32: Games, 4

Broad Game Structure
zap2 (used in diags)
Game strategy and estimated value ("evalLeaf")
The referee

setup: modules we'll need:

MyGame /* the game we're playing */
Player1, Player 2 /* both Human players for now */
Player1 = HumanPlayer(MyGame)
Player2 = HumanPlayer(MyGame)
The referee's "play" procedure

let rec play: state => unit = ...

Consumes a state because we need to know what's happening. Produces a unit because when the game is over, there's nothing more we care about. We'll start out by saying

play(MyGame.initialState);
The referee's "play" procedure

let rec play: state => unit = s =>
{
  .
  .
  .

  play(s2)
}
The referee's "play" procedure

let rec play: state => unit = s =>
{
    switch gameStatus(s) {
    | Draw => print_endline("It's a draw"); ()  
    | Win(p) => print_endline("Player " ++ 
                     stringOfPlayer(p) ++ " wins!"); ()  
    | Ongoing(P1) => ...
    }

    play(s2)
}
The referee's "play" procedure

let rec play: state => unit = s =>
{...
| Ongoing(P1) => { 
  let m = Player1.nextMove(s);
  let s2 = MyGame.nextState(s,m);
  play(s2);
  }
| Ongoing(P2) => { 
  let m = Player2.nextMove(s);
  let s2 = MyGame.nextState(s,m);
  play(s2);
  }
}
The referee's "play" procedure

• Just four steps:
  • Check current status and finish up if appropriate
  • If ongoing, ask the relevant player for a move
  • Compute the next game-state, following that move.
  • *play* the game, starting from that new game-state.
Quiz policy

• Despite what the syllabus says, all in-class quizzes count.
Quiz setup

- Last time we had a list of ints, and list of int-lists
- We wanted to "map" the "cons" operation over these
- But there were more int-lists than ints, so at some point, we had to stop and just copy the remaining items from the int-lists.
- Let's generalize to operations other than "cons" and call this "zap2" rather than "map2".
/* zap2
 * input:
 *   items, a list containing k items
 *   things, a list containing n things, with n >= k.
 *   op: a function that consumes an "item" and an "thing" and
 *       produces some result
 * output:
 *   a list of length n; the first k elements are the results of applying
 *   "op" to corresponding pairs from "items" and "things"; the remaining
 *   n-k elements are the last n-k elements of "things"
 */

checkExpect(zap2([1], [2, 3], (x,y) => x + y), [3, 3]);
checkExpect(zap2([1, 2], [[3, 4], [5, 6]], (u,v) => [u, ... v]),
            [[1,3,4], [2, 5, 6]]);

Quiz: What is the type-signature of "zap2"?
let rec zap2:(list('a), list('b), ('a,'b)=> 'b) => list('b) =
(items, things, op) =>
switch(items, things) {
| ([], _) => things
| ([hd, ... tl], [hd2, ... tl2]) => [op(hd, hd2), ... zap2(tl, tl2, op)]
| (_, []) => failwith("First list must be shorter!")
}
Strategy
Minimax

• We’ve seen how to propagate values from the bottom of the game tree upward to determine a "value" at the top of the tree.

• That value (and its associated move) tell the player at the top of the tree what to do (and how good doing it might be).
Propagating upwards

• Using this "minimax" approach, we can assign values to every single state of the game tree! (This is the "minimax algorithm")

• If the root node has a positive value, we call it a "first-player-win" game; if it has a negative value, it's a second-player-win game. If the value for the root node is zero, it's a no-player-win game.

• Small Theorem: YC, for a non-square starting brick of chocolate, is a first-player-win game.
What does having a value (or computed value) at each node tell you?

• How good is this game for P1 at the start

• Suppose the value (to P1) at the start is +8.
  • P1 should be happy to play the game

• What move should P1 make?
  • Whichever one leads to the "child" state with value +8.
  • *Have to look at all the children again to tell which one that is*
  • Why not record it?
Where are we?

• For small games, we can propagate values from terminal states to starting state to tell us whether the game is first-player-win or not

• *How does this help us actually decide what to do?*

• Idea: instead of just saying, when you have moves leading to states with values 1, 5, 4, that you have (as player 1) a value of "5", you could say
  • *There's a value of 5 to be had*
  • *Move number 2 is the one that gets you that value!*
improved argmax

(*
  inputs: a procedure f that consumes items of type 'a
  a nonempty list alod of items of type 'a
  output: a pair (v, q), where q is the item in the list for which f(q)
        is greatest, and v = f(q).
*)
let argmax: ('a => int, list('a)) => (int, 'a) = ...
Improved minimax algorithm

• Input: a game tree (represented by its top node, s) with values at each final state
• Output: a (value, move option) pair, where the "value" is the value of the game to
  P1 if everyone moves optimally at each state, and the move option is the optimal
  move (if any) for whichever player is supposed to move at state s.
• Algorithm (recursive): if initial state, s, is final, the value is already assigned;
  return (value s, None)
• Otherwise
  • for each move from position s, compute the next-state corresponding to this move, and
    apply minimax to the game tree starting from that state, producing a value (and perhaps a
    move) for each of those child-states
  • If whose_turn(s) is P1: among all moves, find the move m with the largest next-state value/
    cvalue, v; return (v, Some m)  \(<\text{argmax!}>\)
  • If whose_turn(s) is P2: among all moves, find the move m with the smallest next-state
    value/cvalue, v; return (v, Some m)  \(<\text{argmin!}>\)
Real Games and checking for "first-player win"
6 x 8 connect-four

Each node has 8 children (one for dropping a marble in each column)

Guesstimate of number of (final) states: each of 48 cells is either red or blue, so $2^{48}$ possibilities - about $3 \times 10^{14} = 300 \text{ trillion}$. 

We'll run out of memory and time before we can represent the whole game tree.

No practical hope of determining first-player win or not.
Practical 6 x 8 connect 4

• Your friend, who has played a lot, whispers in your ear "put your marble in column 3 ... that'll work out best."
• This kibbitzer has looked at all 8 possible moves, estimated how good each one might be (perhaps by thinking "after we make this move, what's the best move for the opponent, and good/bad will that be?") , and chosen the best of those.
• If only you knew how to estimate the value, you could do this too.
• In fact, you could look ahead a few moves, estimate the values, and propagate those estimates up (via minimax) to get a best-move choice
• Why not just look ahead one move?
  • Because in many cases, minimax hides the problem of mistaken estimates!
How to make a move: setup

- You're at some state, s, in the game
- For each *terminal* state, t, there's a value (to P1), value(t)

- Assume that for any *nonterminal* state n, you can estimate the value of that state to P1, via "evalLeaf(n)"
- What value should evalLeaf(n) provide if you're at a terminal state?

- We'll look at a "subtree", starting at s and going forward, say, 4 moves.
- We'll assign a value to each *leaf* of this subtree, "evalLeaf(n)"
- We'll propagate these values up to get a really good value-guess for s.
Observation

• While we're talking about a game tree, there's no actual tree structure in our program!
• From a state, we construct a list of moves (which are like edges of a tree)
• From each (state, move) pair, we get a new state (which is like a child node!)
• But there's no "tree" datatype or "Node" or "Leaf" constructors