Révisions

OPTION INFORMATIQUE - TP n° 4.5 - Olivier Reynet

À la fin de ce chapitre, je sais :

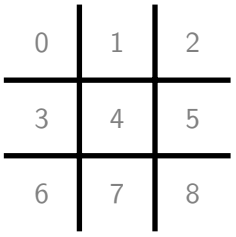
- ☞ manipuler une structure de donnée sous la forme d'un vecteur de bits
- ☞ utiliser les fonction d'itération sur les listes
- ☞ implémenter l'algorithme minimax

A Modélisation d'un jeu de morpion

On se donne les types OCaml suivants :

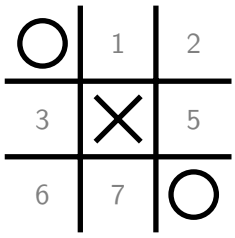
```
type ttt_player = PCross | PNought;;
type ttt_cell = Empty | Cross | Nought;;
type ttt_board = {cross : int; nought : int};;
```

Ils représentent les joueurs (croix ou rond), les cellules (vides, avec une croix ou un rond) et l'aire de jeu séparée en deux camps (croix et ronds). L'aire de jeu est numérotée de bas en haut, de 0 à huit. L'occupation du jeu est implémentée par des entiers nommés **bitboards** : chaque bit du bitboard $b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$ représente une case occupée si le bit est à 1 et inoccupée si le bit est à zéro. Un bitboard est donc un vecteur de neuf bits et se note en hexadécimal.



$\rightarrow b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

■ **Exemple 1 — Exemple de modèle de partie morpion.** La partie



est représentée par `let board = {cross = 0x010; nought = 0x101}`

A1. Écrire une fonction de signature `pos_to_bb : int -> int` qui convertit un numéro représentant une case de 0 à 8 en un bitboard. Par exemple, `pos_to_bb 8 ; ;` renvoie `256`.

Solution :

```
let pos_to_bb position = 1 lsl position;;
```

A2. Écrire une fonction de signature `create_bb : int list -> int` qui prend en paramètre la liste des numéros des cases occupées et qui renvoie le bitboard associé. Par exemple, `create_bb [2 ;4 ;6]` renvoie `84`.

Solution :

```
let create_bb positions =
    let rec aux bb pos =
        match pos with
            | [] -> bb
             | p::t -> aux (bb + pos_to_bb p) t
    in aux 0 positions;;
```

A3. Écrire la fonction précédente à l'aide de la fonction d'itération `List.fold_left`.

Solution :

```
let create_bb positions =
        List.fold_left (fun bb p -> bb + pos_to_bb p) 0 positions;;
```

A4. Écrire une fonction de signature `get_cell : ttt_board -> int -> t_cell` qui renvoie le contenu de la cellule de numéro k dans une partie. Sur la partie de l'exemple 1, l'expression `get_cell board 4` renvoie `Cross`.

Solution :

```
let get_cell board k =
    let c = ((board.cross lsr k) land 0x1) and
    n = ((board.nought lsr k) land 0x1) in
    match (c,n) with
        | (0,0) -> Empty
        | (1,0) -> Cross
        | (0,1) -> Nought
        | _ -> failwith "Cell Error !";;
```

B Utilitaires

B1. Écrire une fonction de signature `show : ttt_board -> unit` qui imprime la partie de morpion sur la console.

Solution :

```
let show board =
    Printf.printf "\n";
        for k = 8 downto 0 do
            let cell = get_cell board k in
            match cell, k mod 3 = 0  with
                | Empty, true -> Printf.printf "_\n"
                | Cross, true  -> Printf.printf "X\n"
                | Nought, true -> Printf.printf "O\n"
                | Empty, false -> Printf.printf "_"
                | Cross, false -> Printf.printf "X"
                | Nought, false -> Printf.printf "O"
    done;;
```

B2. Écrire une fonction de signature `bb_free_moves : ttt_board -> int` qui renvoie le bitboard représentant toutes les positions libres du jeu.

Solution :

```
let bb_free_moves board =
    let occupancy = board.cross lor board.nought in
    (lnot occupancy) land 0x1FF;;
```

B3. Écrire une fonction de signature `free_moves : ttt_board -> int list` qui renvoie les numéros des case des positions libres du jeu sous la forme d'une liste. Par exemple, pour la partie de l'exemple 1, `free_moves board` renvoie `[7 ; 6 ; 5 ; 3 ; 2 ; 1]`.

B4. Créer la liste `winning_bb` des bitboards de tous les motifs gagnants du jeu.

Solution :

```
let winning_bb = [ 0x1C0; 0x038; 0x007; (*lines*)
                   0x124; 0x092; 0x049; (*cols*)
                   0x054; 0x111 (*diags*) ];;
```

B5. Écrire une fonction de complexité constante et de signature `is_winning_bb : int -> bool` qui teste si un bitboard est gagnant.

Solution :

```
let is_winning_bb bb =
    let rec aux remaining =
        match remaining  with
            | [] -> false
            | w::_ when (bb land w) = w -> true
            | _::t -> aux t  in
    aux winning_bb;;
```

B6. Écrire une fonction de signature `winner : ttt_board -> t_player option` qui renvoie le joueur vainqueur d'une partie ou `None` s'il n'y en a pas.

Solution :

```
let winner board =
    match (is_winning_bb board.cross, is_winning_bb board.nought) with
        | (false, false)-> None
        | (true, false) -> Some PCross
        | (false, true) -> Some PNought
        | _ -> failwith "Can not have two winners !" ;;
```

B7. Écrire une fonction de signature `pos_eval : ttt_board -> int` qui évalue la qualité d'une position, c'est-à-dire la différence entre le nombre de triplets gagnants encore accessibles à `PCross` et le nombre de triplets gagnants encore accessibles à `PNought`. Cette fonction constituera l'heuristique pour l'implémentation de minimax.

Solution :

```
let check_w_pos winning bb = (((lnot bb) land 0x1FF) land winning) = winning;;

let pos_eval board =
    let check c w b = if check_w_pos w b then c+1 else c in
    let count_lines bb = List.fold_left (fun c w -> check c w bb) 0 winning_bb in
    let cross_eval = count_lines board.nought and nought_eval = count_lines board.cross in
    (*Printf.printf "%d,%d" cross_eval nought_eval;*)
    cross_eval - nought_eval;;
```

C Implémentation de Minimax

C1. Écrire une fonction de signature `minimax : ttt_board -> int -> t_player -> int list -> int list * int` qui calcule la position à jouer pour un joueur d'après l'algorithme minimax. On choisira la convention suivante : `PCross` est J_{max} et `PNought` J_{min}.

Solution :

```
let rec minimax board depth player moves =
    Printf.printf " depth -> %d, last move -> %d\n" depth (List.hd moves);
    let next_board player bb =
        match player with
            | PCross -> {cross = bb; nought = board.nought}
            | PNought -> {cross = board.cross; nought = bb} in
    let next_move free_cells next_player op =
        let bb = match player with
            | PCross ->  board.cross
            | PNought ->  board.nought in
    let possibles =
        List.map (fun move ->
```

```
            let n_bb = ((pos_to_bb move) lor bb) in
            let n_ttt =  next_board player n_bb in
            minimax n_ttt (depth - 1) next_player (move::moves))
            free_cells in
        List.fold_left
        (fun (macc, sacc) (m, s) -> if  op sacc s then (macc, sacc) else (m,s))
        (List.hd possibles)
        possibles in
        match (winner board, depth, free_moves board) with
            | (Some PCross,_,_) -> Printf.printf "PCross wins !\n"; (moves, 10)
            | (Some PNought,_,_) -> Printf.printf "PNought wins !\n"; (moves, -10)
            | (_,0,_) -> (moves, pos_eval board)
            | (None,_,[]) -> (moves, 0)  (* no more free moves *)
            | (None,_,fm) -> match player with
            | PCross -> next_move fm PNought (>)
            | PNought -> next_move fm PCross (<);;
```