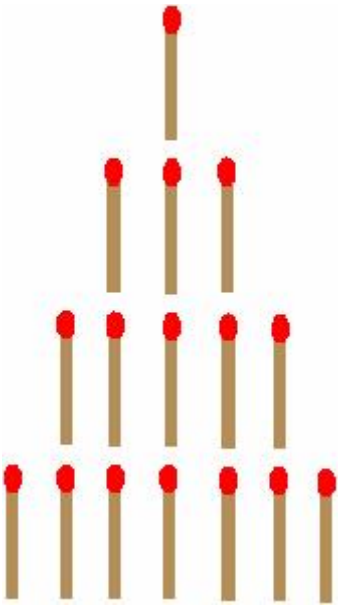
Martin Creuze
A22
Alexandre Labouré
Groupe 2

TP2 IA01

Jeu de Nim



20 novembre 2022

Sommaire

Introduction

Le but de ce TP est de trouver différentes manières de faire gagner une IA au jeu de Nim. Pour rappel, le jeu de Nim est un jeu de pure stratégie. Il se joue avec un tas d'allumettes, chaque joueur peut retirer au maximum 3 allumettes par tour. Le joueur qui prend la dernière allumette a perdu. Le jeu de Nim est ainsi équivalent à se déplacer d'un sommet à un autre dans un arbre : les sommets représentent les diverses positions du jeu et les arêtes les transitions d'une position à une autre. Il montrera qu'il existe une stratégie optimale.

Ce TP est divisé en 2 parties, chacune ayant un objectif différent.

Le but de la première partie est d'étudier la fonction proposée et de déterminer quel type de recherche elle effectue pour la résolution du jeu de Nim. Nous allons également déterminer quelle recherche (en profondeur ou en largeur) semble la plus adaptée pour ce jeu et voir s'il est possible d'améliorer la fonction proposée.

Pour la deuxième partie, le but est de programmer une IA capable d'améliorer sa stratégie au fur et à mesure de jeux successifs. Pour ceci, dès qu'un coup mènera à la victoire de l'IA, il sera ajouté à une base de données contenant les coups que l'IA peut jouer à partir d'un certain état. Ce système est propagé à tous les coups qui ont mené à la victoire. Ainsi, au prochain lancement du jeu, les probabilités que l'IA tombe sur un coup qui mène à la victoire seront augmentées. Ceci permettra de faire émerger des stratégies gagnantes.

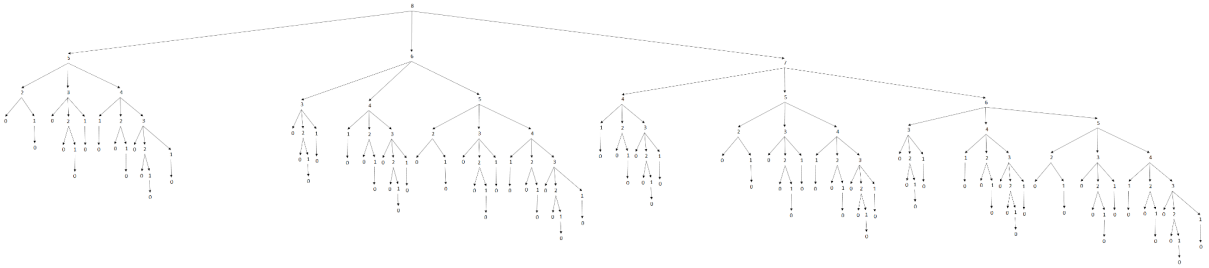
Résolution 1

Question 1

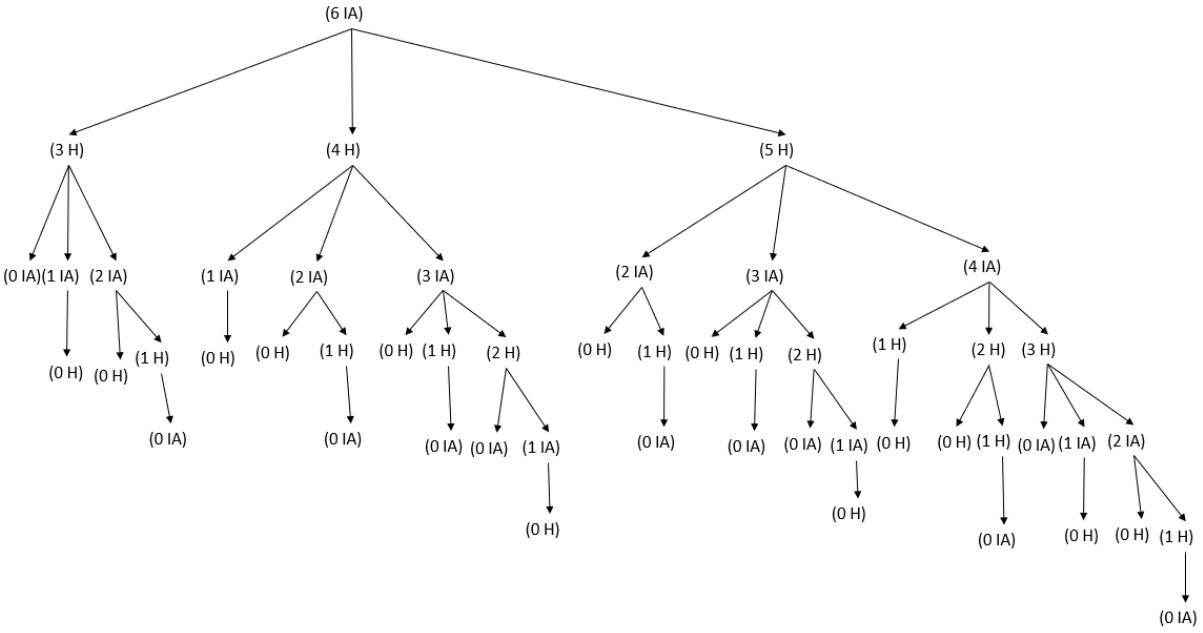
L'IA commence, il y a 16 allumettes soit l'état initial est (16 IA)
La partie se termine lorsque le dernier joueur prend la dernière allumette. On a donc comme états finaux (0 IA) où il reste 0 allumette et c'est au tour de l'IA de jouer (donc victoire de l'IA), et (0 Humain) où il reste 0 allumette et c'est au tour de l'Humain de jouer (donc victoire de l'Humain).
Ainsi, pour chaque fonction d'exploration, on partira de l'état (16 IA) jusqu'à tomber sur un des état finaux, état qui arrêtera la fonction.

Question 2

L'arbre de recherche à partir de l'état où il ne reste plus que 8 allumettes est le suivant :



Pour pouvoir mieux visualiser, voici un arbre où il ne reste que 6 allumettes.



Comme évoqué dans l'introduction, on a bien un jeu de Nim est équivalent à se déplacer d'un sommet à un autre dans un arbre : les sommets représentent les diverses positions du jeu et les arêtes les transitions d'une position à une autre, et avec pour racine l'état initial et comme feuilles les états finaux.

Question 3

Fonction successeurs :
La fonction successeurs prend en paramètre le nombre d'allumettes restants et renvoie une liste des actions possibles, c'est-à-dire retirer 3, 2 ou 1 allumette(s) si possible.

```
(defun successeurs (allumettes actions)
  (cdr (assoc allumettes actions)))

> (successeurs 15 actions)
(3 2 1)
> (successeurs 2 actions)
(2 1)
```

Fonction explore :
Si c'est à l'humain de jouer et qu'il ne reste plus d'allumettes, c'est que l'IA a tiré la dernière allumette. L'IA a donc perdu et la fonction retourne nil.
Dans le cas inverse, si c'est l'humain qui a tiré la dernière allumette, l'IA a gagné et la fonction retourne true.
Cela correspond bien à nos états finaux (0 IA) et (0 Humain)

On parcourt l'arbre tant qu'il reste des allumettes.
A chaque appel, la fonction initialise une variable "sol" à nil et une liste "coup" avec les actions possibles.
Pour chacune des actions possibles, on rappelle la fonction avec le nombre d'allumettes restantes après l'action, en changeant le joueur, et on affecte sa valeur de retour (t ou nil) à sol.
On parcourt ainsi l'arbre en profondeur.
Si l'IA a perdu (sol==nil), on revient à l'état précédent et on essaye avec l'action suivante.
Si aucune action ne convient, on remonte encore au niveau au-dessus, et ainsi de suite…
On parcourt ainsi l'arbre jusqu'à obtenir une victoire de l'IA.

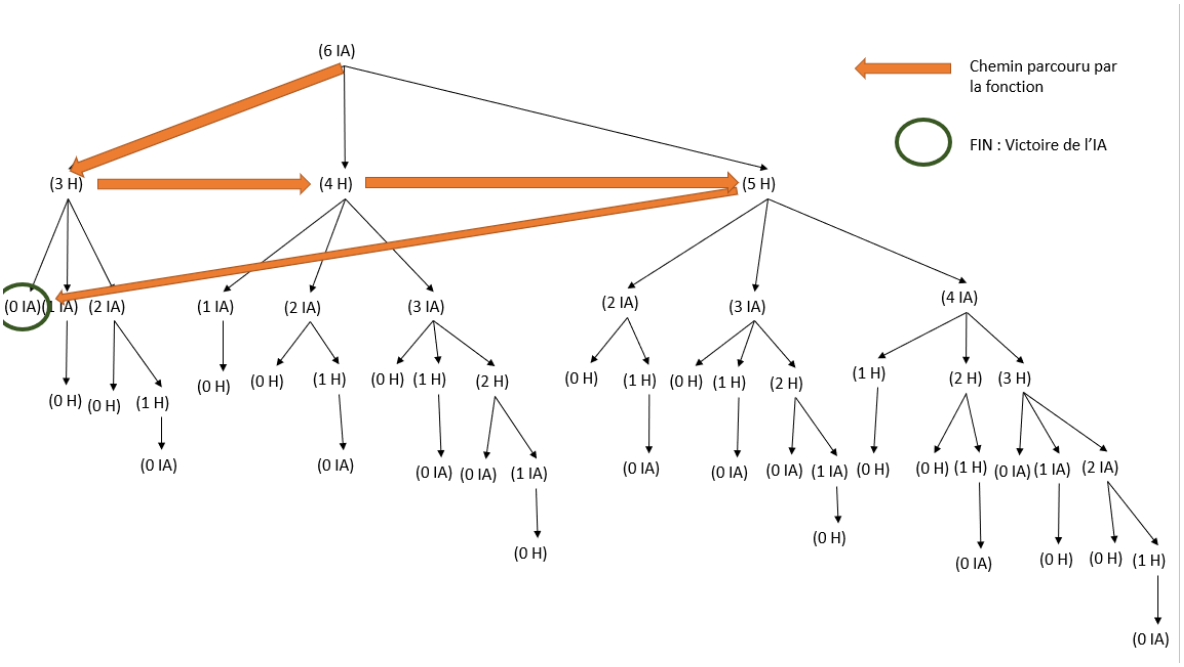
La fonction retourne le nombre d'allumettes que le joueur passé en argument doit tirer au premier coup pour que l'IA ait une chance de gagner.

On crée ensuite une variable nbCoupsAJouer initialisé à nil. On lui affecte les valeurs de retour de la fonction explore pour 16, 8 et 3 allumettes où c'est l'IA qui commence le jeu à chaque fois.

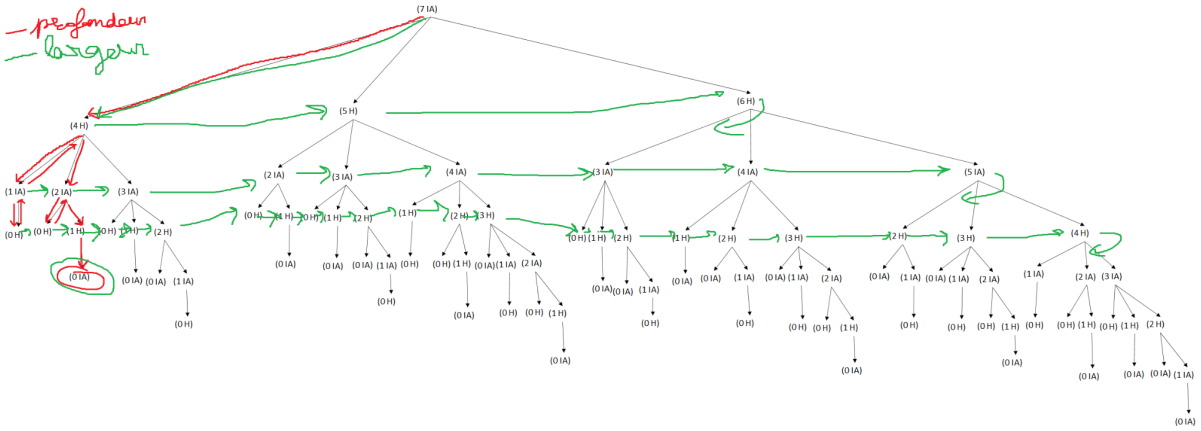
C’est une recherche en profondeur. Si on ne trouve pas la solution dans la première branche la plus à gauche de l’arbre, on remonte au niveau au-dessus et on cherche la solution dans la deuxième branche directement à droite de la première.

Question 4

On suppose qu’on garde la même finalité de l’algorithme : parcourir l’arbre jusqu’à trouver un nœud où l’IA gagne.
Par définition, l’exploration en largeur et en profondeur ont la même complexité en espace et en temps dans le pire des cas. Mais compte-tenu de l’exercice, nous n’arrivons jamais au pire cas, cas où le nœud de sortie est la feuille de la dernière branche : on ne va donc pas explorer tous les nœuds de l’arbre. On s'arrête dès qu’il y a une victoire de l’IA .
La question est donc : quelle exploration mène le plus rapidement à cette victoire?
La recherche est plus rapide par une exploration en profondeur. En effet, pour un arbre partant de l’état (6 IA), on remarque que l’exploration en largeur prend plus de temps.



On peut généraliser ce résultat en partant de n’importe quel nombre d’allumettes de départ.
Compte tenu de l’exercice, l’exploration en profondeur sera toujours plus rapide et l'exploration en largeur sera exponentiellement plus longue. L’arbre pour 7 allumettes parle de lui-même :



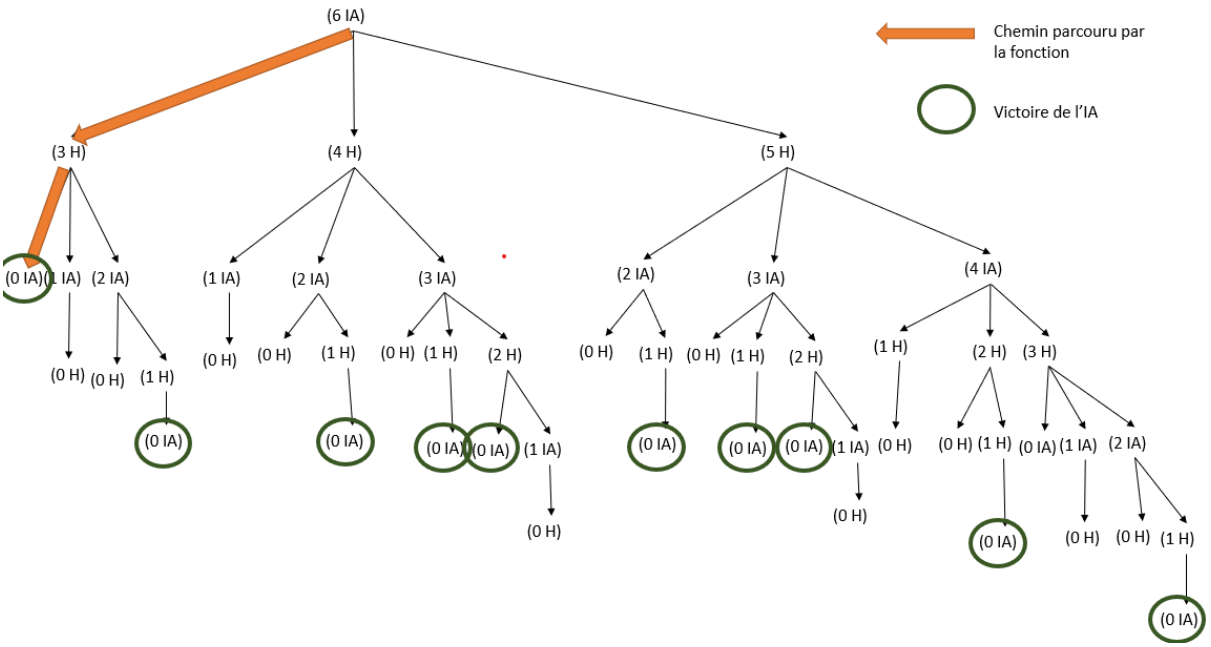
Question 5

La fonction s'arrête dès qu'il y a une possibilité de victoire, ce n'est pas la meilleure stratégie.

Prenons l'exemple de 6 allumettes :

```
> (explore 6 actions 'IA 0)
Joueur IA joue 3 allumettes - il reste 3 allumette(s)
   Joueur HUMAIN joue 3 allumettes - il reste 0 allumette(s)
      sol = 3

 sol = 3
3
```



Pour 6 allumettes, on remarque bien avec l'arbre qu'il aurait fallu retirer 1 allumette au premier coup pour obtenir le plus de chances de victoire alors que la fonction retourne 3. De même, lorsqu'il reste 7 allumettes, la fonction devrait retourner 2 au lieu de 3.

Dans la fonction explore proposée, on choisit l'action dès qu'il y a possibilité de victoire et pas forcément l'action où il y a le plus de chances de victoire.
Afin d'améliorer la fonction, il faudrait donc parcourir tout l'arbre et voir pour quelle première action il y a le plus de chances de victoire pour l'IA. Il faudrait créer une stratégie de meilleur choix.

Résolution 2

Nous allons maintenant réaliser une IA capable d'améliorer sa stratégie face à un joueur humain. L'IA commence le jeu.

Lorsque c'est son tour, l'IA choisit une action aléatoire parmi celles possibles. Cela correspond au code suivant :

```
(defun Randomsuccesseurs (actions)
  (let ((r (random (length actions))))
      ;;(format t "~&~2t Res du random ~s~&" r)
      (nth r actions)))
```

```
> (Randomsuccesseurs (successeurs 10 actions))
2
> (Randomsuccesseurs '(3 2 1))
3
```

Question 2

On demande à l'utilisateur quel coup il souhaite jouer et on s'assure que ce coup est possible (sinon la fonction redemande un coup à jouer si ce dernier n'est pas possible). La fonction retourne le coup joué.

```
(defun JeuJoueur (allumettes actions)
;; initialisation d'une variable coup à 0 et d'une variable possibilites
avec la liste des coups possibles (allumettes que l'on peut retirer en
fonction du nombre d'allumettes restantes)
  (let ( (coup 0) (possibilites (successeurs allumettes actions)))
      ;; boucle demandant à l'utilisateur de saisir une valeur tant que
celle ci n'est pas comprise dans la liste possibilites
      (loop do
          (progn
               ;; affichage pour l'utilisateur
               (format t "Saisir nombre d' allumettes à tirer : ")
               ;; coup prend la valeur saisie par l'utilisateur
               (setq coup (read))
          )
      while (not (member coup possibilites)))
   ;; retour du coup joué
   coup)
 )
```

```
> (JeuJoueur 2 actions)
Saisir nombre d allumettes a tirer : 3

Saisir nombre d allumettes a tirer : 1
1
```

On a donc maintenant un moyen d'obtenir l'action réalisée pour l'IA et celle réalisée pour l'humain à chaque tour.

Question 3

Avant de considérer le renforcement, on écrit le code d'exploration. Cette fonction renvoie échec (ou nil) si l'IA perd ou la liste actions si l'IA gagne.

```
(defun explore-renf (allumettes actions)
  ;; on initialise une variable joueur, vaut IA au début
  ;; une variable i pour permettre un affichage listé
  ;; et une variable correspond au nombre d'allumettes retirées par un
joueur
  (let ((joueur 'IA) (i 1) (retire 0))
      ;; tant qu'il y a des allumettes ...
      (while allumettes
        ;; ... on affiche le nombre d'allumettes
        (format t "~%~s.   Il y a ~s allumettes, " i allumettes)
        (setq i (+ i 1))
        ;; si c'est à l'IA de jouer :
        (if (eq joueur 'IA)
            ;; on récupère aléatoirement le nombre d'allumettes retirées
et on l'affiche. Puis on soustrait ce nombre au nombre total
d'allumettes
            (progn
            (setq retire (Randomsuccesseurs (successeurs allumettes
actions)))
            (format t "l'IA tire ~s allumettes" retire)
            (setq allumettes (- allumettes retire))
            ;; si l'IA a pris la derniere, c est perdu, sinon c'est à l'
humain de jouer
            (if (eq allumettes 0) (return nil))
            (setq joueur 'humain))

      ;; sinon, c'est à l'humain de jouer
      (progn
            (format t "~%")
            ;; il retire le nombre d'allumettes désirés, nombre qu'on
soustrait au nombre total d'allumettes
```

```
            (setq retire (JeuJoueur allumettes actions))
            (format t "l'humain tire ~s allumettes ~%" retire)
            (setq allumettes (- allumettes retire))
            ;; s'il a pris la dernière, on retourne les actions
            (if (eq allumettes 0) (return actions)
            ;; sinon c'est à l'IA de jouer
            (setq joueur 'IA))))
      )))
```

Les deux joueurs retirent chacun leur tour des allumettes jusqu'à ce qu'il n'en reste plus.

Test pour une victoire de l'IA :

```
> (explore-renf 16 actions)

1.   Il y a 16 allumettes,l'IA tire 1 allumettes
2.   Il y a 15 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes

3.   Il y a 12 allumettes,l'IA tire 3 allumettes
4.   Il y a 9 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes

5.   Il y a 6 allumettes,l'IA tire 1 allumettes
6.   Il y a 5 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes

7.   Il y a 2 allumettes,l'IA tire 1 allumettes
8.   Il y a 1 allumettes,

Saisir nombre d allumettes a tirer : 1
l'humain tire 1 allumettes
((16 3 2 1 1) (15 3 2 1) (14 3 2 1) (13 3 2 1) (12 3 2 1 3) (11 3 2 1)
(10 3 2 1) (9 3 2 1) (8 3 2 1) (7 3 2 1 2) ...)
```

Test pour une défaite de l'IA :

```
> (explore-renf 16 actions)

1.   Il y a 16 allumettes,l'IA tire 1 allumettes
```

```
2.   Il y a 15 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes

3.   Il y a 12 allumettes,l'IA tire 2 allumettes
4.   Il y a 10 allumettes,

Saisir nombre d allumettes a tirer : 2
l'humain tire 2 allumettes

5.   Il y a 8 allumettes,l'IA tire 2 allumettes
6.   Il y a 6 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes

7.   Il y a 3 allumettes,l'IA tire 3 allumettes
NIL
```

Question 4

On introduit dans le code de la fonction explore précédente l'étape de renforcement du dernier coup gagnant dans le cas où l'IA gagne.

```
(defun explore-renf (allumettes actions)
  ;; on initialise une variable joueur, vaut IA au début
  ;; une variable i pour permettre un affichage listé
  ;; et deux variables correspondants au nombre d'allumettes retirées
par un joueur ou par l'IA
  (let ((joueur 'IA) (i 1) (retire_IA 0) (retire_h 0))
      ;; tant qu'il y a des allumettes ...
      (while allumettes
      ;; ... on affiche le nombre d'allumettes
      (format t "~%~s.   Il y a ~s allumettes," i allumettes)
      (setq i (+ i 1))
      ;; si c'est à l'IA de jouer :
      (if (eq joueur 'IA)
            ;; on récupère aléatoirement le nombre d'allumettes retirées
et on l'affiche. Puis on soustrait ce nombre au nombre total
d'allumettes
            (progn
            (setq retire_IA (Randomsuccesseurs (successeurs allumettes
actions)))
```

```
            (format t "l'IA tire ~s allumettes" retire_IA)
            (setq allumettes (- allumettes retire_IA))
            ;; si l'IA a pris la dernière, c'est perdu, sinon c'est à l'
humain de jouer
            (if (eq allumettes 0) (return nil))
            (setq joueur 'humain))

      ;; sinon, c'est à l'humain de jouer
      (progn
            (format t "~%")
            ;; il retire le nombre d'allumettes désiré, nombre qu'on
soustrait au nombre total d'allumettes
            (setq retire_h (JeuJoueur allumettes actions))
            (format t "l'humain tire ~s allumettes ~%" retire_h)
            (setq allumettes (- allumettes retire_h))
            ;; s'il a pris la dernière, on retourne les actions
            (if (eq allumettes 0)
            (progn
                  ;; ajoute l' action réalisé au dernier coup de l' IA
dans la liste d'actions pour (retire_IA + retire_h) allumettes soit le
nombre d'allumettes avant que l'IA ne joue
                  (nconc (assoc (+ retire_IA retire_h) actions) (list
retire_IA))
                  (return actions)
            )
            ;; sinon c'est à l'IA de jouer
            (setq joueur 'IA))))
      )))
```

Il a donc seulement fallu ajouter une étape à la fonction de la question 3. Cette étape permet d'ajouter dans la liste d'actions principale l'action réalisée au dernier coup de l'IA pour le nombre d'allumettes avant qu'elle ne joue.

```
> (explore-renf 16 actions)
1.   Il y a 16 allumettes,l'IA tire 1 allumettes
2.   Il y a 15 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes

3.   Il y a 12 allumettes,l'IA tire 3 allumettes
4.   Il y a 9 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes
```

```
5.   Il y a 6 allumettes,l'IA tire 3 allumettes
6.   Il y a 3 allumettes,

Saisir nombre d allumettes a tirer : 3
l'humain tire 3 allumettes
((16 3 2 1) (15 3 2 1) (14 3 2 1) (13 3 2 1) (12 3 2 1) (11 3 2 1) (10 3
2 1) (9 3 2 1) (8 3 2 1) (7 3 2 1) ...)

> (assoc 6 actions)
(6 3 2 1 3)
```

Question 5

La fonction de renforcement correspond donc à ce qui a été ajouté entre la question 3 et la question 4, soit :

```
(defun renforcement (nb_allumettes coup_gagnant actions)
  (nconc (assoc nb_allumettes actions) (list coup_gagnant))
  actions)
```

```
> (renforcement 16 3 actions)
((16 3 2 1 3) (15 3 2 1) (14 3 2 1) (13 3 2 1) (12 3 2 1) (11 3 2 1) (10
3 2 1) (9 3 2 1) (8 3 2 1) (7 3 2 1) ...)
> (renforcement 15 2 actions)
((16 3 2 1 3) (15 3 2 1 2) (14 3 2 1) (13 3 2 1) (12 3 2 1) (11 3 2 1)
(10 3 2 1) (9 3 2 1) (8 3 2 1) (7 3 2 1) ...)
```

Question 6

On allons transformer la fonction en une fonction récursive.

```
(defun explore-renf-rec (allumettes actions joueur i)
  (cond
   ;; si l'IA a pris la dernière allumettes > perdu
   ((and (eq joueur 'humain) (eq allumettes 0)) nil)
   ;; si l'humain a pris la dernière allumettes > gagné
   ((and (eq joueur 'IA) (eq allumettes 0)) t)
   ;; sinon : il reste des allumettes
   (t (progn
```

```
      ;; on initialise deux variables correspondants au nombre
d'allumettes retirées par un joueur ou par l'IA (une seule aurait pu
suffir)
      ;; et une variable solution
      (let ((retire_IA 0) (retire_h 0) (sol nil))
            ;; si c'est à l'IA de jouer
            (if (eq joueur 'IA) (progn
                        ;; on récupère aléatoirement le nombre
d'allumettes retirées et on l'affiche
                        (setq retire_IA (Randomsuccesseurs (successeurs
allumettes actions)))
                        (format t "~%~V@tIl y a ~s allumettes, l'IA tire
~s allumettes" i allumettes retire_IA)
                        ;; on rappelle la fonction avec le nb
d'allumettes restants pour l'humain
                        (setq sol (explore-renf-rec (- allumettes
retire_IA) actions 'humain (+ i 3)))
                              ;; si l'humain a pris la dernière
allumette, on renforce chaque action réalisé par l'IA
                              (if sol (progn
                                          (format t "~%~V@t On ajoute l
action ~s quand il reste ~s allumettes" i retire_IA allumettes)
                                          (renforcement allumettes
retire_IA actions))))

            ;; sinon c'est à l'humain de jouer
            (progn
            (format t "~%~V@tIl y a ~s allumettes~%" i allumettes)
            ;; il retire le nombre d'allumettes désirées
            (setq retire_h (JeuJoueur allumettes actions))
            (format t "~%~V@t l'humain tire ~s allumettes" i retire_h)
            ;; on rappelle la fonction avec le nb d'allumettes restants
pour l'IA
            (setq sol (explore-renf-rec (- allumettes retire_h) actions
'IA (+ i 3)))))))))))
```

A chaque appel, si le nombre d'allumettes ne correspond pas à un état final, on fait jouer le joueur mis en argument. On renvoie la fonction avec l'autre joueur et le nombre d'allumettes restantes, qu'on applique à une variable solution. On effectue ceci jusqu'à tomber sur 0 allumette. Si le dernier appel renvoie nil, c'est que l'IA a perdu. On ne fait donc rien. Sinon, pour chaque appel qu'on a effectué, on renforce la stratégie.

Test pour une victoire de l'IA :

```
> (explore-renf-rec 16 actions 'IA 0)
Il y a 16 allumettes, l'IA tire 2 allumettes
```

```
   Il y a 14 allumettes

Saisir nombre d allumettes a tirer : 3

      l'humain tire 3 allumettes
        Il y a 11 allumettes, l'IA tire 2 allumettes
          Il y a 9 allumettes

Saisir nombre d allumettes a tirer : 3

            l'humain tire 3 allumettes
              Il y a 6 allumettes, l'IA tire 2 allumettes
                Il y a 4 allumettes

Saisir nombre d allumettes a tirer : 1

                  l'humain tire 1 allumettes
                    Il y a 3 allumettes, l'IA tire 2 allumettes
                      Il y a 1 allumettes

Saisir nombre d allumettes a tirer : 1

                        l'humain tire 1 allumettes
                  On ajoute l action 2 quand il reste 3 allumettes
              On ajoute l action 2 quand il reste 6 allumettes
        On ajoute l action 2 quand il reste 11 allumettes
 On ajoute l action 2 quand il reste 16 allumettes
((16 3 2 1 3 2) (15 3 2 1 2) (14 3 2 1) (13 3 2 1) (12 3 2 1) (11 3 2 1
2) (10 3 2 1) (9 3 2 1) (8 3 2 1) (7 3 2 1) ...)
```

Test pour une défaite de l'IA :

```
> (explore-renf-rec 16 actions 'IA 0)
Il y a 16 allumettes, l'IA tire 3 allumettes
   Il y a 13 allumettes

Saisir nombre d' allumettes à tirer : 3

      l'humain tire 3 allumettes
      Il y a 10 allumettes, l'IA tire 2 allumettes
      Il y a 8 allumettes

Saisir nombre d' allumettes à tirer : 3

            l'humain tire 3 allumettes
```

```
            Il y a 5 allumettes, l'IA tire 1 allumettes
            Il y a 4 allumettes
Saisir nombre d' allumettes à tirer : 3

                 l'humain tire 3 allumettes
                 Il y a 1 allumettes, l'IA tire 1 allumettes
NIL
```

Conclusion

Avec ce TP nous avons tout d'abord consolidé nos connaissances du langage lisp.
Nous avons aussi amélioré notre savoir dans le domaine des recherches dans un espace d'état. Nous avons pû comprendre les avantages et inconvénients des recherches en profondeur et en largeur.
Nous avons également codé une IA qui apprend de ses expériences, ce qui fut très intéressant.

Les deux résolutions que nous avons utilisées sont très différentes et ne nous apportent pas le même résultat.
La première effectue une recherche en profondeur dans un espace d'état mais ne nous a pas semblé très cohérente étant donné qu'elle renvoie le nombre d'allumettes à retirer dès qu'il y a possibilité de victoire. Face à un humain débrouillard, elle risque de ne jamais gagner.
La deuxième résolution est une approche de modification de la base des actions possibles au fur et à mesure des expériences de jeu de l'IA. Cette résolution est très intéressante car elle permet de faire émerger la meilleure stratégie à utiliser, en fonction du nombre d'allumettes qu'il reste, afin que l'IA maximise ses chances de gagner. Une fois qu'on a joué une infinité de fois, les stratégies gagnantes convergent vers un état (5 Humain). A cet état, peu importe l'action choisie par le joueur, l'IA gagnera. Si l'humain retire 1, l'IA retirera 3 et il devra prendre la dernière allumette. Si l'humain retire 2, l'IA retirera 2 etc… Donc à 6 allumettes, l'IA retire 1, à 7 elle retire 2 et au-dessus elle en retire 3.