Game Playing
AI Class 8 — Ch. 5.1-5.3, 5.4.1, 5.5

Today’s Class
- Game playing
  - State of the art and resources
  - Framework
- Game trees
  - Minimax
  - Alpha-beta pruning
  - Adding randomness

Why Games?
- Clear criteria for success
- Offer an opportunity to study problems involving {hostile / adversarial / competing} agents.
- Interesting, hard problems which require minimal setup
- Often define very large search spaces
  - chess $35^{100}$ nodes in search tree, $10^{40}$ legal states
- Historical reasons
- Fun! (Mostly.)

State-of-the-art
- How good are computer game players?
  - Chess:
    - Deep Blue beat Gary Kasparov in 1997
    - Kasparov vs. Deep Junior (Feb 2003) tie!
  - Kasparov vs. X3D Fritz (November 2003): tie!
- Checkers:
  - Chinook (an AI program with a very large endgame database) is the world champion and can provably never be beaten. Retired in 1995
  - Computer players have finally reached tournament-level play
- Bridge
  - "Expert-level" computer players exist (but no world champions yet!) Good places to learn more:
  - http://www.cs.ualberta.ca/~chinook/
  - http://www.cs.unimass.nl/sga

Chinook
- World Man-Machine Checkers Champion, developed by researchers at the University of Alberta.
- Earned this title by competing in human tournaments, winning the right to play for the world championship, eventually defeating the best players in the world.
- Visit http://www.cs.ualberta.ca/~chinook/ to play!
- Developers have fully analyzed the game of checkers, and can provably never be beaten http://www.cs.ualberta.ca/~sga/sga/sga.html
Typical Games

- 2-person game
- Players alternate moves
- Zero-sum: one player's loss is the other's gain
- Perfect information: both players have access to complete information about the state of the game. No information is hidden from either player.
- Deterministic: No chance (e.g., dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- Not: Bridge, Solitaire, Backgammon, ...

How to Play (How to Search)

- Obvious approach:
  - From current game state:
    - Consider all the legal moves you can make
    - Compute new position resulting from each move
    - Evaluate each resulting position
    - Decide which is best
    - Make that move
    - Wait for your opponent to move and repeat
- Key problems are:
  - Representing the "board"
  - Generating all legal next boards
  - Evaluating a position

Evaluation function

- Evaluation function or static evaluator is used to evaluate the "goodness" of a game position
  - Unlike heuristic search, where evaluation function is a positive estimate of cost from start node to a goal, passing through n
- Zero-sum assumption allows one evaluation function to describe goodness of a board for both players (how?)
  - $f(n) \gg 0$: position n is good for me and bad for you
  - $f(n) \ll 0$: position n is bad for me and good for you
  - $f(n) = 0 \pm \epsilon$: position n is a neutral position
  - $f(n) = +\infty$: win for me
  - $f(n) = -\infty$: win for you

Evaluation function examples

- Example of an evaluation function for Tic-Tac-Toe:
  - $f(n) = [3-lengths open for \times] - [3-lengths open for O]
  - A 3-length is a complete row, column, or diagonal
- Alan Turing's function for chess
  - $f(n) = w(n)/b(n)$
  - $w(n) = \text{sum of the point value of white's pieces}$
  - $b(n) = \text{sum of black's}$
**Evaluation function examples**

- Most evaluation functions are specified as a **weighted sum** of position features:
  \[ f(n) = w_1 \times \text{feat}_1(n) + w_2 \times \text{feat}_2(n) + ... + w_n \times \text{feat}_k(n) \]
- Example features for chess: piece count, piece placement, squares controlled, ...
- Deep Blue had over **8000** features in its evaluation function!

**Game trees**

- Problem spaces for typical games are represented as trees
- Player must decide best single **move** to make next
- Root node = the current board configuration
- Arcs = possible legal moves for a player

**Static evaluator function**
- Rates a board position
- \( f(\text{board}) = R \) with \( R > 0 \) “white” (me), \( R < 0 \) for black (you)
- If it is **my turn** to move:
  - Root is labeled “**MAX**” node
  - Otherwise it is a “**MIN**” node (**opponent’s turn**)
- Each level’s nodes are all **MAX** or all **MIN**
- Nodes at level \( i \) are opposite those at level \( i + 1 \)

**Minimax Procedure**

- Create start node: **MAX** node, current board state
- Expand nodes down to a **depth** of lookahead
- Apply evaluation function at each leaf node
- “Back up” values for each non-leaf node until a value is computed for the root node
  - **MIN**: backed-up value is **lowest** of children's values
  - **MAX**: backed-up value is **highest** of children's values
- Pick operator associated with the child node whose backed-up value set the value at the root

**Minimax Algorithm**

- [YouTube Video](https://www.youtube.com/watch?v=6ELUvkSkCts)
Example: Nim

- In Nim, there are a certain number of objects (coins, sticks, etc.) on the table – we’ll play 7-coin Nim
- Each player in turn has to pick up either one or two objects
- Whoever picks up the last object loses

Partial Game Tree for Tic-Tac-Toe

- \( f(n) = +1 \) if position is a win for X.
- \( f(n) = -1 \) if position is a win for O.
- \( f(n) = 0 \) if position is a draw.

Minimax Tree

Nim Game Tree

- **In-class exercise:**
  - Pair up (from ends, please)
  - Draw minimax search tree for 4-coin Nim
- **Things to consider:**
  - What’s your start state?
  - What’s the maximum depth of the tree? Minimum?

Alpha-beta Pruning

- **What is Pruning?**
- We can improve on the performance of the minimax algorithm through **alpha-beta pruning**
- Basic idea: "If you have an idea that is surely bad, don’t take the time to see how truly awful it is.” – Pat Winston

Alpha-beta Pruning

- **Traverse search tree in depth-first order**
- At each **MAX** node \( n \), \( \alpha(n) = \) maximum value found so far
- At each **MIN** node \( n \), \( \beta(n) = \) minimum value found so far
  - \( \alpha \) starts at \(-\infty\) and increases, \( \beta \) starts at \(+\infty\) and decreases
- **\( \beta \)-cutoff:** Given a **MAX** node \( n \),
  - Cut off search below \( n \) (i.e., don’t look at any more of \( n \)’s children) if:
    - \( \alpha(n) \geq \beta(i) \) for some **MIN** node ancestor \( i \) of \( n \)
- **\( \alpha \)-cutoff:**
  - Stop searching below **MIN** node \( n \) if:
    - \( \beta(n) \leq \alpha(i) \) for some **MAX** node ancestor \( i \) of \( n \)
**Alpha-beta Example**

- MAX
- MIN
- **Effectiveness of Alpha-beta**
  - Alpha-beta is guaranteed to:
    - Compute the same value for the root node as minimax
    - With $\leq$ computation
  - **Worst case**: no pruning, examining $b^d$ leaf nodes, where each node has $b$ children and a $d$-ply search is performed
  - **Best case**: examine only $(2b)^{d/2}$ leaf nodes.
    - Result is you can search twice as deep as minimax!
    - When each player's best move is the first alternative generated
  - In Deep Blue, empirically, alpha-beta pruning took average branching factor to 6 from about 35!

**Games of Chance**

- Backgammon: a two-player game with uncertainty
- Players roll dice to determine what moves to make
- White has just rolled 5 and 6 and has four legal moves:
  - 5-10, 5-11
  - 5-11, 10-16
  - 5-11, 11-16
- Good for decision making in adversarial problems with skill and luck

**Game trees with chance nodes**

- Chance nodes (shown as circles) represent random events
- For a random event with $N$ outcomes, each chance node has $N$ distinct children; a probability is associated with each
  - (For 2 dice, there are 21 distinct outcomes)
- Use minimax to compute values for MAX and MIN nodes
- Use expected values for chance nodes
  - For chance nodes over a max node, as in C:
    $$\text{expectimax}(C) = \sum_i (P(d_i) \times \text{maxvalue}(i))$$
  - For chance nodes over a min node:
    $$\text{expectimin}(C) = \sum_i (P(d_i) \times \text{minvalue}(i))$$

**Meaning of the evaluation function**

- Dealing with probabilities and expected values means we have to be careful about the “meaning” of values returned by the static evaluator.
- A “relative-order preserving” change of values would not change decision of minimax, but would change the decision with chance nodes.

**Example: Oopsy-Nim**

- Starts out like Nim
  - Each player in turn has to pick up either one or two objects
  - Sometimes (probability = 0.25), when you try to pick up two objects, you drop them both
  - Picking up a single object always works
- Question: Why can’t we draw the entire game tree?
- Exercise: Draw the 2-ply game tree (2 moves per player)
**Nim Game Tree**

- **In-class exercise:**
  - Pair up (from ends, please)
  - Draw minimax search tree for 4-coin Nim

- **Things to consider:**
  - What's your start state?
  - What's the maximum depth of the tree? Minimum?