Mini-Max Game Player

15-150
Principles of Functional Programming
April 14, 2020
Michael Erdmann
Recall: Game as tree of alternating player moves
Recall: Optimal Play from Mini-Max

Maxie

Minnie

Maxie
Recall: Optimal Play from Mini-Max

Need to explore these subtrees to determine root’s value and thereby optimal initial move for Maxie.
Recall: GAME

signature GAME =
sig
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play

  type state (* abstract *)
  type move (* abstract *)

  val start : state

  val moves : state -> move Seq.seq
  val make_move : state * move -> state

  val status : state -> status
  val player : state -> player

  datatype est = Definitely of outcome | Guess of int
  val estimate : state -> est

  . . .

end
As a reminder, from last time:

structure Nim : GAME =
struct
  datatype player = Minnie | Maxie
  datatype outcome = Winner of player | Draw
  datatype status = Over of outcome | In_play

  datatype state = State of int * player
  datatype move = Move of int

  val start = State (15, Maxie)

  fun moves (State (n, _)) =
    Seq.tabulate (fn k => Move (k+1)) (Int.min (n,3))

  fun flip Maxie = Minnie
    | flip Minnie = Maxie

  fun make_move (State (n, p), Move k) = State (n-k, flip p)

  datatype est = Definitely of outcome | Guess of int

  fun estimate (State (n, p)) =
    if n mod 4 = 1 then Definitely (Winner (flip p))
      else Definitely (Winner p)

  ...
end
signature SETTINGS =
  sig
    structure Game : GAME  (* parameter *)
    val depth : int
  end

signature PLAYER =
  sig
    structure Game : GAME  (* parameter *)
    val next_move : Game.state -> Game.move
  end
functor MiniMax (Settings : SETTINGS) : PLAYER =
struct
  structure Game = Settings.Game
  structure G = Game

  type edge = G.move * G.est
  fun emv (m,v) = m
  fun evl (m,v) = v

An edge represents a move from the current state, along with a value attributed to the resulting state:

\[
\text{make\_move} \ (s,m) \cong t
\]

(\(v\) is \(t\)'s MiniMax value computed recursively)
Functorize MiniMax Player

functor MiniMax (Settings : SETTINGS) : PLAYER =
struct
  structure Game = Settings.Game
  structure G = Game

  type edge = G.move * G.est
  fun emv (m,v) = m
  fun evl (m,v) = v

  (* leq : G.est * G.est -> bool *)
  fun leq (x, y) = 

  implements this ordering (including int ordering):

    Definitely(Winner Maxie)
    Guess(positive int)
    Definitely(Draw)   Guess(0)
    Guess(negative int)
    Definitely(Winner Minnie)
functor MiniMax (Settings : SETTINGS) : PLAYER =
struct
  structure Game = Settings.Game
  structure G = Game

  type edge = G.move * G.est
  fun emv (m,v) = m
  fun evl (m,v) = v

  (* leq : G.est * G.est -> bool *)
  fun leq (x, y) = . . .

  (* max, min : edge * edge -> edge *)
  fun max (e1, e2) = if leq (evl e2, evl e1) then e1 else e2
  fun min (e1, e2) = if leq (evl e1, evl e2) then e1 else e2

  (* choose : G.player -> edge Seq.seq -> edge *)
  fun choose G.Maxie = Seq.reduce1 max
  | choose G.Minnie = Seq.reduce1 min
Mini-Max at a Maxie Node

\[ v = \max\{v_1, \ldots, v_k\} \]
mutual recursion

**search** hands **evaluate** a best edge \((m_i, v_i)\). **evaluate** returns best \(v\) to its calling **search**.

\[ v = \max\{v_1, \ldots, v_k\} \]

\((m, v) = (m_i, v_i)\) with \(i\) index maximizing \(v_i\)

*Maxie*
Functorize MiniMax Player  (cont)

(* search : int -> G.state -> edge                  *)
(* REQUIRES: depth d > 0 and G.status(s) == In_play. *)
fun search d s =  
  choose (G.player s)  
  (Seq.map  
    (fn m => (m, evaluate (d-1) (G.make_move(s,m))))  
    (G.moves s))

(* evaluate : int -> G.state -> G.est *)
(* REQUIRES : d ≥ 0.                      *)
and evaluate d s =  
  case (G.status s, d) of  
    (G.Over(v), _) => G.Definitely(v)  
  | (G.In_play, 0) => G.estimate(s)  
  | (G.In_play, _) => evl (search d s)

val next_move = emv o (search Settings.depth)

Check whether the game is over! (Don’t rely on estimator to detect this.)

This is the function specified in the PLAYER signature, accessible to the outside world.
Functorize MiniMax Player (cont)

(* search : int -> G.state -> edge *)
(* REQUIRES: depth d > 0 and G.status(s) == In_play. *)
fun search d s =
    choose (G.player s)
    (Seq.map
        (fn m => (m, evaluate (d-1) (G.make_move(s,m))))
        (G.moves s))

(* evaluate : int -> G.state -> G.est *)
(* REQUIRES : d >= 0. *)
and evaluate d s =
    case (G.status s, d) of
        (G.Over(v), _) => G.Definitely(v)
        | (G.In_play, 0) => G.estimate(s)
        | (G.In_play, _) => evl (search d s)

val next_move = emv o (search Settings.depth)
end (* functor MiniMax *)
signature TWO_PLAYERS =
sig
  structure Maxie : PLAYER  (* parameter *)
  structure Minnie : PLAYER  (* parameter *)
  sharing type Maxie.Game.state = Minnie.Game.state
  sharing type Maxie.Game.move = Minnie.Game.move
end

signature GO =
sig
  val go : unit -> unit
end
TWO_PLAYERS & GO

signature TWO_PLAYERS =
sig
  structure Maxie : PLAYER (* parameter *)
  structure Minnie : PLAYER (* parameter *)
  sharing Maxie.Game = Minnie.Game
end

(alternate sharing form)

signature GO =
sig
  val go : unit -> unit
end
functor Referee (P : TWO_PLAYERS) : GO =
struct
  structure G = P.Maxie.Game
  structure H = P.Minnie.Game

  (* run : G.state -> string *)
  fun run s =
    case (G.status s, G.player s) of
      (G.Over(v), _) => G.outcome_to_string(v)
    | (G.In_play, G.Maxie) =>
        run (G.make_move (s, P.Maxie.next_move s))
    | (G.In_play, G.Minnie) =>
        run (H.make_move (s, P.Minnie.next_move s))

  fun go () = print (run (G.start) ^ "\n")
end
Human vs depth-3 MiniMax for Nim

structure NimHuman = HumanPlayer(Nim) (* Nim : GAME *)

structure NimSet3 : SETTINGS =
struct
    structure Game = Nim
    val depth = 3
end

structure Nim3MM = MiniMax(NimSet3)

structure HvM : TWO_PLAYERS =
struct
    structure Maxie = NimHuman
    structure Minnie = Nim3MM
end

structure Nim_RefHvM = Referee(HvM)

Nim_RefHvM.go()
(be sure to look at the AlphaBeta slides as well)