A new_mcts.ml

```
#open "hextile";;
#open "randomutil";;
#open "player";;
#open "board";;
#open "timeutil";;
#open "lstutil";;

type arbre =
    | Racine
    | Noeud of noeud
and noeud = {
    mutable parent:arbre;
    mutable tester: int;
    mutable victoires: int;
    mutable sous_noeuds: noeud list;
    plateau: color vect vect;
    case_joue: tile;
    joueur_actuelle: color;
    couleur_interesse: color
}
;;

let genererPremierNoeud couleur_interesse premier_joueur size =
    {
        parent=Racine;
        tester=0;
        victoires=0;
        sous_noeuds=[];
        plateau=genBoard size;
        case_joue={x=(-1);y=(-1)};
        joueur_actuelle=premier_joueur;
        couleur_interesse=couleur_interesse
    }
;;
```

```
let estVictoireApresPartieAleatoireNoeud noeud =
    let rec estVictoireApresPartieAleatoirePlateau plateau joueur_actuelle =
        let cases_jouables = avoirCasesJouables plateau in
        match cases_jouables with
            | [] ->
                winOnBoard plateau noeud.couleur_interesse
            | cases_jouables ->
                let prochain_joueur = getOtherColor joueur_actuelle in
                jouerCoupAleatoire plateau joueur_actuelle;
                estVictoireApresPartieAleatoirePlateau plateau prochain_joueur
    in
    let plateau_auxiliaire = copierPlateau noeud.plateau in
    estVictoireApresPartieAleatoirePlateau plateau_auxiliaire noeud.joueur_actuelle
;;

let rec propagationNoeud noeud victoire =
    if victoire then noeud.victoires <- noeud.victoires + 1;
    noeud.tester <- noeud.tester + 1;
    match noeud.parent with
        | Racine -> ()
        | Noeud noeud_parent ->
            propagationNoeud noeud_parent victoire
;;


let simulationNoeud noeud =
    let victoire = estVictoireApresPartieAleatoireNoeud noeud in
    propagationNoeud noeud victoire
;;

let ajouterNoeud noeud case_joue =
    let nouveau_plateau = copierPlateau noeud.plateau in
    setTileColor nouveau_plateau noeud.joueur_actuelle case_joue;
    let nouveau_noeud = {
        parent=Noeud noeud;
        tester=0;
        victoires=0;
        sous_noeuds=[];
        plateau=nouveau_plateau;
        case_joue=case_joue;
        joueur_actuelle=getOtherColor noeud.joueur_actuelle;
        couleur_interesse=noeud.couleur_interesse;
    } in
    noeud.sous_noeuds <- nouveau_noeud::noeud.sous_noeuds;
    nouveau_noeud
;;
```

```
82
83  let avoirSousNoeudDeCase noeud case =
84      let rec aux sous_noeuds =
85          match sous_noeuds with
86              | [] ->
87                  ajouterNoeud noeud case
88              | sous_noeud::reste_sous_noeuds when sous_noeud.case_joue = case ->
89                  sous_noeud
90              | _::reste_sous_noeuds ->
91                  aux reste_sous_noeuds
92      in
93      aux noeud.sous_noeuds
94  ;;
95
96
97  let rec testerSousNoeudAleatoire noeud =
98      let coups_jouables = avoirCasesJouables noeud.plateau in
99      if coups_jouables = [] then
100         simulationNoeud noeud
101     else
102         let case_joue = avoirElementAleatoire coups_jouables in
103         let rec aux sous_noeuds =
104             match sous_noeuds with
105                 | [] ->
106                     let nouveau_noeud = ajouterNoeud noeud case_joue in
107                     simulationNoeud nouveau_noeud
108                 | sous_noeud::reste_sous_noeuds
109                         when sous_noeud.case_joue = case_joue ->
110                     testerSousNoeudAleatoire sous_noeud
111                 | _::reste_sous_noeuds ->
112                     aux reste_sous_noeuds
113         in
114         aux noeud.sous_noeuds
115 ;;
116
117 let testerSousNoeudAleatoirePendant temp noeud =
118     let start = avoirTemp () in
119     while avoirTemp () -. start < temp do
120         testerSousNoeudAleatoire noeud
121     done
122 ;;
123
124 let estMeilleurNoeud noeud1 noeud2 =
125     noeud1.victoires / noeud1.tester > noeud2.victoires / noeud2.tester
126 ;;
127
```

```
128 let avoirMeilleurSousNoeudAvant temp noeud =
129     testerSousNoeudAleatoirePendant temp noeud;
130     max estMeilleurNoeud noeud.sous_noeuds
131 ;;
132
133 let printNoeud noeud =
134     print_string "Noeud:"; print_string "\n";
135     print_string "    tester:"; print_int noeud.tester; print_string "\n";
136     print_string "    victoires:"; print_int noeud.victoires; print_string "\n";
137     print_string "    case_joue:"; printTile noeud.case_joue; print_string "\n";
138     print_string "    joueur_actuelle:"; afficherCouleur noeud.joueur_actuelle
139          print_string "\n";
140     print_string "    couleur_interesse:";
141         afficherCouleur noeud.couleur_interesse;
142         print_string "\n"
143 ;;
```

B player.ml

```
1  #open "randomutil";;
2  #open "hextile";;
3  #open "hex";;
4  #open "board";;
5  #open "lstutil";;
6  #open "graph";;
7
8  (* On dit que deux cases sont connectes si elles sont relies par une
9      suite de case de mme couleur*)
10 let winOnBoard board color =
11     (* Indique si `color` a gagn sur `board` *)
12     let isConnectedToEndSide =
13         isConnectedToEnd (getSameColorNeighbourTiles board)
14             (isNextToEndSide board color) in
15     (* fonction indique si une case est connecte  une case de fin *)
16     let filled_start_tiles = filter (isColor board color)
17         (getColorStartTiles board color) in
18     any isConnectedToEndSide filled_start_tiles
19 ;;
```

```
21 let winner board = findIf (winOnBoard board) Empty [Red; Blue];;
22     (* Indique qui a gagn sur le plateau `board`, si personne n'a gagne
23         renvoie `Empty`*)
24
25 let rec getWinningPlayer board will_play=
26     (* Indique si une position du plateau `board` est bonne pour le joueur
27         `will_play` qui va jouer *)
28     let empty_tiles = getTilesOfColor board Empty in
29     match empty_tiles with
30         | [] -> winner board
31         | lst when any (isWinningPlay board will_play) lst -> will_play
32         | _ -> getOtherColor will_play
33
34 and isWinningPlay board color tile =
35     (* Renvoie si le la case `tile` est un bon coup pour le joueur
36         `color` sur le plateau `board` *)
37     let board_after_play = setTileColor board color tile in
38     let next_player = getOtherColor color in
39     let is_winning_play = getWinningPlayer board_after_play next_player = color in
40     setTileColor board Empty tile;
41     is_winning_play
42 ;;
43
44 let getWinningPlay board color =
45     (* Renvoie le coup  jouer sur le plateau `board` pour le joueur
46         `color` *)
47     let empty_tiles = getTilesOfColor board Empty in
48     findIf (isWinningPlay board color) (hd empty_tiles) empty_tiles
49 ;;
50
51 (* La valeur d'un plateau est `(r, b, w)` ou `r` est le nombre de
52     faon que `r` peut gagner en partant de ce plateau `b` est la
53     mme mais pour bleu et `w` la personne qui gagne si elle jout
54     parfaitement en partant du plateau *)
55
56 let getFullBoardValue board =
57     let winner_of_board = winner board in
58     match winner_of_board with
59         | Blue -> (0, 1, Blue)
60         | Red -> (1, 0, Red)
61         | Empty -> raise (Failure "The board is not valid or not full")
62 ;;
```

```
64  let addBoardsValue has_played v1 v2 =
65      match v1, v2 with
66          | (r1, b1, w1), (r2, b2, w2) when w1 <> w2 ->
67              (r1 + r2, b1 + b2, has_played)
68          | (r1, b1, _), (r2, b2, _) ->
69              (r1 + r2, b1 + b2, getOtherColor has_played)
70  ;;
71
72  let rec getBoardValue board will_play=
73      (*Renvoie la valeur du plateau `board` sachant que le joueur
74          `will_play` va jouer*)
75      let empty_tiles = getTilesOfColor board Empty in
76      match empty_tiles with
77          | [] -> getFullBoardValue board
78          | lst ->
79              let after_play_values = map (getBoardValueAfterPlay board will_play)
80                  empty_tiles in
81              (* liste des valeurs des plateaux ateignable depuis le
82                  plateau `board` *)
83              reduce (addBoardsValue will_play) (0, 0, getOtherColor will_play)
84                  after_play_values
85
86  and getBoardValueAfterPlay board playing_color tile =
87      (* Renvoie la valeur du tableau obtenue en faisant jou `color`
88          sur la case `tile` sur le plateau `board` *)
89      setTileColor board playing_color tile;
90      let after_play_value = (getBoardValue board (getOtherColor playing_color)) in
91      setTileColor board Empty tile;
92      after_play_value
93  ;;
94
95  let isBetterFor color v1 v2 =
96      match v1, v2 with
97          | ((r1, b1, w1), _), ((r2, b2, w2), _) ->
98              w1 = color && w2 <> color
99              || color = Blue && b1 > b2
100             || color = Red && r1 > r2
101 ;;
```

```
103 let getBestPlay board color =
104     (* Renvoie le coup  jouer sur le plateau 'board' pour le joueur
105         'color' *)
106     let empty_tiles = getTilesOfColor board Empty in
107     (* Cherche le coups qui amne le joueur au plateau ayant la plus grande
108         valeur *)
109     snd (max (isBetterFor color)
110         (getImagesAndAntecedants (getBoardValueAfterPlay board color) empty_tiles))
111 ;;
112
113 let jouerCoupAleatoire plateau joueur =
114     let cases_jouables = avoirCasesJouables plateau in
115     let case_joue = avoirElementAleatoire cases_jouables in
116     setTileColor plateau joueur case_joue
117 ;;
118
```

C board.ml

```
1  #open "hextile";;
2
3  type color = Red | Blue | Empty;;
4
5  let genBoard size =
6      let board = make_vect size [||] in
7      for i = 0 to size - 1 do
8          board.(i) <- make_vect size Empty
9      done;
10     board
11 ;;
12
13 let getBoardSize board =
14     vect_length board
15 ;;
```

```
let copierPlateau plateau =
    let taille = (getBoardSize plateau) in
    let nouveau_plateau = make_vect taille [||] in
    for i = 0 to taille - 1 do
        nouveau_plateau.(i) <- make_vect taille Empty;
        for j = 0 to taille - 1 do
            nouveau_plateau.(i).(j) <- plateau.(i).(j)
        done
    done;
    nouveau_plateau
;;

let getOtherColor color =
    match color with
        | Red -> Blue
        | Blue -> Red
        | _ -> raise (Failure "Empty does not have an opposite color")
;;

let setTileColor board color tile =
    board.(tile.x).(tile.y) <- color;
    board
;;

let getTileColor board tile =
    board.(tile.x).(tile.y)
;;

let isColor board color tile =
    color = getTileColor board tile
;;

let rec fillBoardWith board color_and_tiles =
    match color_and_tiles with
        | [] -> board
        | color_and_tile::rst_color_and_tiles ->
            let board_parially_filled = fillBoardWith board rst_color_and_tiles in
            let color, tile = color_and_tile in
            setTileColor board_parially_filled color tile;
            board
;;
```

```
59  let getTilesOfColor board color =
60      (* Renvoie la liste des coordone des case de couleur ‘color‘ sur
61          le plateau ‘board‘ *)
62      let rec aux tile =
63          (* Itre sur chaque case du plateau *)
64          match tile with
65              | {x=(-1); y=_} -> []
66              | {x=n; y=(-1)} -> aux {x=(n - 1); y=(getBoardSize board - 1)}
67              | {x=xn; y=yn} when isColor board color tile ->
68                  tile::(aux {x=xn; y=(yn - 1)})
69              | {x=xn; y=yn} -> aux {x=xn; y=(yn - 1)}
70      in
71      aux {x=(getBoardSize board - 1); y=(getBoardSize board - 1)}
72  ;;
73
74  let avoirCasesJouables board = getTilesOfColor board Empty;;
75
76
77  let isOnBoard board tile =
78      tile.x >= 0 && tile.x < getBoardSize board
79      && tile.y >= 0 && tile.y < getBoardSize board
80  ;;
81
82
83  let isFull board =
84      getTilesOfColor board Empty = []
85  ;;
86
87  let printBoard board =
88      for i = 0 to getBoardSize board - 1  do
89          for j = 0 to getBoardSize board - 1 do
90              match board.(i).(j) with
91                  | Blue -> print_string "Blue  "
92                  | Red -> print_string "Red   "
93                  | Empty -> print_string "Empty ";
94          done;
95          print_newline ()
96      done
97  ;;
98
99  let afficherCouleur couleur =
100     match couleur with
101         | Blue -> print_string "Blue"
102         | Red -> print_string "Red"
103         | Empty -> print_string "Empty"
104 ;;
```

D hextile.ml

```
type tile = {x:int; y:int};;

let add tile1 tile2 = {x=tile1.x + tile2.x; y=tile1.y + tile2.y};;

let printTile tile =
    print_char '(';
    print_int tile.x;
    print_char ',';
    print_int tile.y;
    print_char ')'
;;
```

E randomutil.ml

```
#open "lstutil";;

let initierAleatoire () =
    let system_random_number_generator = open_in "/dev/urandom" in
    random__init (input_binary_int system_random_number_generator)
;;

let avoirElementAleatoire lst =
    avoirNEmeElement lst (random__int (list_length lst))
;;
```

F lstutil.ml

```
let rec list_length lst =
    match lst with
        | [] -> 0
        | _::rlst -> 1 + list_length rlst
;;

let rec filter predicat lst =
    (* Renvoie la liste des lment verifiant le predicat 'predicat'*)
    match lst with
        | [] -> []
        | x::rlst when predicat x -> x::(filter predicat rlst)
        | _::rlst -> (filter predicat rlst)
;;
```

```
let renverser lst =
    let rec aux lst sortie =
        match lst with
            | [] -> sortie
            | x::rlst -> aux rlst (x::sortie)
    in
    aux lst []
;;

let map f lst =
    (* Applique f  tout les lment de la liste *)
    let rec aux f lst sortie =
        match lst with
            | [] -> sortie
            | x::rlst -> (aux f rlst ((f x)::sortie))
    in
    renverser (aux f lst [])
;;

let rec reduce f a lst =
    (* Si lst = [a1; ... an], renvoit f( a1 f( ... f( an a) ) ) *)
    match lst with
        | [] -> a
        | x::rlst -> f x (reduce f a rlst)
;;

let rec applicationSuccessive f a n =
    (* renvoit [f^n (a); f^{n - 1} (a); ... ; f^2 (a); f (a); a] *)
    match n with
        | 0 -> [a]
        | n ->
            let reste_app_successive = applicationSuccessive f a (n - 1) in
            (f (hd reste_app_successive))::reste_app_successive
;;

let rec any predicat lst =
    match lst with
        | [] -> false
        | x::rlst -> (predicat x) || (any predicat rlst)
;;
```

```
let rec isIn lst x =
    match lst with
        | [] -> false
        | elm::rlst when x = elm -> true
        | _::rlst -> isIn rlst x
;;

let rec findIf predicat default lst =
    (* Renvoie le premier lment verifiant `predicat` ou `default` si aucun *)
    match lst with
        | [] -> default
        | x::rlst when predicat x -> x
        | _::rlst -> findIf predicat default rlst
;;

let rec getImagesAndAntecedants f lst =
    match lst with
        | [] -> []
        | x::rlst -> (f x, x)::(getImagesAndAntecedants f rlst)
;;

let rec max isGreater lst =
    (* Renvoie le maximum de tel que {a > b <=> isGreater a b} *)
    let rec aux current_max lst =
        match lst with
            | [] -> current_max
            | x::rlst when isGreater x current_max -> aux x rlst
            | _::rlst -> aux current_max rlst
    in
    match lst with
        | [] -> raise (Failure "Empty list has no max")
        | x::rlst -> aux x rlst
;;

let rec avoirNEmeElement lst n =
        match n, lst with
            | _, [] -> raise (Failure "Out Of Bound")
            | 0, elm::_ -> elm
            | n, _::rlst -> avoirNEmeElement rlst (n - 1)
;;
```

```
let rec supprimer liste valeur =
    match liste with
        | [] -> []
        | element::reste_liste when element = valeur -> reste_liste
        | element::reste_liste -> element::(supprimer reste_liste valeur)
;;

let rec appliquerProcedure procedure liste =
    match liste with
        | [] -> ()
        | element::reste_liste ->
            (procedure element); appliquerProcedure procedure reste_liste
;;
```

G timeutil.mli

```
value avoirTemp: unit -> float = 1 "avoirTemp";;
```

H timeutil.c

```
#include </usr/local/lib/caml-light/mlvalues.h>
#include </usr/local/lib/caml-light/alloc.h>
#include <time.h>

value avoirTemp(value unit)
{
    return copy_double(((double) clock()) / CLOCKS_PER_SEC);
}
```