15-150 Fall 2019

Stephen Brookes
schedule

• 12 Nov: alpha-beta
• 14 Nov: lazy
• 19 Nov: guest lecture (Ron Minsky, Jane Street Capital)
• 21 Nov: lazy
• 26 Nov: imperative
• 28 Nov: THANKSGIVING
• 3 Dec: case study
• 5 Dec: review
today

• 2-person games, continued
referee

• A referee takes 2 players *for the same game* and *alternates* them from a start state, until the game is over

• Produces a *list* of the game states
signature EVENT =
  sig
    structure Game : GAME
    val player1 : Game.state -> Game.move
    val player2 : Game.state -> Game.move
  end

signature REFEREE =
  sig
    structure Game : GAME
    val player1 : Game.state -> Game.move
    val player2 : Game.state -> Game.move
    val run : Game.state -> Game.state list
  end
functor FairReferee(Event : EVENT) : REFEREE =

struct
  structure Game = Event.Game
  val moves = Game.moves
  val step = Game.step
  val player1 = Event.player1
  val player2 = Event.player2

  fun loop (p1, p2) s =
    if null(moves s) then [s] else
      let
        val m = p1 s
        val s1 = step (s, m)
      in
        s :: loop(p2, p1) s1
      end

  fun run s = loop (player1, player2) s

end
example

- structure R = FairReferee(MiniMax(Nim));
structure R : REFEREE

- R.run 15;
val it = [15,13,12,9,8,5,4,1,0] : Nim.state list

Start with 15.
Maxie takes 2, leaving 13.
Minnie takes 1, leaving 12.
Maxie takes 3, leaving 9.
Minnie takes 1, leaving 8.
Maxie takes 3, leaving 5.
Minnie takes 1, leaving 4.
Maxie takes 3, leaving 1.
Minnie (no other choice) takes 1, losing.
reflection

• We used very simple signatures

• Could have added extra features
  
  to print states and moves

• Could have allowed for a more general type of outcome or estimation value, e.g.
  
  score, estimate : state -> real

• Could design referee to use exceptions...
  
  ... handle Invalid _ => ...
runtime issues

- Can use *depth-bounding* to limit runtime, but this may lead to sub-optimal results
  - the quality of results will depend on *search depth* and how well you *estimate*
  - even *bounded* minimax may *waste time* searching fruitlessly...
doing better

Redesign F and G
to maintain and propagate extra information
and use it to avoid wasteful computation

• alpha-beta pruning

Alpha–beta pruning is a search algorithm that seeks to decrease the number of game tree nodes that get evaluated by the minimax algorithm. It stops evaluating a move when at least one possibility is found that proves the move to be worse than a previously examined move.
the plan

• alpha-beta minimax is a sequential algorithm, so we will look first at a sequential version of standard minimax

• Then we will modify, to keep track of information that tells us when it is safe to stop evaluating early

(these ideas also work with bounded minimax)
lists or sequences

• Our implementation of alpha-beta minimax will use *sequential* evaluation, to allow short-circuiting.

• We will want to deal with `moves(s)` as a *list*, not as a *sequence*.

• We’ll assume given a function

  \[
  \text{toList} : 'a \text{ seq} \rightarrow 'a \text{ list}
  \]

  provides a view of an abstract type.
We assumed a structure `Seq : SEQ` and used

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

To turn a **sequence** into a **list** we can use

```
toList : 'a Seq.seq -> 'a list
```

To “reduce” and “map” a list we can use

```
foldl, foldr : ('a * 'a -> 'a) -> 'a -> 'a list -> 'a
List.map : ('a ->'b) -> 'a list -> 'b list
```

We’ll assume a special value `infty : int` for use as a default base value for reducing.
fun F(s:state) : int = 
  let
  val S = Seq.map(fn m => step(s, m))(moves s)
  in
  if (Seq.null S) then (score s) else
    Seq.reduce1 Int.max (Seq.map G S)
  end

and G(s:state) : int = 
  let
  val S = Seq.map(fn m => step(s, m))(moves s)
  in
  if (Seq.null S) then ~(score s) else
    Seq.reduce1 Int.min (Seq.map F S)
  end
minimax  
using lists

REQUIRES For all terminal states \( s \), \( -\infty \leq \text{score} \, s \leq \infty \)
ENSURES For all states \( s \), \( F_0 \, s = F \, s \) and \( G_0 \, s = G \, s \)

**fun** \( F_0(s:\text{state}) : \text{int} = \)**

```
let
  val S = List.map(fn m => step(s,m))(toList(moves s))
in
  if (List.null S) then (score s) else foldl Int.max (~infty) (List.map G_0 S)
end
```

**and** \( G_0(s:\text{state}) : \text{int} = \)**

```
let
  val S = List.map(fn m => step(s,m))(toList(moves s))
in
  if (List.null S) then ~score(s) else foldl Int.min (infty) (List.map F_0 S)
end
```
minimax

using lists, with commentary

REQUIRES  For all terminal states s, \( \sim \text{infty} \leq \text{score } s \leq \text{infty} \)
ENSURES    For all states s, \( F_1 s = F s \) and \( G_1 s = G s \). Prints the call pattern.

fun \( F_1(s:state) : \text{int} = \) (print ("F " ^ Int.toString s ^ " \n ");
  let
    val S = List.map(fn m => step(s,m))(toList(moves s))
  in
    if (List.null S) then (score s) else
      foldl Int.max (~infty) (List.map G_1 S)
  end)
and \( G_1(s:state) : \text{int} = \) (print ("G " ^ Int.toString s ^ " \n ");
  let
    val S = List.map(fn m => step(s,m))(toList(moves s))
  in
    if (List.null S) then ~ (score s) else
      foldl Int.min (infty) (List.map F_1 S)
  end)
drawing game trees

leaves showing scores

leaves showing outcomes
- `F 3;`
  `val it = 1 : int`

- `F1 3;`

```
  F 3
  G 0
  G 1
  F 0
  G 2
  F 0
  F 1
  G 0
```

```
val it = 1 : int
```

**F1** prints the call pattern
Nim example

\[-F_1 4;\]

\[
\begin{array}{c}
F 4 \\
G 1 \\
F 0 \\
G 2 \\
F 0 \\
F 1 \\
G 0 \\
G 3 \\
F 0 \\
F 1 \\
G 0 \\
F 2 \\
G 0 \\
G 1 \\
F 0 \\
\end{array}
\]

\[
\begin{array}{c}
4 \\
1 \\
2 \\
3 \\
\end{array}
\]

\[
\begin{array}{c}
0 \\
0 \\
0 \\
0 \\
0 \\
0 \\
\end{array}
\]

\[
\begin{array}{c}
+ \mid \\
+ \mid \\
+ \mid \\
\end{array}
\]

\[
\begin{array}{c}
- \mid \\
- \mid \\
- \mid \\
\end{array}
\]

\[
\begin{array}{c}
0 \\
1 \\
2 \\
\end{array}
\]

\[
\begin{array}{c}
+ \mid \\
\end{array}
\]

val it = 1 : int

\textbf{F}_1 \textit{ is a sequential } F, \textbf{ with print instructions}
Nim example
so what?

Sequential minimax ($F_1, G_1$) explores the entire game tree (in depth-first traversal)

- $F_1$ 4 “visits” the optimal move (to state 1) early, but carries on making recursive calls even though we can tell from the game tree that no better move will be found later
minimax F

Let \( \langle p_1, \ldots, p_n \rangle = \text{map}(\text{fn } m \Rightarrow \text{step}(p, m))(\text{moves } p) \)

Define \( F_1 p = (\text{score } p) \) if \( n=0 \)

Otherwise, compute the increasing sequence

\[
\alpha_1 = G_1 p_1 \\
\alpha_2 = \text{Int.max}(\alpha_1, G_1 p_2) \\
\vdots \\
\alpha_n = \text{Int.max}(\alpha_{n-1}, G_1 p_n)
\]

then define \( F_1 p = \alpha_n \)
minimax G

Let \( \langle p_1, \ldots, p_n \rangle = \text{map}(\text{fn } m \mapsto \text{step}(p, m))(\text{moves } p) \)

Define \( G_1 p = \sim(\text{score } p) \) if \( n=0 \)

Otherwise, compute the decreasing sequence

\[
\beta_1 = F_1 p_1 \\
\beta_2 = \text{Int.min}(\beta_1, F_1 p_2) \\
\ldots \\
\beta_n = \text{Int.min}(\beta_{n-1}, F_1 p_n)
\]

then define \( G_1 p = \beta_n \)
when to stop?

What information can we propagate to help figure out when it’s OK to stop?

• Idea: keep track of the largest Maxie score $\alpha$ and the smallest Minnie score $\beta$, based on what has been seen so far

• Redesign F and G to maintain these two parameters correctly…

• …and stop exploring as soon as possible when current prediction is worse than best opponent score so far
alpha-beta F

Let $\langle p_1, ..., p_n \rangle = \text{map}(\text{fn } m \Rightarrow \text{step}(p, m))(\text{moves } p)$

Define $F_2(p, \alpha, \beta) = (\text{score } p)$ if $n=0$

Otherwise, compute the increasing sequence

$$\alpha_1 = \text{Int.max}(\alpha, G_2(p_1, \alpha, \beta))$$
$$\alpha_2 = \text{Int.max}(\alpha_1, G_2(p_2, \alpha_1, \beta))$$
... 
$$\alpha_i = \text{Int.max}(\alpha_{i-1}, G_2(p_i, \alpha_{i-1}, \beta))$$

and stop when $\alpha_i \geq \beta$ or $i = n$

then define $F_2(p, \alpha, \beta) = \alpha_i$
alpha-beta $G$

Let $\langle p_1, \ldots, p_n \rangle = \text{map}(\text{fn} \ m \Rightarrow \text{step}(p, m))(\text{moves} \ p)$

Define $G_2(p, \alpha, \beta) = \sim(\text{score} \ p)$ if $n=0$

Otherwise, compute the decreasing sequence

$\beta_1 = \text{Int.min}(\beta, G_2(p_1, \alpha, \beta))$
$\beta_2 = \text{Int.min}(\beta_1, G_2(p_2, \alpha, \beta_1))$
$\ldots$
$\beta_i = \text{Int.min}(\beta_{i-1}, G_2(p_i, \alpha, \beta_{i-1}))$

and **stop** when $\beta_i \leq \alpha$ or $i = n$

then define $G_2(p, \alpha, \beta) = \beta_i$
sequential minimax

leaves labelled with outcomes, i.e. +/- score

F_A = max \langle G_B, G_C \rangle

G_B = min \langle F_D, F_E \rangle

F_E = max \langle G_G, G_H, G_I \rangle

G_C = \ldots\ldots\ldots
Can tell that \( F(\text{E}) \geq 6 \) without evaluating \( G(\text{H}) \) and \( G(\text{L}) \)

Since \( 4 < 6 \) can tell that \( G(\text{B}) = 4 \)

\[
F(\text{A}) = \max \langle G(\text{B}), G(\text{C}) \rangle
\]

\[
G(\text{B}) = \min \langle F(\text{D}), F(\text{E}) \rangle
\]
pruning

\[
\begin{align*}
F_A &= \text{reduce}_1 \max \{G_B, G_C\} \\
G_B &= \text{reduce}_1 \min \{F_D, F_E\} \\
F_E &= \text{reduce}_1 \max \{G_G, G_H, G_I\}
\end{align*}
\]

Can tell that \(F_E \geq 6\) even without finding \(G_H\) and \(G_I\)

Since \(4 < 6\) can tell that \(G_B = 4\)
We can tell that $G \circ C \leq 3$ without evaluating $F \circ F$.

Since $3 < 4$ we can tell that $F \circ A = 4$.
pruning

Can tell that
F ⊕ A = 4
without searching the
“pruned” subtrees
Nim example

Maxie can win +1 by moving over here (take 3)...

... and can win at most +1 over here...

... and can win at most +1 over here...

... so F 4 = +1
Nim example

Can show $F_4 = 1$
without visiting the dotted regions
alpha-beta minimax

Design mutually recursive functions

\[ F_2, G_2 : \text{state} \times \text{int} \times \text{int} \rightarrow \text{int} \]

with integer parameters \( \alpha, \beta \) such that

\[ \alpha = \text{highest possible Maxie score} \]
\[ \beta = \text{lowest possible Minnie score} \]

based on what’s been visited so far

\( F_2 \) and \( G_2 \) mimic \( F \) and \( G \)
but recursive calls also
increase \( \alpha \), decrease \( \beta \)
alpha-beta minimax

Design mutually recursive functions

\[ F_2, G_2 : \text{state} \times \text{int} \times \text{int} \rightarrow \text{int} \]

with integer parameters \( \alpha, \beta \) such that

\[ \alpha = \text{highest guaranteed Maxie score} \]
\[ \beta = \text{lowest guaranteed Minnie score} \]

based on what’s been visited so far

Start with

\[ \alpha \leq \text{lowest possible score in game}, \]
\[ \beta \geq \text{highest possible score in game} \]

and maintain \( \alpha \leq \beta \)
## alpha-beta spec

\[
F_2(p, \alpha, \beta) = \begin{cases} 
F(p) & \text{if } \alpha < F(p) < \beta \\
\leq \alpha & \text{if } F(p) \leq \alpha \\
\geq \beta & \text{if } F(p) \geq \beta
\end{cases}
\]

(similarly for \( G_2 \))

\[
F_2(p, \alpha, \beta) \text{ is exact if } F(p) \text{ is between } \alpha, \beta
\]

and is good enough, otherwise
Nim example

sequential minimax

- F1 4;

+|- 4
F 4
G 1
F 0
G 2
F 0
F 1
G 0
G 3
F 0
F 1
G 0
G 2
F 0
G 0
G 1
F 0

val it = 1 : int

alpha-beta minimax

- F2(4, ~2, 2);

F (4, ~2, 2)
G (1, ~2, 2)
F (0, ~2, 2)
G (2, 1, 2)
F (0, 1, 2)

pruning
G (3, 1, 2)
F (0, 1, 2)

val it = 1 : int
Nim example

Maxie can win \(+1\) by moving over here (take 3)...

... and can win at most \(+1\) over here, so PRUNE...

... and can win at most \(+1\) over here, so PRUNE...

... so \(F 4 = +1\)
correctness

REQUIRES $\sim \text{infty} < \text{score} \ s < \text{infty}$ for all terminal states $s$

ENSURES $F_2(s, \sim \text{infty}, \text{infty}) = F(s)$
$G_2(s, \sim \text{infty}, \text{infty}) = G(s)$ for all states $s$

and $F_2, G_2$ are more efficient
panacea?

• alpha-beta minimax may perform better
  • regular minimax $O(k^d)$
  • alpha-beta best case $O(k^{d/2})$

• But many significant factors….
  • *move order* exploration is crucial!
a bad case

worst moves tried first

no pruning
a bad case

worst moves tried first

no pruning
a good case

Max
Min

best moves tried first