CMSC 471

Games : Part 1

Why study games?

- Interesting, hard problems requiring minimal “initial structure”
- Clear criteria for success
- Study problems involving {hostile, adversarial, competing} agents and uncertainty of interacting with the natural world
- People have used them to assess their intelligence

Classical vs. Statistical approach

- We'll look first at the classical approach used from the 1940s to 2010
- Then at newer statistical approaches, of which AlphaGo is an example
- These share some techniques

Typical simple case for a game

- **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 state of game. No information hidden from either player.
- **No chance** (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- But not: Bridge, Solitaire, Backgammon, Poker, Rock-Paper-Scissors, ...

Can we use …

- Uninformed search?
- Heuristic search?
- Local search?
- Constraint based search?

None of these model the fact that we have an **adversary** …

How to play a game

- A way to play such a game is to:
  - Consider all the legal moves you can make
  - Compute new position resulting from each move
  - Evaluate each to determine which is best
  - Make that move
  - Wait for your opponent to move and repeat
- Key problems are:
  - Representing the "board" (i.e., game state)
  - Generating all legal next boards
  - Evaluating a position

Evaluation function

- **Evaluation function** or **static evaluator** used to evaluate the "goodness" of a game position

  Contrast with heuristic search, where evaluation function estimates **cost** from start node to goal passing through given node

- Zero-sum assumption permits single function to describe goodness of board for both players
  - **f(n) >> 0**: position n good for me; bad for you
  - **f(n) << 0**: position n bad for me; good for you
  - **f(n) near 0**: position n is a neutral position
  - **f(n) = +infinity**: win for me
  - **f(n) = -infinity**: win for you

Evaluation function examples

- **For Tic-Tac-Toe**

  f(n) = [# my open 3lengths] - [# your open 3lengths]

  Where a 3length is complete row, column or diagonal that has no opponent marks

- **Alan Turing's function for chess**
  - **f(n) = w(n)/b(n)** where w(n) = sum of point value of white's pieces and b(n) = sum of black's
  - Traditional piece values: pawn:1; knight:3; bishop:3; rook:5; queen:9

Evaluation function examples

- Most evaluation functions specified as a weighted sum of positive features

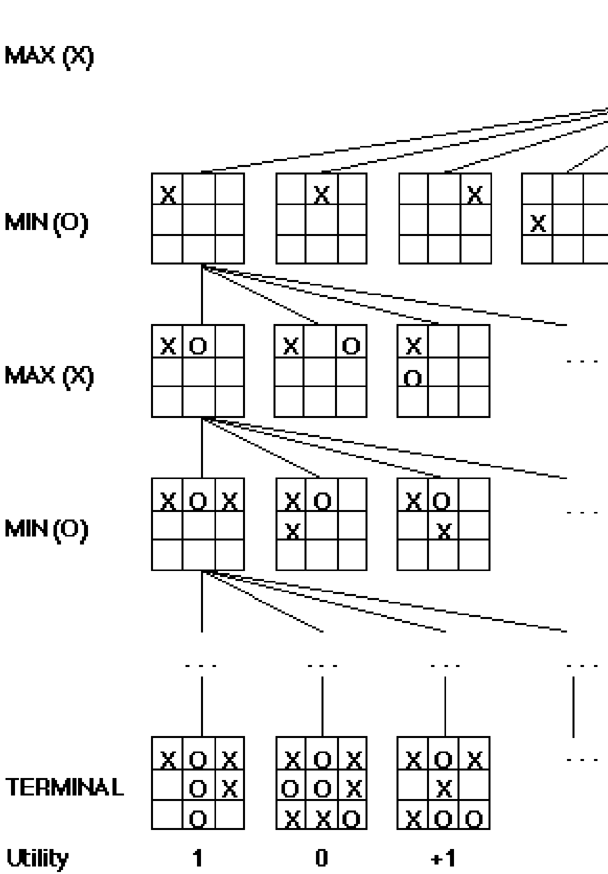
  $f(n) = w_1 * feat_1(n) + w_2 * feat_2(n) + \ldots + w_n * feat_k(n)$

- Typical chess features are piece count, piece values, piece placement, squares controlled, etc.
- IBM's chess program Deep Blue (circa 1996) had >8K features in its evaluation function

But, that’s not how people play

- People also use *look ahead*

  i.e., enumerate actions, consider opponent’s possible responses, REPEAT
- Producing a *complete* **game tree** is only possible for simple games
- So, generate a partial game tree for some number of plys
  - Move = each player takes a turn
  - Ply = one player’s turn
- What do we do with the game tree?



- We can easily generate a complete game tree for Tic-Tac-Toe
- Taking board symmetries into account, there are 138 terminal positions
- 91 wins for X, 44 for O and 3 draws

Minimax theorem

- Intuition: assume your opponent is at least as smart as you and play accordingly
  - If she’s not, you can only do better!
- You can think of this as:
  - Minimizing your maximum possible loss
  - Maximizing your minimum possible gain

Minimax Algorithm

- Mini-max algorithm is a recursive or backtracking algorithm that is used in decision-making and game theory.
- Mini-Max algorithm uses recursion to search through the game-tree.
- In this game, 2-players play the game. One is called **MAX**. The other is called **MIN**.

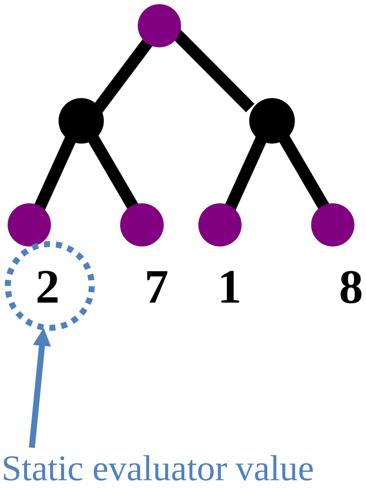
Minimax Algorithm

- Both players’ objective is to maximize their benefit and minimize the opponents benefit.
- The minimax algorithm performs a DFS for exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as recursion.

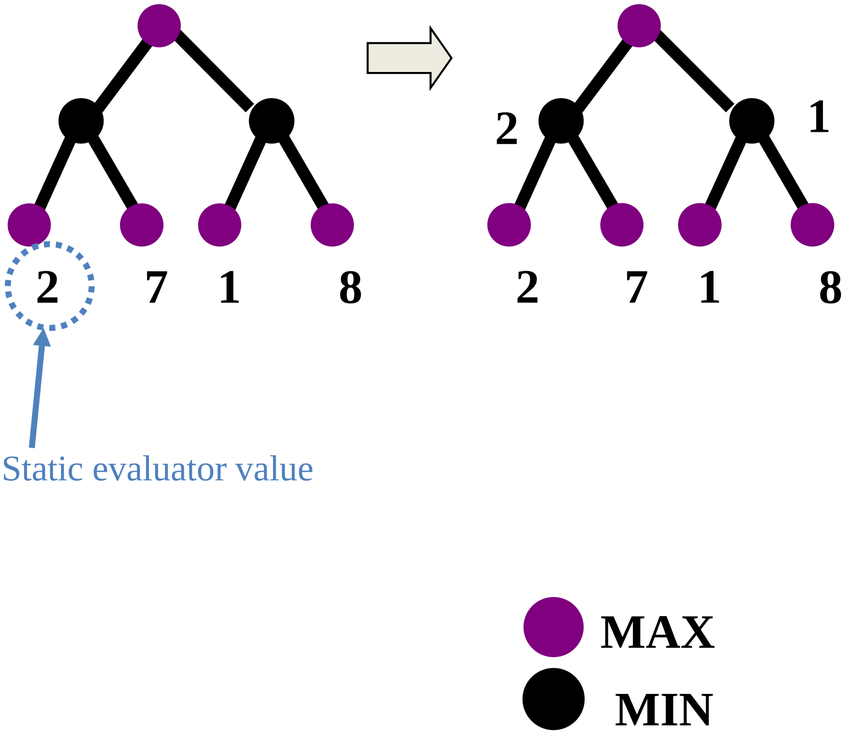
Minimax procedure

- Create MAX node with current board configuration
- Expand nodes to some **depth** (a.k.a. **plys**) of lookahead in game
- Apply evaluation function at each **leaf** node
- *Back up* values for each non-leaf node until value is computed for the root node
  - At MIN nodes: value is **minimum** of children's values
  - At MAX nodes: value is **maximum** of children's values
- Choose move to child node whose backed-up value determined value at root

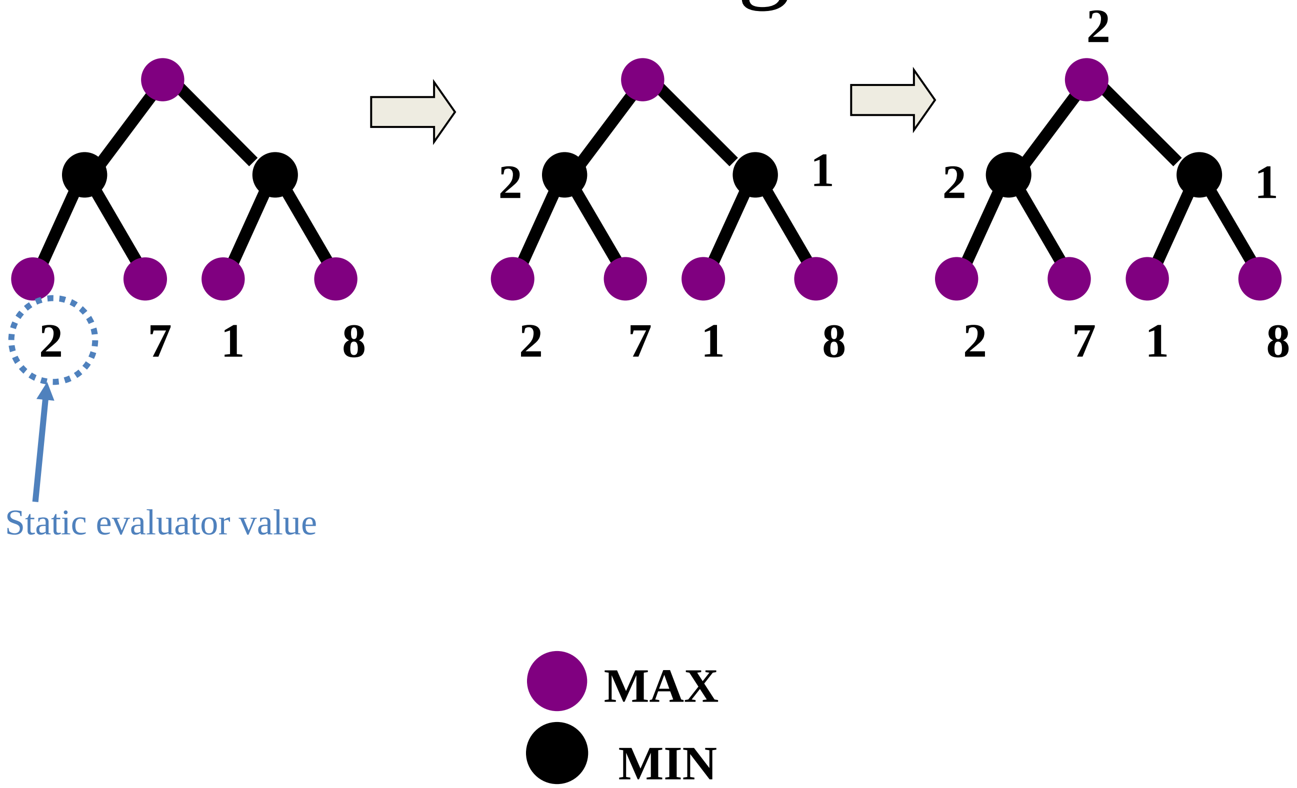
Minimax Algorithm



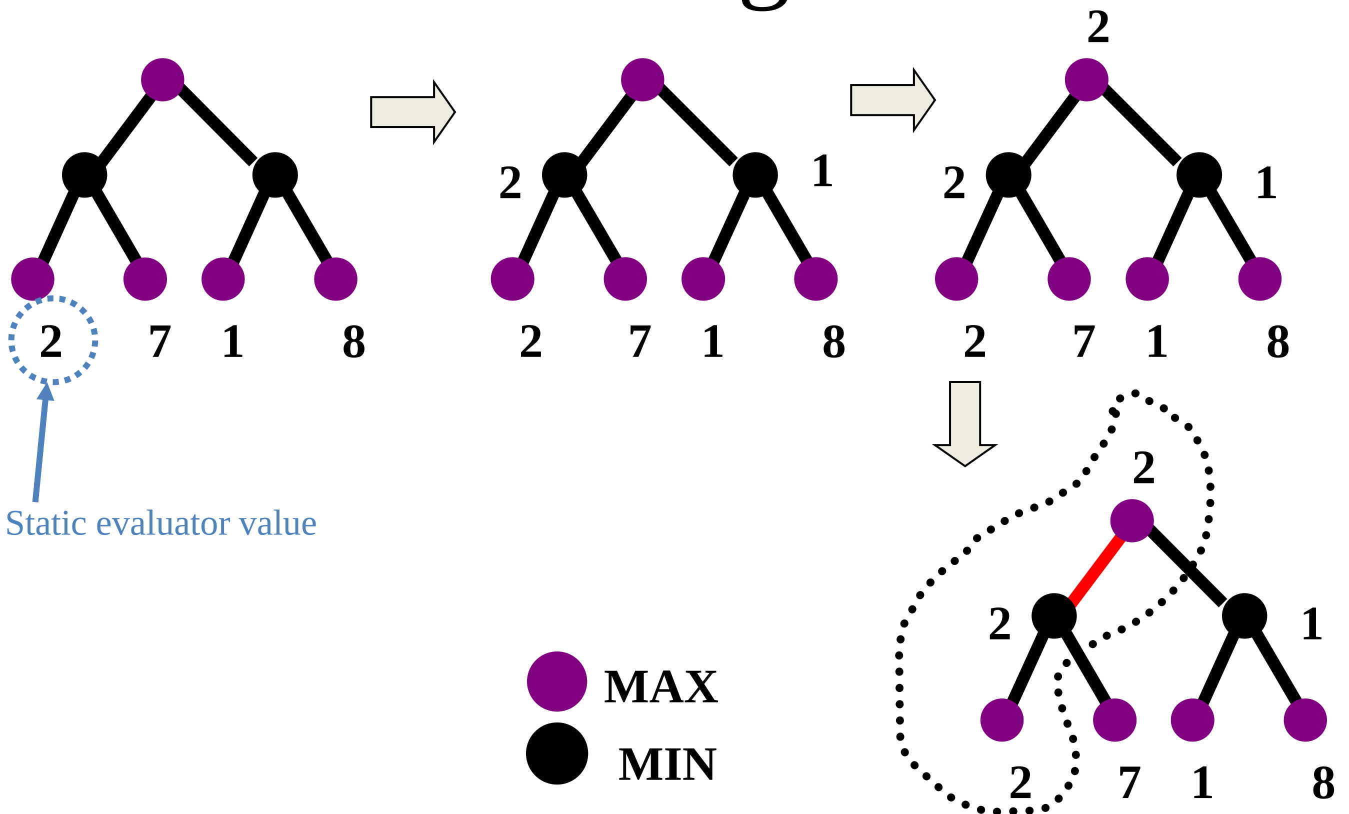
Minimax Algorithm



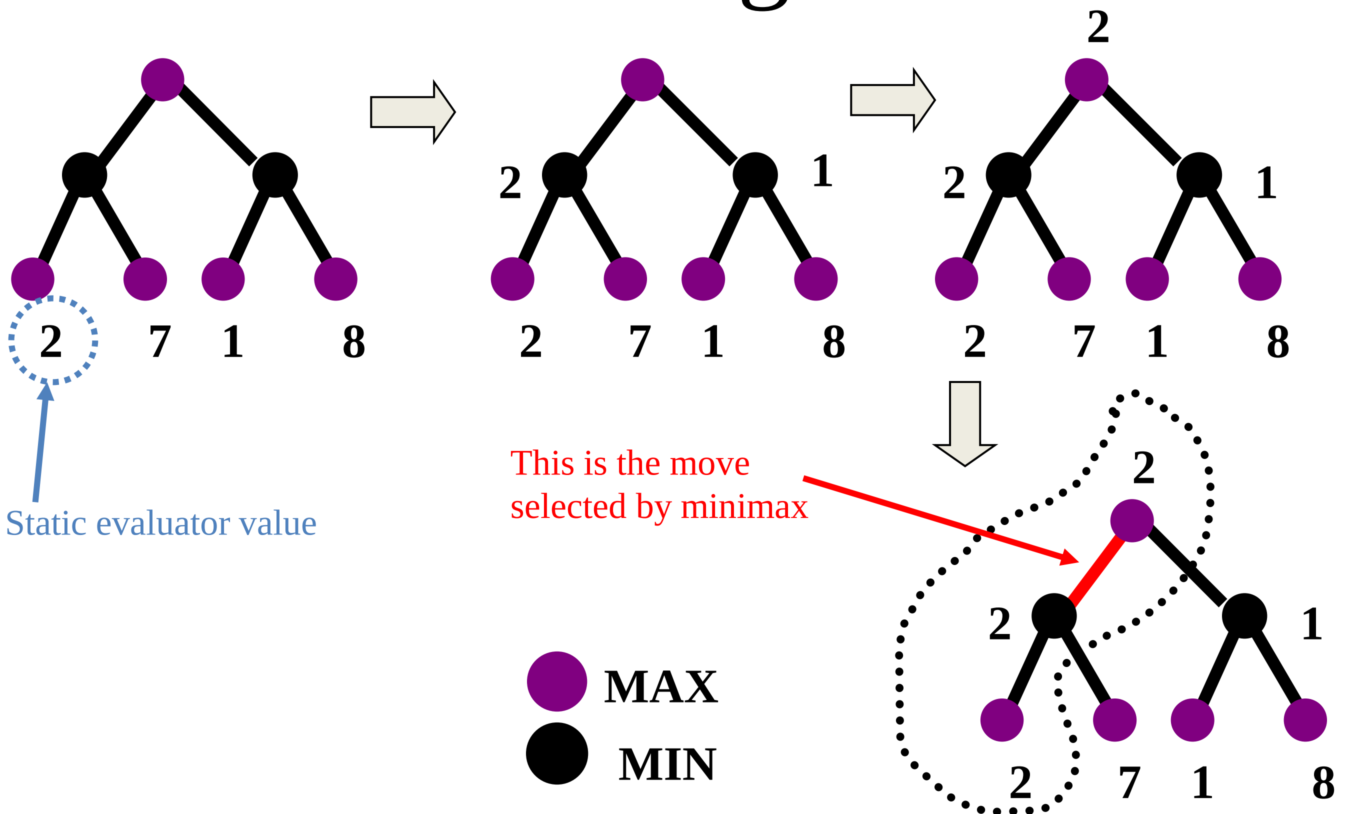
Minimax Algorithm



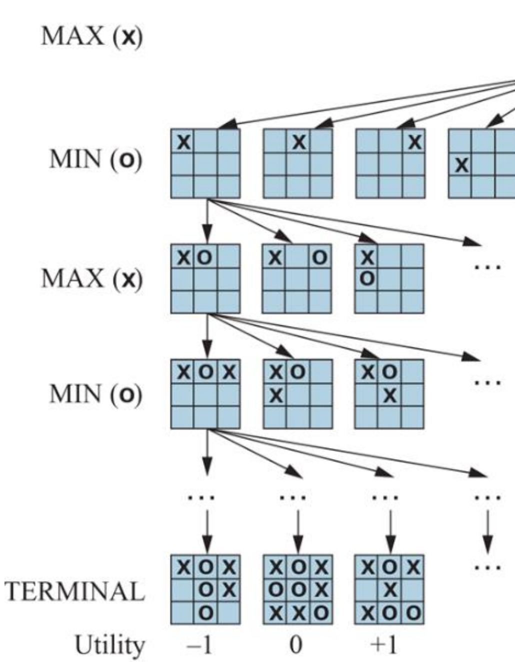
Minimax Algorithm



Minimax Algorithm



Partial Game Tree for Tic-Tac-Toe



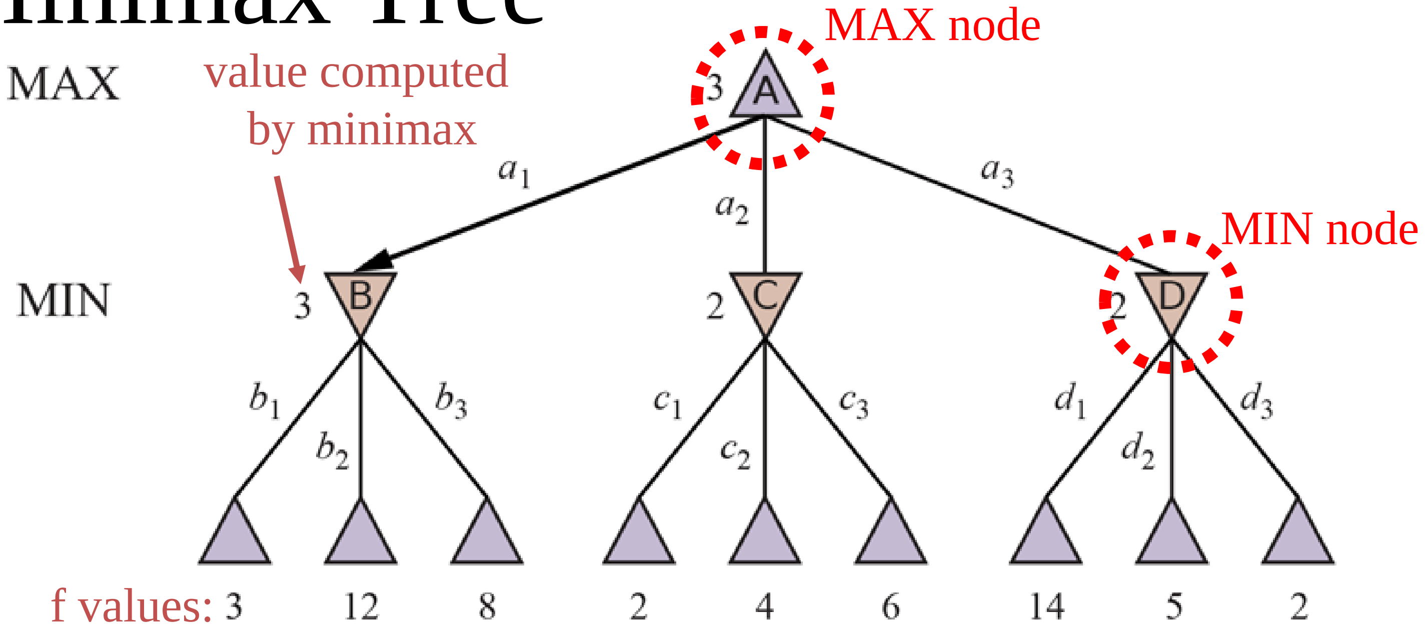
f(n)=+1 if position win for X

f(n)=-1 if position win for O

f(n)=0 if position a draw

Partial game tree for tic-tac-toe. Top node is the initial state, and max moves first, placing an X in an empty square. Only part of the tree shown, giving alternating moves by min (O) and max (X), until we reach terminal states, which are assigned utilities {-1,0,+1} for {loose, draw, win}

Minimax Tree



Two-ply game tree. △ nodes are “max nodes,” in which it is max’s turn to move, and ▽ nodes are “min nodes.” The terminal nodes show utility values for max; the other nodes are labeled with their minimax values. max’s best move at root is a1 because it leads to state with the highest minimax value. min’s best reply is b1 since it leads to the state with the lowest minimax value.