**Checkers**

Homework1-Chekers

Professor Davi Geiger

Due Wednesday, September 19\(^{th}\), 2012.

**Write a program that will execute a game of checkers.** The function below called \texttt{Expand} (S, RB) will be the one to respond to each play provided by the adversary. Here RB=0 means time for red pieces to move and RB=1 means time for the black pieces to move. \( S \) is the state of the checker before a move is to be made. **Within Expand there is a function called \texttt{Evaluate-Moves} (list-of-moves). Do not focus on the “evaluate” question, i.e., how to make decision of what moves to make.**

The function \texttt{SimulateGame} will alternate the RB=0 and RB=1 turns to play/expand and thus simulate a game.

\texttt{CheckersScenarios.pdf} provides four scenarios for you to check your program is working satisfactorily. For each scenario, start with state 1 and it is for red to move. Then, State 2 should result. Then, it is black to move and Final State should result.

/* Neighbors of x */

/* Checkers Board and Coordinates */

```
(a,b)  2,1  4,1  6,1  8,1
  1,2  3,2  5,2  7,2
  2,3  4,3  6,3  8,3

1,6  3,6  5,6  7,6
  2,7  4,7  6,7  8,7
  1,8  3,8  5,8  7,8
```
/* States
S(a, b)=0,
S(a, b)=1 (red)
S(a, b)=2 (black)
S(a, b)=3 (kingred)
S(a, b)=4 (kingblack)
*/

/* Variable RB: RB=0 -> red, RB=1 -> black */
/* Variable FB: FB=0 -> backward, FB=1 -> forward */

SimulateGame()
    Set MAX_PLAY, normally 100 is enough
    roundCount = 1
    S = InitializeGame; /* initialize */
    While roundCount <= MAX_PLAY
        [S] = Expand(S, RB = 0); % RB = 0 for red
        if (S == 0) /* if no possible moves, then red lose the game */
            roundCount = MAX_PLAY;
            drawFlag = 0;
        else
            [S] = Expand(S, RB = 1); % RB = 1 for black
            if (S == 0) /* if no possible moves, then black lose the game */
                roundCount = MAX_PLAY;
                drawFlag = 0;
            roundCount = roundCount + 1;
        endif
    endwhile
    /* If the number of round exceeds MAX_PLAY and no player have won, then agree to draw */
if (roundCount > MAX_PLAY and drawFlag == 1)
    disp('Agree to draws');

InitializeGame()

    S = zeros(8); % two dimension array initialized to be all zeros
    for a=1:8
        for b=1:3
            if (mod(a+b,2) ~= 0)
                S(b,a) = 1; % red pieces
            end
        end
        for b=6:8
            if (mod(a+b,2) ~= 0)
                S(b,a) = 2; % black pieces
            end
        end
    end
    Return [S]

Expand(S, RB) % function to move one more step given board states and player information

    list_of_moves = 0;
    Jump = 0;
    for a = 1:8
        for b = 1:8
            if (mod(a+b,2) == 1) % Check if it is a R/B piece
                if (S(b,a) == 1+RB or S(b,a) == 3+RB) % Exploit grid with R/B pieces
                    [S, Jump, list_of_moves] = ExploitMoves(S, a, b, Jump, list_of_moves, RB);
                end
            end
        end
    end
    if (Jump == 0) % if no jump happened
        if (list_of_moves ~= 0)
            best = EvaluateMoves(list_of_moves);
            S = MakeMove(S, best(1), best(2), best(3), best(4), RB);
        end
    end
else
    \( S = 0; \) % list_of_moves ==0, Game Over
end

end

Return \([S]\)

**MakeMove**\((S, a, b, aa, bb, RB)\) % function to move piece from \((a,b)\) to \((aa,bb)\)

\[
\text{piece} = S(b, a);
\]
\[
S(b, a) = 0;
\]
\[
\text{if} \ (bb == 1+7*(1-RB) \ \text{and} \ \text{piece} == 1+RB) \ % \text{King promotion for simple pieces}
\]
\[
S(bb, aa) = 3+RB;
\]
\[
\text{else}
\]
\[
S(bb, aa) = \text{piece};
\]
\]
\]
\]
Return \([S]\)

**EvaluateMoves**\((\text{list_of_moves})\)

% Just return first element in the list, this function is left for further refinement

\[
\text{best} = \text{list_of_moves}(1);
\]

Return \([\text{best}]\)

% Given board state \(S\), current position & player color, adds possible moves to list_of_moves

**ExploitMoves**\((S, a, b, \text{Jump}, \text{list_of_moves}, RB)\)

\[
\text{if} \ (S(b,a) == 3+RB) \ % \text{If king piece, then check backward move first}
\]
\[
\text{for diagonal} = 0:1
\]
\[
\text{ng} = \text{Neighbor}(a, b, \text{diagonal}, RB, FB = 0); \ % \text{FB = 0 check backward}
\]
\[
\text{if} \ (\text{ng} != 0) \ % \text{neighbor is inside the board}
[S, Jump, list_of_moves] = CheckMove(S, a, b, ng(1), ng(2), diagonal, Jump, list_of_moves, RB, 0);

if ((S(b,a) == 1 + RB) || (S(b,a) == 3 + RB)) % Check there is a piece to move forward
    for diagonal = 0:1
        ng = Neighbor(a, b, diagonal, RB, FB = 1); % FB = 1 check forward
        if (ng != 0) % neighbor is inside the board
            [S, Jump, list_of_moves] = CheckMove(S, a, b, ng(1), ng(2), diagonal, Jump, list_of_moves, RB, 1);
            Return [S, Jump, list_of_moves]
    Return [S, Jump, list_of_moves]

Neighbor(a, b, d, RB, FB) % Get the coord of a place given diagonal, RB and FB
/* FB=1 forward, 0 backward */
/* diagonal or “d’ is either 0 or 1 */
NeighborListOutput=NeighborList(a,b);
neighbor=mod(RB+FB,2);
if (d==0) (neighbor? NeighborListOutput(3): NeighborListOutput (1));
else if (d==1) (neighbor? NeighborListOutput (4): NeighborListOutput (2));
/* mod(RB+FB,2) = (1, redforward, blackbackward; 0 redbackward, blackforward) */

/* In the Cartesian notation, we want [n1(a,b)=(a-1,b-1);n2(a,b)=(a+1,b-1);n3(a,b)=(a+1,b+1);n4(a,b)=(a-1,b+1)]
and the boundary condition is to be inside the board. */

NeighborList(a,b)
n1(a,b) = ( if (((a-1) >= 1) and ((b-1) >= 1))
        (a-1,b-1)
    else 0);
n2(a,b)= ( if (((a-1) >= 1) and (b+1) <= 8))
        (a-1,b+1)
\texttt{else 0);}
\texttt{n3(a,b)= (if (((a+1) <= 8) and ((b+1) <= 8))}
\texttt{(a+1,b+1);}
\texttt{else 0);}
\texttt{n4(a,b)= (if (((a+1) <= 8) and ((b-1) >= 1))}
\texttt{(a+1,b-1);}
\texttt{else 0);}
\texttt{Return(list(n1,n2,n3,n4));}

\texttt{CheckMove(S, a, b, aa, bb, diagonal, Jump, list\_of\_moves, RB, FB)}
\texttt{\quad if ((S(bb,aa) == 0) && (Jump == 0)) \% Empty spot to move and no Jump occured}
\texttt{\quad \quad SS = SimulateMove(S, a, b, aa, bb, RB);} \quad \% SS encodes the new board configuration
\texttt{\quad \quad list\_of\_moves = ListUpdate(SS, a, b, aa, bb, list\_of\_moves);}
\texttt{\quad else if ((S(bb,aa) == 2-RB) || (S(bb,aa) == 4-RB)) \% Black/red piece in front, so Jump maybe possible}
\texttt{\quad \quad ng = Neighbor(aa, bb, diagonal, RB, FB); \% Keep same direction "diagonal"}
\texttt{\quad \quad \% Neighbor must be inside the board & empty}
\texttt{\quad \quad if (ng != 0 and S(ng(2),ng(1)) == 0)}
\texttt{\quad \quad \quad if (Jump == 0)}
\texttt{\quad \quad \quad \quad \quad Jump = 1; \% jump is obliged}
\texttt{\quad \quad \quad \quad \quad list\_of\_moves = 0; \% initialize the list\_of\_moves}
\texttt{\quad \quad \quad \quad \quad [S, list\_of\_moves] = MakeJump(S, a, b, aa, bb, ng(1), ng(2), Jump, list\_of\_moves, RB);}
\texttt{\quad \quad \quad Return [S, Jump, list\_of\_moves]}

\texttt{MakeJump(S, a, b, aa, bb, aaa, bbb, Jump, list\_of\_moves, RB)}
\texttt{\quad piece = S(b,a);}
\texttt{\quad S(b,a) = 0; \% Empty previous occupied spot by RB piece}
\texttt{\quad S(bb, aa) = 0; \% Empty previous occupied spot by 1-RB piece}
promotion = 0;

if ((bbb == 1+7*(1-RB)) and (piece == 1+RB))  % King promotion for simple pieces
    promotion = 1;
    S(bbb,aaa) = 3+RB;
else
    S(bbb,aaa) = piece;

[list_of_moves] = ListUpdate(S, a, b, aaa, bbb, list_of_moves);

if (promotion != 1)  % If not a promotion, must check if a NEW jump is obliged
    [S, Jump, list_of_moves] = ExploitMoves(S, aaa, bbb, Jump, list_of_moves, RB);
    Return [S, list_of_moves]

SimulateMove(S, a, b, aa, bb, RB)

SS = S;

SS(b,a) = 0;  % Move piece to (aa,bb)
if ((bb == 1+7*(1-RB)) and (S(b,a) == 1+RB))  % King promotion for simple pieces
    SS(bb,aa) = 3 + RB;  % Move to aa, bb and king promotion
else
    SS(bb,aa) = S(b,a);  % Move to aa, bb
Return [SS]

ListUpdate(S, a, b, aaa, bbb, list_of_moves)  % Function to update the list_of_move

candidate = [S, a, b, aaa, bbb];

if (list_of_moves == 0)
    list_of_moves = candidate;
else
    list_of_moves = append(candidate, list_of_moves);

Return [list_of_moves]