Stochastic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect information</td>
<td>Chess, checkers, go</td>
<td>Backgammon, monopoly</td>
</tr>
<tr>
<td>(fully observable)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imperfect information</td>
<td>Battleships</td>
<td>Scrabble, poker, bridge</td>
</tr>
<tr>
<td>(partially observable)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alternating two-player zero-sum games

- Players take turns
- Each game outcome or terminal state has a utility for each player (e.g., 1 for win, 0 for loss)
- The sum of both players’ utilities is a constant
Games vs. single-agent search

• We don’t know how the opponent will act
  – The solution is not a fixed sequence of actions from start state to goal state, but a strategy or policy (a mapping from state to best move in that state)

• Efficiency is critical to playing well
  – The time to make a move is limited
  – The branching factor, search depth, and number of terminal configurations are huge
    • In chess, branching factor $\approx 35$ and depth $\approx 100$, giving a search tree of $10^{154}$ nodes
      – Number of atoms in the observable universe $\approx 10^{80}$
  – This rules out searching all the way to the end of the game
Game tree

- A game of tic-tac-toe between two players, “max” and “min”
COMPLETE MAP OF OPTIMAL TIC-TAC-TOE MOVES

YOUR MOVE IS GIVEN BY THE POSITION OF THE LARGEST RED SYMBOL ON THE GRID. WHEN YOUR OPPONENT PICKS A MOVE, ZOOM IN ON THE REGION OF THE GRID WHERE THEY WENT. REPEAT.

MAP FOR X:

http://xkcd.com/832/
A more abstract game tree

Terminal utilities (for MAX)

A two-\textit{ply} game
• **Minimax value of a node**: the utility (for MAX) of being in the corresponding state, assuming perfect play on both sides

• **Minimax strategy**: Choose the move that gives the best worst-case payoff
Computing the minimax value of a node

• \( \text{Minimax}(node) = \)
  - \( \text{Utility}(node) \) if \( node \) is terminal
  - \( \max_{\text{action}} \text{Minimax}(\text{Succ}(node, action)) \) if \( \text{player} = \text{MAX} \)
  - \( \min_{\text{action}} \text{Minimax}(\text{Succ}(node, action)) \) if \( \text{player} = \text{MIN} \)
Optimality of minimax

• The minimax strategy is optimal against an optimal opponent
• What if your opponent is suboptimal?
  – Your utility can only be higher than if you were playing an optimal opponent!
  – A different strategy may work better for a sub-optimal opponent, but it will necessarily be worse against an optimal opponent

Example from D. Klein and P. Abbeel
More general games

- More than two players, non-zero-sum
- Utilities are now tuples
- Each player maximizes their own utility at their node
- Utilities get propagated (backed up) from children to parents
Alpha-beta pruning

• It is possible to compute the exact minimax decision without expanding every node in the game tree
Alpha-beta pruning

• It is possible to compute the exact minimax decision without expanding every node in the game tree.
Alpha-beta pruning

• It is possible to compute the exact minimax decision without expanding every node in the game tree.
Alpha-beta pruning

• It is possible to compute the exact minimax decision without expanding every node in the game tree.
Alpha-beta pruning

• It is possible to compute the exact minimax decision without expanding every node in the game tree.
Alpha-beta pruning

• It is possible to compute the exact minimax decision without expanding every node in the game tree
Alpha-beta pruning

• $\alpha$ is the value of the best choice for the MAX player found so far at any choice point above node $n$

• We want to compute the MIN-value at $n$

• As we loop over $n$’s children, the MIN-value decreases

• If it drops below $\alpha$, MAX will never choose $n$, so we can ignore $n$’s remaining children

• Analogously, $\beta$ is the value of the lowest-utility choice found so far for the MIN player
Alpha-beta pruning

**Function** $\text{action} = \text{Alpha-Beta-Search}(\text{node})$

$v = \text{Min-Value}(\text{node}, -\infty, \infty)$

return the action from node with value $v$

$\alpha$: best alternative available to the Max player
$\beta$: best alternative available to the Min player

**Function** $v = \text{Min-Value}(\text{node}, \alpha, \beta)$

if Terminal(\text{node}) return Utility(\text{node})

$v = +\infty$

for each action from node

$v = \text{Min}(v, \text{Max-Value}(\text{Succ}(\text{node}, \text{action}), \alpha, \beta))$

if $v \leq \alpha$ return $v$

$\beta = \text{Min}(\beta, v)$

end for

return $v$
Alpha-beta pruning

Function $\text{action} = \text{Alpha-Beta-Search}(\text{node})$

$v = \text{Max-Value}(\text{node}, -\infty, \infty)$

return the action from node with value $v$

$\alpha$: best alternative available to the Max player

$\beta$: best alternative available to the Min player

Function $v = \text{Max-Value}(\text{node}, \alpha, \beta)$

if Terminal($\text{node}$) return Utility($\text{node}$)

$v = -\infty$

for each action from node

$v = \text{Max}(v, \text{Min-Value}(\text{Succ}(\text{node}, \text{action}), \alpha, \beta))$

if $v \geq \beta$ return $v$

$\alpha = \text{Max}(\alpha, v)$

end for

return $v$
Alpha-beta pruning

• Pruning does not affect final result
• Amount of pruning depends on move ordering
  – Should start with the “best” moves (highest-value for MAX or lowest-value for MIN)
  – For chess, can try captures first, then threats, then forward moves, then backward moves
  – Can also try to remember “killer moves” from other branches of the tree
• With perfect ordering, the time to find the best move is reduced to $O(b^{m/2})$ from $O(b^m)$
  – Depth of search is effectively doubled
Evaluation function

• Cut off search at a certain depth and compute the value of an evaluation function for a state instead of its minimax value
  – The evaluation function may be thought of as the probability of winning from a given state or the expected value of that state
• A common evaluation function is a weighted sum of features:

\[ \text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

  – For chess, \( w_k \) may be the material value of a piece (pawn = 1, knight = 3, rook = 5, queen = 9) and \( f_k(s) \) may be the advantage in terms of that piece
• Evaluation functions may be learned from game databases or by having the program play many games against itself
Cutting off search

• **Horizon effect:** you may incorrectly estimate the value of a state by overlooking an event that is just beyond the depth limit
  – For example, a damaging move by the opponent that can be delayed but not avoided

• Possible remedies
  – **Quiescence search:** do not cut off search at positions that are unstable – for example, are you about to lose an important piece?
  – **Singular extension:** a strong move that should be tried when the normal depth limit is reached
Advanced techniques

- **Transposition table** to store previously expanded states
- **Forward pruning** to avoid considering all possible moves
- **Lookup tables** for opening moves and endgames
Chess playing systems

• Baseline system: 200 million node evaluations per move (3 min), minimax with a decent evaluation function and quiescence search
  – 5-ply ≈ human novice
• Add alpha-beta pruning
  – 10-ply ≈ typical PC, experienced player
• Deep Blue: 30 billion evaluations per move, singular extensions, evaluation function with 8000 features, large databases of opening and endgame moves
  – 14-ply ≈ Garry Kasparov
• Recent state of the art (Hydra, ca. 2006): 36 billion evaluations per second, advanced pruning techniques
  – 18-ply ≈ better than any human alive?
Monte Carlo Tree Search

• What about games with deep trees, large branching factor, and no good heuristics – like Go?
• Instead of depth-limited search with an evaluation function, use randomized simulations
• Starting at the current state (root of search tree), iterate:
  – Select a leaf node for expansion using a tree policy (trading off exploration and exploitation)
  – Run a simulation using a default policy (e.g., random moves) until a terminal state is reached
  – Back-propagate the outcome to update the value estimates of internal tree nodes

C. Browne et al., *A survey of Monte Carlo Tree Search Methods*, 2012
Stochastic games

• How to incorporate dice throwing into the game tree?
Stochastic games

MAX

CHANCE

MIN

CHANCE

MAX

TERMINAL

2 -1 1 -1 1
Stochastic games

- **Expectiminimax**: for chance nodes, sum values of successor states weighted by the probability of each successor

- **Value**(node) =
  - Utility(node) if node is terminal
  - $\max_{\text{action}} \text{Value}(\text{Succ}(\text{node}, \text{action}))$ if type = MAX
  - $\min_{\text{action}} \text{Value}(\text{Succ}(\text{node}, \text{action}))$ if type = MIN
  - $\sum_{\text{action}} P(\text{Succ}(\text{node}, \text{action})) \times \text{Value}(\text{Succ}(\text{node}, \text{action}))$ if type = CHANCE
Stochastic games

• **Expectiminimax**: for chance nodes, sum values of successor states weighted by the probability of each successor
  – Nasty branching factor, defining evaluation functions and pruning algorithms more difficult

• **Monte Carlo simulation**: when you get to a chance node, simulate a large number of games with random dice rolls and use win percentage as evaluation function
  – Can work well for games like Backgammon
Fig. 1. Portion of the extensive-form game representation of three-card Kuhn poker (16). Player 1 is dealt a queen (Q), and the opponent is given either the jack (J) or king (K). Game states are circles labeled by the player acting at each state (“c” refers to chance, which randomly chooses the initial deal). The arrows show the events the acting player can choose from, labeled with their in-game meaning. The leaves are square vertices labeled with the associated utility for player 1 (player 2’s utility is the negation of player 1’s). The states connected by thick gray lines are part of the same information set; that is, player 1 cannot distinguish between the states in each pair because they each represent a different unobserved card being dealt to the opponent. Player 2’s states are also in information sets, containing other states not pictured in this diagram.
Stochastic games of imperfect information

• Simple Monte Carlo approach: run multiple simulations with random cards pretending the game is fully observable
  – “Averaging over clairvoyance”
  – Problem: this strategy does not account for bluffing, information gathering, etc.
Game AI: Origins

- Minimax algorithm: Ernst Zermelo, 1912
- Chess playing with evaluation function, quiescence search, selective search: Claude Shannon, 1949 (paper)
- Alpha-beta search: John McCarthy, 1956
- Checkers program that learns its own evaluation function by playing against itself: Arthur Samuel, 1956
Game AI: State of the art

• Computers are better than humans:
  – **Checkers**: solved in 2007
  – **Chess**: IBM Deep Blue defeated Kasparov in 1997

• Computers are competitive with top human players:
  – **Backgammon**: TD-Gammon system used reinforcement learning to learn a good evaluation function
  – **Bridge**: top systems use Monte Carlo simulation and alpha-beta search
Game AI: State of the art

• Computers are not competitive with top human players:
  – **Poker**
    • **Heads-up limit hold’em poker has been solved** (Science, Jan. 2015)
      – Simplest variant played competitively by humans
      – Smaller number of states than checkers, but partial observability makes it difficult
      – *Essentially weakly solved* = cannot be beaten with statistical significance in a lifetime of playing
    • Huge increase in difficulty from limit to no-limit poker
    • **But online poker bots proliferate**
  – **Go**
    • Branching factor 361, no good evaluation functions have been found
    • Best existing systems use Monte Carlo Tree Search and pattern databases
    • New approaches: **deep learning** (44% accuracy for move prediction, can win against other strong Go AI)
DIFFICULTY OF VARIOUS GAMES FOR COMPUTERS

EASY

SOLVED
COMPUTERS CAN PLAY PERFECTLY

SOLVED FOR ALL POSSIBLE POSITIONS

SOLVED FOR STARTING POSITIONS

COMPUTERS CAN BEAT TOP HUMANS

HARD

http://xkcd.com/1002/

See also: http://xkcd.com/1263/