Games II: The Minimax Algorithm

15-150 M21

Lecture 0726
26 July 2021

- Implemented playable games in SML
- Our game implementation consisted of:
  - A `GAME` (specifying rules, how to make moves, etc.)
  - `PLAYER`s (plays a particular `GAME`, provides function `next_move` assigning a "choice" of move to each state)
  - Games are refereed by a `CONTROLLER`, who facilitates play between two `PLAYER`s playing the same game.
- Implemented `Nim`, where states were of the form `(s,Minnie)` or `(s,Maxie)` for `s:int` nonnegative. A move is a positive `int` `i` which is less than or equal to `Int.min(3,s)`.

We'll deal with 4 different kinds of players:

- Human players (our game library includes utilities to accept user input to determine `next_move`)
- Directly-implemented players (`NimPlayer` from tomorrow's lab)
- MiniMax players (this lecture)
- Alphabeta players (Wednesday's lecture, `games` homework)

0 How to Build Smart `PLAYER`s

What do we mean by smart?

dis·cern·ing
/dəˈsərniNG/

adjective

having or showing good judgment.
"the restaurant attracts discerning customers"

Similar: discriminating selective judicious tasteful refined cultivated

We want to design our `PLAYER`s such that their `next_move` function makes decisions which generally lead to it winning the game more often.

Demonstration:

`RunNim.play RunNim.HvM`

vs.

`RunNim.play RunNim.HvP`

What do we mean by smart?

dis·cern·ing

/dəˈsərniNG/

adjective

having or showing good judgment.

"the restaurant attracts discerning customers"

Similar: discriminating selective judicious tasteful refined cultivated

We want to design our `PLAYER`s such that their `next_move` function makes decisions which generally lead to it winning the game more often.

So what we want to do is build a player who "knows what's good for her": who is able to assess the moves available to her, decide which one has the most favorable outcome, and make the corresponding move her `next_move`.

Formally, we make sense of games mathematically by examining the corresponding *game tree*. A game tree is a finitely-branching tree where

- The nodes represent *game states*
- The edges represent *moves*
- The root node is the current state of the game, and the rest of the tree represents different outcomes achieveable by a certain series of moves from the two players
- The children of a given node are the states reachable from that game state by the current player making a valid move.

We'll call our players '**Maxie**' and '**Minnie**'.

I want to Maximize!

I want to Minimize!

Demonstration:

Nim Game Tree

Observation:

A good player is thinking a few moves into the future

Problem:

It's not always tractable to search through the *entire* game tree

- Problem: it's impractical (and often impossible) to visit every node of the tree
- Solution: explore some of the tree, and guess
  - Have a fixed 'search depth' d
  - Explore the top d levels of the tree (i.e. the game states than can be reached from the current one in d moves or fewer)
  - When you hit your search depth, use your knowledge of the game to assign an appropriate value to that state, and treat that value as the value of the node.
- More precisely: we'll have a function `estimate` which takes a game state (for instance, a value of type `Nim.State.t`) and returns a "guess" of the goodness or badness of that state.

Estimators : Some design principles

An *estimator* for a game `G` is a function assigning "guesses" to each state to (perhaps roughly) indicate who's winning.

- The "guesses" will usually be numerical (e.g. `int`s): lower numbers better for **Minnie**, larger numbers better for **Maxie**. The scale is arbitrary: all that matters is the relative ordering of states.
- The goal here is to induce an ordering on states, i.e. articulate a sense in which states are "better" or "worse" than each other (from one player's perspective).
- We want "better" to mean "more likely to win" (as best as possible)
- A given `GAME` will have many possible estimators, with varying degrees of sophistication, and which may weight different factors differently.

0726.0 (lib/game/estimate/ESTIMATOR.sig)

```
signature ESTIMATOR =
sig
  structure Game : GAME

  type guess
  datatype est = Definitely of Game.Outcome.t
               | Guess of guess
  val compare : est * est -> order
  val toString : guess -> string

  val estimate : Game.State.t -> guess
```

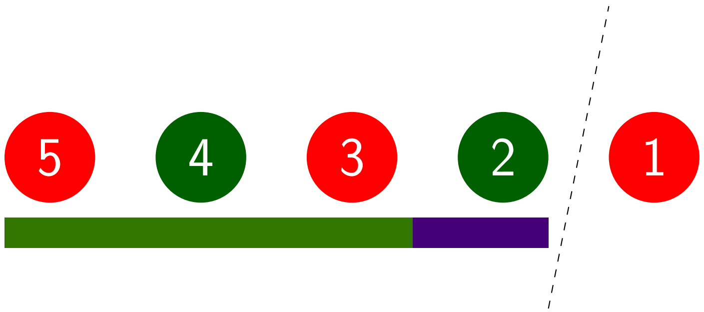
- Note that the only operation on values of type `est` is comparison (the function `compare`). We don't – in general – require `guesses` to be numbers at all, we just require that they be ordered.
- We **transparently** ascribe to this signature. While we don't require in general that `guess` is implemented as `int` or `real`, if we do happen to implement it that way we want to have access to the associated methods (e.g. from the basis structures `Int` and `Real`).

Player `p` can guarantee a win from `(s, flip p)`

iff

`s mod 4` $\cong$ `1`

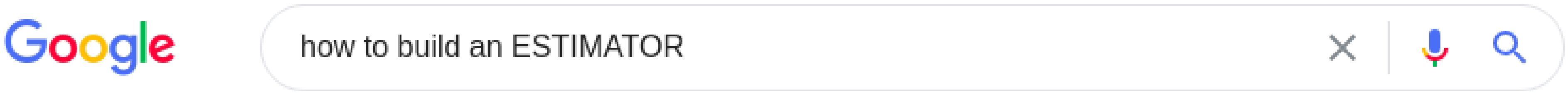
(remember
```
fun flip Maxie = Minnie | flip Minnie = Maxie)
```



Nim has a perfect estimator

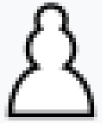
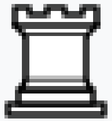
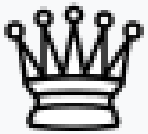
So, assuming the other player plays optimally, whoever's turn it is when `s` is of the form `(4*k)+1` for some `k:int` will *lose*.

```
(* recall a value of Nim.State.t is (s,p)
   for some nonnegative int s and either
   p=Minnie or p=Maxie *)
(* estimate : Nim.State.t -> int *)
fun estimate (s,p) =
      case (s mod 4, p) of
          (1,Minnie) => 1
        | (1,Maxie) => ~1
        | (_,Minnie) => ~1
        | (_,Maxie) => 1
```



This is somewhat *too* clean of an example: most games don't have perfect estimators. Rather, the best we can do is make pretty good guesses! To design an estimator, we'll usually use some combination of simple heuristics and more sophisticated theory.

For instance, here's a common heuristic for chess: for a chess piece $\mathtt{p}$, let $\mathtt{v(p)}$ be given by the following chart

Symbol					
Piece	pawn	knight	bishop	rook	queen
Value	1	3	3	5	9

Then put

$$\mathtt{estimate(S)} = \left(\sum_{\substack{\text{Pieces } \mathtt{p} \ \textbf{Maxie} \\ \text{has in play (in } \mathtt{S}\text{)}}} \mathtt{v(p)} \right) - \left(\sum_{\substack{\text{Pieces } \mathtt{p} \ \textbf{Minnie} \\ \text{has in play (in } \mathtt{S}\text{)}}} \mathtt{v(p)} \right)$$

1 The MiniMax Algorithm

- We should assign each node an estimator guess, its “value”.
- The value of a node should reflect who’s winning from that node, which depends on the moves available from that state.
- From there, we can fill in the rest of the game tree by assuming the players play optimally

Demonstration: Minimax

Fix a search depth d.

1. Traverse the game tree down to the d-th level.(For every node encountered where the game is over, assign such nodes the value `Definitely` of whoever the winner is.)
2. Call the estimator to assign values to the d-th level.
3. Work upwards, assigning values to nodes according to the Minnie and Maxie principles described above
   - For **Minnie** nodes: the value should be the *minimum* of the values of the child nodes
   - For **Maxie** nodes: the value should be *maximum* of the values of the child nodes.

Once we've filled all the way to the top of the tree (our current state), then we can decide which move to make based on the estimated values.

5 Minute Break

2 SML Implementation

0726.3 (lib/game/estimate/MiniMax.fun)

```
functor MiniMax (Settings : SETTINGS) :> PLAYER
```

0726.1 (lib/game/estimate/SETTINGS.sig)

```
signature SETTINGS =
sig

  structure Est : ESTIMATOR

  val search_depth : int

end
```

Recall: the PLAYER signature

0726.2 (lib/game/core/PLAYER.sig)

```
signature PLAYER =
sig

  structure Game : GAME

  val next_move : Game.State.t -> Game.Move.t

end
```

0726.3 (lib/game/estimate/MiniMax.fun)

```
functor MiniMax (Settings : SETTINGS) :> PLAYER
where Game = Settings.Est.Game =
struct
  structure Est = Settings.Est
  structure Game = Est.Game
```

0726.4 (lib/game/estimate/MiniMax.fun)

```
  type edge = Game.Move.t * Est.est
  fun valueOf ((_,value) : edge) = value
  fun moveOf ((move,_) : edge) = move
  fun max ((m1,v1):edge, (m2,v2):edge):edge =
    case Est.compare (v1, v2) of
      LESS => (m2, v2)
    | _ => (m1, v1)
  fun min ((m1,v1):edge, (m2,v2):edge):edge =
    case Est.compare (v1, v2) of
      GREATER => (m2, v2)
    | _ => (m1, v1)
```

Encoding the difference between Minnie and Maxie

`reduce1 : ('a * 'a -> 'a) -> 'a Seq.seq -> 'a`

REQUIRES: g is total and associative, S is nonempty

ENSURES:

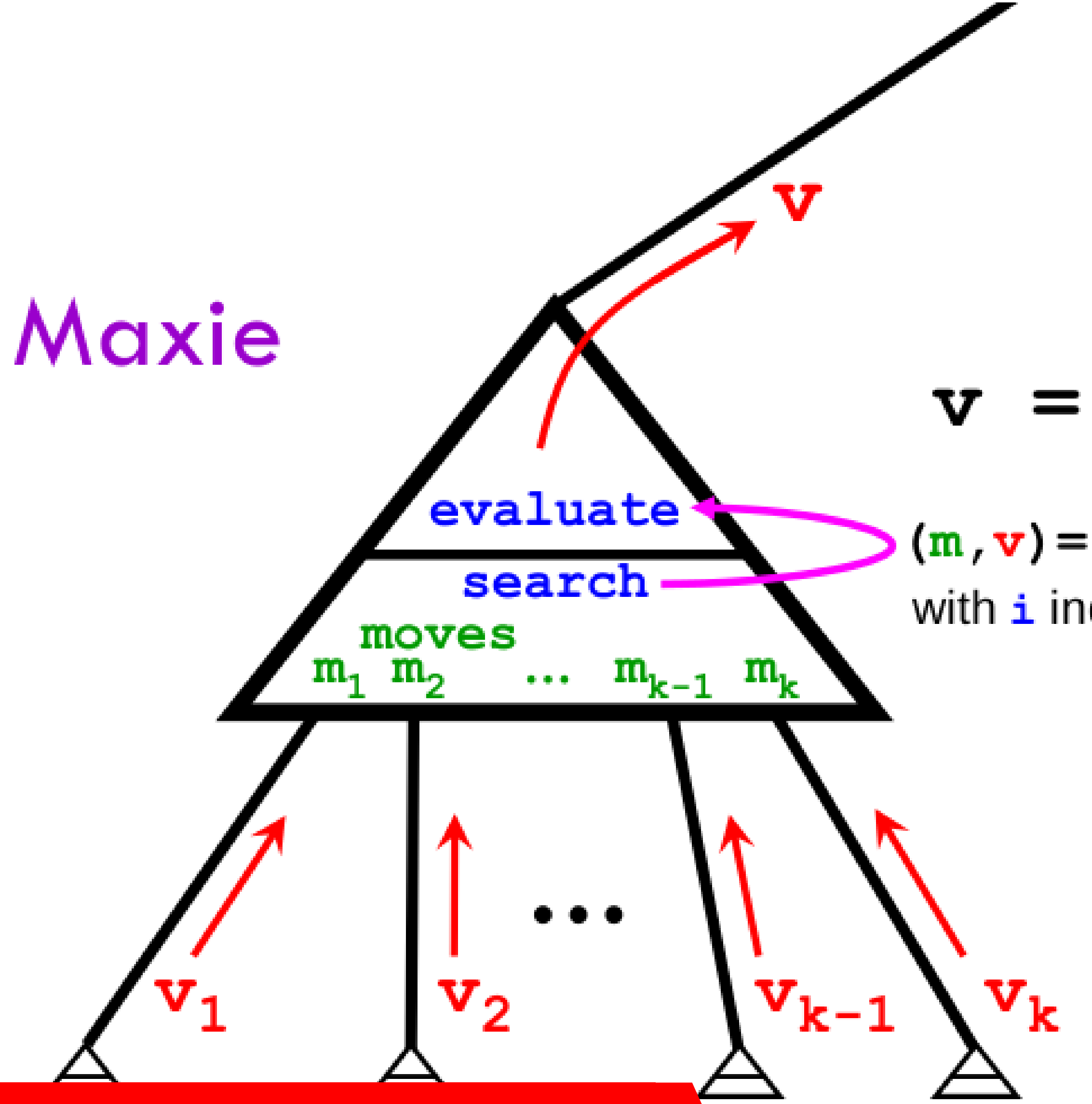
`reduce1 g` $\langle$`x1,...,xn`$\rangle \cong$ `g(x1,g(x2,g(...,, xn)))`

0726.5 (lib/game/estimate/MiniMax.fun)

```
(* choose : Player.t -> edge Seq.seq -> edge *)
fun choose Player.Maxie  = Seq.reduce1 max
  | choose Player.Minnie = Seq.reduce1 min
```

Mutual Recursion

```
fun even 0 = true
  | even n = odd(n-1)
and odd 0 = false
  | odd n = even (n-1)
```



$$v = \max\{v_1, \ldots, v_k\}$$

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

```
(* search : int -> G.State.t -> edge        *)
(* REQUIRES: d > 0                          *)
fun search (d:int) (s:Game.State.t) : edge =
   choose (Game.player s)
      (
      Seq.map
         (fn m =>
            (m,
             evaluate (d-1) (Game.play(s,m)))
            )
         (Game.moves s)
      )
```

0726.7 (lib/game/estimate/MiniMax.fun)

```
 (* evaluate:int -> Game.status -> Est.est *)
 (* REQUIRES: d >= 0                       *)
and evaluate (d : int) (st : Game.status) : Est.est =
    case st of
       Game.Playing s => (
          case d of
             0 => Est.Guess (Est.estimate s)
           | _ => valueOf (search d s)
       )
     | Game.Done oc => Est.Definitely oc
```

0726.8 (lib/game/estimate/MiniMax.fun)

```
val next_move =
  moveOf o search Settings.search_depth
```

Tomorrow's Lab:

Estimators & Minimax

Next Time

Optimizing the complexity of minimax, to avoid unnecessary work

Thank you!