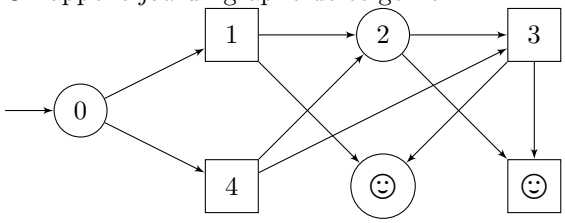
Jeux et stratégies gagnantes

La théorie des jeux est une discipline très sérieuse. Elle trouve des applications dans des domaines aussi variés que l'économie, la théorie des nombres et l'informatique. En informatique, elle sert à analyser des systèmes devant interagir avec un environnement dont on ne connaît pas le comportement de manière exacte, par exemple un utilisateur éventuellement mal intentionné. On a alors affaire à des jeux combinatoires sur des objets tels que des arbres ou des graphes. Aujourd'hui on va s'intéresser au calcul des stratégies gagnantes dans les jeux sur les graphes

1 Graphes de jeux

On appelle *jeu* un graphe de ce genre :



Où les ronds sont les états appartenant au joueur 1 et les carrés ceux du joueur 2. Quand un joueur possède un état, c'est lui qui choisit dans lequel des états possibles on va au coup suivant. Les objectifs sont pour le joueur 1 d'arriver à l'état ☺ et à l'état ☺ pour le joueur 2.

Formellement un jeu peut être représenté par un 6-uplet $\langle V_1, V_2, i, E, \Omega_1, \Omega_2 \rangle$. Où $(V_1 \cup V_2, E)$ est un graphe avec V_j représentant les états contrôlés par le joueur j, $i \in V_1 \cup V_2$ est l'état initial, $\Omega_j \subset V_1 \cup V_2$ représente l'objectif du joueur j. On dit que le joueur j gagne si un des états de Ω_j est atteint à la fin de la partie.

On représenteras un jeu de la façon suivante en Caml, sans préciser pour l'instant le type que prendront les états.

```
type 'a game =
  { state : 'a list ;
    owner : 'a -> player ;
    init : 'a ;
    trans: 'a -> 'a list ;
    win1 : 'a list ;
    win2 : 'a list }
```

Le champ trans associe à un état, la liste de ses successeurs possibles. Le type player est définie par :

```
type player = P1 | P2
```

▶ **Question 1** *Donner la représentation en Caml de l'exemple du dessin.*

val example : int game

Une stratégie (sans mémoire) pour le joueur 1 est une fonction qui à chaque état contrôlé par joueur 1 associe une transition possible à partir de cet état. Une stratégie serait représenté en Caml par le type suivant

```
type 'a strategy = ('a -> 'a)
```

Une stratégie est dite *gagnante* si quelle que soit la stratégie du second joueur, on atteint un de nos objectifs. Par exemple dans le jeu représenté sur le dessin, joueur 2 a une stratégie gagnante mais pas joueur 1.

▶ **Question 2** *Représenter en Caml la stratégie gagnante de joueur 2.*

val winning_strat : int strategy

▶ **Question 3** *Écrire un fonction* execute *qui étant donné un jeu et deux stratégies fait avancer la partie en suivant les instructions donnés par les stratégies et s'arrête quand un objectif a été atteint, dans ce cas là on renverra cet état. Dans le cas où une stratégie propose une transition non autorisée il faudra lever une exception.*

val execute : 'a game -> 'a strategy -> 'a strategy -> 'a

Vous pouvez tester cette fonction avec la stratégie gagnante de joueur 2 et d'autre stratégies de joueur 1 et vérifier que l'on arrive toujours dans l'état gagnant de joueur 2.

2 Attracteurs

Pour décider si un joueur à une stratégie gagnante on procède par un calcul d'attracteur. L'ensemble Attr_i^j représente l'ensemble des états du jeu à partir desquels le joueur j possède une stratégie qui le fait gagner en i étapes ou moins. Le calcul se fait récursivement :

- Au rang 0, c'est exactement l'objectif : $\mathsf{Attr}_0^j = \Omega_j$
- Au rang $i+1$ on ajoute les états contrôlés par j où au moins un successeur est gagnant (en i étapes) ainsi que les états contrôlés par l'adversaire où tout les successeurs sont gagnants : $\mathsf{Attr}_{i+1}^j = \mathsf{Attr}_i^j \cup \{s \in V_j \mid \exists (s, s') \in E.\ s' \in \mathsf{Attr}_i^j\} \cup \{s \in V_{3-j} \mid \exists s' \in V.\ (s, s') \in E \text{ et } \forall (s, s') \in E.\ s' \in \mathsf{Attr}_i^j\}$

On s'arrête à partir du rang i où l'attracteur ne grandit plus : $\mathsf{Attr}_{i+1}^j = \mathsf{Attr}_i^j$, car à ce moment on a atteint un point fixe et les états gagnants de j sont exactement les états de Attr_i^j.

On aura besoin d'une structure de donnée pour représenter les ensembles. On a déjà utiliser plusieurs fois des listes pour cela, mais le fait que l'accès se fasse

en temps linéaire les rend peu efficace pour des gros ensembles. Vous devez aussi connaître les arbres binaires équilibrés dans lesquels l'accès et en temps logarithmique. Le meilleur temps est atteint avec les tableaux pour lesquels l'accès se fait en temps constant, c'est donc ce qu'il y a de mieux pour des ensembles très grand. Leur inconvénient est qu'ils ne peuvent être indexés que par des entiers, ce qui nous empêche de les utiliser pour représenter des ensembles d'états qui peuvent avoir un type quelconque. La solution serait d'utiliser des tables de hachage, cette structure se comporte comme un tableau (accès en temps constant) mais présente l'avantage de pouvoir être indexé par des clés de type quelconque et de ne pas avoir de taille fixe. En Caml elles ont le type `Hashtbl.t` et se manipule grâce aux fonctions suivantes :

- `Hashtbl.create n` crée une table destiné à contenir environ `n` éléments ;
- `Hashtbl.add t a b` ajoute dans la table `t` l'élément ayant pour clé `a` et pointant vers `b` ;
- `Hashtbl.find t a` renvoie l'élément pointé par `a` ;
- `Hashtbl.fold f t init` calcule `(f kN dN ( ... (f k1 d1 init ))` où les `k1 ... kN` sont les clés et les `d1 ... dN` sont les valeurs de tout les éléments présents dans la table.

Pour représenter les ensembles on pourra donc utiliser une table à valeur booléenne.

```
type 'a set = ('a,bool) Hashtbl.t
```

▶ **Question 4** *Implémenter les fonctions de base sur les ensembles.*

```
val empty_set : unit -> 'a set
val is_empty : 'a set -> bool
val add : 'a set -> 'a -> unit
val mem : 'a set -> 'a -> bool
val remove : 'a set -> 'a -> unit
```

▶ **Question 5** *Écrivez deux fonctions prenant en argument un jeu, un état et une fonction booléenne. La première vérifie que tout les successeurs de l'état sont évalués à vrai par la fonction booléenne, et que l'état en question à au moins un successeur. La deuxième vérifie qu'il existe un successeur évalué à vrai.*

```
val forall_succ : 'a game -> 'a -> ('a -> bool) -> bool
val exists_succ : 'a game -> 'a -> ('a -> bool) -> bool
```

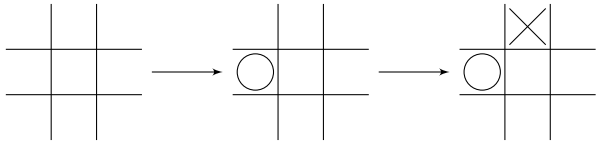
▶ **Question 6** *Écrivez une fonction réalisant une étape du calcul de l'attracteur : elle doit ajouter à l'ensemble en argument (représentant Attr_i) les états qui sont dans l'attracteur à l'étape $i+1$. On renverra un booléen, vrai si des états ont été ajouté, faux sinon.*

```
val step : 'a game -> player -> 'a set -> bool
```

▶ **Question 7** *Écrivez une fonction qui étant donné un jeu et un joueur, renvoie l'ensemble des états du jeu qui sont dans l'attracteur de ce joueur. Testez votre fonction sur le jeu donné en exemple.*

```
val attractor : 'a game -> player -> 'a set
```

3 Tic-tac-toe



Tic-tac-toe est un jeu se jouant sur une grille 3×3. Tour à tour les 2 joueurs marquent une case avec un rond pour le joueur 1, une croix pour le joueur 2. Le premier à aligner 3 marques (en colonne, ligne ou diagonale) gagne la partie.
On représenteras les cases par le type `cell`.

```
type cell = V | O | X
```

Et les configurations du jeu par une liste de listes (de taille 3).

```
type conf = cell list list
```

▶ **Question 8** *Écrire en Caml la configuration vide, une fonction qui ajoute une marque dans la case donnée, et une fonction renvoyant la marque contenu dans une case.*

```
val empty_conf : conf
val set : conf -> int -> int -> cell -> conf
val get : conf -> int -> int -> cell
```

Il nous faut maintenant générer la liste des états appartenant à chaque joueur et la liste des états gagnants.

▶ **Question 9** *Commencer par calculer la liste des (3^9) configurations possibles.*

```
val all_states : conf list
```

▶ **Question 10** *Écrire une fonction* `owner` *qui étant donné une configuration, renvoyé le joueur qui doit jouer au coup suivant, et qui lance une exception si la configuration n'est pas correcte. Et une autre fonction qui renvoie vrai si cette configuration est gagnante pour le joueur.*

```
owner : conf -> player -> bool
winning : conf -> player -> bool
```

▶ **Question 11** *Écrire une fonction qui renvoie les successeurs possibles d'une configuration.*

```
val successors : conf -> conf list
```

▶ **Question 12** *Écrire le jeu tic-tac-toe en Caml et calculer les attracteurs des deux joueurs. Est-ce que l'un des deux joueurs à une stratégie gagnante ?*

```
val tictactoe : conf game
```

4 Calcul de la stratégie gagnante

Pour calculer la stratégie gagnante il nous manque de l'information : quand on a ajouté un état dans l'attracteur car il existait un successeur dans l'attracteur il faut conserver ce successeur comme témoin, pour savoir quelle est la stratégie à jouer.
Pour cela on va conserver au long du calcul une table contenant pour les états dans l'attracteur, la stratégie à jouer.

▶ **Question 13** *Modifier les fonctions* `step` *et* `attractor` *pour qu'en plus de l'attracteur elles remplissent la table.*

```
val step : 'a game -> player -> 'a set -> ('a,'a) Hashtbl.t -> bool
val attractor : 'a game -> player -> ('a set * ('a,'a) Hashtbl.t)
```

▶ **Question 14** *Calculer une stratégie gagnante pour le jeu tic-tac-toe.*

```
win_strat : conf strategy
```

Jeux et stratégies gagnantes

Un corrigé

▶ Question 1

```
let example =
  { state = [0;1;2;3;4;5;6];
    owner = (function
        | 0 | 2 | 5 -> P1
        | 1 | 3 | 4 | 6 -> P2 );
    init = 0;
    trans = (function 0 -> [1;4] | 1 -> [2;5] | 2 -> [3;6]
      | 4 -> [2;3] | _ -> []);
    win1 = [5];
    win2 = [6] }
```

▶ Question 2

```
let winning_strat = function
  | 1 -> 2
  | 4 -> 2
  | 3 -> 6
```

▶ Question 3

```
let execute g s1 s2 =
  let rec aux i =
    if List.mem i g.win1 || List.mem i g.win2
    then i
    else
      let t = match g.owner i with P1 -> s1 i | P2 -> s2 i
      in
if List.mem t (g.trans i) then aux t else failwith "not_allowed"
  in aux g.init
```

▶ Question 4

```
let empty_set () = Hashtbl.create 1000

let is_empty t =
  Hashtbl.fold (fun x b r -> b || r) t true

let add t a = Hashtbl.add t a true

let mem t a =
  try Hashtbl.find t a
  with Not_found -> false

let remove t a = Hashtbl.add t a false
```

▶ Question 5

```
let forall_succ g s p =
  let rec aux = function
    | [] -> false
    | a :: [] -> p a
    | a :: s -> p a && aux s
  in aux (g.trans s)

let exists_succ g s p =
  List.fold_left (fun b v -> p v || b) false (g.trans s)
```

▶ Question 6

```
let step g p a =
  let modif =
    List.fold_left
      ( fun b s ->
          if (g.owner s = p) && (not (mem a s))
            && exists_succ g s (mem a)
          then (add a s; true)
          else b )
      false g.state
  in
    List.fold_left
      ( fun b s ->
          if (g.owner s <> p) && (not (mem a s))
            && forall_succ g s (mem a)
          then (add a s; true)
          else b )
      modif g.state
```

▶ Question 7

```
let attractor g p =
  let a = empty_set () in
    List.iter (add a)
      (match p with P1 -> g.win1 | P2 -> g.win2);
    while (step g p a) do () done;
    a
```

▶ Question 8

```
let empty_conf =
  let l = [ V ; V ; V] in
    [ l ; l ; l ]

let rec get_line c i = match c with
  | a :: s -> if i = 0 then a else get_line s (i-1)

let get c i j =
  get_line (get_line c i) j

let set c i j a =
  let rec aux i x = function
    | a :: s -> if i = 0 then x :: s else a :: aux (i-1) x s
  in aux i (aux j a (get_line c i)) c
```

▶ Question 9

```
let all_states =
  let s = empty_conf :: [] in
  let rec aux x res =
    if x = 9 then res
    else
      let i,j = x mod 3 , x / 3
      in
        aux (x+1)
          (List.fold_left
            (fun l t ->
              let u , v = set t i j O , set t i j X
                in u :: v :: l)
            res res)
  in aux 0 s
```

▶ Question 10

```
let nb x c =
  List.fold_left
    (fun n a ->
       List.fold_left
 (fun n e -> if e = x then n+1 else n)
 n a) 0 c

exception InvalidState

let owner c =
  if nb O c = nb X c then P1
  else if nb O c = nb X c + 1 then P2
  else raise InvalidState

let winning s p =
  let x = if p = P1 then O else X in
    List.exists
      (fun i ->
 List.for_all
   (fun j -> get s i j = x) [0;1;2] )
      [0;1;2]
    ||
      List.exists
      (fun j ->
 List.for_all
   (fun i -> get s i j = x) [0;1;2] )
      [0;1;2]
    ||
      List.for_all (fun i -> get s i i = x) [0;1;2]
    ||
      List.for_all (fun i -> get s i (2-i) = x) [0;1;2]
```

► Question 11

```
let successors s =
  let aux x c =
    let res = ref [] in
      for i = 0 to 2 do
for j = 0 to 2 do
  if get c i j = V
  then res := set c i j x :: !res
done;
      done;
      !res
  in
    if owner s = P1
    then aux O s
    else aux X s
```

► Question 12

```
let states =
  List.filter
    (fun c -> try (owner c; true) with _ -> false)
    all_states

let win1 , win2 =
  List.filter (fun s -> winning s P1) states ,
  List.filter (fun s -> winning s P2) states

let tictactoe =
  { state = states ;
    owner = owner ;
    trans = successors ;
    init = empty_conf;
    win1 = win1 ;
    win2 = win2 }


let attr1 = attractor tictactoe P1
let has_win_strat = Hashtbl.mem attr1 empty_conf
```

► Question 13

```
let step g p a strat =
  let modif =
    List.fold_left
      ( fun b s ->
  if (g.owner s = p) && (not (mem a s))
    && exists_succ g s (mem a)
  then (
    Hashtbl.add strat s (exists_succ_t g s (mem a));
    add a s; true)
  else b )
      false g.state
  in
    List.fold_left
      ( fun b s ->
  if (g.owner s <> p) && (not (mem a s))
    && forall_succ g s (mem a)
    && g.trans s <> []
  then (add a s; true)
  else b )
      modif g.state

let attractor g p =
  let a = empty_set () in
  let strat = Hashtbl.create 1000 in
    List.iter (add a)
      (match p with P1 -> g.win1 | P2 -> g.win2);
    while (step g p a strat) do () done;
    a, strat
```

► Question 14

```
let a1,s1 = attractor tictactoe P1

let strat1 c = Hashtbl.find s1 c
```