Games and adversarial search
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 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 (fully observable)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Imperfect information (partially observable)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Types of game environments

<table>
<thead>
<tr>
<th>Perfect information (fully observable)</th>
<th>Deterministic</th>
<th>Stochastic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess, checkers, go</td>
<td></td>
<td>Backgammon, monopoly</td>
</tr>
<tr>
<td>Imperfect information (partially observable)</td>
<td>Battleships</td>
<td>Scrabble, poker, bridge</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
  – 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-ply* game
A more abstract game tree

- **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 state

\[
\text{Minimax}(\text{state}) =
\begin{align*}
\text{Utility}(\text{state}) & \quad \text{if state is terminal} \\
\max \text{ Minimax}(\text{successors}(\text{state})) & \quad \text{if player} = \text{MAX} \\
\min \text{ Minimax}(\text{successors}(\text{state})) & \quad \text{if player} = \text{MIN}
\end{align*}
\]
The minimax strategy is optimal against an optimal opponent
- If the opponent is sub-optimal, the utility can only be higher
- A different strategy may work better for a sub-optimal opponent, but it will necessarily be worse against an optimal opponent
More general games

- More than two players, non-zero-sum
- Utilities are now tuples
- Each player maximizes their own utility at each 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

```
MAX

3

MIN

3

≥3

3 12 8
```
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 $n$
• 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 take this branch, 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

• 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:

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

  – 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
Additional 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): 36 billion evaluations per second, advanced pruning techniques
  - 18-ply ≈ better than any human alive?
Games of chance

• How to incorporate dice throwing into the game tree?
Games of chance
Games of chance

• **Expectiminimax**: for chance nodes, average values weighted by the probability of each outcome
  – 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
Partially observable games

- Card games like bridge and poker
- Monte Carlo simulation: deal all the cards randomly in the beginning and pretend the game is fully observable
  - “Averaging over clairvoyance”
  - Problem: this strategy does not account for bluffing, information gathering, etc.
Game playing algorithms today

- 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
- Computers are not competitive
  - **Go:** branching factor 361. Existing systems use Monte Carlo simulation and pattern databases
Origins of game playing algorithms

• Ernst Zermelo (1912): Minimax algorithm
• Claude Shannon (1949): chess playing with evaluation function, quiescence search, selective search (paper)
• John McCarthy (1956): Alpha-beta search
• Arthur Samuel (1956): checkers program that learns its own evaluation function by playing against itself
Review: Games

• What is a zero-sum game?
• What’s the optimal strategy for a player in a zero-sum game?
• How do you compute this strategy?
Review: Minimax

• **Minimax**(state) =
  - Utility(state) if state is terminal
  - max Minimax(successors(state)) if player = MAX
  - min Minimax(successors(state)) if player = MIN
Review: Games

- Efficiency of alpha-beta pruning
- Evaluation functions
- Horizon effect
- Quiescence search
- Additional techniques for improving efficiency
- Stochastic games, partially observable games