ARTIFICIAL INTELLIGENCE

Russell & Norvig
Chapter 5: Adversarial Search
Why study games?

- Games can be a good model of many competitive activities
- Games are a traditional hallmark of intelligence
- State of game is easy to represent and there are a small number of actions with precise rules
- Unlike "toy" problems, games are interesting because they are too hard to solve (e.g. search).
## 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>Chess, checkers, Connect 4</td>
<td>Backgammon, monopoly</td>
</tr>
<tr>
<td>Imperfect information (partially observable)</td>
<td>Battleship</td>
<td>Scrabble, poker, bridge</td>
</tr>
</tbody>
</table>
Alternating two-player zero-sum games

• Two players: Max and Min
• Players take turns, Max goes first
• Alternate until end of game
• Each game outcome or terminal state has a utility for each player (e.g., +1 for win, 0 for tie, -1 for loss)
• Zero-sum is where the total payoff to all players is the same for every instance of the game. In Chess 1+0, 0+1, ½+½
Games as search

- $S_0$ is initial state (how game setup at start)
- Player(s) which player has move in state $s$
- Actions(s) is set of legal moves in a state $s$
- Result(s,a) is result of a move (transition model)
- Terminal-Test(s) returns true when game is over; else false
- Utility(s,p) is utility/objective/payoff function defines the numeric value for a game that ends in a terminal state $s$ for a player $p$
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
• A game of tic-tac-toe between two players, “max” and “min”
Game Playing - Minimax

- Game Playing: An opponent tries to thwart your every move

- Minimax is a search method that maximizes your position while minimizing your opponents position

- We need a method of measuring “goodness” of a position, a utility function (or payoff function)
  - e.g. outcome of a game; win 1, loss -1, draw 0

- Uses recursive DFS solution
Minimax for two-ply game

Terminal utilities (for MAX)

Gives best achievable payoff if both players play perfectly
Minimax Strategy

- 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
Properties of minimax

- **Complete?** Yes (if tree is finite)
- **Optimal?** Yes (against an optimal opponent)
- **Time complexity?** $O(b^m)$
- **Space complexity?** $O(bm)$ (depth-first exploration)
- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games → exact solution completely infeasible
- Do we need to explore every path? NO!
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

![Game tree diagram with values 3, 12, 8, 2, 14, 5, 2]
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.
function ALPHA-BETA-SEARCH(state) returns an action
    \( v \leftarrow \text{MAX-VALUE}(state, -\infty, +\infty) \)
    return the action in ACTIONS(state) with value \( v \)

function MAX-VALUE(state, \( \alpha, \beta \)) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    \( v \leftarrow -\infty \)
    for each \( a \) in ACTIONS(state) do
        \( v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(	ext{RESULT}(s,a), \alpha, \beta)) \)
        if \( v \geq \beta \) then return \( v \)
        \( \alpha \leftarrow \text{MAX}(\alpha, v) \)
    return \( v \)

function MIN-VALUE(state, \( \alpha, \beta \)) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    \( v \leftarrow +\infty \)
    for each \( a \) in ACTIONS(state) do
        \( v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(	ext{RESULT}(s,a), \alpha, \beta)) \)
        if \( v \leq \alpha \) then return \( v \)
        \( \beta \leftarrow \text{MIN}(\beta, v) \)
    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, branching factor can be cut in two, or depth of search 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:

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

- 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
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?
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
Games of chance

MAX

CHANCE

MIN

CHANCE

MAX

TERMINAL

2 -1 1 -1 1
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.