# IA01 : Intelligence Artificielle - Représentation des connaissances

## TP2 : Jeu de Nim

ASSAF Nora & GAJAN Antoine




ALLIANCE SORBONNE UNIVERSITÉ utc

# Table des matières

# Introduction

Ce TP a pour objectif de mobiliser les compétences vues en cours magistraux et en travaux dirigés afin de répondre à un problème complexe.

En effet, il s'agit ici du jeu de Nim. C'est un jeu de stratégie qui se joue à 2 joueurs. Les joueurs se trouvent face à 16 allumettes. Ce jeu se joue chacun son tour. Chaque joueur peut retirer 1, 2 ou 3 allumettes. Celui qui prend la dernière allumette a perdu.

Grâce à l'étude de deux méthodes de résolutions (la recherche dans un espace d'états et une intelligence artificielle capable de s'améliorer sa stratégie), nous établirons la meilleure stratégie de résolution.

Ce rapport présentera la démarche entreprise afin de répondre au problème.

# 1 Modélisation du problème

Nous adopterons la modélisation proposée dans l'énoncé pour représenter les actions. Pour chaque nombre d'allumettes entre 1 et 16, on établira une sous-liste dont le premier élément sera le nombre d'allumettes sur la table et les suivants seront les actions de retrait possibles. La liste des actions est alors composée de 16 sous listes, chacune ayant la structure proposée ci-dessus.

Par exemple, s'il reste de 16 à 3 allumettes, on peut retirer 1, 2 ou 3 allumettes. Ceci sera représenter par la sous-liste suivante :

```
(x 1 2 3) ;; pour x allant de 3 a 16
```

Lorsqu'il reste 2 allumettes, on ne peut retirer que 1 ou 2 allumettes. On aura donc :

```
(2 1 2)
```

Lorsqu'il ne reste qu'une seule allumette, on ne peut en retirer qu'une. La représentation proposée est la suivante :

```
(1 1)
```

On obtient donc la liste des actions 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)))
```

# 2 Résolution 1 : Recherche dans un espace d'états

## 2.1 Définition de l'espace d'états

Nous allons étudier un parcours dans un espace d'états du problème où chaque état est représenté formellement par deux variables représentant le nombre d'allumettes disponibles et le joueur qui a la main. L'IA démarrant la partie, on aura l'état initial suivant :

```
(16 'IA)
```

Chaque état est de la forme (nb_allumettes joueur). On a 16 allumettes et 2 joueurs. Au total, il y aura 32 états possibles pour le jeu de Nim.

Il y aura deux états finaux. En effet, en fin de partie, il reste 0 allumette sur la table, et le joueur associé est le gagnant (soit l'humain, soit l'IA). On a donc les états finaux suivants :

```
(0 'humain)
(0 'IA)
```

## 2.2 Arbre de recherche

Le passage d'un état à un autre étant obtenu à partir de l'application d'une action, on obtient l'arbre de recherche suivant pour le jeu de Nim :

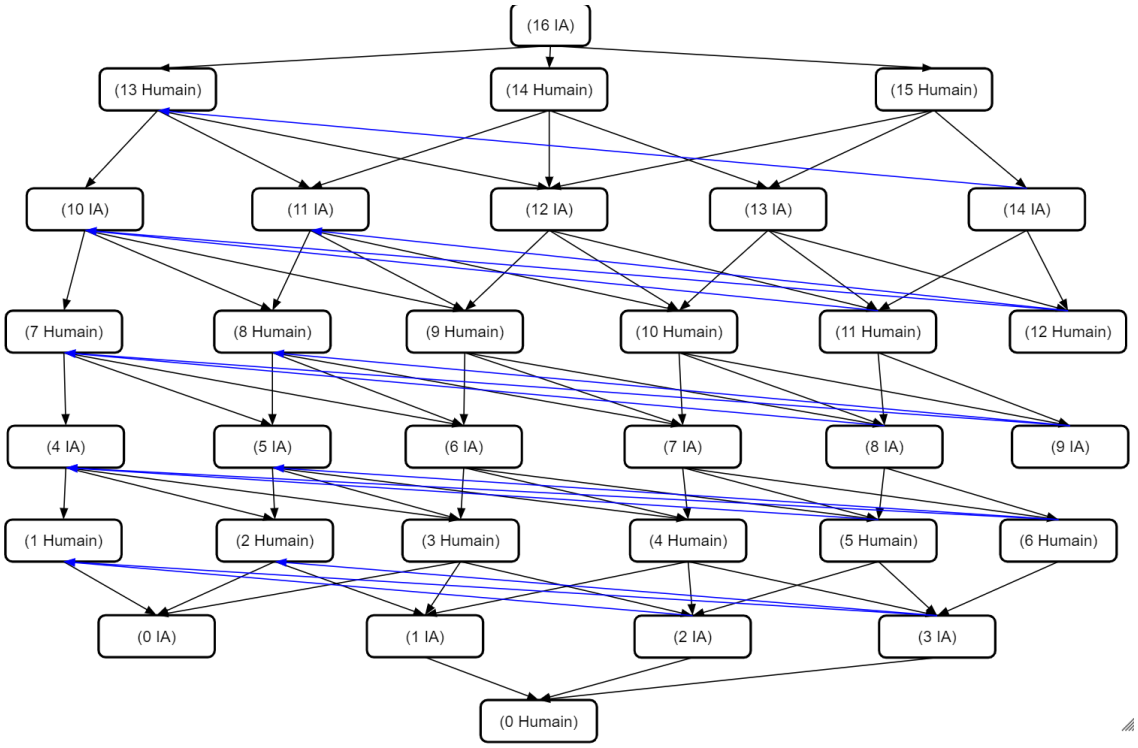


FIGURE 1 – Arbre de recherche du jeu de Nim (16 allumettes)

Afin d'améliorer la lisibilité de l'arbre, voici l'arbre de recherche lorsqu'on joue au jeu de Nim avec 8 allumettes et que l'IA commence la partie :

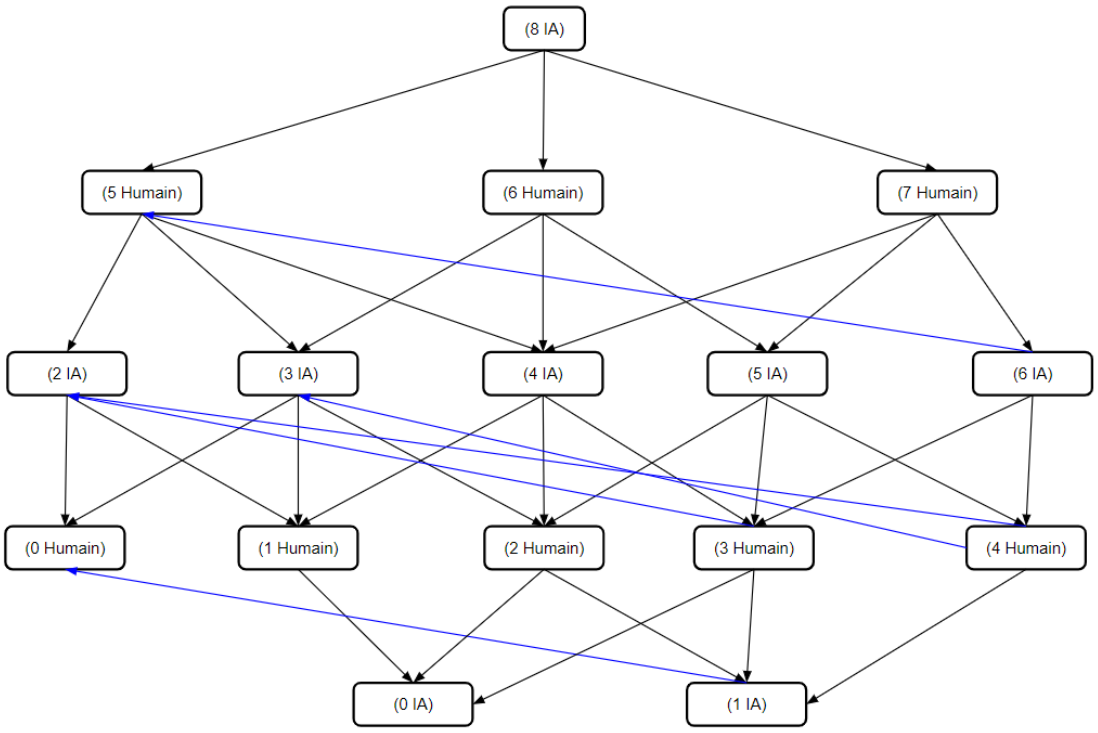


FIGURE 2 – Arbre de recherche du jeu de Nim (8 allumettes)

On remarque instantanément que l'on est en présence d'un arbre de recherche conséquent, avec de nombreux états et beaucoup d'associations.

## 2.3 Analyse des fonctions

Le code suivant est proposé dans l'énoncé :

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

;; EXPLORE
;; Fonction qui effectue la recherche d'etat

(defun explore (allumettes actions joueur i) ;declaration de la
    fonction
  (cond
   ((and (eq joueur 'humain) (eq allumettes 0)) nil) ;si c'est l'humain
     qui joue et que le nombre d'allumettes = 0 renvoyer nil (l'IA a
    perdu)
   ((and (eq joueur 'IA) (eq allumettes 0)) t) ;si c'est l'IA qui joue
    et que le nombre d'allumettes = 0 renvoyer vrai (l'IA a gagne)

   (t
    (progn
         (let ((sol nil) (coups (successeurs allumettes actions))) ;on
    met sol a nil et coups = successeurs de l'etat actuel
           (while (and coups (not sol)) ;tant que la liste coup est non
     vide et que sol = nil
             (progn
               ;affichage du joueur qui est en train de jouer et du
    nombre d'allumettes qu'il reste
               (format t "~%~V@tJoueur ~s joue ~s allumettes - il reste
     ~s allumette(s) " i joueur (car coups) (- allumettes (car coups)))
               ;on rappelle recursivement explore avec le nombre d'
    allumettes restant et l'autre joueur et on met le resultat dans sol
               (setq sol (explore (- allumettes (car coups)) actions (
    if (eq joueur 'IA) 'humain 'IA) (+ i 3)))

                (if sol
                    (setq sol (car coups))) ;si l'IA a gagne (sol = t)
    alors on affecte sol avec le 1er element de la liste coups (le coup
     gagnant)
               (format t "~%~V@t sol = ~s~%" i sol)
               (pop coups) ;on retire le coup que l'on vient de traiter
               )
             )
           ;; On renvoie le coup a jouer par l'IA, menant a une
    victoire
          sol)))))

(defvar nbCoupsAJouer nil)

;on appelle la fonction avec differents nombres d'allumettes
(setq nbCoupsAJouer (explore 16 actions 'IA 0))
(setq nbCoupsAJouer (explore 8 actions 'IA 0))
(setq nbCoupsAJouer (explore 3 actions 'IA 0))
```

Ce code mérite une analyse minutieuse afin de bien comprendre son fonctionnement. Afin de faciliter sa compréhension, nous avons ajouté des commentaires directement sur le code proposé.

### 2.3.1 Fonction successeur

Prenons d'abord quelques exemples.

```
(successeurs 16 actions) ;; renvoie (3 2 1)
(successeurs 5 actions) ;; renvoie (3 2 1)
(successeurs 2 actions) ;; renvoie (2 1)
(successeurs 1 actions) ;; renvoie (1)
(successeurs 16 NIL) ;; Aucun coup possible, renvoie NIL
```

On remarque que la fonction successeur renvoie la valeur associée à la clé allumettes (nombre d'allumettes sur la table) dans la liste actions définie ci-dessus. Pour cela, la fonction assoc a été utilisée. Elle retourne une liste correspondant à un couple (clé valeur). En prenant le cdr (ie, la liste sans son premier élément), on obtient une liste qui n'est autre que la valeur associée à la clé allumettes.

### 2.3.2 Fonction explore

A nouveau, prenons quelques exemples pour comprendre le fonctionnement de la fonction explore.

```
(explore 16 actions 'IA 0) ;; renvoie 3, le premier coup joue par l'IA
   quand il restait 16 allumettes sur la table
(explore 8 actions 'IA 0) ;; renvoie 3, le premier coup joue par l'IA
   quand il restait 8 allumettes sur la table
(explore 3 actions 'IA 0) ;; renvoie 2, le premier coup joue par l'IA
   quand il restait 3 allumettes sur la table
```

La fonction explore comporte un cond, équivalent à switch dans de nombreux autres langages. Il comporte 3 conditions. Si le joueur est l'humain et que le nombre d'allumettes est 0, alors on retourne NIL car l'IA a pris la dernière allumette et a donc perdu. Dans le cas contraire, où l'IA gagne, on retourne true. Sinon, on va procéder comme suit. On définit la solution à NIL (aucune solution encore trouvée pour que l'IA gagne) et les coups possibles avec les successeurs de l'état actuel (fonction expliquée ci-dessus).

Tant que l'on n'a pas trouvé de solution permettant à l'IA de gagner et qu'il reste des allumettes sur la table, on affiche le joueur devant retirer des allumettes, ainsi que le nombre d'allumettes restantes. On prend un "coup" possible, le premier de la liste des successeurs. On appelle alors de manière récursive la fonction explore avec le nombre d'allumettes restantes suite à l'action du joueur. Si on a trouvé une

solution menant à la victoire de l'IA, on définit la solution comme le coup ayant permis la victoire. On retire le coup de la liste des coups possibles puisque déjà traité. Si l'on n'a pas obtenu une solution, on reprend le même processus avec un autre coup défini dans la liste des coups possibles, et ce, jusqu'à trouver une solution pour que l'IA gagne ou que la liste des coups possibles soit vide. Enfin, on retourne la solution. Ce sera un nombre si on a trouvé une solution permettant à l'IA de gagner et NIL le cas échéant.

Cette fonction explore renvoie donc un coup pour lequel on a trouvé au moins une configuration permettant à l'IA de gagner. Si aucun coup ne permet la victoire, alors la fonction renvoie NIL.

## 2.4 Type de recherche

Ici l'algorithme effectue une recherche en profondeur. En effet, on appelle récursivement la fonction explore sur le premier successeur de chaque état actuel à partir de l'état initial. Lorsqu'on arrive à un état final, s'il s'agit de celui souhaité (victoire de l'IA), on retourne le premier coup joué par l'IA. Sinon, si l'on n'a pas trouvé de configuration gagnante, on "remonte" à l'état précédent (situé sur l'étage au-dessus), et on appelle de manière récursive la fonction explore sur le successeur suivant, et ce, jusqu'à trouver une configuration gagnante pour l'IA.

En d'autres termes, l'algorithme choisit un chemin, va au bout de ce chemin, puis revient sur ses pas et fait pareil pour tous les chemins possibles dans le graphe donné jusqu'à trouver un état solution. Ce parcours peut donc être résumé par le schéma ci-dessous, modélisant un parcours en profondeur "par la gauche".

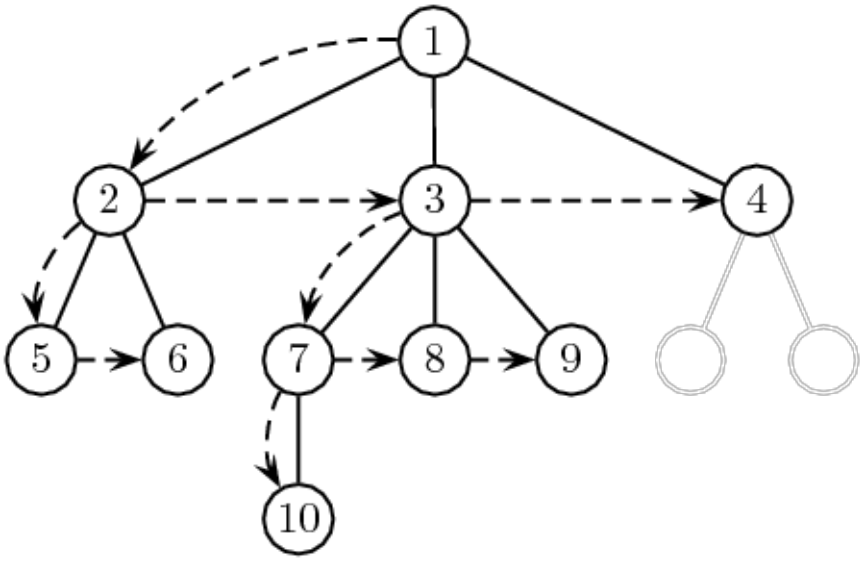


FIGURE 3 – Schéma général de la recherche en profondeur

Dans notre cas, le "par la gauche" représente les situations où l'on tire le nombre maximum d'allumettes possibles. La recherche en profondeur se fait donc comme

suit, en suivant les arcs oranges. Le parcours entier de l'arbre en profondeur est résumé sur le schéma suivant :

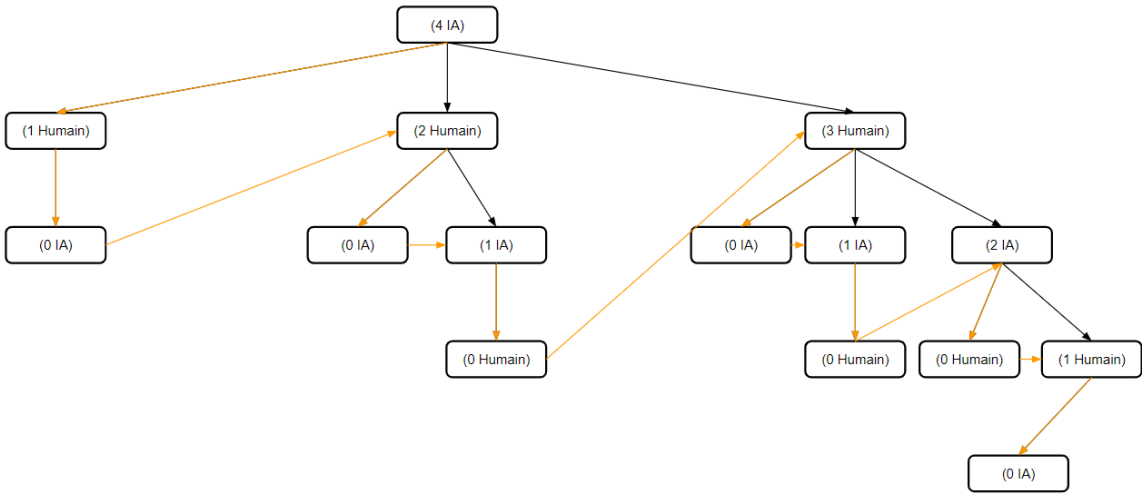


FIGURE 4 – Parcours en profondeur de l'arbre à partir de 4 allumettes

## 2.5 Comparaison des modes de recherche

Les deux types de parcours permettent de parcourir tous les chemins dans le pire des cas. Cependant, la profondeur permet plus facilement de remonter tous les chemins en les concaténant tandis que la largeur est plus rapide sur les gros graphes. Les deux parcours ont la même complexité temporelle, en O(S + A) où S et A représentent respectivement le nombre de sommets et le nombre d'arêtes du graphe. La recherche en largeur se fait comme suit :

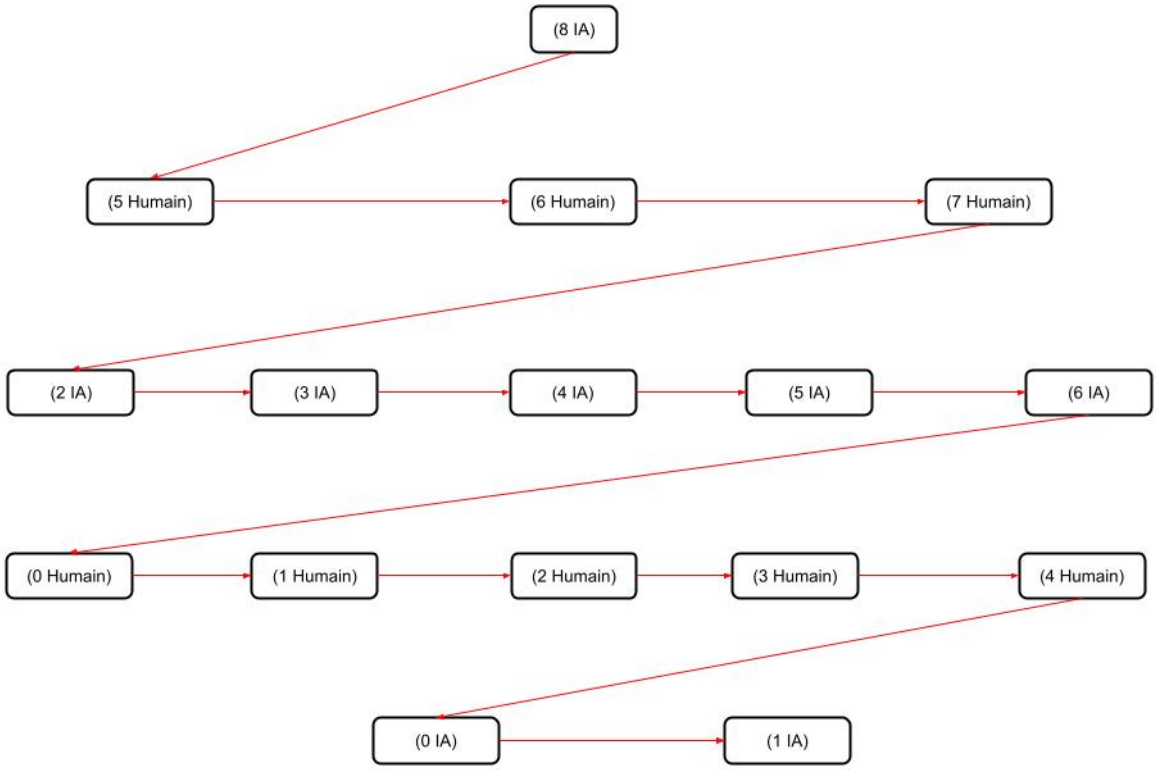


FIGURE 5 – Schéma de la recherche en largeur à partir de 8 allumettes

Le parcours en profondeur semble donc ici être le plus adapté. En effet, même si on est en présence d'un graphe de grande taille, on souhaite trouver une solution et non toutes les solutions. Ici, le parcours en profondeur explore d'abord un chemin jusqu'à l'état final, puis un autre chemin, et ce, jusqu'à ce qu'il trouve une configuration gagnante pour l'IA. Le parcours en largeur nous aurait fait explorer directement l'ensemble des chemins possibles, augmentant la complexité spatiale, ce qui est inutile. En effet, on souhaite seulement trouver un coup à jouer tel qu'il existe une configuration amenant à une victoire possible de l'IA.

## 2.6 Optimisation du parcours

Il serait toutefois possible d'optimiser le parcours. En effet, en plus de parcourir prioritairement les états successeurs qui correspondent à "a retiré 3 allumettes", on pourrait éviter d'explorer les chemins qui se rejoignent. Par exemple, à partir de 8 allumettes restantes (cf. figure 2), si l'IA tire 1 allumette, puis l'humain en tire 1 et enfin l'IA en retire 1, cela est équivalent à ce que l'IA ait retiré directement 3 allumettes. Retirer les chemins qui se rejoignent permettrait de limiter le nombre d'appels de la fonction, tout en évitant de parcourir plusieurs fois le même chemin. Ceci améliorera la complexité temporelle de la fonction. Le graphe résultant de l'optimisation du parcours est en fait donc à enlever les transitions modélisées en bleu sur les figures 1 et 2. On obtient donc le graphe suivant (à partir de 8 allumettes pour améliorer la lisibilité) :

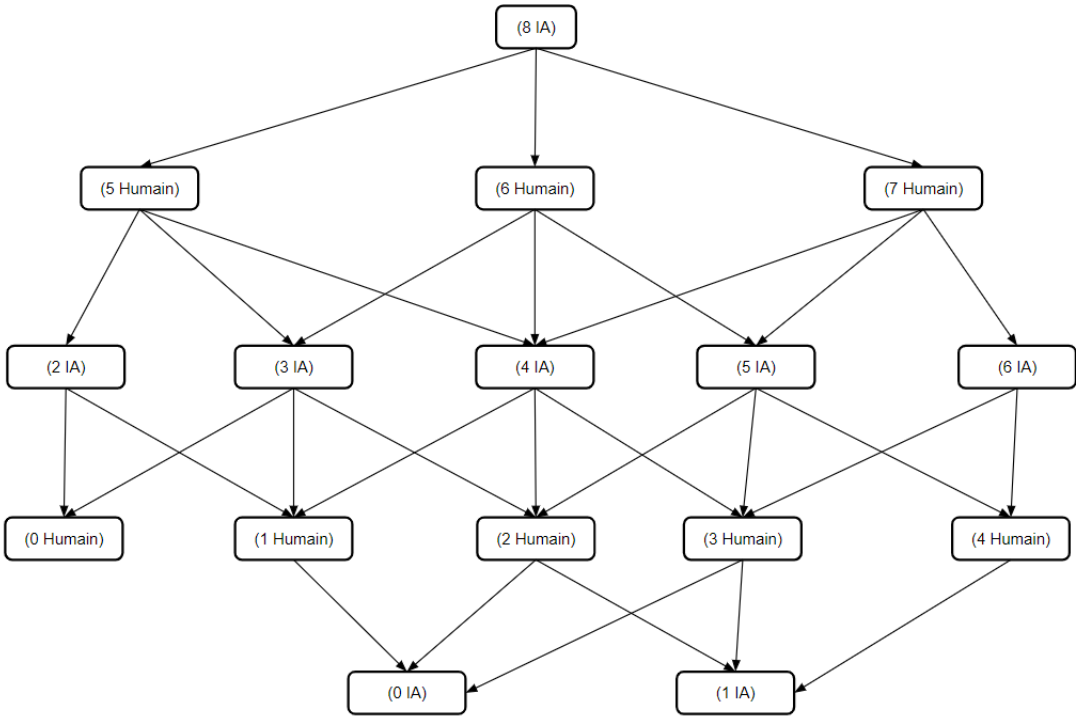


FIGURE 6 – Graphe résultant de l'optimisation de l'algorithme (avec 8 allumettes)

Il serait cependant possible d'améliorer l'algorithme au détriment de la complexité de la fonction. En effet, une idée d'amélioration consisterait non plus à retourner la première solution trouvée, mais celle qui a la plus grande probabilité de victoire future pour l'IA. Il faudrait donc, pour un nombre d'allumettes donné, explorer l'ensemble des possibilités et calculer pour chaque possibilité le ratio nb_chemins_victoire / nb_chemins_total. La possibilité avec le plus grand ratio de victoire est alors retournée.

# 3 Résolution 2 : Intelligence artificielle

## 3.1 Présentation du modèle

Nous allons à présent concevoir une intelligence artificielle avec apprentissage par renforcement. Celle-ci sera d'abord sans stratégie de meilleur choix et effectuera pour commencer des choix aléatoires. Cependant, au fil des parties, l'intelligence artificielle devra s'améliorer, prenant en considération les résultats des précédentes parties.

## 3.2 Jeu du joueur

Tout d'abord, quand l'humain joue, il faut lui demander quel coup il souhaite jouer et vérifier ce celui-ci est bien possible. Pour cela, nous avons créé la fonction JeuJoueur. En fonction du nombre d'allumettes restantes sur la table et la liste des actions possibles définie précédemment, la fonction retourne le coup s'il est valide ou en redemande un jusqu'à ce qu'il soit possible.

Afin de protéger l'utilisateur d'une mauvaise utilisation de la fonction, nous avons ajouté un pour vérifier qu'il existe des coups possibles dans actions associés au nombre d'allumettes restants. Dans le cas contraire, la fonction renverra directement NIL sans demander ce que souhaite jouer l'utilisateur.

```
;; Fonction qui renvoie le coup valide demande par l'utilisateur
(defun JeuJoueur(allumettes actions)
    ;; Definition des variables
    (let ((coup NIL) (succ (successeurs allumettes actions)) (possible
   NIL))
        ;; Si la liste des successeurs est non nulle, on demande a l'
   utilisateur
        (if succ
            (progn
                (format t "~% Combien d'allumettes voulez-vous retirer
   ? ")
                (setq coup (read))
                (if (member coup succ)
                    (setq possible t)
                )
                ;; Tant que le coup n'est pas valide
                (while (not possible)
                    ;; On en redemande un et on regarde s'il est valide
                    (format t "Ce coup ne peut pas etre joue. Combien d
   'allumettes voulez-vous retirer ? ")
                    (setq coup (read))
                    (if (member coup succ)
                        (setq possible t)
                    )
                )
            )
        )
```

```
        ;; On retourne le coup
        coup
    )
)
```

Afin de vérifier le bon fonctionnement de la fonction, nous proposons le jeu de tests suivant :

```
;; Jeu de tests
(JeuJoueur 16 actions) ;; demande a l'utilisateur un nombre d'
    allumettes jusqu'a ce qu'il donne un nombre entre 1 et 3 (actions
    valides)
(JeuJoueur 2 actions) ;; demande a l'utilisateur un nombre d'allumettes
     jusqu'a ce qu'il donne un nombre entre 1 et 2 (seules actions
    valides)
(JeuJoueur 16 NIL) ;; aucun coup possible car actions = NIL donc on ne
    demande rien a l'utilisateur et on renvoie NIL
```

## 3.3 Déroulement d'une partie

A présent, nous savons décider comment doit jouer l'IA et comment s'assurer que l'humain joue un coup valide. On peut donc définir une fonction qui modélisera le déroulement d'une partie. Cette fonction alternera les coups de l'IA et de l'humain jusqu'à ce que l'un des 2 joueurs prenne la dernière allumette et perde.

On a donc l'algorithme de la fonction explore-renf suivant :

```
Fonction explore-renf(allumettes joueur actions):
    Si allumettes = 0 et joueur = humain : retourner echec
    Sinon si allumettes = 0 et joueur = IA : retourner actions
    Sinon:
        // Demande du coup a jouer
        Si joueur = humain:
            coup = JeuJoueur(allumettes actions)
        Sinon :
            coup = randomSuccesseurs(successeurs(allumettes actions))
        // Au tour du joueur suivant de jouer
        Si joueur = humain alors joueur = IA sinon joueur = humain
        Retourner explore-renf((- allumettes coup) joueur actions))
```

NB : nous avons modifié le prototype de la fonction afin de prendre en compte le joueur qui doit jouer plus facilement.

Nous avons également fait le choix d'ajouter dans notre code Lisp des vérifications avancées, afin de s'assurer de la cohérence des différents paramètres. Par exemple, on vérifie que actions est bien une liste non vide, que le nombre d'allumettes est positif ou nul, et que le joueur est soit un humain soit une IA. De ce fait, nous obtenons le code suivant :

```
(defun explore-renf(allumettes joueur actions)
```

```
    (if (AND (listp actions) actions (OR (eq joueur 'IA) (eq joueur '
humain)) (>= allumettes 0)(<= allumettes 16))
        (cond
            ;; Si l'humain gagne
            (
                (and (eq allumettes 0) (eq joueur 'humain))
                (progn
                    (format t "~%~%Il reste 0 allumette.~%Vous avez
gagne.")
                    NIL
                )
            )
            ;; Si l'IA gagne
            (
                (and (eq allumettes 0) (eq joueur 'IA))
                (progn
                    (format t "~%~%Il reste 0 allumette.~%L'
intelligence artificielle a gagne.")
                    actions
                )
            )
            ;; Sinon
            (t
                ;; Creation des variables
                (let ((coup NIL))
                    ;; Affichage du nombre restant d'allumettes
                    (format t "~%~%Il reste ~s allumettes." allumettes)

                    ;; Si l'humain doit jouer
                    (if (eq joueur 'humain)
                        ;; Demande du coup a jouer et appel recursif
                        (progn
                            (setq coup (JeuJoueur allumettes actions))
                            (format t "~%Vous avez retire ~s allumettes
." coup)
                            (explore-renf (- allumettes coup) 'IA
actions)
                        )

                        ;; Si l'IA doit jouer
                        (progn
                            ;; L'IA prend un coup random
                            (setq coup (randomSuccesseurs (successeurs
allumettes actions)))
                            (format t "~%L'IA a retire ~s allumettes."
coup)
                            (explore-renf (- allumettes coup) 'humain
actions)
                        )
                    )
                )
            )
        )
    )
)
```

On peut proposer les tests suivants afin de s'assurer du bon fonctionnement de la

fonction :

```
(explore-renf 16 'IA actions) ;; L'IA commence a jouer, il reste 16
    allumettes
(explore-renf 8 'Humain actions) ;; L'humain commence a jouer, il reste
    8 allumettes
(explore-renf -1 'IA actions) ;; retourne NIL car -1 n'est pas un
    nombre d'allumettes valide
(explore-renf 13 'moi actions) ;; retourne NIL car "moi" n'est pas un
    joueur valide
(explore-renf 12 'IA NIL) ;; retourne NIL car actions est une liste
    vide
```

## 3.4 Ajout d'une stratégie de renforcement

Nous savons maintenant jouer une partie avec une IA qui effectue ses choix de manière aléatoire. Nous aimerions à présent l'améliorer au fil des parties. Pour cela, nous allons opter pour la stratégie suivante. Si l'IA gagne, on ajoute les actions ayant permis à l'IA de gagner dans la liste actions. Ainsi, si l'IA a gagné en prenant 3 allumettes quand il en restait 9 sur la table et 2 allumettes quand il en restait 3, alors la liste action devient :

```
(...(9 3 2 1 3)...(3 3 2 1 2)...)
```

En fait, le choix sera toujours effectué de manière aléatoire, mais les différentes possibilités n'auront plus la même probabilité d'apparaître. Par exemple, avec la liste actions ci-dessus, l'IA a plus de chance de prendre 2 allumettes quand il en reste 3 sur la table puisque le chiffre 2 apparaît le plus souvent dans la sous-liste associée au nombre d'allumettes 3.

Dans un premier temps, nous allons d'abord ajouter uniquement le dernier coup dans la liste des actions comme proposé par l'énoncé à l'étape 9. On peut donc proposer l'algorithme de renforcement suivant :

```
Fonction explore-renf(allumettes joueur actions dernier_coup_IA) :
    Si allumettes = 0 et joueur = humain : retourner NIL
    Sinon si allumettes = 0 et joueur = IA :
        Ajouter dernier_coup_IA a actions
        retourner actions
    Sinon:
        Si joueur = humain:
            coup = JeuJoueur(allumettes actions)
            // Au tour de l'IA de jouer
            explore-renf((- allumettes coup) 'IA actions dernier_coup_IA
    )
        Sinon :
            coup = randomSuccesseurs(successeurs(allumettes actions))
            dernier_coup_IA = (allumettes coup)
```

```
            // Au tour de l'humain
            explore-renf((- allumettes coup) 'humain actions
    dernier_coup_IA)
```

NB : Nous avons rajouté un argument pour le dernier coup joué par l'IA. Ainsi, si l'IA gagne, nous pouvons ajouter le dernier coup à la liste actions plus facilement. Nous avons fait ce choix pour écrire la fonction de manière récursive, ce qui limite les efforts de codage. Par conséquent, lors du premier appel, on a dernier_coup_IA = NIL.

En transposant l'algorithme précédent en code Lisp, et en ajoutant des tests par sécurité, on obtient :

```
(defun explore-renf(allumettes joueur actions dernier_coup_IA)
    (if (AND (listp actions) actions (OR (eq joueur 'IA) (eq joueur '
    humain)) (>= allumettes 0)(<= allumettes 16))
        (cond
            ;; Si l'humain gagne
            (
                (and (eq allumettes 0) (eq joueur 'humain))
                (progn
                    (format t "~%~%Il reste 0 allumette.~%Vous avez
    gagne.")
                    NIL
                )
            )
            ;; Si l'IA gagne
            (
                (and (eq allumettes 0) (eq joueur 'IA))
                (progn
                    (format t "~%~%Il reste 0 allumette.~%L'
    intelligence artificielle a gagne.")
                    ;; Ajout du dernier coup a actions
                    (push (cadr dernier_coup_IA) (cdr(assoc (car
    dernier_coup_IA) actions)))
                    actions
                )
            )
            ;; Sinon
            (t
                ;; Creation des variables
                (let ((coup NIL))
                    ;; Affichage du nombre restant d'allumettes
                    (format t "~%~%Il reste ~s allumettes." allumettes)

                    ;; Si l'humain doit jouer
                    (if (eq joueur 'humain)
                        ;; Demande du coup a jouer et appel recursif
                        (progn
                            (setq coup (JeuJoueur allumettes actions))
                            (format t "~%Vous avez retire ~s allumettes
    ." coup)
                            (explore-renf (- allumettes coup) 'IA
    actions dernier_coup_IA)
                        )
```

```
                        ;; Si l'IA doit jouer
                        (progn
                            ;; L'IA prend un coup random
                            (setq coup (randomSuccesseurs (successeurs
    allumettes actions)))
                            (setq dernier_coup_IA (list allumettes coup
    ))
                            (format t "~%L'IA a retire ~s allumettes."
    coup)
                            (explore-renf (- allumettes coup) 'humain
    actions dernier_coup_IA)
                        )
                    )
                )
            )
        )
    )
)
```

On peut vérifier le bon fonctionnement de la fonction à l'aide des tests suivants :

```
(explore-renf 16 'IA actions NIL) ;; L'IA commence a jouer, il reste 16
    allumettes
(explore-renf 8 'Humain actions NIL) ;; L'humain commence a jouer, il
    reste 8 allumettes
(explore-renf -1 'IA actions NIL) ;; retourne NIL car -1 n'est pas un
    nombre d'allumettes valide
(explore-renf 13 'moi actions NIL) ;; retourne NIL car "moi" n'est pas
    un joueur valide
(explore-renf 12 'IA NIL NIL) ;; retourne NIL car actions est une liste
    vide
```

La détection des erreurs est particulièrement intéressante puisqu'elle permet d'éviter toute mauvaise manipulation de la fonction de la part de l'utilisateur du programme.

## 3.5 Fonction de renforcement

Afin de coder la fonction précédente en Lisp, il nous faut être capable de faire l'action : "ajouter coup à actions". Pour cela, nous allons d'abord coder une fonction renforcement(nombre d'allumettes en jeu, coup joué gagnant, liste actions). Cette fonction permettra d'ajouter le coup gagnant aux coups possibles pour le nombre d'allumettes en jeu dans la liste actions. Par exemple, (Renforcement 7 1 actions) renvoie la liste actions (.... (7 1 3 2 1)...) ou (... (7 3 2 1 1)...).

On veillera à la cohérence des paramètres de la fonction grâce à des tests. On obtient donc le code suivant :

```
(defun renforcement(allumettes coup_gagnant actions)
    ;; Si les parametres sont valides
```

```
    (if (AND (listp actions) actions (>= allumettes 0) (<= allumettes
    16))
        (let((liste (assoc allumettes actions)))
            ;; Si la liste est bien non vide
            (if liste
                (push coup_gagnant (cdr liste))
            )
            actions
        )
    )
)
```

On peut tester le fonctionnement du code à l'aide des tests suivants :

```
(renforcement 16 2 actions) ;; Ajoute 2 dans les valeurs associees a la
    cle 16 de actions
(renforcement -1 2 actions) ;; renvoie NIL car -1 n'est pas un nombre d
   'allumettes valides
(renforcement 16 1 NIL) ;; renvoie NIL car actions = NIL
```

## 3.6 Etendre le renforcement

Nous souhaitons étendre les principales étapes de l'algorithme de la fonction explore-renf précédente afin de mettre en œuvre le renforcement récursif décrit à l'étape 10 de l'énoncé. En d'autres termes, on souhaite que l'ensemble des coups joués par l'IA ayant mené à une victoire soit ajouté à la liste des actions.

```
Fonction explore-renf-rec(allumettes joueur actions)
   Si allumettes = 0 et joueur = humain : retourner echec
   Si allumettes = 0 et joueur = IA : retourner actions // Sinon si ???
   Sinon:
       Si joueur = humain:
           coup = JeuJoueur(allumettes actions)
           (explore-renf-rec (- allumettes coup) 'IA actions)
       Sinon :
           coup = randomSuccesseurs(successeurs(allumettes actions))
           sol = (explore-renf-rec (- allumettes coup) 'Humain actions)
           Si sol :
               actions = (renforcement(allumettes coup actions))
```

En transposant cet algorithme en code Lisp, on obtient le code suivant :

```
(defun explore-renf-rec(allumettes joueur actions)
    (if (AND (listp actions) actions (OR (eq joueur 'IA) (eq joueur '
   humain)) (>= allumettes 0) (<= allumettes 16))
        (cond
            ;; Si l'humain gagne
            (
                (and (eq allumettes 0) (eq joueur 'humain))
                (progn
                    (format t "~%~%Il reste 0 allumette.~%Vous avez
   gagne.")
```

```
                    NIL
                )
            )
            ;; Si l'IA gagne
            (
                (and (eq allumettes 0) (eq joueur 'IA))
                (progn
                    (format t "~%~%Il reste 0 allumette.~%L'
    intelligence artificielle a gagne.")
                    actions
                )
            )
            ;; Sinon
            (t
                ;; Creation des variables
                (let ((coup NIL) (sol NIL))
                    ;; Affichage du nombre restant d'allumettes
                    (format t "~%~%Il reste ~s allumettes." allumettes)

                    ;; Si l'humain doit jouer
                    (if (eq joueur 'humain)
                        ;; Demande du coup a jouer et appel recursif
                        (progn
                            (setq coup (JeuJoueur allumettes actions))
                            (format t "~%Vous avez retire ~s allumettes
    ." coup)
                            (explore-renf-rec (- allumettes coup) 'IA
    actions)
                        )

                        ;; Si l'IA doit jouer
                        (progn
                            ;; L'IA prend un coup random
                            (setq coup (randomSuccesseurs (successeurs
    allumettes actions)))
                            (format t "~%L'IA a retire ~s allumettes."
    coup)
                            (setq sol (explore-renf-rec (- allumettes
    coup) 'humain actions))
                            ;; Si l'IA gagne
                            (if sol
                                ;; MAJ de actions en ajoutant le coup a
     actions
                                (renforcement allumettes coup actions)
                                ;; Si l'IA perd, on renvoie NIL
                                NIL
                            )
                        )
                    )
                )
            )
        )
    )
)
```

On peut proposer les tests suivants qui permettent de s'assurer du bon fonctionnement du code :

```
(explore-renf-rec 16 'IA actions) ;; L'IA commence a jouer, il reste 16
    allumettes
(explore-renf-rec 8 'Humain actions) ;; L'humain commence a jouer, il
   reste 8 allumettes
(explore-renf-rec -1 'IA actions) ;; retourne NIL car -1 n'est pas un
   nombre d'allumettes valide
(explore-renf-rec 13 'moi actions) ;; retourne NIL car "moi" n'est pas
   un joueur valide
(explore-renf-rec 12 'IA NIL) ;; retourne NIL car actions est une liste
    vide
```

Si l'IA gagne, la fonction renforcement sera appelée sur chacun des coups joués par l'IA. La liste actions sera alors mise à jour, permettant l'apprentissage par renforcement.

# 4 Conclusion

Ainsi, ce TP nous a permis de comparer la résolution d'un problème complexe à l'aide de 2 méthodes différentes. La première consiste en une recherche dans un espace d'états. Elle nous a permis de déterminer un coup possible permettant à l'IA de gagner la partie. La deuxième méthode consiste en la création d'une intelligence artificielle avec apprentissage par renforcement. Cela signifie qu'elle est capable de se renforcer au fil des parties. En effet, elle détermine le coup à jouer grâce à son apprentissage des parties précédentes. Les coups ayant mené aux victoires sont ajoutés à la liste des actions, augmentant ainsi la probabilité de ces coups d'être choisi.

La première solution semble donc plus pertinente dans le cas où on a très peu de parties jouées. En effet, on est sûr que le programme renverra un coup pour lequel il y a au moins une possibilité de gagner pour l'IA. Quant à la deuxième solution, elle semble donc plus efficace après un apprentissage sur de nombreuses parties. En effet, avec un nombre de parties qui tend vers l'infini, la probabilité de choisir le meilleur coup possible, amenant à une victoire de l'IA, sera proche de 1.