15-150 Fall 2019
Games (part 2)

Stephen Brookes *

*Building on notes by Dan Licata
1 Minimax

In the last lecture we introduced the Minimax algorithm for evaluating game states and used it to define players Maxie and Minnie who make moves guaranteed to lead to the best outcome for each one, assuming that the opponent also tries their best. We showed that our functions do indeed work as specified.

We used the game Nim (with a single pile of sticks) as illustration. This choice was made because Nim is a very simple game to understand and play, and there is a well known strategy for winning at Nim, making it easy to check that our evaluation functions are producing correct results.

You should easily be able to adapt the minimax code to deal with some more sophisticated games. Obvious choices would be a more general version of Nim where you can have multiple piles of sticks. Again there is a winning strategy, but this required a bit more math (to do with so-called “nimbers”).

Now we look at some issues concerning efficiency. In class we looked at the work and span for the minimax evaluation functions, under some simple (and common) assumptions about the game.

These notes don’t repeat everything from class, so please consult the slides! Here we discuss some of the main ideas, but not all the details.

In homework you will explore some of these ideas in more depth.
2 Bounded minimax

If we ask the ML runtime system (using the Nim game implementation) to evaluate \( F_{150} \) the computation takes a (ridiculously) long time! Reason: the game tree with root 150 has depth 150, so this function makes an enormous number of recursive calls.

Next, we modify the MiniMax functor design to obtain a player who uses minimax to a chosen depth, then estimation, using a specific function \( \text{estimate} : \text{state} \rightarrow \text{int} \). Again this functor is applicable to an arbitrary \text{GAME} structure, but the game needs to be coupled with an integer, a search depth parameter. For this purpose it will be convenient to introduce an extra signature \text{BOUND}, given by:

\[
\text{signature BOUND =}
\]
\[
\text{sig}
\]
\[
\text{val depth : int}
\]
\[
\text{end}
\]

We also introduce a signature for games-with-estimation:

\[
\text{signature EST_GAME =}
\]
\[
\text{sig}
\]
\[
\text{structure Game : GAME}
\]
\[
\text{val estimate : Game.state \rightarrow int}
\]
\[
\text{end}
\]

Here is an example of a structure with this signature, a game-with-estimation: Nim, with the “genius” estimation function:

\[
\text{structure SmartNim : EST_GAME =}
\]
\[
\text{struct}
\]
\[
\text{structure Game = Nim}
\]
\[
\text{fun estimate p = if (p mod 4 = 1) then ~1 else 1}
\]
\[
\text{end;}
\]

The implementation of bounded minimax is very similar in spirit to the previously presented unbounded minimax. We’ve deliberately written the code and explanation to make that comparison simple.
functor BoundedMaxiMe(structure E : EST_GAME and B : BOUND) : PLAYER = 
  struct
    structure Game = E.Game
    open Game

    fun F d p = 
      let
        val M = moves p
      in
        if (null M) then (score p) else
        if d=0 then (E.estimate p) else
          reduce1 Int.max (map (fn m => G(d-1) (step(p,m))) M)
      end
    and
    G d p = 
      let
        val M = moves p
      in
        if (null M) then ~(score p) else
        if d=0 then ~(E.estimate p) else
          reduce1 Int.min (map (fn m => F(d-1) (step(p,m))) M)
      end
  end

  type edge = move * int

  fun maxedge ((m1,v1), (m2,v2)) = if v1 < v2 then (m2,v2) else (m1,v1)
  fun maxbestedge L = reduce1 maxmove L

  fun player p = 
    let
      val M = moves p
      val (m,_) = maxbestedge(map (fn m => (m, G B.depth (step(p,m)))) M)
    in
      m
    end
end
Note the changes from the earlier unbounded version: the state evaluation functions have an extra argument representing search depth, and this decreases in recursive calls. When the depth is zero, estimation is used.

Try using this functor to build various players for games of your choosing, with different choices of depth and different estimation functions.

Here is a (somewhat silly) example.

```haskell
structure SillyNim : EST_GAME =
strict
  structure Game = Nim
  fun estimate p = ~1
end;

structure BadNim : PLAYER =
  BoundedMaxiMe(structure E = SillyNim and B = struct val depth = 2 end);
```

This builds a player for the Nim game who only uses minimax to depth 2 then guesses using a very pessimistic estimator.

The value of BadNim.player 8 is 2, meaning that this player would take 2 sticks from 8, leaving 6. The opponent could then take 1, leaving 5, and we know that from 5 sticks the player who goes next cannot force a win (unless his opponent is careless).

In contrast, if we do

```haskell
structure GoodNim : PLAYER =
  BoundedMaxiMe(structure E = SmartNim and B = struct val depth = 2 end);
```

we’ll get GoodNim.player 8 = 3. The good player takes 3 from 8, leaving 5, and the opponent cannot force a win.

(In fact because the estimator in SmartNim is perfect, we’d even get this behavior of we chose depth 0.)
3 Practical matters

As we have mentioned, many games have huge game trees. Even though our state evaluation and move selection functions don’t actually build trees, the recursive call patterns do correspond to tree traversals and huge game trees correlate with huge numbers of recursive calls.

It’s easy to see that even in some simple example games, our minimax functions waste time exploring irrelevant parts of the game tree.

Suppose we have a terminating game with states \( a, b, c, d, e, f, g, h, i, \ldots \), the same set of moves, and

\[
\begin{align*}
\text{moves } a &= \langle b, e \rangle \\
\text{moves } b &= \langle c, d \rangle \\
\text{moves } e &= \langle f, g \rangle \\
\text{moves } g &= \langle h, i \rangle \\
\text{moves } c &= \text{moves } d = \text{moves } f = \langle \rangle 
\end{align*}
\]

and we don’t bother to specify what moves \( h \) or moves \( i \) are(!). Suppose that \( \text{step}(x,y) = y \) for all states \( x \) and moves \( y \).

Then, with the \( F \) and \( G \) functions defined as usual, consider how we could reason about the value of \( F_a \). (Note that this is not the same as analyzing how ML would actually evaluate this expression! We’re reasoning mathematically about the value of this expression. Of course, we need to appeal to the function definitions along the way.)

\[
F_a = \text{reduce1 Int.max } <G_b, G_e>
\]

\[
G_b = \text{reduce1 Int.min } <F_c, F_d>
\]

\[
= \text{reduce1 Int.min } <3, 5>
\]

\[
= 3
\]

So we know

(1) \( F_a = \text{reduce1 Int.max } <3, G_e> \).

Similarly,
\[ \text{G e} = \text{reduce1 Int.min} < \text{F f}, \text{F g}> = \text{reduce1 Int.min} < 2, \text{F g}> \]

Even without analyzing the value of \( \text{F g} \) (which we know exists, because this game is terminating) we can tell from this equation that the value of \( \text{G e} \) is at most 2. That's because \( \text{Int.min}(2, v) \leq 2 \) no matter what \( v \) is. So, looking again at (1), it follows that the value of \( \text{F a} \) is 3. That's because \( \text{Int.max}(3, v') = 3 \) when \( v' \leq 2 \).

Thus, without examining \( \text{F g} \) we can figure out that \( \text{F a} = 3 \). And if we don’t need to evaluate \( \text{F g} \) we also don’t need to evaluate \( \text{G h, G i} \), or looks anywhere deeper.

If the game tree from node \( g \) is very large, avoiding any evaluation of that part of the game tree may save lots of work and time!

This discussion motivates the development of an even more sophisticated form of minimax, known as minimax with alpha-beta pruning.

The main idea is to define a pair of mutually recursive functions \( F_2, G_2 \) of type \text{int * int -> state -> int} whose extra arguments represent lower and upper bounds on the best found scores for MaxiMe and MiniMe so far discovered. These functions examine the successor states of a current state \text{iteratively} (sequentially) rather than by means of a parallel \text{reduce}.

When \( F \) and \( G \) are the usual minimax functions of type \text{state -> int}, the new functions satisfy the following specification:

\[
\text{REQUIRES: } \alpha < \beta \\
\text{ENSURES: } F_2 (\alpha, \beta) p = F(p) \text{ if } \alpha < F(p) < \beta \\
\phantom{\text{ENSURES: }} \leq \alpha \text{ if } F(p) \leq \alpha \\
\phantom{\text{ENSURES: }} \geq \beta \text{ if } F(p) \geq \beta \\
\]

and \( G_2 (\alpha, \beta) p = \neg (F_2 (\neg \beta, \neg \alpha) p) \). In particular, if we have integers \( \alpha \) and \( \beta \) such that \( \alpha \) is less than the score of all states in the game and \( \beta \) is greater than the scores of all states in the game, for all states \( p \) we’ll get

\[ F_2 (\alpha, \beta) p = F p \text{ and } G_2 (\alpha, \beta) p = G p. \]

So the new functions compute the optimal values. But the big advantage is that the new functions can be designed so that they never explore irrelevant states, where the concept of irrelevance has a nice formulation in terms of the two bounding values.
The move selection functions in the MiniMax functor can also be adapted in a similar way to pay attention to alpha and beta.
For homework you will implement a version of alpha-beta minimax.

4 References

Minimax is a well known algorithm, much discussed in AI courses. Our functional code design is based most closely on a classic paper:


This paper begins by discussing the functions F and G for state evaluation, but the paper uses a pseudo-code which contains assignments, not directly suitable for transcription into a functional programming language.

Nevertheless the paper serves as an invaluable source for the mathematical background, and the section on alpha-beta pruning is useful if you want to improve the efficiency of game evaluation. Many more details concerning how to implement the F2 and G2 functions mentioned above are discussed in the Knuth-Moore paper, although again their pseudo-code is imperative in style.