Correction DS4 : Jeux de soustraction

Xavier Badin de Montjoye

1 Jeux Impartiaux

1. a) Non, car les joueurs ne bougent pas les mêmes pièces. b) Non car les joueurs jouent en simultanées. c) Non, car présence de hasard et informations incomplètes.

2. Figure 1 et Figure 2.

3. On note $[n]$ l'ensemble $\{0;\dots;n\}$. Les jeux Al_n, N_n et W_n, m correspondent aux graphes simplifiés $([n], E_A)$, $([n], E_N)$ et $([n] \times [m], E_W)$ avec

$$E_A = \{r \to s \mid r > s \geq 0 \wedge s \geq r - 3\}$$

$$E_N = \{r \to s \mid r > s \geq 0\}$$

$$E_W = \{(r, r' \to (s, s') \mid (r \geq s \geq 0, r' \geq s' \geq 0) \wedge (r > s \vee r' > s' \vee r - s = r' - s')\}$$

4.

```
let soustraction a1 a2 =
  let k = Array.length a1 in
  let a3 = Array.make k 0 in
  for i =0 to (k-1) do
    a3.(i) <- a1.(i)-a2.(i)
  done;a3
```

5.

```
let positif a =
  let k = Array.length a in
  let rec aux i =
    if i>=k then true
    else if a.(i)<0 then false
    else aux (i+1)
  in aux 0
```

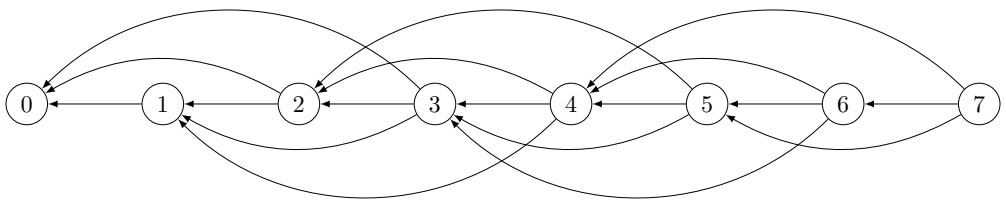

FIGURE 1 – Graphe simplifié de Al_7

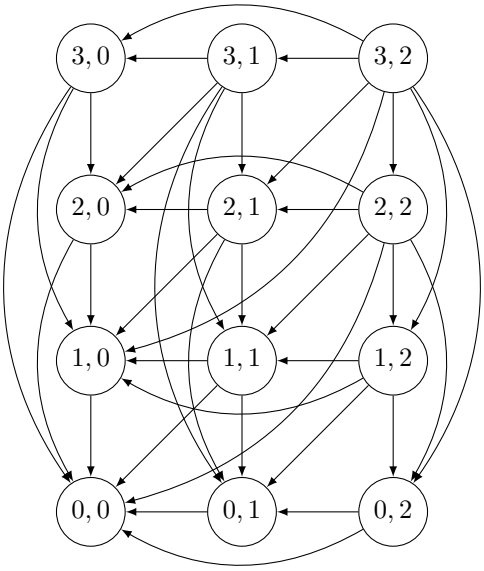

FIGURE 2 – Graphe simplifié de $W_{3,2}$

6. Pour Nim et Wythoff, les ensembles de règles sont infinis.

$$R_N = \mathbb{N}^* \quad R_W = \{(0,n) \mid n > 0\} \cup \{(n,0) \mid n > 0\} \cup \{(n,n) \mid n > 0\}$$

7.

```
let voisins_al a =
  let rec aux i j =
    if i<0||j<0 then []
    else [|i|]::aux (i-1) (j-1)
  in aux (a.(0)-1) 2
```

8.

```
let voisins_nim a =
  let rec aux i =
    if i <0 then []
    else [|i|]::aux (i-1)
  in aux (a.(0)-1)
```

9. On utilise une fonction auxiliaire pour calculer le fait de juste enlever des allumettes à gauche, un pour enlever juste à droite et une pour enlever autant dans les deux piles.

```
let voisins_wyt a =
  let rec gauche i =
    if i<0 then []
    else [|i;a.(1)|]::gauche (i-1)
  in let rec droite i =
       if i < 0 then gauche (a.(0)-1)
       else [|a.(0);i|]::droite (i-1)
```

```
  in let rec double i j =
       if i<0||j<0 then droite (a.(1)-1)
       else [|i;j|]::double (i-1) (j-1)
  in double (a.(0)-1) (a.(1)-1)
```

10. Soit $pos = (x_0, \ldots, x_{k-1})$ on définit $\varphi_{n,k}(pos) = \sum_{i=0}^{k-1} pos_i \cdot (n+1)^{k-i-1}$. Cette fonction est bien bijective. On note que pos est l'écriture en base $n+1$ de $\varphi_{n,k}(pos)$.

11.

```
let arr2int n a =
  let k = Array.length a in
  let rec aux i =
    if i<0 then 0
    else (n+1)*(aux (i-1))+a.(i)
  in aux (k-1)
```

12.

```
let int2arr n k i =
  let a = Array.make k 0 in
  let rec aux i j =
    if j=0 then a.(0)<-i
    else (a.(j)<-(i mod (n+1));aux (i/(n+1)) (j-1))
  in (aux i (k-1);a)
```

13. Pour `arr2int`, on peut considérer l'invariant de récurrence suivant : `aux i` renvoie $\sum_{j=0}^{i} \texttt{a.(j)}(n+1)^{i-j}$. Ainsi, `aux (k-1)` calcul bien $\varphi_{n,k}(pos)$.

Pour `int2arr`, on remarque qu'après j appels récursifs, $\varphi_{n,k}(\texttt{a}) = i \bmod (n+1)^j$. Ainsi, à la fin des k appels récursifs, on a bien $\varphi_{n,k}(\texttt{a}) = \texttt{i}$.

14. Les deux fonctions font appels à une fonction récursive ave un paramètre décrémentant à chaque appel. Elles ont donc une complexité en $O(k)$.

15. On utilise une fonction auxiliaire `puissance` qui permet de calculer les puissances entières. Notons qu'il est possible d'accélérer notre algorithme en utilisant l'exponentiation rapide.

```
let rec puissance a b =
  if b = 0 then 1
  else if b < 0 then 0
  else a*puissance a (b-1)

let initialiser n k x =
  let p = puissance (n+1) k in
  Array.make p x
```

16.

```
let maxArr a =
  let k = Array.length a in
```

```
  let rec aux i m =
    if i>=k then m
    else aux (i+1) (max m a.(i))
  in aux 0 a.(0)
```

17. Supposons (p, i) gagnant pour i. On a alors une stratégie gagnante σ pour le joueur i. Définissons la stratégie τ pour le joueur $3-i$ comme $\tau(q, 3-i) = (q', 3-i)$ avec q' définie comme $\sigma(q, i) = (q', 3-i)$. Supposons maintenant que i dispose d'une réponse à τ qui amène dans un état terminal appartenant à $3-i$. Autrement dit, une réponse à τ entraînant une défaite de $3-i$. Alors, le joueur $3-i$ pourrait jouer cette même stratégie face à σ et atteindre un état terminal de i et faisant perdre le joueur i. C'est une contradiction. De ce fait, $(p, 3-i)$ est bien gagnant pour $3-i$.

18. Il faut ici bien sur comprendre descendant comme descendant direct. Tous les enfants d'une position perdante sont gagnants. En effet, si tel n'était pas le cas, le joueur en position perdante pourrait jouer vers une position perdante, ce qui signifierait que les deux joueurs ont dû jouer depuis une position perdante, ce qui entraînerait par définition la défaite des deux joueurs, ce qui est impossible.

19. De même, chaque position gagnante a au moins un enfant qui est une position perdante.

20. Les $\mathcal{P}-$ positions d'un jeu de Wythoff avec au plus 5 allumettes dans chaque pile, sont : $(0, 0)$, $(1, 2)$, $(2, 1)$, $(5, 3)$, $(3, 5)$.

21. On effectue un parcours en profondeur de notre graphe simplifié en stockant dans un tableau les sommets déjà rencontrés. On leur attribue la valeur 1 pour signifier qu'ils sont gagnants et -1 pour signifier qu'ils sont perdants.

Notons que si on remarque qu'un enfant est perdant, on peut directement conclure que le sommet est gagnant.

```
let gagnant r p =
  let k = Array.length p
  and n = maxArr p in
  let vu = initialiser n k 0 in
  let rec aux a = let id = arr2int n a in
    if vu.(id) <> 0 then vu.(id)
    else let rec descendant l = match l with
        |[] -> -1
        |t::q -> if aux t = -1 then 1
            else descendant q
      in let v = descendant (r a)
      in (vu.(id)<-v;v)
  in (aux p > 0)
```

22. Une stratégie gagnante consiste, dans chaque position gagnante, à se rendre dans une $\mathcal{P}$-position. Le jeu étant acyclique, la partie se termine. De plus, on a montré précédemment que d'une $\mathcal{P}-$position, il n'est pas possible de se rendre dans une $\mathcal{P}-$position, et que d'une position non perdante, on peut toujours se rendre dans une $\mathcal{P}-$position. Les positions terminales étant des $\mathcal{P}-$positions, il en résulte qu'il s'agit bien d'une stratégie gagnante.

23. On se contente de regarder tous les voisins de p et de tester s'il s'agit de $\mathcal{P}-$positions. Notons qu'il aurait été plus efficace, par un facteur d, degré du graphe, de refaire un parcours.

```
let strategie r p =
  (* Si la case considérée est terminale on la renvoie *)
  let rec aux l der = match l with
    |[] -> der
```

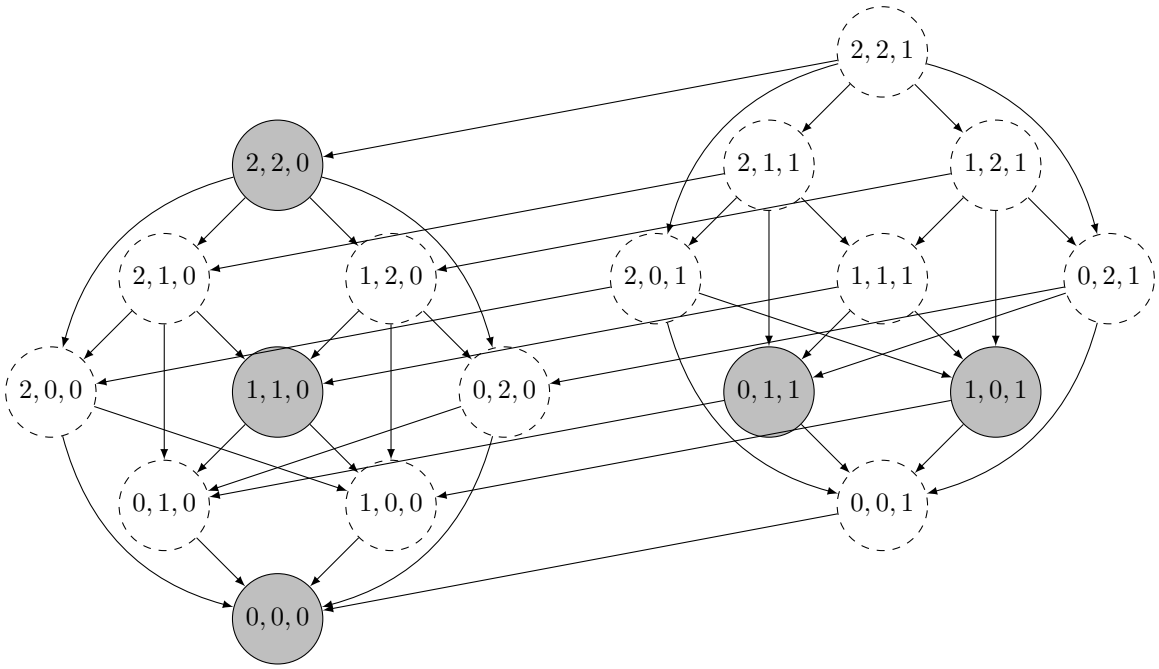

FIGURE 3 – Graphe simplifié de $N_2 + N_2 + N_1$. On a représenté en gris les $\mathcal{P}-$positions

```
    |t::q -> if not (gagnant r t) then t
        else aux q t
  in aux (r p) p
```

2 Somme de Jeux de Nim

24. Figure 3

25. Les positions terminales de $J + K$ sont les (p, q) avec p et q positions terminales de respectivement J et K.

26.

```
let concatener a1 a2 =
  let k1 = Array.length a1 and k2 = Array.length a2 in
  let a3 = Array.make (k1+k2) 0 in
  for i=0 to (k1-1) do
    a3.(i)<-a1.(i)
  done;
  for i=0 to (k2-1) do
    a3.(i+k1)<-a2.(i)
  done;
  a3
```

27.

```
let extraire a i j =
  let res = Array.make (j-i+1) 0 in
  for k = i to j do
```

```
    res.(k-i)<-a.(k)
  done;
  res
```

28. Pour cette fonction, on utilise `List.map` qui nous permet d'appliquer une fonction sur tous les arguments d'une liste. De plus, on introduit la fonction `concatener_g` qui permet d'inverser l'ordre des paramètres de `concatener`. Notons qu'on renvoie bien une fonction.

```
let concatener_g a1 a2 =
  concatener a2 a1

let somme r1 r2 k1 k2 =
  let r3 a =
    let a1 = extraire a 0 (k1-1) and
    a2 = extraire a k1 (k1+k2-1) in
    (List.map (concatener a1) (r2 a2))@(List.map (concatener_g a2) (r1 a1))
  in r3
```

29. Deux cas se présentent à nous.
— Si p est une $\mathcal{P}-$position et pas q, ou l'inverse par symétrie, alors (p, q) est une position gagnante.
— p et q sont des $\mathcal{P}-$positions, et dans ce cas (p, q) est une $\mathcal{P}-$positions.
Pour le prouver, il suffit de remarquer que dans le premier cas, le joueur peut toujours se ramener à une position où les deux termes sont des $\mathcal{P}-$positions et dans le second, le premier joueur se retrouve nécessairement après un coup dans le premier cas. De plus, les positions terminales sont un couple de positions terminales, donc un couple de $\mathcal{P}-$positions.

30. On ne peut rien dire si p et q sont des positions gagnantes. Pour s'en convaincre, il suffit de regarder les jeux $N_1 + N_1$ et $N_1 + N_2$. Le premier est une $\mathcal{P}-$position tandis que le second est une position gagnante.

31. Les $\mathcal{P}-$positions de $N_3 + N_4$ sont $(0, 0), (1, 1), (2, 2)$ et $(3, 3)$.

32. Les $\mathcal{P}-$positions de $N_n + N_m$ sont les $E_= = \{(k, k) \mid n, m \geq k \geq 0\}$. Pour le prouver, il nous suffit de noter que $E_=$ possède les propriétés inductives des $\mathcal{P}-$positions. On a $(0, 0) \in E_=$, tout élément non dans $E_=$ a un enfant dans cet ensemble et tout élément de $E_=$ n'a pas de voisins sortant dans $E_=$.

33. On a $14 = \overline{1110}^2$ et $5 = \overline{101}^2$. Donc

$$14 \oplus 5 = \overline{1011}^2 = 11$$

34.

```
let rec xor x y =
  if x*y = 0 then x+y
  else ((x+y) mod 2)+2* (xor (x/2) (y/2))
```

35.On va noter P la propriété que le vecteur $(k_1, \ldots, k_n)$ vérifie $k_1 \oplus \ldots \oplus k_n = 0$. Soit $(k_1, \ldots, k_n)$ une position ne satisfaisant pas P. On note x l'entier obtenu à partir du XOR bit à bit des k_i. Soit $m \in \mathbb{N}$ tel que $2^m \leq x < 2^{m+1}$. On numérote les bits à partir de 0 à avec le bits 0 de poids faible. Le bit m est ici égale à 1. Par définition du XOR, il existe i tel que k_i ait son m-ième bit égale à 1. Soit $b_0, \ldots, b_l$ valant 0 ou 1 tel que :

$$k_i = \sum_{j=0}^{l} b_j 2^j$$

On pose alors :

$$\tilde{k_i} = \sum_{j=m+1}^{l} b_j 2^j$$

On pose maintenant $\tilde{x} = k_1 \oplus \ldots \oplus k_{i-1} \oplus \tilde{k_i} \oplus k_{i+1} \oplus \ldots \oplus k_n$. On note qu'à partir du $m+1$-ième bit, les valeurs n'ont pas changé, donc elle reste nulle pour $\tilde{x}$. De plus, le m-ième bit de $\tilde{x}$ est maintenant nulle. Il en résulte $\tilde{x} < 2^m$. De ce fait $k'_i = \tilde{k_i} + \tilde{x} = \tilde{k_i} \oplus \tilde{x} < k_i$ et on a alors $(k_1, \ldots, k_{i-1}, k'_i, k_{i+1}, \ldots, k_n)$ satisfait $\mathcal{P}$.

36. L'ensemble des éléments possédant la propriété P satisfait les propriétés inductives des $\mathcal{P}-$positions et donc sont les $\mathcal{P}-$positions.

37. On utilise une fonction auxiliaire qui calcul le XOR de tous les éléments d'un tableau.

```
let xor_total a =
  let k = Array.length a in
  let rec aux i =
    if i < 0 then 0
    else xor a.(i) (aux (i-1))
  in aux (k-1)

let gagnant_somme a = (xor_total a <> 0)
```

38. Pour simplifier notre algorithme on présente plusieurs fonctions auxiliaires. La première `puiss2` calcul la valuation 2-adique d'un entier. La seconde, `is1ini` prend en entrée x et i et regarde s'il y a un 1 en i-ème position de la représentation binaire de x. Enfin, `prendre_allumette` renvoie juste une position pouvant être joué dans le cas où nous nous trouvons dans une position perdante.

```
let rec puiss2 x =
  if x < 2 then 0
  else 1 + puiss2 (x/2)

let is1ini x i =
  let rec aux y j =
    if j = 0 then (y mod 2 = 1)
    else aux (y/2) (j-1)
  in aux x i

let prendre_allumette a =
  let rec aux i =
    if i<0 then (0,0)
    else if a.(i)>0 then (i,1)
    else aux (i-1)
  in aux ((Array.length a)-1)

let coup_optimal a =
  let x = xor_total a and k = Array.length a in
  let m = puiss2 x in
  let rec aux i =
    if i<0 then prendre_allumette a
    else if is1ini a.(i) m then (i, a.(i)- xor a.(i) x)
    else aux (i-1)
  in aux (k-1)
```

39. Soit k le nombre de colonnes et, soit n le nombre maximal d'éléments dans une même colonne. La fonction `xor_total` a une complexité en $O(k \log(n))$. Le calcul de la valuation 2-adique du résultat du xor

coûte également $\log(n)$, puisque `x`$< 2n$. Vérifier si un entier à un 1 dans sa représentation binaire en position i coûte $\log(n)$. Enfin, la fonction `aux` est appelée au plus k fois pour une complexité totale en $O(k\log(n))$. Ainsi, la complexité total de notre algorithme est en $O(k\log(n))$.

40.

```
let destination_optimal a =
  let (i,j)=coup_optimal a in
  let b= Array.copy a in (b.(i)<-a.(i)-j;b)
```

3 Somme de jeux de soustraction quelconques

41. Le mex de $\{3;0;2;7;1;5\}$ est le plus petit entier non dans l'ensemble, donc 4.

42. Le mex d'un ensemble fini est borné par la taille de cet ensemble. En effet, si $\text{MEX}(E) > |E|$, il en résulte que tous les entiers de $\{0;\dots;|E|\}$ sont dans $|E|$. Or $Card(\{0;\dots;|E|\}) = |E|+1$, on a donc une contradiction.

43.

```
let mex l =
  let n = List.length l in
  let a = Array.make (n+1) 0 in
  let rec premier i =
    if a.(i)=0 then i
    else premier (i+1) in
  let rec aux l = match l with
    |[] -> premier 0
    |t::q -> (if t <= n then a.(t)<-1;aux q)
  in aux l
```

44. On effectue un parcours de notre graphe en stockant dans un tableau les positions déjà rencontrées. On utilise `List.map` pour calculer la valeur des enfants.

```
let grundy r p =
  let k = Array.length p
  and n = maxArr p in
  let vu = initialiser n k (-1) in
  let rec aux a = let id = arr2int n a in
    if vu.(id) >= 0 then vu.(id)
    else let v = mex (List.map aux (r a))
      in (vu.(id)<-v;v)
  in aux p
```

45. Prouvons par induction que les $\mathcal{P}-$positions ont valeur de Grundy 0. Tout d'abord, notons que les états terminaux n'ont pas d'enfants. Or le mex d'un ensemble vide vaut 0. Ainsi, les terminaux ont bien valeur de Grundy 0.

Soit p une position ayant pour enfant $p_1,\dots,p_n$. Supposons que $g(p) > 0$. On a alors $0 \in \{g(p_1),\dots,g(p_n)\}$. Ainsi, p a un enfant p_i ayant 0 comme valeur de Grundy. Par hypothèse d'induction, p_i est une $\mathcal{P}-$position, donc p n'est pas une $\mathcal{P}-$position.

Si $g(p) = 0$, pour tout i, $g(p_i) \neq 0$ et p_i est une position gagnante. Donc p est une $\mathcal{P}$-position.

46. Erreur d'énoncé, (p,q) est une position **perdante** si $g(p) = g(q)$. On le réalise vite si p et q sont terminaux.

On procède par induction. Si p et q sont des états terminaux, alors $g(p) = g(q) = 0$ et (p, q) est une position perdante.

Sinon, supposons qu'au moins p n'est pas terminal dans J. Si $g(p) = g(q)$, tout coup emmène dans une position (p', q) ou (p, q') avec p' enfant de p et q' enfant de q. On suppose qu'on est dans le premier cas. Or $g(p') \neq g(p)$, donc $g(p') \neq g(q)$ et (p', q) est une position gagnante. Ainsi, tous les enfants de (p, q) sont des positions gagnantes, donc (p, q) est une position perdante.

Si $g(p) \neq g(q)$, on peut supposer que $g(p) > g(q)$. Ainsi, par définition du mex, p a un enfant p' tel que $g(p') = g(q)$. Par hypothèse d'induction, (p', q) est une $\mathcal{P}-$position, donc (p, q) a une position perdante comme enfant et est donc une position gagnante.

47. On va montrer par induction sur les positions de $J + K$ que $g((p, q)) = g(p) \oplus g(q)$.

Si (p, q) est un état terminal, alors p et q sont des états terminaux et donc des états perdants. Donc $g((p, q) = 0 = 0 \oplus 0 = g(p) \oplus g(q)$.

Sinon, pour toute position accessible (p', q') à partir de (p, q), on a par hypothèse d'induction, $g(p', q') = g(p') \oplus g(q')$ or, on sait que soit $g(p') = g(p)$ et $g(q) \neq g(q')$, soit $g(q') = g(q)$ et $g(p) \neq g(p')$. Donc aucun enfant de (p, q) n'a comme valeur de Grundy, $g(p) \oplus g(q)$. Il nous reste à montrer que pour tout $x < g(p) \oplus g(q)$, il existe un enfant de (p, q) de valeur de Grundy x.

Pour ce faire, considérons le premier bit i, par ordre décroissant de poids, tel que x et $g = g(p) \oplus g(q)$ diffère. Comme $x < g$, ce bit i est nul pour x et égale à 1 pour g. Par définition du XOR, soit $g(p)$, soit $g(q)$ a le même bit égal à 1. Par symétrie, on peut supposer qu'il s'agit de p. Considérons maintenant $g(q) \oplus x$. Son $i-$ème bit est donc égale à 0, et les bits de poids plus grands ont la même valeur que $g(p)$. Il en résulte que $g(q) \oplus x < g(p)$. Ainsi, par définition du mex, il existe p' enfant de p tel que $g(p') = x \oplus g(q)$. Ainsi, la position (p', q) est accessible à partir de (p, q) et a par induction pour valeur de Grundy $g(p') \oplus g(q) = x$.

De ce fait, on a bien $g((p, q)) = g(p) \oplus g(q)$.

48. Si on connaît la valeur de Grundy de deux jeux, on connaît la valeur de Grundy de leur somme et on peut donc aisément calculer les $\mathcal{P}-$positions. De plus, pour tout jeu J dans une position p, et pour tout jeu K, la valeur du jeu $J + K$ est la même que la valeur du jeu $N_{g(p)} + K$.