31: Games, 2

Game review
Project Structure
Game strategy and Estimated Value
Warmup: a tree game

• It's red's turn.
• Red can go either left or right
• The reward (how much blue pays red) is in the circle below.
• What should red choose?
Warmup: a tree game
Warmup: a tree game

- What should blue choose?
- Payoff numbers still tell you how much blue pays red.
Insight

• The "tree game" is ... every game.
• **If** you can draw out the whole tree the strategy is always the same...
  • It's the "minimax" thing you worked out on the previous slides
• If you can't draw out the whole tree, what do you do?
• Instead of knowing the "value" of some state... you guess it!
  • To be kinder: you "estimate" it.
  • Good chess players are great at this.
  • Bad ones just sum up the point values of pieces captured: Q = 10, R = 4, K,B = 3, ... and compare their value to the opponents' value.
• Then you propagate upwards using minimax!
Review

• We're working with two-person, deterministic, finite, zero-sum games of perfect information
• Archetypes: Yucky chocolate, tic-tac-toe, connect-4.
Representation

• There's a nice visual representation of a game like this: a tree (typically not binary!)
• Each "node" is a game-state
• Edges labelled by legal moves
• Nodes with only leaf-children are "terminal", and labelled by who wins (or "tie"); other nodes are "ongoing"
• Terminal nodes are labelled with their "value" to player 1 (the "value" to player 2 is the negative of this): if player1 wins some game by 10 points, then the value is +10. For a win/lose game like YC, values of +1/-1 suffice.
• Which "row" of the tree determines whose move it is
Code for Game Representation: YC
type whichPlayer =
  | P1
  | P2;

type state = (int, int, whichPlayer);
let initialState = (2, 2, P1);


type move =
  | Row(int)
  | Col(int);

let legalMoves: state => list(move) = (n, k, w) => ...

let nextState: (state, move) => state = ...


type status =
  | Win(whichPlayer)
  | Draw
  | Ongoing(whichPlayer);

let gameStatus: state => status = (n, k, w) => ...;
Additional bits of code

stringOfPlayer: whichPlayer => string
stringOfState: state => string
stringOfMove: move => string
moveOfStringString: string => move
Additional pieces

stringOfPlayer: whichPlayer => string
• For tic-tac-toe, might produce "X" for P1 and "O" for P2.
• For other games, perhaps "Player 1" and "Player 2"
Additional pieces

stringOfState: state => string

• For 2 x 2 yucky chocolate's starting state, that might be
  "[ ][ ][X][ ][ ]\n", which prints as

  [ ][ ]
  [X][ ]

• Can be surprisingly messy (but straightforward) to write
Additional pieces

stringOfMove: move => string

• For Yucky Chocolate, string_of_move (Row 3) might be "3 rows", as in "Player 1 makes the move: 3 rows"

Recall:

type move = Row(int) | Col(int);

let stringOfMove : move => string = fun ...
moveOfString \( (s:\text{string}) : \text{move} \)

• Used to transform human input into the internal representation of a move.

• For connect 4, moveOfString("4") might produce \text{Col 4}, representing a move in which the player puts a marble in column 4.

• For Yucky Chocolate, moveOfString("R 3") might be \text{Row 3}

• What happens if the string is nonsense?
  • Procedure should fail.
The Game module

- All of these types and procedures will be gathered together in one module, with a name like YCGame.
- We have a module type, Game, that mentions everything in the past few slides, so that to create a usable game for this assignment, your YCGame must match the Game module type.
- In lab, you'll actually go through this for the game "Nim" --- good practice for the more substantial game you'll be writing later (probably Connect-four)
Strategy
What happens near the end of yucky chocolate?

• When board is 2 x 2:
  • Player says "If I take one row, then he'll take one column and I'll lose"
  • Player says "If I take one columns, he'll take one row and I'll lose"
  • Player says "Even though this state doesn't have an official "value" because it's not a terminal state, I can see it's a really bad state for me to be in!"

• Player has "propagated" values from the bottom of the game tree upward!
One-level propagation

• You're player 1; it's your turn; there are three possible moves. They lead to terminal states with values 3, 5, -4.
  • Recall "value" means "value to player 1"

• Which move do you pick?
  • The one that leads to value 5!

• What's the resulting value to you?
  • 5

• What's the value to you of being in your current state?
  • 5, because I can always ensure that I win at least that much.

• What's the value, to player 2, of the game being in that state?
  • -5
You're player 2; it's your turn; there are three possible moves. They lead to terminal states with values 3, 5, -4.

- Recall "value" means "value to player 1"

- Which move do you pick?
  - The one that leads to value -4

- What's the resulting value to you?
  - 4

- What's the value to you of being in your *current* state?
  - 4, because I can always ensure that I win at least that much.

- What's the value, to player 1, of the game being in that state?
  - -4
With this approach, we can associate a computed value to any node whose children are all terminal

• Now every terminal or near-terminal node has either a value or an nvalue (for "new value") --- the value, to player 1, or being in that state.

• Suppose you're player 1, and one or two steps away from the end of the game; you have 4 moves.

• The values/nvalues of the next-states for these moves are value are -3, 5, 4, 2

• Which do you take?
  • The one that leads to value/nvalue 5.

• How happy are you to be in this state (i.e., what is this state's value to you)?
  • +5
Computing a state value

• To compute the nvalue of a state where it's player 1's turn:
  • If state is terminal: use the value!
  • Else: Consider the values/nvalues of all possible next states
  • Take the \textit{max} of these!

• To compute the nvalue of a state where it's player 2's turn:
  • If terminal, use the state's value.
  • Consider the values/nvalues of all possible next states
  • These are "how good it is for player 1", so player 2 wants to make this number as small as possible...and will choose that move
  • Player 1 knows that if player 2 gets to this position, player 2 will chose the option that makes things worst for player 1.
  • So the value (to player 1) is the \textit{min} of all possible next-state values/nvalues.
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)  \(<\text{argmin!}>\)