Rapport TP2 :
Jeu de Nim

Tobias SAVARY

Camille BAUVAIS

16 novembre 2022

Année universitaire 2022/2023

Table des matières

1 Introduction

1.1 Contexte

Durant ce TP, nous allons résoudre le jeu de Nim de 2 façons différentes. Le jeu de Nim est un jeu de stratégie qui se joue à 2 joueurs. Au départ, les joueurs se trouvent face à 16 allumettes. Chaque joueur peut retirer 1, 2 ou 3 allumettes à tour de rôle. Celui qui prend la dernière allumette a perdu.

1.2 Représentation du problème

Le jeu de Nim est représenté sous forme d'une liste Lisp dans laquelle figure toutes les actions possibles lors du jeu. On propose donc d'implémenter une liste "actions" composée de 16 sous-listes. Chaque sous-liste représente les nombres de retrait possibles pour un nombre d'allumettes en jeu donné. Une sous-liste est de la forme suivante :

```
(Nombre_allumettes_en_jeu RetraitPossible1 RetraitPossible2(optionnel)
RetraitPossible3(optionnel))
```

Les champs RetraitPossible2 et RetraitPossible3 peuvent ne pas être précisés (notamment pour les cas ou il y a 1 ou 2 allumettes en jeu).

On obtient donc la représentation suivante :

```
(setq actions '((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) (6 3 2 1) (5 3 2 1) (4 3 2 1) (3 3 2 1) (2 2 1) (1 1)))
```

On a bien pour chaque état de 16 à 3, 3 actions possibles : retirer 1, 2 ou 3 allumettes. Pour l'état 2, la possibilité de ne retirer que 1 ou 2 allumettes et enfin pour l'état 1 de ne retirer qu'une seule allumette.

1.3 Explication des résolutions

1.3.1 Résolution 1 :

La première résolution consiste à étudier un parcours dans un espace d'états. Pour cela, on dispose d'un état initial et d'états finaux ainsi que d'un algorithme d'exploration. Cet algorithme réalise un parcours en profondeur sur l'ensemble des états possibles.

1.3.2 Résolution 2 :

La seconde résolution, utilise une IA sans stratégie du meilleur choix. Cette IA commencera le jeu et sera capable d'améliorer sa stratégie face à un joueur humain.

Lorsque c'est son tour, cette IA choisira au hasard parmi les possibilités un nombre d'allumettes à retirer. Une fois ce choix réalisé, les allumettes seront retirées et le jeu passera au joueur suivant.

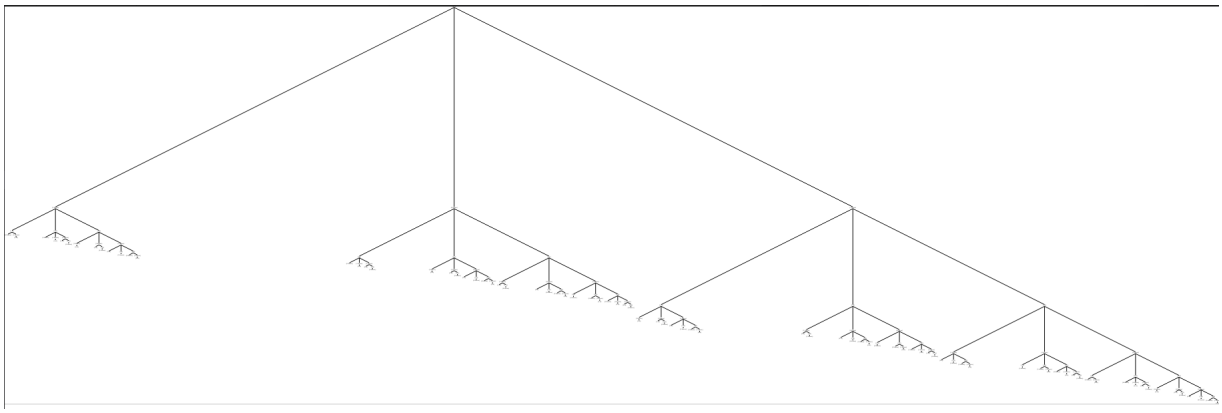
A la fin de la partie, si l'IA gagne, elle voudra renforcer sa stratégie gagnante. L'idée est d'ajouter le coup gagnant à la position qui a rendu possible sa victoire. Ce renforcement sera propagé aux positions qui ont mené à cette victoire. Au prochain tour de jeu, les probabilités de tomber sur un coup qui mène à la victoire sont augmentées. Des stratégies gagnantes devraient émerger.

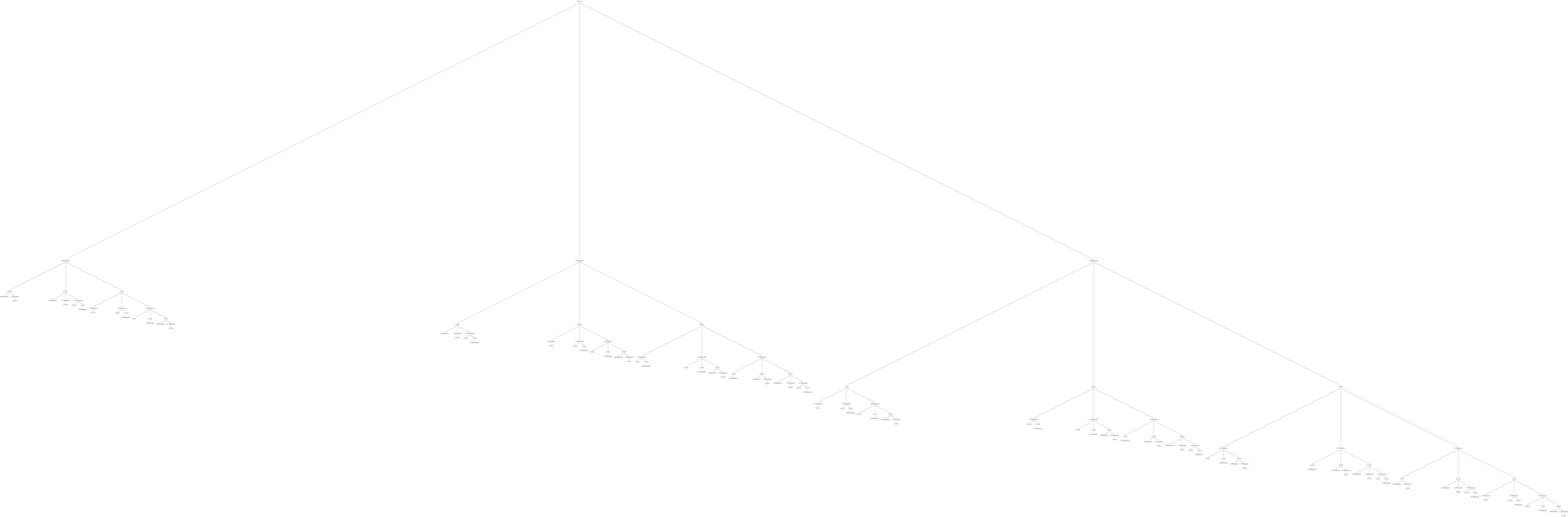
2 Résolution 1 :

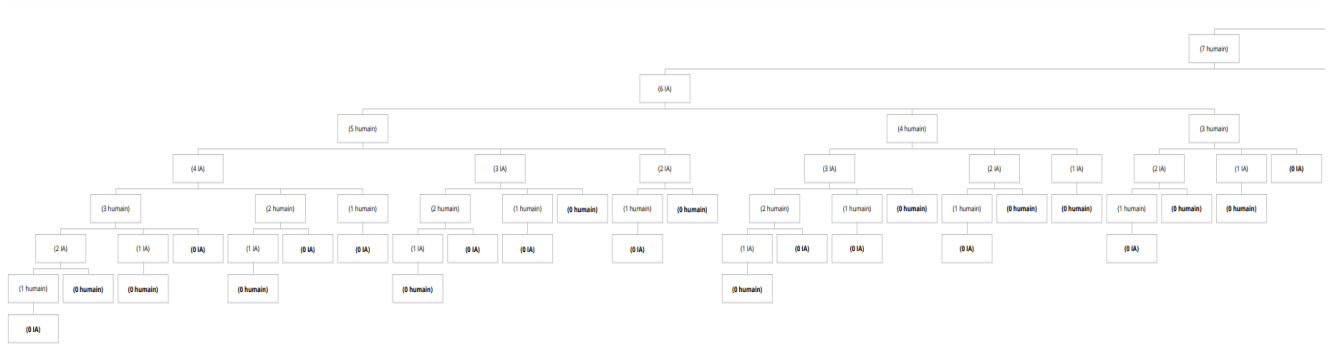
La première résolution consiste à étudier un parcours dans un espace d'états. Dans cette résolution, chaque état est représenté formellement par deux variables qui sont respectivement : le nombre d'allumettes disponibles et le joueur qui a la main.

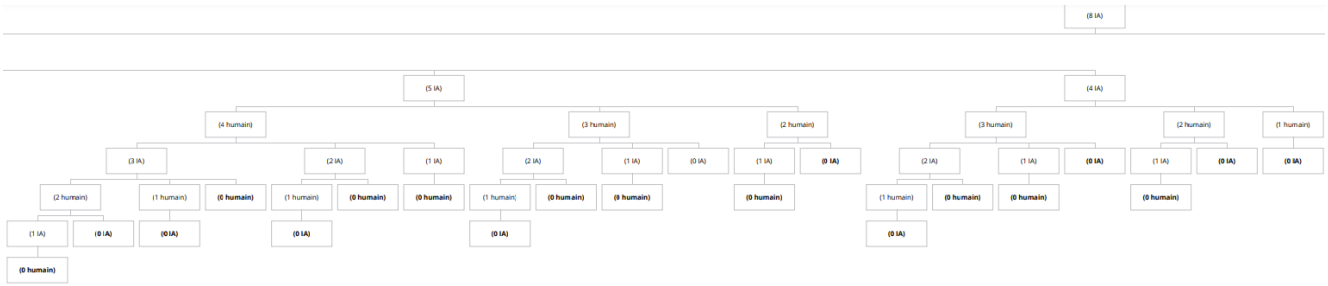
1. Sachant que l'IA démarre la partie, l'état initial est **(16 IA)** car au début de la partie il y a 16 allumettes sur la table et c'est l'IA qui démarre. Les états finaux correspondent aux états **(0 IA)**, l'IA gagne, et **(0 humain)**, l'humain gagne.
2. Arbre de recherche à partir de l'état initial :

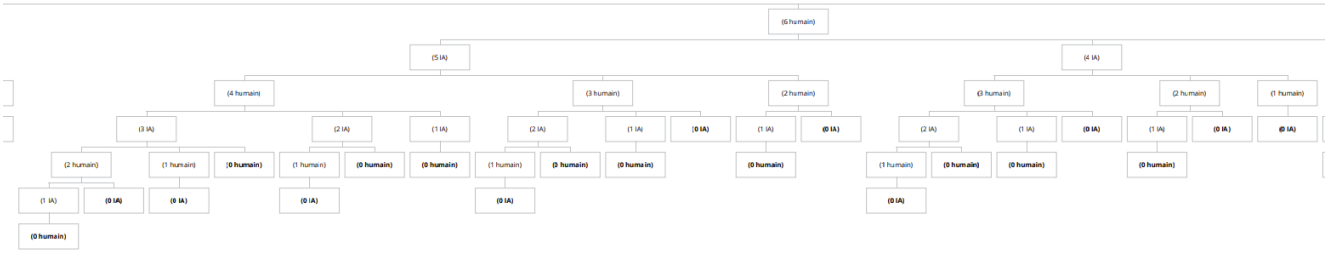
Voici l'arbre demandé. Afin de faciliter la lecture de celui-ci, nous avons inclu l'arbre original dans la page suivante sur lequel il est possible de zoomer ainsi qu'une seconde implémentation qui sera peut-être plus lisible.

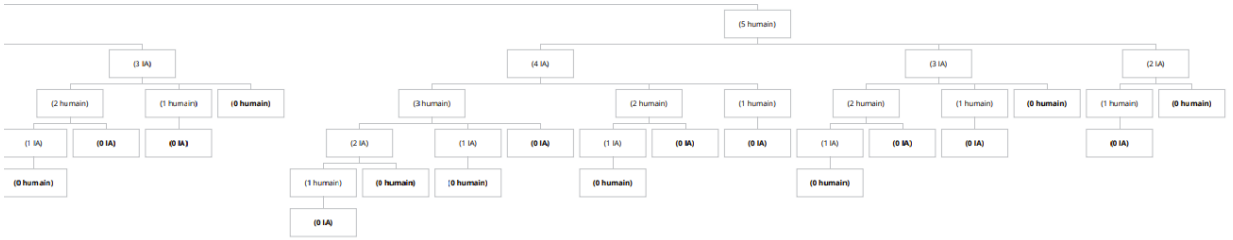





(7 humain)
(6 IA)
(5 humain)
(4 humain)
(3 humain)
(4 IA)
(3 IA)
(2 IA)
(3 IA)
(2 IA)
(1 IA)
(2 IA)
(1 IA)
(0 IA)
(3 humain)
(2 humain)
(1 humain)
(2 humain)
(1 humain)
(0 humain)
(1 humain)
(0 humain)
(2 humain)
(1 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(2 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 humain)
(0 humain)
(0 humain)
(0 humain)
(0 humain)
(0 humain)
(0 humain)
(0 humain)
(0 humain)
(0 IA)


(8 IA)
(5 IA)
(4 IA)
(4 humain)
(3 humain)
(2 humain)
(3 humain)
(2 humain)
(1 humain)
(3 IA)
(2 IA)
(1 IA)
(2 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(2 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(0 IA)
(2 humain)
(1 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(0 humain)
(1 IA)
(0 IA)
(0 IA)
(0 IA)
(0 IA)
(0 IA)
(0 IA)
(0 humain)


(6 humain)
(5 IA)
(4 IA)
(4 humain)
(3 humain)
(2 humain)
(3 humain)
(2 humain)
(1 humain)
(3 IA)
(2 IA)
(1 IA)
(2 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(2 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(0 IA)
(2 humain)
(1 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(0 humain)
(1 IA)
(0 IA)
(0 IA)
(0 IA)
(0 IA)
(0 IA)
(0 humain)


(5 humain)
(3 IA)
(4 IA)
(3 IA)
(2 IA)
(2 humain)
(1 humain)
(3 humain)
(2 humain)
(1 humain)
(2 humain)
(1 humain)
(0 humain)
(1 humain)
(0 humain)
(1 IA)
(0 IA)
(0 IA)
(2 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(1 IA)
(0 IA)
(0 IA)
(1 IA)
(0 IA)
(0 humain)
(1 humain)
(0 humain)
(0 humain)
(0 humain)
(0 humain)
(0 humain)
(0 IA)

3. Explication du code :

Algorithm 1 Successeurs

```
;; fonction qui donne tous les successeurs d'un état :
(defun successeurs (allumettes actions)
  (cdr (assoc allumettes actions)))
```

Algorithm 2 Explore

```
;; EXPLORE
(defun explore (allumettes actions joueur i)
  (cond
   ((and (eq joueur 'humain) (eq allumettes 0)) nil)
   ; Si le joueur est un humain et qu'il n'y a plus d'allumettes
   ; sur la table, l'IA a perdu la partie.
   ((and (eq joueur 'IA) (eq allumettes 0)) t)
   ; Si le joueur est l'IA et qu'il n'y a plus d'allumettes sur la table, l'IA a gagné la partie.
   ; Les deux conditions précédentes représentent les conditions d'arrêt de la récursivité.
    (t (progn
         (let ((sol nil) (coups (successeurs allumettes actions)))
         ; Lorsque nous ne sommes pas dans les conditions d'arrêt décrites précédemment, on
         ; initialise des variables locales sol et coups. Coups est une liste qui représente
         ; l'ensemble des successeurs possibles de l'état courant et sol est une liste
         ; initialement vide. sol permettra de stocker le chemin vers la solution si celle-ci
         ; est trouvée lors du parcours.
           (while (and coups (not sol))
           ; On boucle tant qu'il reste des successeurs et que sol est nul.
             (progn
               (format t "~%~V@tJoueur ~s joue ~s allumettes - il reste ~s
               allumette(s) " i joueur (car coups) (- allumettes (car coups)))
               ; On affiche le joueur courant, le nombre d'allumettes jouées et
               ; le nombre d'allumettes restantes après le tirage.
                (setq sol (explore (- allumettes (car coups)) actions
                (if (eq joueur 'IA) 'humain 'IA) (+ i 3)))
                ; On rappelle la fonction explore avec le nombre d'allumettes
                ; moins le nombre d'allumettes tirées, les actions possibles,
                ; le joueur suivant et i + 3 afin d'indenter l'affichage. Le résultat
                ; de l'appel de la fonction explore est stocké dans sol.
                (if sol
                    (setq sol (car coups)))
                ; Suite aux appels récursifs de la fonction explore, Si une
                ; solution a été trouvée, sol prend la valeur du premier élément de coup
               (format t "~%~V@t sol = ~s~%" i sol)
               ; On affiche la solution trouvée
               (pop coups)
               ;Suppression du premier élément de coups, permet de passer à l'élément
               ; suivant de la liste de successeurs (incrémentation du while)
               ))
          sol)))))    ; On retourne sol
```

```
(defvar nbCoupsAJouer nil)
(setq nbCoupsAJouer (explore 16 actions 'IA 0))
(setq nbCoupsAJouer (explore 8 actions 'IA 0))
(setq nbCoupsAJouer (explore 3 actions 'IA 0))
```

La fonction regarde d'abord si on est dans un état final :

- Si nous sommes dans un état final et que le joueur est l'IA alors on renvoie T
- Si nous sommes dans un état final et que le joueur est l'humain alors on renvoie NIL.

Ensuite, on initialise la variable sol, qui indique si la solution a été trouvée ou non. Cette variable est à nil tant qu'aucune solution n'a été trouvée. De même, la variable coups est initialisée et elle prend tous les successeurs de l'état en question, c'est à dire, renvoie le nombre de coups possibles pour un nombre d'allumettes donné.
Tant qu'il reste des coups et que la solution n'a pas été trouvée, on rappelle la fonction explore avec le nombre d'allumettes auquel on retire le nombre de coups, les actions, et le joueur qui devient humain si le joueur courant était IA et vice-versa.
Si la solution a été trouvée, on met sol au nombre de coups utilisé.
On passe au nombre de coups suivant.
On renvoie la solution.
Cet algorithme effectue donc une recherche en profondeur. En effet, un parcours en largeur aurait exploré toutes les possibilités du niveau n+1 avant de passer aux états du niveau n+2.

En résumé, cette fonction explore permet de faire jouer chacun leur tour l'IA et l'Humain grâce à un parcours en profondeur.

4. Les deux solutions permettent de parvenir au résultat et de parcourir toutes les solutions.
   Pour effectuer un parcours en largeur il faudra tout de même penser à modifier la condition d'arrêt. Le parcous en profondeur a pour avantage de remonter facilement tous les chemins en les concaténant. Mais le parcours en largeur est notamment intéressant pour les graphes importants car il permet d'être plus rapide sur ce type de graphe, ce qui est le cas ici.
5. Il serait possible d'optimiser le parcours en utilisant une statégie bien connue du jeu de Nim. En effet, au jeu de Nim il est possible de toujours laisser un nombre d'allumettes multiples de 4 (4 8 12 16) sur la table afin de gagner plus souvent. Néanmoins, comme l'IA commence à jouer, elle n'est pas sûre de gagner à tous les coups car cette technique permet de gagner à tous les coups uniquement pour le 2eme joueur. Si le joueur en face utilise aussi cette stratégie, il sera donc impossible pour l'IA de gagner la partie.

3 Résolution 2 :

Dans cette seconde résolution, nous allons programmer une IA sans stratégie du meilleur choix. Cette IA commencera le jeu et sera capable d'améliorer sa stratégie face à un joueur humain.

3.1 Principe de l'IA :

Lorsque c'est son tour, cette IA choisira, au hasard parmi les possibilités, un nombre d'allumettes à retirer. Une fois ce choix réalisé, les allumettes seront retirées et le jeu passera au joueur suivant.

A la fin de la partie, si l'IA gagne, elle voudra renforcer sa stratégie gagnante. L'idée est d'ajouter le coup gagnant à la position qui a rendu possible la victoire. Ce renforcement sera propagé aux positions qui ont mené à cette victoire. Au prochain jeu, les probabilités de tomber sur un coup qui mène à la victoire sont augmentées. Des stratégies gagnantes devraient émerger.

2. Fonction de lecture d'un coup que veut jouer l'utilisateur (humain) :

Spécification : JeuJoueur (NbAllumettes, listeActions)
Entree :
- NbAllumettes correspond au nombre d'allumettes sur la table.
- listeActions correspond à la liste des actions possibles.

Sortie : Entier lu au clavier compatible avec la liste d'actions.
Objectif : Cette fonction permet de lire au clavier un choix de l'utilisateur. Elle vérifie si l'utilisateur rentre bien une valeur comprise dans le spectre de valeur souhaité. Il faut donc que la valeur entrée par l'utilisateur appartienne à une valeur possible dans la liste actions.

Deux solutions ont été implémentées.

Algorithm 3 JeuJoueur

```
(defun JeuJoueur (NbAllumettes listeActions)
    (let ((lecture 5)
        (possible (assoc NbAllumettes listeActions)))
        (while (not (member lecture (cdr possible)))
            (format t "Rentrer le coup à jouer (il reste ~s allumettes): ~&" NbAllumettes)
            (setq lecture (read))
        )
        lecture
    )
)
```

Algorithm 4 JeuJoueur

```
(defun JeuJoueur (allumettes actions)
  (let ((choix_joueur nil)
        (coups nil))
    (while (not choix_joueur)
      (format t "Rentrer le coup à jouer (il reste ~s allumettes): ~&" allumettes)
      (setq coups (read))
      (setq choix_joueur (member coups (successeurs allumettes actions)))
      )
    coups)
  )
```

Exemple d'execution :

```
> (JeuJoueur 5 actions)
> Rentrer le coup à jouer (il reste 5 allumettes) :
> 2
> 2

> (JeuJoueur 8 actions)
> Rentrer le coup à jouer (il reste 8 allumettes) :
> 5
> Rentrer le coup à jouer (il reste 8 allumettes) :
> 1
> 1

> (JeuJoueur 2 actions)
> Rentrer le coup à jouer (il reste 2 allumettes) :
> 3
> Rentrer le coup à jouer (il reste 2 allumettes):
> 2
> 2
```

Nous avons trouvé deux implémentations de la fonction JeuJoueur. Ces deux fonctions sont fonctionnelles. Néanmoins, nous avons préféré garder dans la suite du TP la seconde version puisque celle-ci utilise la fonction *successeurs* que nous avons défini précédemment.

3. Algorithme de l'exploration renforcée :

Spécification : explore-renf1 (nb_allumettes, actions)
Entree :

— nb_allumettes correspond au nombre d'allumettes sur la table.

— actions correspond à la liste des actions possibles

Sortie : Renvoie la liste actions si l'IA gagne, nil sinon.
Objectif : Cette fonction effectue l'exploration en profondeur des possibilités du jeu de Nim.
Elle permet de faire jouer l'IA qui choisit un nombre d'allumettes à retirer aléatoirement et le joueur qui lui peut entrer au clavier le nombre d'allumettes qu'il souhaite retirer.

Algorithm 5 explore-renf1

```
explore-renf1 (nb_allumettes actions)
    IA_coups <- Nil
    humain_coups <- Nil
    Si nb_allumettes == 0 alors :
        Retourner actions
    Sinon si nb_allumettes == 1 alors :
        Retourner nil
    Sinon :
        IA_coups <- Randomsuccesseurs (successeurs (nb_allumettes, actions))
        Affichage ("L'IA joue IA_coups")
        nb_allumettes <- nb_allumettes - IA_coups
        Si nb_allumettes > 0 alors :
            humain_coups <- JeuJoueur (nb_allumettes, actions)
            nb_allumettes <- nb_allumettes - humain_coups
            explore-renf1 (nb_allumettes, actions)
        Sinon :
            explore-renf1 (1, actions)
```

Algorithm 6 explore-renf1

```
(defun explore-renf1 (nb_allumettes actions)
  (let ((IA_coups) (humain_coups))
    (cond
     ((= nb_allumettes 0) actions) ;; IA gagne
     ((= nb_allumettes 1) nil) ;; IA perd
     (t (progn;; IA qui joue
            (setq IA_coups (Randomsuccesseurs (successeurs nb_allumettes actions)))
        (format t "L'IA joue ~s ~&" IA_coups)
            (setq nb_allumettes (- nb_allumettes IA_coups))
        ;; humain qui joue
        (if (> nb_allumettes 0)
            (progn
            (setq humain_coups (JeuJoueur nb_allumettes actions))
              (setq nb_allumettes (- nb_allumettes humain_coups))
            (explore-renf1 nb_allumettes actions)
              )
          (explore-renf1 1 actions))
            )
        ))
    ))
```

Exemple d'execution :

```
> (explore-renf1 16 actions)
L'IA joue 2
Rentrer le coup à jouer (il reste 14 allumettes):
2
L'IA joue 3
Rentrer le coup à jouer (il reste 9 allumettes):
3
L'IA joue 2
Rentrer le coup à jouer (il reste 4 allumettes):
1
L'IA joue 1
Rentrer le coup à jouer (il reste 2 allumettes):
1
NIL
```

```
> (explore-renf1 16 actions)
L'IA joue 2
Rentrer le coup à jouer (il reste 14 allumettes):
3
L'IA joue 2
Rentrer le coup à jouer (il reste 9 allumettes):
3
L'IA joue 3
Rentrer le coup à jouer (il reste 3 allumettes):
3
((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) ...)
```

4. Amélioration de l'algorithme précédent afin de le renforcer avec des stratégies déjà rencontrées comme décrite dans l'étape 9 :

Spécification : explore-renfDernierCoup (nb_allumettes, actions)
Entree :

— nb_allumettes correspond au nombre d'allumettes sur la table.
— actions correspond à la liste des actions possibles.

Sortie : la liste actions si l'IA gagne, nil sinon.
Objectif : Cette fonction effectue l'exploration en profondeur des possibilités du jeu de Nim.
Elle permet de faire jouer l'IA qui choisit un nombre d'allumettes à retirer aléatoirement et le joueur qui lui peut entrer au clavier le nombre d'allumettes qu'il souhaite retirer.
Si l'IA gagne, son dernier coup joué est ajouté à la sous-liste correspondante de actions grace à la fonction *renforcement.*

Algorithm 7 explore-renfDernierCoup

```
explore-renfDernierCoup (nb_allumettes, actions)
    IA_coups <- Nil
    humain_coups <- Nil
    Si nb_allumettes == 0 alors :
        Retourner actions
    Sinon si nb_allumettes == 1 alors :
        Retourner nil
    Sinon :
        IA_coups <- Randomsuccesseurs (successeurs(nb_allumettes, actions))
        Affichage ("L'IA joue IA_coups")
        dernier_coup <- Liste(nb_allumettes, IA_coups)
        nb_allumettes <- nb_allumettes - IA_coups
        Si nb_allumettes > 0 alors :
            humain_coups <- JeuJoueur (nb_allumettes, actions)
            nb_allumettes <- nb_allumettes - humain_coups
            Si nb_allumettes == 0 alors:
                Effectuer le renforcement de dernier_coup
            explore-renfDernierCoup (nb_allumettes, actions)
        Sinon :
            explore-renfDernierCoup (1, actions)
```

Algorithm 8 explore-renfDernierCoup

```
(defun explore-renfDernierCoup (nb_allumettes actions)
  (let ((IA_coups ) (humain_coups))
    (cond
     ((= nb_allumettes 0) actions) ;; IA gagne
     ((= nb_allumettes 1) nil) ;; IA perd
     (t (progn;; IA qui joue
        (setq IA_coups (Randomsuccesseurs (successeurs nb_allumettes actions)))
        (format t "L'IA joue ~s ~&" IA_coups)
        (setq dernier_coup (list nb_allumettes IA_coups))
        (setq nb_allumettes_IA nb_allumettes)
        (setq nb_allumettes (- nb_allumettes IA_coups))
        ;; humain qui joue
        (if (> nb_allumettes 0)
            (progn
            (setq humain_coups (JeuJoueur nb_allumettes actions))
              (setq nb_allumettes (- nb_allumettes humain_coups))
              (when (= nb_allumettes 0)
                  (renforcement nb_allumettes_IA IA_coups actions)
              )
            (explore-renfDernierCoup nb_allumettes actions)
            )
          (explore-renfDernierCoup 1 actions))
        )))))
```

Exemple d'execution :

```
> (explore-renfDernierCoup 16 actions)
L'IA joue 2
Rentrer le coup à jouer (il reste 14 allumettes):
3
L'IA joue 3
Rentrer le coup à jouer (il reste 8 allumettes):
2
L'IA joue 2
Rentrer le coup à jouer (il reste 4 allumettes):
1
L'IA joue 3
NIL
```

```
> (explore-renfDernierCoup 16 actions)
L'IA joue 2
Rentrer le coup à jouer (il reste 14 allumettes):
2
L'IA joue 1
Rentrer le coup à jouer (il reste 11 allumettes):
2
L'IA joue 1
Rentrer le coup à jouer (il reste 8 allumettes):
3
L'IA joue 3
Rentrer le coup à jouer (il reste 2 allumettes):
2
((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) ...)
```

```
;;vérification
> (successeurs 5 actions)
(3 2 1 3)
```

5. Pour cette question, nous avons implémenté une fonction lisp qui permet d'ajouter le coup gagnant aux coups possibles pour le nombre d'allumettes en jeu dans la liste actions.

Spécification : renforcement (nbAllumettes, coup, actions)
Entree :

- nbAllumettes correspond au nombre d'allumettes sur la table à renforcer.
- coup correspond au coup à insérer dans la liste actions.
- actions correspond à la liste de toutes les actions possibles.

Sortie : liste actions modifiée avec le coup donné en paramètre, ajouté dans la sous-liste correspondante au nombre d'allumettes nbAllumettes.
Objectif : Cette fonction permet d'ajouter le coup gagnant aux coups possibles pour le nombre d'allumettes en jeu dans la liste actions.

Deux fonctions ont été implémentées :

Algorithm 9 renforcement

```
(defun renforcement (nbAllumettes coup actions)
    (let ((toAdd (append (list nbAllumettes coup) (cdr (assoc nbAllumettes actions)))))
    ; On prépare l'element a ajouter (liste composée du nombre d'allumettes,
    ; du coup et des autres possibilitées déjà présentes dans actions)
    (setf (nth  (* (- nbAllumettes 16) -1) actions) toAdd)
    ; On récupère l'indice ou se trouve la sous-liste correspondante au nombre d'allumettes
    ; et on la modifie grâce au setf par la sous-liste toAdd cree précédement.
    actions)); On retourne actions
```

Algorithm 10 renforcement

```
(defun renforcement (nbAllumettes coup actions)
  (setf (cdr (assoc nbAllumettes actions))
    (nconc (cdr(assoc nbAllumettes actions)) (list coup)))
    actions)
```

Exemple d'exécution :

```
(Renforcement 7 1 actions) -> (... (7 3 2 1 1) ...)
(Renforcement 16 3 actions) -> ((16 3 2 1 3) ...)
```

Explication : Les deux fonctions utilisent un setf afin d'accéder à la mémoire et de changer l'élément voulu. Dans la première fonction l'accès se fait grace à l'indice de l'emplacement de la sous-fonction d'actions correspondant à nbAllumettes. La deuxième accède à la mémoire grâce à un cdr. Ainsi, la deuxième fonction reste plus simple. En effet, elle permet de modifier la liste actions sans calculer l'indice de la sous-liste correspondante. C'est donc cette fonction que l'on gardera pour la suite.

6. Dans cette dernière partie, nous allons implémenter une fonction qui permet de renforcer l'IA avec tous les coups qu'elle a pu jouer dès qu'elle gagne la partie.

Spécification : explore-renf-rec (nb_allumettes, actions)
Entree :

— nb_allumettes correspond au nombre d'allumettes sur la table.
— actions correspond à la liste de toutes les actions possibles.

Sortie : liste actions modifiée suite au renforcement si l'IA gagne, nil sinon.
Objectif : Sur le même principe que explore-renf1 (cf question 3), explore-renf-rec simule une partie de jeu de Nim.
Une liste a_renf enregistre tous les coups de l'IA. Si elle gagne cette liste est parcourue pour être utilisée dans la fonction renforcement (definie à la question 5).

Commentaire : A part l'étape du renforcement décrite ci-dessus, explore-renf fonctionne de la même manière que explore-renf1.
De plus, nous utilisons la fonction explore-renf-rec durant l'appel afin de remettre à Nil la variable globale a_renf.

Algorithm 11 explore-renf-rec

```
explore-renf-rec (nb_allumettes, action)
  a_renf <- nil //(variable globale)
  explore-renf(nb_allumettes, actions)
```

Algorithm 12 explore-renf

```
 explore-renf (nb_allumettes actions)
    IA_coups <- Nil
    humain_coups <- Nil
    Si nb_allumettes == 0 alors :
        Pour tout élément x de a_renf faire :
            renforcement de x
        Retourner a_renf
    Sinon si nb_allumettes == 1 alors:
        Retourner nil
    Sinon
        IA_coups <- Randomsuccesseurs (successeurs(nb_allumettes, actions))
        Affichage ("L'IA joue IA_coups")
        Ajouter dans a_renf la liste (IA_coups, nb_allumettes)
        nb_allumettes <- nb_allumettes - IA_coups
        Si nb_allumettes > 0 alors :
            humain_coups <- JeuJoueur (nb_allumettes, actions)
            nb_allumettes <- nb_allumettes - humain_coups
            explore-renf (nb_allumettes, actions)
        Sinon :
            explore-renf (1, actions)
```

Algorithm 13 explore-renf-rec

```
(defun explore-renf-rec (nb_allumettes actions)
  (defparameter a_renf nil)
    (explore-renf nb_allumettes actions))
```

Algorithm 14 explore-renf

```
(defun explore-renf (nb_allumettes actions)
  (let ((IA_coups ) (humain_coups))
    (cond
     ((= nb_allumettes 0)
        (dolist (x a_renf actions)
          (renforcement (car x) (cadr x) actions))) ;; IA gagne
     ((= nb_allumettes 1) nil) ;; IA perd
     (t (progn;; IA qui joue
            (setq IA_coups (Randomsuccesseurs (successeurs nb_allumettes actions)))
            (format t "L'IA joue ~s ~&" IA_coups)
            (push (list nb_allumettes IA_coups) a_renf)
            (setq nb_allumettes (- nb_allumettes IA_coups))
        (if (> nb_allumettes 0);; humain qui joue
            (progn
            (setq humain_coups (JeuJoueur nb_allumettes actions))
            (setq nb_allumettes (- nb_allumettes humain_coups))
            (explore-renf nb_allumettes actions)
              )
          (explore-renf 1 actions)))))))
```

Exemple d'exécution :

```
> (explore-renf-rec 16 actions)
L'IA joue 3
Rentrer le coup à jouer (il reste 13 allumettes):
2
L'IA joue 1
Rentrer le coup à jouer (il reste 10 allumettes):
1
L'IA joue 1
Rentrer le coup à jouer (il reste 8 allumettes):
3
L'IA joue 2
Rentrer le coup à jouer (il reste 3 allumettes):
2
NIL
```

```
> (explore-renf-rec 16 actions)
L'IA joue 2
Rentrer le coup à jouer (il reste 14 allumettes):
1
L'IA joue 2
Rentrer le coup à jouer (il reste 11 allumettes):
3
L'IA joue 1
Rentrer le coup à jouer (il reste 7 allumettes):
2
L'IA joue 3
Rentrer le coup à jouer (il reste 2 allumettes):
2
((16 3 2 1 2) (15 3 2 1) (14 3 2 1) (13 3 2 1 2)
 (12 3 2 1) (11 3 2 1) (10 3 2 1) (9 3 2 1) (8 3 2 1 1)
 (7 3 2 1) ...)
```

Explication : La fonction explore-renf-rec définit une variable globale a_renf puis fait appel à la fonction explore-renf. Ainsi, a_renf sera reconnue dans explore-renf et sera remise à nil à chaque appel de explore-renf-rec.
Pour procéder au renforcement des coups de l'IA, nous enregistrons dans a_renf chaque coup réalisé par l'IA avec son nombre d'allumettes associés. La variable a_renf étant globale, elle peut être réutilisée à chaque appel récursif de explore-renf. Ainsi, lorsque l'IA gagne (condition nb_allumettes = 0) cette liste est parcourue pour pouvoir renforcer la stratégie de l'IA.

4 Conclusion

Lors de ce TP, nous avons eu l'occasion de tester deux résolutions différentes du jeu de Nim. La première résolution constitue une résolution naïve alors que lors de la seconde résolution, l'IA se renforce et s'améliore au fil des parties.

Après plusieurs essais des deux résolutions, nous avons remarqué qu'au début les deux résolutions sont équivalentes. Néanmoins, au bout d'un certain nombre de parties, la seconde résolution devient beaucoup plus efficace car l'IA applique des choix qui lui permettent de gagner grâce aux parties précédentes qui ont renforcé sa stratégie. Lors de la première partie les deux résolutions sont donc similaires mais au bout d'un grand nombre de partie, l'IA de la seconde résolution devient beaucoup plus forte alors que l'IA de la première résolution garde le même niveau.