CPSC 201: Introduction to Computer Science
Carsten Schürmann
Date: May 14, 2002

Homework 7

Solution

In this assignment we implement an interactive and an automated version of the Connect Four game. The objective of this assignment is to let you design and implement a game search procedure based on techniques we discussed in class. We will have another tournament and you can get extra points if your system is among the very best players. I, of course, will be competing, too.

We have prepared a few files for you to get started on this assignment. They can be found in the course directory `/c/cs201/lib/code/ass7/*`. Copy all files in your working directory. From the SML toplevel (once SML is started in your working directory), simply type `CM.make ();` to load all files.

Connect Four is a game that two players play against each other. Each player picks one of two color Red and Yellow respectively. The player then take turns and insert coins into a vertical game board. The coins drop down to the bottom of the board which has seven slots. Each slot can maximally hold seven coins. The winner of the game is the player who first manages to line up four coins of his or her color in a straight line, vertically, horizontally, or diagonally.

Important: Do not edit the signature files. They must remain unchanged otherwise we cannot run the competition and let your implementations play against each other. If you do change them you are automatically disqualified from the competition.

Problem 1: Game setup (20 points)

Implement the structure `Game` in file `game.fun`. Take note of how we represent the configuration of the game. The board is represented by four different entities (r, y, t, l): r is the maximal number of red coins in a line and y the maximal number of yellow coins. t is a list of seven integers, each marks the top of an individual column, and l is a list of lines, including direction, color, length, and the left and right end points. The initial board can hence be represented by the following configuration.

```
  val initialBoard  : board = ((0,0), [0,0,0,0,0,0,0], [])
```

Implement the following four functions:

```
  makeMove  : board -> move -> board
  validMove : board -> move -> bool
  boardToString : board -> string
```

`makeMove` is used to insert a coin into the board, `validMove` decides if a move was valid, and `boardToString` converts a board into a string that can be printed on demand.

```
structure Game : GAME  =
  struct
    exception Error of string
   datatype Color = Red | Yellow
    type square = int * int
    (* sq = (i,j) with 1 <= i,j <= 7 *)
    datatype direction = Horizontal | Rising | Vertical | Falling
    (* printed as -,/,|, respectively *)
    (* Line (d,c,n,(left,right)) : line *)
    (* consisting of the direction d, Color c, length n and *)
    (* left and right endpoints *)
    datatype line =
      Line of direction * Color * int * (square * square)
    (* board (rmax, ymax,tops,lines) consisting of *)
    (* Reds max, Yellows max line length, *)
    (* top of each of the 7 columns, *)
    (* and a list of current connections *)
    type board = (int * int) * int list * line list
    type move = int
    (* The intial board *)
    val initialBoard  : board = ((0,0), [0,0,0,0,0,0,0], [])
    (* val adjacent : direction -> square * square -> bool *)
    (* adjacent d (left,right) returns true *)
    (* if left and right are adjacent in direction d *)
    fun adjacent Horizontal ((i,j),(i',j')) = (i+1=i' andalso j=j')
      | adjacent Rising ((i,j),(i',j')) = (i+1=i' andalso j+1=j')
      | adjacent Vertical ((i,j),(i',j')) = (i=i' andalso j+1=j')
      | adjacent Falling ((i,j),(i',j')) = (i+1=i' andalso j-1=j')
    (*
     val findNeighbors : square -> player -> line list
     -> (line list * line list)
     -> (line list * line list)
     findNeighbors sq p l (accN,accO) >=> (neighbors, others)
       where neighbors are the lines which would be extended by sq,
       others are the remaining lines.  accN and accO accumulate
       the result.
       *)
    fun findNeighbors sq p (nil) (neighbors,others) =
        (neighbors,others)
      | findNeighbors sq p
        ((l as Line(d', p', n', (left, right)))::lines)
        (neighbors,others) =
        findNeighbors sq p lines
        (if p = p'
           then if adjacent d' (right,sq)
             orelse adjacent d' (sq, left)
                  then (l::neighbors, others)
```

```
            else (neighbors, l::others)
       else (neighbors,l::others))
 (* val extendR : square -> direction -> player -> line list
   -> line *)
(*
 extendR sq d p lines, where each line in lines is from
 player p, returns a new line which extends a given at the
 right in direction d by adding sq, or creates a new
 singleton line if no such line exists.
 *)
fun extendR sq d p nil = Line(d, p, 1, (sq,sq))
  | extendR sq d p (Line(d', p', n, (left,right))::lines) =
    if d = d' (* p = p' guaranteed by invariant *)
      andalso adjacent d (right,sq)
      then Line(d', p', n+1, (left,sq))
    else extendR sq d p lines
(* extendL like extendR, but extend on left *)
fun extendL sq d p nil = Line(d, p, 1, (sq,sq))
  | extendL sq d p (Line(d', p', n, (left,right))::lines) =
    if d=d' (* p = p' guaranteed by invariant *)
      andalso adjacent d (sq,left)
      then Line(d', p', n+1, (sq,right))
    else extendL sq d p lines
(* join : line * line -> line *)
(* Join two lines (leftLine, rightLine) adjacent lines *)
(* with same player and direction *)
fun join (Line(d1, p1, n1, (left1,right1)),
          Line(d2, p2, n2, (left2,right2))) =
  (* d1 = d2, p1 = p2, right1 = left2 *)
  Line(d1, p1, n1+n2-1, (left1,right2))
(* incTop : int list * int -> int list *)
(* incTop (tops,i) increments i'th element of tops *)
(* 1 <= i <= |tops| *)
fun incTop (j::tops,1) = (j+1)::tops
  | incTop (j::tops,i) = j::incTop(tops,i-1)
(* val findMax : player -> int -> line list -> int *)
(* findMax p lmax lines *)
(* find maximum of lmax and length player p's lines *)
fun findMax p lmax (nil) = lmax
  | findMax p lmax (Line(_, p', n, _)::lines) =
  if p = p' then findMax p (Int.max(lmax,n)) lines
  else findMax p lmax lines
(* moveSquare : square -> player -> board -> board *)
(*
 move sq p board >=> board', which results from player p
   moving on square p.  We assume the move is legal.  Exception
   Lost is raised, if other player has line of 4 or more
   *)
 fun moveSquare (sq as (i,j)) p ((pmax,omax),tops,lines) =
   let
     val (neighbors,others) =
           findNeighbors sq p lines (nil,nil)
     val newLines =
           List.map (fn d => join (extendR sq d p neighbors,
                                   extendL sq d p neighbors))
           [Horizontal,Rising,Vertical,Falling]
     val tops' = incTop(tops,i)
     val maxs' = (case p
                    of Red => (findMax p pmax newLines,omax)
                  | Yellow => (pmax, findMax p omax newLines))
   in
     (maxs', tops', newLines @ others)
   end
 (* moves : player -> board -> move list *)
 (* Return list of possible moves of player on given board *)
 fun moves Red (board as ((pmax,omax),tops,lines)) =
     if omax >= 4 then nil (* loss---no moves *)
     else moves' 1 tops
   | moves Yellow (board as ((pmax,omax),tops,lines)) =
     if pmax >= 4 then nil (* loss---no moves *)
     else moves' 1 tops
 and moves' i nil = nil
   | moves' i (7::tops') = moves' (i+1) tops'
   | moves' i (j::tops') = i::moves' (i+1) tops'
 (* move : player -> board -> move -> board *)
 fun move p (board as ((pmax,omax),tops,lines)) i =
   moveSquare (i, List.nth(tops,i-1)) p board
(* validMove (B, m) = b
    Invariant:
    and  B is a board
    and  m is a move
    then b = true if m is a valid move on B
         b = false otherwise
 *)
 fun validMove (_, tops, _) m =
   (1 <= m) andalso (m <= 7) andalso (List.nth (tops, m-1) <= 6)
 (* makeMove (C, B, m) = B'
    Invariant:
    If   C is a color
    and  B is a board
    and  m is a move
    then B' is the board after a move has been done
 *)
 fun makeMove C (board as ((pmax,omax),tops,lines)) i =
    let
      val r = moveSquare (i, List.nth(tops,i-1)) C board
    in
      r
    end
  fun colorToString (Red) = "x "
    | colorToString (Yellow) = "o "
  fun lineToCoord (Horizontal, C, (left as (x, y), right)) =
      if left = right then [(left, C)]
      else (left, C) :: lineToCoord (Horizontal, C, ((x+1, y), right))
    | lineToCoord (Vertical, C, (left as (x, y), right)) =
      if left = right then [(left, C)]
      else (left, C) :: lineToCoord (Vertical, C, ((x, y+1), right))
    | lineToCoord (Rising, C, (left as (x, y), right)) =
      if left = right then [(left, C)]
      else (left, C) :: lineToCoord (Rising, C, ((x+1, y+1), right))
    | lineToCoord (Falling, C, (left as (x, y), right)) =
      if left = right then [(left, C)]
      else (left, C) :: lineToCoord (Falling, C, ((x+1, y-1), right))
  fun linesToCoord nil = nil
    | linesToCoord (Line(d,C,n,(left,right)) :: lines) =
        linesToCoord lines @ lineToCoord (d, C, (left, right))
  fun coordToString (8, 0) coord = "\n"
    | coordToString (8, j) coord = "\n" ^ coordToString (1, j-1) coord
    | coordToString (i, j) coord =
        (case (List.find (fn ((i',j'), C) => i = i' andalso j = j') coord)
           of NONE => ". "
            | (SOME (_, C)) => colorToString C) ^ coordToString (i+1, j) coord
  (* boardToString B = s
     Invariant:
     If   B is a board
     then s is a string that represents the board
  *)
  fun boardToString ((pmax,omax),tops,lines) =
    let
      val coord = linesToCoord lines
      val s =  coordToString (1, 6) coord ^ "1 2 3 4 5 6 7\n"
    in
      s
    end
end
```

Problem 2: Interactive player (20 points)

Implement the function

```
play   : int -> Game.Color -> Game.board -> Game.move option
```

in functor `Interactive` in file `player.fun`. This function will print the current board, and prompt the user for input, a number between `1` and `7`. The user can also type in `quit`, and the game is to be aborted (which means), the opponent won. Make sure to ask the player as long as he or she mistypes or makes an invalid move.

```
functor Interactive (structure Game : GAME
                     val name : string) : PLAYER =
  struct
    structure Game = Game
    val name = name
    exception Error of string
    (* play d C B = m
       Invariant:
       If   d is maximal search depth
       and  B is a board
       and  C is the color to be used
       then m is NONE if the player gives up
                 SOME m' where m' is a move, otherwise
    *)
    fun read d C B =
      let
        val _ = print ("Your move (or quit)? ")
        val _ = TextIO.flushOut TextIO.stdOut
        val input = TextIO.inputLine TextIO.stdIn
      in
        case input
          of "quit\n" => NONE
           | _ => (case Int.fromString(input)
                     of NONE => (print ("Illegal input\n"); read d C B)
                      | SOME(i) => (if Game.validMove B i then SOME i
                                    else (print ("Illegal move\n"); read d C B)))
      end
    fun play d C B =
        (print (Game.boardToString B); read d C B)
  end
```

Problem 3: Automatic player (40 points)

Complete the implementation of

```
  play   : int -> Game.Color -> Game.board -> Game.move option
```

in functor `Automatic` in file `player.fun`. This function will look at the game board, apply analysis techniques to your liking and return a move (`SOME m`) where `m` is a move, or resigns (`NONE`). This is the meat of the assignment. The better you perform here, the better chances you have to win the tournament.

Implement the MINIMAX technique discussed in class with depth limited pruning and your choice of evaluation function for non-terminal symbols. Feel free to refine this technique as you see fit.

```
functor Automatic (structure Game : GAME
                   structure Random : RANDOM
                   val name : string) : PLAYER =
  struct
    structure Game = Game
    val name = name
    val rgen = Random.rand (34234,123123)
    exception Error of string
    (* moves : player -> board -> move list *)
    (* Return list of possible moves of player on given board *)
    fun moves Game.Red (board as ((pmax,omax),tops,lines)) =
        if omax >= 4 then nil (* loss---no moves *)
        else moves' 1 tops
      | moves Game.Yellow (board as ((pmax,omax),tops,lines)) =
        if pmax >= 4 then nil (* loss---no moves *)
        else moves' 1 tops
    and moves' i nil = nil
      | moves' i (7::tops') = moves' (i+1) tops'
      | moves' i (j::tops') = i::moves' (i+1) tops'
    (* evaluate : player -> board -> real *)
    (* approximately evaluate board assuming it's player's move *)
    (* real simple: maximum length line *)
    fun evaluate p ((pmax,omax),tops,lines) =
      if pmax >= 4 then 1.0
      else if omax >= 4 then ~1.0
           else real(pmax-omax)/2.5
    (* val eval : G.player -> int -> int -> G.board -> real *)
    (* d >= 0 search to depth d, d < 0 searches whole tree *)
    (* prune is the cut-off value from sibling nodes *)
    fun eval p 0 prune board = evaluate p board
      | eval Game.Red d prune board =
          evalList Game.Red d prune ~1.0 board (moves Game.Red board)
      | eval Game.Yellow d prune board =
          evalList Game.Yellow d prune 1.0 board (moves Game.Yellow board)
    and evalList Game.Red d prune maxv board nil = maxv
      | evalList Game.Red d prune maxv board (m::ms) =
        let
          val v = evalMove Game.Yellow (d-1) maxv Game.Red board m
        in
          if v > prune then v (* prune *)
          else evalList Game.Red d prune (Real.max(maxv,v)) board ms
        end
      | evalList Game.Yellow d prune minv board nil = minv
      | evalList Game.Yellow d prune minv board (m::ms) =
        let val v = evalMove Game.Red (d-1) minv Game.Yellow board m
        in
          if v < prune then v (* prune *)
          else evalList Game.Yellow d prune (Real.min (minv,v)) board ms
        end
    and evalMove p d prune q board m = (eval p d prune (Game.makeMove q board m))
    fun maxVal (l) = foldr Real.max ~1.0 l
    fun minVal (l) = foldr Real.min 1.0 l
    (* val pick : real -> real list -> G.move list
     -> G.move option *)
    (* pick v vs ms  where |vs| = |ms| *)
    fun pick v values moves =
      let
        fun bestMoves nil nil bms = bms
          | bestMoves (v'::vs) (m::ms) bms =
              bestMoves vs ms (if (Real.== (v, v')) then m::bms else bms)
        val bms = bestMoves values moves nil
      in
        case bms
          of nil => NONE
        | _ => SOME(List.nth (bms, Random.randRange (0,List.length bms -1) rgen))
      end
    (* play d C B = m
       Invariant:
       If   d is maximal search depth
       and  B is a board
       and  C is the color to be used
       then m is NONE if the player gives up
                 SOME m' where m' is a move, otherwise
    *)
    fun play d Game.Red board =
        let
          val moves = moves Game.Red board
          val values = map (evalMove Game.Yellow (d-1) ~1.0 Game.Red board) moves
          val maxv = maxVal values
          val m = pick maxv values moves
        in
          m
        end
      | play d Game.Yellow board =
        let
          val moves = moves Game.Yellow board
          val values = map (evalMove Game.Red (d-1) 1.0 Game.Yellow board) moves
          val minv = minVal values
          val m = pick minv values moves;
        in
          m
        end
  end
```

Problem 4: Connect Four game (20 points)

Complete the implementation of

```
playerRed : int -> string
playerYellow : int -> string
```

in functor `Connect4` in file `connect4.fun`. In essence you are implementing a referee, that passes moves from one player to the other. If `playerRed` is called this means that red makes the first move, otherwise with `playerYellow` yellow begins. We need this functionality for the competition.

```
functor Connect4 (structure Game : GAME
                  structure PlayerRed : PLAYER
                    sharing PlayerRed.Game = Game
                  structure PlayerYellow : PLAYER
                    sharing PlayerYellow.Game = Game) : CONNECT4 =
  struct
    exception Error of string
    structure PlayerRed = PlayerRed
    structure PlayerYellow = PlayerYellow
    val timeRed = ref Time.zeroTime;
    val timeYellow = ref Time.zeroTime;
    fun validTime t = Time.toSeconds t < 300
    fun timerRed f =
        let
          val startTime = Time.now ()
          val r = f ()
          val endTime = Time.now ()
        in
          (timeRed := Time.+ (!timeRed, Time.- (endTime,startTime)); r)
        end
    fun timerYellow f =
        let
          val startTime = Time.now ()
          val r = f ()
          val endTime = Time.now ()
        in
          (timeYellow:= Time.+ (!timeYellow, Time.- (endTime,startTime)); r)
        end
    fun timeToString t = Int32.toString (Time.toSeconds t)
    fun has_won Game.Red ((y, r), _, _) = (r = 4)
      | has_won Game.Yellow  ((y, r), _, _) = (y = 4)
    fun runRed d B =
        if validTime (!timeRed) then
          (print (Game.boardToString B);
           if has_won Game.Yellow B then
              (print ("Winner: " ^ PlayerYellow.name ^ "\n"); PlayerYellow.name)
           else
          (print (PlayerRed.name ^ "'s move [" ^ (timeToString (! timeRed)) ^
             "]\n" );checkRed d (timerRed (fn () => PlayerRed.play d Game.Red B)) B))
      else
        (print (PlayerRed.name ^ " disqualified [out of time]\n"); PlayerYellow.name)
  and checkRed d NONE B = (print  (PlayerRed.name ^ " gave up!\n"); PlayerYellow.name)
    | checkRed d (SOME m) B =
      if Game.validMove B m then
        runYellow d (Game.makeMove Game.Yellow B m)
      else (print (PlayerRed.name ^ " disqualified [invalid move]\n"); PlayerYellow.name)
  and runYellow d B =
      if validTime (!timeYellow) then
        (print (Game.boardToString B);
         if has_won Game.Red B then
           (print ("Winner: " ^ PlayerRed.name ^"\n"); PlayerRed.name)
         else
           (print (PlayerYellow.name ^ "'s move [" ^ (timeToString (! timeYellow)) ^  "]\n");
            checkYellow d (timerYellow (fn () => PlayerYellow.play d Game.Yellow B)) B))
      else
        (print (PlayerYellow.name ^ " disqualified [out of time]\n");PlayerRed.name)
  and checkYellow d NONE B = (print  (PlayerYellow.name ^ " gave up!\n"); PlayerRed.name)
    | checkYellow d (SOME m) B =
      if Game.validMove B m then
        runRed d (Game.makeMove Game.Red B m)
      else (print (PlayerYellow.name ^ " disqualified [invalid move]\n"); PlayerRed.name)
  (* playerRed d = s
     Invariant:
     If   d is the search depth
     and  playerRed makes the first move
     then s is the name of the winner
  *)
  fun playerRed d =
    (timeRed := Time.zeroTime;
     timeYellow := Time.zeroTime;
     runRed d Game.initialBoard)
  (* playerYellow d = s
     Invariant:
     If   d is the search depth
     and  playerYellow makes the first move
     then s is the name of the winner
  *)
  fun playerYellow d =
    (timeRed := Time.zeroTime;
     timeYellow := Time.zeroTime;
     runYellow d Game.initialBoard)
end
```

Problem 5: Tournament (up to 40 extra credit points)

Your program will participate in the CS201 tournament. In addition to the ordinary credit for writing a valid implementation of the signature below, we offer a first prize of a tasteful Yale mug plus 40 extra credit points. You obtain 20 ec points for reaching the semi-final, 10 ec points for reaching the quarterfinals, and 5 ec points for reaching the round of the last sixteen. To this end, edit the file `connect4.sml` and replace the string `YourFirstName` by your first name. Hand-in all files in the directory, including `sources.cm`.

Your program loses a match if it raises an uncaught exception, attempts an illegal move, if the cumulative CPU time spent by your program for a match exceeds 300 seconds, or (the usual condition) it cannot make a legal move.

The tournament is played in single elimination format, with two matches (with either program making the initial move) between randomly paired programs in each round. In case of a tie the program which used less CPU time overall advances to the next round. (See structure `Timer`.) In each match, since there can be at most 49 moves, there can be no ties.

1. Xin, Jeremy

2. Eleanor,Pamela

3. Carsten, James Kim, James Wu, Christopher

Hand-in Instructions

In the course directory /c/cs201/bin, there are five programs that support you in submitting your solution to the homework.

```
submit    assignment-number file(s)
unsubmit  assignment-number file(s)
check     assignment-number
protect   assignment-number file(s)
unprotect assignment-number file(s)
```

`submit` allows you to submit one of several files. For example

```
/c/cs201/bin/submit 7 game.sig game.fun ...
```

submits your file `game.sig` and `game.fun`. `check` can give you the peace of mind that your solution has really been submitted, and if you would like to make last minute changes to your solution, use `unsubmit` before submitting the updated version. `protect` and `unprotect` give you the power to protect/unprotect submitted solutions from deletion.