Feuille d’exercices n°13 - Jeux

Notions abordées

- jeu de Chomp, jeu des allumettes généralisé
- calcul des attracteurs
- algorithme MinMax (dont une version avec mémoïsation)
- mesurer le temps en OCaml
- manipuler un dictionnaire en OCaml

Exercice 1 : Attracteurs pour le jeu des allumettes généralisé

On considère le jeu des allumettes généralisé et paramétré par deux entiers $n \in \mathbb{N}^\star$ et $k \in [\![1, n]\!]$:

- le jeu se déroule avec $n \in \mathbb{N}^\star$ allumettes initiales ;
- à son tour, chaque joueur doit prendre entre 1 et k allumettes.

Le jeu présenté en classe est donc le jeu pour les paramètres $n = 13$ et $k = 3$. On représente un état du jeu par un couple de $\{\mathsf{A}, \mathsf{B}\} \times [\![0, n]\!]$, qui donne le joueur dont c’est le tour de jouer et le nombre d’allumettes restantes. Alice commence la partie.

Dans la suite de cet exercice, on note $(\mathcal{A}_p)_{p\in\mathbb{N}}$ la suite des attracteurs d’Alice et $(\mathcal{B}_p)_{p\in\mathbb{N}}$ celle des attracteurs de Bob. On note alors $\mathcal{A} = \bigcup_{p\in\mathbb{N}} \mathcal{A}_p$ et $\mathcal{B} = \bigcup_{p\in\mathbb{N}} \mathcal{B}_p$.

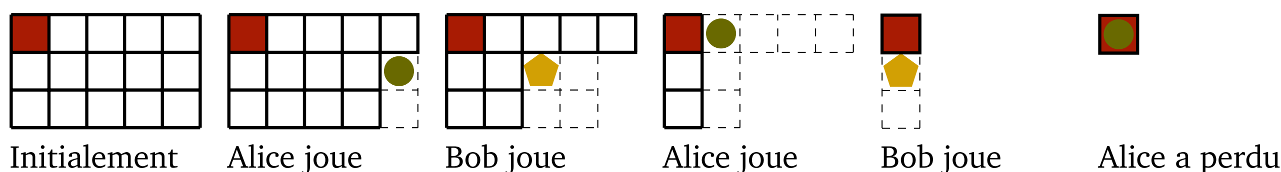
Q. 1 Montrer que le jeu des allumettes généralisé n’admet pas de partie nulle ni de partie infinie.

Q. 2 Démontrer que $\mathcal{A} = \{(\mathsf{A}, i) \mid i \in [\![0, n]\!], i \not\equiv 1[k+1]\} \cup \{(\mathsf{B}, i) \mid i \in [\![0, n]\!], i \equiv 1[k+1]\}$
et que $\mathcal{B} = \{(\mathsf{B}, i) \mid i \in [\![0, n]\!], i \not\equiv 1[k+1]\} \cup \{(\mathsf{A}, i) \mid i \in [\![0, n]\!], i \equiv 1[k+1]\}$.

Q. 3 En déduire qui de Alice et Bob admet une stratégie gagnante selon la valeur de n.
Donner cette stratégie.

Exercice 2 : Chomp

Chomp est un jeu se jouant avec une tablette de chocolat de dimensions $n \times m$ (*i.e.* à n lignes et m colonnes). Chaque joueur, à son tour, doit choisir un carreau non encore mangé de la tablette, et doit manger tous les carrés se trouvant en dessous à droite de celui-ci (au sens large). Le carreau en haut à gauche est pimenté, celui qui doit le manger a perdu. Ci-dessous un exemple de déroulé d’une partie de Chomp sur une tablette 3×5.



Dans cet exercice on étudie les stratégies gagnantes pour le premier joueur.

Q. 1 Quelle est la stratégie gagnante dans le cas d'une tablette de dimensions $1 \times m$ avec $m > 1$?

Q. 2 Proposer une représentation en machine d'un l'état du jeu, par exemple avec un type en C.

Q. 3 À l'aide d'un graphe qui décrit tous les coups possibles, trouver une stratégie gagnante pour une tablette 2×2.

Q. 4 Si la stratégie gagnante pour le cas $2 \times m$ avec m quelconque n'est pas claire, compléter le graphe de la question précédente pour qu'il représente le jeu pour une tablette initiale de taille 2×3 et décrire la stratégie gagnante.

Q. 5 On se propose d'étudier le cas $2 \times m$. Déterminer une stratégie gagnante.

Q. 6 Montrer que quelle que soit la tablette initiale, l'un des deux joueurs possède une stratégie gagnante.

Q. 7 Déterminer une stratégie gagnante si la tablette est de dimensions $n \times n$.

Exercice 3 : Calcul de min-max

On prend comme convention dans cet exercice que les nœuds ronds représentent un état du jeu où c'est à Alice de jouer, tandis que ceux pentagonaux représentent un état où c'est à Bob de jouer. De plus on suppose que les scores sont donnés de sorte qu'Alice cherche à maximiser le score et Bob cherche à le minimiser. Autrement dit un score élevé représente une situation favorable à Alice.

Q. 1 Compléter l'arbre ci-dessous au moyen de l'algorithme du min-max.

On souhaite fournir une implémentation en OCaml de l'algorithme mis en œuvre dans la question précédente. On définit donc le type suivant pour représenter les arbres de min-max.

```
type joueur = Alice | Bob

type mm_tree =
  | Leaf of joueur * int
  | Node of joueur * mm_tree list (* liste non vide *)
```

On remarque que dans de tels arbres les nœuds internes ne contiennent pas de valeurs. En effet seules les feuilles sont annotées avec une valeur, que l'on peut imaginer être celle fournie par une fonction d'heuristique. On souhaite "compléter" l'arbre, autrement dit associer une valeur à chaque nœud interne de l'arbre. On définit donc le type suivant pour représenter les arbres de min-max complétés.

```
type mm_tree_completed =
  | LeafC of joueur * int
  | NodeC of joueur * int * mm_tree_completed list (* liste non vide *)
```

À titre d'exemple le sous-arbre de hauteur 2 le plus à gauche de l'arbre de la question 1 est représenté au moyen de la valeur OCaml ci-dessous.

```
let ex =
  Node(Alice,[
      Node(Bob,[
          Leaf(Alice, 3)]);
      Node(Bob,[
          Leaf(Alice, 1); Leaf(Alice, 2); Leaf(Alice, 9)]);
      Node(Bob,[
          Leaf(Alice, 4)])])
```

Sa version complétée est quant à elle représentée par la valeur suivante.

```
let ex_c =
  NodeC(Alice, 4, [
      NodeC(Bob, 3, [
          LeafC(Alice, 3)]);
      NodeC(Bob, 1, [
          LeafC(Alice, 1); LeafC(Alice, 2); LeafC(Alice, 9)]);
      NodeC(Bob, 4, [
          LeafC(Alice, 4)])])
```

Q. 2 Proposer une fonction `complete_mm : joueur -> mm_tree -> mm_tree_completed` prenant en arguments un joueur j et un arbre de min-max, et qui retourne une version complétée de cet arbre lorsque c'est le joueur j qui cherche à maximiser son score.

Q. 3 Compléter l'arbre ci-dessous au moyen de l'algorithme du min-max **avec élagage** $\alpha - \beta$.

Q. 4 Proposer une fonction `lazy_score_tree : joueur -> mm_tree -> int` prenant en arguments un joueur j et un arbre de min-max et calculant, de manière **paresseuse grâce à l'élagage** $\alpha - \beta$, le score de cet arbre lorsque c'est le joueur j qui cherche à maximiser son score.

Exercice 4 : Calcul de min-max avec mémoïsation

Dans les exercices précédents nous avons appliqué l'algorithme du min-max de manière purement récursive et naïve, de sorte que l'arbre des appels récursifs pouvait présenter de nombreuses occurrences des mêmes appels, autrement dit de sorte que l'algorithme pouvait être amené à refaire plusieurs fois les mêmes calculs. Pour le jeu des allumettes par exemples, il est clair qu'un même état peut être considéré plusieurs fois car il y a plusieurs manières d'aboutir à cet état.

On se propose dans cet exercice de faire une étude statistique de ces re-calculs dans le cas du jeu du morpion. Le jeu du morpion se joue sur une grille 3×3 de cases pouvant être ou bien vides, ou bien marquées par Alice, ou bien marquées par Bob. Initialement toutes les cases de la grille sont vides. Chaque joueur, à son tour, marque une case vide. Le jeu s'arrête lorsque trois cases alignées (*i.e.* sur une même ligne, une même colonne ou une même diagonale) sont marquées par le même joueur, jouer qui a alors gagné la partie.
Dans le fichier `compagnon.ml` on fournit :
- la définition d'un type `joueur` pour représenter les deux joueurs ;
- la définition d'un type `case` pour représenter l'état d'une case ;
- la définition d'un type `morpion` pour représenter les états du jeu (par un tableau de cases et un joueur) ;
- une fonction `est_gagnant : morpion -> joueur` permettant de tester si un état est gagnant pour un joueur ;
- une fonction `next : morpion -> morpion list` retournant la liste des états accessibles en un coup depuis un état donné.

Q. 1 Définir alors un algorithme "naïf" de min-max effectuant les appels récursifs jusqu'aux sommets terminaux. *On remarque que cela revient en fait au calcul des attracteurs. Un état à score strictement positif appartient à l'attracteur du joueur, un état à score strictement négatif appartient à l'attracteur de l'autre joueur, un état de score nul à aucun des deux.*

Q. 2 Enrichir la fonction précédente de manière à compter, pour chaque état du jeu, les appels récursifs sur cet état générés par l'algorithme du min-max. *On pourra utiliser le module* `Hashtbl` *pour définir une table de hachage globale, et incrémenter à chaque appel récursif sur une entrée* e *l'entier associé à l'entrée* e *dans cette table.*

Q. 3 Donner :

- le nombre total d'appels récursifs qui sont des re-calculs ;
- le pire nombre de redondance d'appels récursifs ;
- la proportion des appels récursifs qui sont des re-calculs.

Q. 4 Mettre en place, au moyen du module `Hashtbl`, une mémoïsation des appels de l'algorithme de min-max. Comparer expérimentalement les temps d'exécution des deux fonctions. *On pourra utiliser la fonction* `(Sys.time ())` *dont le fonctionnement est rappelé en fin d'exercice.*

Mesurer le temps en OCaml

La fonction `time` du module `Sys` donne le temps en secondes écoulé lors de l'exécution du programme.

```
Sys.time : unit -> float
```

On obtient alors un temps de calcul par différence entre l'instant de début et l'instant de fin.

```
let rec fibo_bete (n : int) : int =
  match n with
  | 0 -> 0
  | 1 -> 1
  | _ -> (fibo_bete (n-1)) + (fibo_bete (n-2))

let fibo_affiche_tps (n : int) : unit =
  let debut = Sys.time () in
  let _     = fibo_bete n in
  let fin   = Sys.time () in
  print_string (string_of_float (fin-.debut))
```

Dans l'exemple ci-dessus on a défini une fonction qu'on sait être longue à éxecuter sur des grandes valeurs. On a ensuite encapsulé un appel à cette fonction dans une deuxième fonction qui prend soin de noter l'instant avant l'appel, l'instant après l'appel, et qui affiche ensuite la différence entre les deux (on note l'usage de `-.` et non de `-`) soit la durée d'exécution de cet appel.

Créer une table de hachage en OCaml

Le module `Hashtbl` d'OCaml implémente le type abstrait Dictionnaire qui permet la gestion d'ensembles dynamiques d'associations clés-valeurs, dont les clés sont 2 à 2 distinctes.

Les `Hashtbl` permettent ainsi de représenter :

→ des ensembles dynamiques (les éléments sont les clés, on peut leur associer une valeur booléenne indiquant la présence ou non, on peut aussi imaginer associer une valeur (quelconque) aux clés qui sont dans l'ensemble et ne pas en associer à celles qui ne sont pas dans l'ensemble) ;

→ des multi-ensembles (en remplaçant la présence booléenne par un nombre d'occurrences entier) ;

→ ou plus généralement une fonction sur un ensemble fini.

Une table dont les clés sont de type `'a` et dont les valeurs sont de type `'b` est de type `('a, 'b) Hashtbl.t`.

On crée une table de hachage comme suit, en précisant `n` une estimation du nombre de clés que contiendra la table. Cette valeur est donnée seulement en vue d'améliorer les performances de la table, elle ne limite pas le nombre de clés.

```
let tbl = Hashtbl.create n
```

Pour manipuler la table, on dispose des fonctions élémentaires suivantes.

- `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` qui teste la présence d'une clé dans la table.
- `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` qui donne la valeur associée à une clé présente dans la table. Si la clé n'est pas présente dans la table, l'exception `Not_found` est levée.
- `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` qui permet d'ajouter une association clé-valeur à la table. Si la clé était déjà présente, la valeur associée est masquée par la nouvelle valeur.
- `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` agit comme `add` sauf que si la clé était déjà présente, la valeur qui lui était associée est écrasée.

De plus on dispose aussi d'une fonctionnelle `fold` pour itérer sur les associations de la table.

```
Hashtbl.fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) Hashtbl.t -> 'c -> 'c
```

On remarque que l'ordre des arguments dans la fonction d'accumulation diffère de celui pour la fonctionelle `List.fold_left`, en effet ici l'accumulateur est passé en troisième argument.