Rapport TP2

Jeu de Nim

ALLIANCE SORBONNE UNIVERSITÉ utc

Sommaire

Introduction

Nous avons été amenés à travailler sur le second TP de IA01 durant les mois d'octobre et de novembre 2022. Ce second TP est une mise en pratique des notions de parcours dans un espace d'états vues en cours, mais aussi une introduction à un nouveau type d'algorithme. Pour cela, le sujet propose de coder deux algorithmes permettant de joueur au jeu de Nim, le premier basant sa stratégie sur une exploration dans un espace d'états du problème, et le deuxième en apprenant à jouer à partir d'une technique appelée apprentissage par renforcement.

Ce rapport contient les codes et algorithmes correspondant aux deux types d'algorithmes demandés.

Le jeu de Nim 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, nous utilisons donc ici la version "Fort Boyard". Nous avons trouvé lors de nos recherches un site expliquant très clairement ce jeu et sa complexité. Il explique également le lien entre ce jeu et la théorie des jeux. Il est disponible ici. On y apprend que le jeu de Nim est un jeu résolu. C'est-à-dire que peu importe l'état où l'on est, il existe un moyen de parvenir à un état gagnant ou un état perdant (gagnant donc pour l'autre). En effet, le jeu de Nim est un jeu à somme nulle, c'est-à-dire que soit on gagne, soit on perd, il n'y a pas d'égalité possible.

Les différentes fonctions, jeux de données de test et commentaires explicatifs sont présents dans les fichiers suivants :

- *resolution1.cl* : pour le premier type de résolution
- *resolution2.cl* : pour le second type de résolution
- *graph.py* : contenant notre code python permettant de générer le graphe demandé dans la première résolution
- *render* : contenant des exemples de graphes générés par notre code
- *actions_test.txt* : Contenant le résultat de la base d'actions une fois entraînée après avoir joué 1000 parties.

1 Résolution 1 : Exploration dans un espace d'états

1.1 États initiaux et finaux

On peut décider de représenter les états de cette manière : (n j), avec n le nombre d'allumettes restantes, et j le joueur qui a la main, par exemple 0 pour l'IA et 1 pour l'humain.

Ainsi, l'état initial serait (16 0), 16 allumettes restantes et l'IA a la main. Pour les états finaux, il y en a deux, la partie se termine quand il ne reste plus d'allumettes restantes, et cela peut arriver lors du tour du joueur comme lors du tour de l'IA. Ainsi, les deux états finaux possibles sont (0 0) et (0 1). Il est également possible d'utiliser une représentation où on utilise directement le nom du joueur à la place de 0 ou 1.

Pour résumer, on a donc :
- État initial : (16 0)
- États finaux : (0 0) (0 1)

ou alors :
- État initial : (16 IA)
- États finaux : (0 IA) (0 Humain)

Nous avons préféré utiliser dans la suite, la seconde version utilisant les termes IA et Humain.

1.2 Arbre de recherche

Afin de rendre l'affichage le plus simple et le plus propre possible, nous avons choisi de créer un petit code python qui dessine l'arbre des possibilités en fonction de critères que demande l'utilisateur.

Ainsi, nous avons codé le fichier "graph.py", disponible dans les fichiers rendus avec ce rapport. Pour l'utiliser, assurez-vous d'avoir bien installé le module graphviz.

```
pip install graphviz
```

Avant de lancer le code, assurez-vous d'avoir un dossier nommé "render" dans le même dossier que le fichier "graph.py", c'est dans ce dossier que seront générer les images correspondant aux arbres de recherche.

Le programme vous demandera le joueur de départ, le nombre d'allumettes de départ, et si le graphe doit être complet ou tronqué. Nous avons laissé la possibilité de générer des graphes tronqués pour des questions de lisibilité. Un graphe tronqué n'afficher pas le sous arbre d'un nœud déjà affiché ailleurs, mais plutôt un nœud contenant "(...)". Quelques exemples d'arbres sont fournis dans le dossier "render" fourni avec ce rapport. Si vous en générer de nouveaux par vous-même, sachez que tenter de générer un graphe complet (donc non tronqué) avec plus de 10 allumettes

au départ peut faire planter le programme, au vu du très grand nombre de nœuds.

Voici un petit exemple d'arbre créé par le programme, en faisant commencer l'IA, avec 4 allumettes au départ et en demandant un graphe complet :

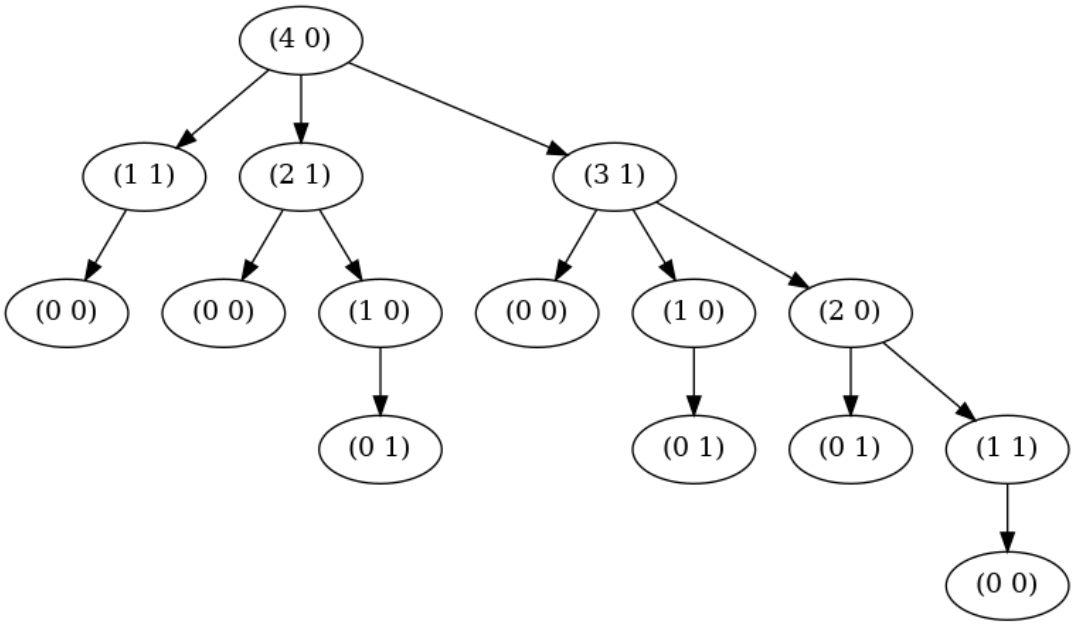

FIGURE 1 – Exemple d'arbre de recherche

À voir aussi dans le dossier "render" :
— 6-complet.png et 6-tronque.png, pour voir la différence entre les deux types d'affichage
— 8-complet.png, pour avoir une idée de l'étendue des possibilités
— 16-tronque.png, pour avoir un arbre qui représente tous les états de notre problème

1.3 Explication de code

Ce code est une proposition de stratégie pour l'IA. La fonction explore prend en entrée un nombre d'allumettes restantes et le joueur qui a la main. La fonction cherche un chemin menant à une victoire de l'IA dans l'espace d'états.

Cependant, elle ne permet pas à l'IA de gagner à tous les coups, elle fait des suppositions sur le comportement de l'humain, en simulant ses choix et en le faisant jouer de sorte à faire gagner l'IA. Ceci s'explique par le fait que la fonction est récursive et cherche à trouver un coup amenant à faire gagner l'IA, que le joueur courant soit l'IA ou l'humain. Les simulations des coups de l'humain ne reflètent donc pas vraiment la réalité.

La fonction renvoie le nombre d'allumettes que doit prendre le joueur courant (IA ou humain donc) pour faire gagner l'IA, car elle aura trouvé un chemin depuis cette action amenant à la victoire de l'IA. Nous pouvons vérifier cela avec cet appel :

```
(setq nbCoupsAJouer (explore 4 actions 'HUMAIN 0))
```

ou pareil mais en faisant jouer l'IA en premier :

```
(setq nbCoupsAJouer (explore 7 actions 'IA 0))
```

Nous voyons ici que ces appels amènent l'humain à changer son comportement, car lors de la simulation/exploration, le premier choix de l'humain n'offre aucune chance de gagner à l'IA. L'humain change donc son choix pour proposer un chemin menant à la victoire de l'IA.

Le détail du fonctionnement (explication ligne par ligne) est fourni à la fin du fichier *resolution1.cl*

Ce code est une recherche en profondeur, car la fonction n'explore pas toutes les possibilités étage par étage dans l'arbre des possibilités, mais recherche à la place en profondeur jusqu'au bout de l'arbre le premier chemin menant à la victoire.

1.4 Profondeur ou largeur

La fonction largeur offrirait la possibilité de considérer tous les nœuds se trouvant à un étage donné de l'arbre de recherche, donc de considérer tous les choix possibles de l'humain et donc jouer mieux. Cependant, le nombre d'états d'un étage à l'autre augmente de façon exponentielle, ce qui pourrait augmenter considérablement le temps de calcul.

Pour palier à cela, on pourrait se servir du fait qu'il y a des états en commun à plusieurs endroits de l'arbre, états ayant les mêmes sous arbres.

On pourrait ainsi faire en sorte de ne pas traiter plusieurs fois les états déjà traités ailleurs dans l'arbre, en mettant les états déjà traités dans une liste par exemple. C'est d'ailleurs avec ce principe que nous avons codé la fonction d'affichage de l'arbre de recherche tronqué.

Le choix en profondeur quant à lui prend peu de temps de calcul étant donné qu'il ne prend pas en compte toutes les possibilités, mais à la place juste un chemin pouvant mener à la victoire. Il est cependant moins intéressant d'un point de vue stratégie, car il ne prend pas en compte tous les choix possibles de l'humain.

Pour résumer, l'exploration en profondeur prend moins de temps de calcul, mais offre une stratégie peu efficace. L'exploration en largeur demande beaucoup de temps de calcul, mais offre une stratégie beaucoup plus efficace. De plus, le temps de calcul peut être considérablement réduit en ne traitant pas deux fois les mêmes états. L'exploration en largeur semble de ce fait être la plus adaptée pour ce jeu.

1.5 Optimisation

Nous avons deux propositions pour améliorer le parcours dans l'arbre de recherche :

1.5.1 Amélioration du parcours largeur

Cette amélioration a déjà été décrite dans la question précédente, elle consiste à enregistrer les états déjà visités pour ne pas les visiter plusieurs fois, ce qui fait gagner un temps de calcul considérable.

1.5.2 Proposition d'un parcours mixte

On pourrait également imaginer un parcours largeur-profondeur, qui pour le tour de l'IA explore en profondeur, et pour le tour de l'humain explore en largeur.

On combinerait ainsi la rapidité du parcours en profondeur et la bonne stratégie du parcours en largeur. Pour le tour de l'humain, on simule tous ses coups possibles, et pour l'IA, on explore en profondeur. On saurait ainsi s'il existe un chemin, à partir d'un coup de l'IA, qui lui permette de gagner à tous les coups en choisissant bien ses actions, indépendamment des choix de l'humain.

On pourrait de plus minimiser encore plus le temps de calcul en appliquant ici aussi l'enregistrement des états déjà visités.

2 Résolution 2 : Apprentissage par renforcement

Dans cette résolution, l'IA se dote d'un historique de ses positions gagnantes et va ainsi pouvoir améliorer son jeu de partie en partie. En effet, nous allons modifier la liste des actions si les choix de l'IA lui permettent de gagner.

Afin, de permettre ce changement de valeur dans la variable contenant la liste d'actions, nous avons choisi de transformer le setq en defparameter (variable globale). En effet, le defparameter va nous permettre de redéfinir la liste des actions déjà existante tout en gardant son accès via la variable *actions*.

Pour cela, nous allons dans un premier temps explorer et retourner la liste des actions si l'IA gagne, puis nous allons appliquer un renforcement en modifiant en mémoire la liste des actions.

Nous avons dans le sujet le code suivant permettant à l'IA de choisir aléatoirement un coup :

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

FIGURE 2 – Randomsuccesseurs

2.1 JeuJoueur

Pour cette fonction, nous allons utiliser la fonction read qui permet de lire une saisie de l'utilisateur. Nous utilisons les fonctions princ et format afin de proposer un affichage guidant l'utilisateur dans son choix de coup (affichage du min et du max).

Dans le but de récupérer et retourner la valeur du coup, nous utilisons deux variables locales :

- coup : permettant de récupérer la saisie de l'utilisateur
- succ : gardant les successeurs, entendu ici en tant que coup possible, pour le nombre d'allumettes en cours

Nous demandons à l'utilisateur de réitérer sa saisie si celle-ci n'est pas dans l'ensemble succ. À l'aide des boucles itérative while ou loop while.

Nos codes sur cette fonction sont les suivants :

```
(defun JeuJoueur (nbAllu actions)
  (if (AND (< nbAllu 17) (> nbAllu -1) (integerp nbAllu) (listp
 actions))
    (let ((coup nil) (succ (successeurs nbAllu actions)))
      (format t "min:~d, max:~d~&" (car (last succ)) (car succ))
      (princ "Entrez le nombre d'allumettes a enlever :")
      (setq coup (read))

      (while (not (member coup succ))
        (format t "Erreur saisie le nombre~&")
        (format t "min:~d,max:~d~&" (car(last succ)) (car succ))
        (princ "Entrez le nombre d'allumettes a enlever :")
        (setq coup (read))
      )
      coup
    )
  )
)
```

FIGURE 3 – JeuJoueur : utilisant un while

```
(defun JeuJoueur (allumettes actions)
  (if (AND (< allumettes 17) (> allumettes -1) (integerp allumettes)
 (listp actions))
      (let ((choix NIL)(choix_possibles (successeurs allumettes
 actions)))
        (loop while (not (member choix choix_possibles)) do
          (format t "Nombre d'allumettes ~s~& min : ~s~& max : ~s~&"
 allumettes (car (last choix_possibles)) (car choix_possibles))
          (princ "Entrez le nombre d'allumettes : ")
          (setq choix (read))
          (if (not (member choix choix_possibles))
              (format t "Erreur saisie nombre~&")
          )
        )
        choix
      )
  )
)
```

FIGURE 4 – JeuJoueur : utilisant un loop while

2.2 Exploration : liste d'actions

Pour cette fonction, nous nous sommes inspirés de la fonction explore et nous l'avons modifié pour les besoins de cet exercice. Nous avons codé deux versions de cette fonction.

La première version permet de gérer les différents cas indépendamment et correspond donc à l'état à une itération donnée. Notre seconde version permet de gérer le couple : IA joue puis humain joue, dans le même appel récursif. En effet, nous

savons d'après l'énoncé que l'IA commence ainsi, nous n'avons qu'à alterner entre IA humain puis recommencer.
Les explications de ces deux fonctions sont présentes après l'algorithme d'exploration.

Vous pouvez retrouver ci-dessous, l'algorithme de la première version de la fonction explore-renf sans renforcement pour le moment, mais retournant simplement nil ou la liste des actions :

Algorithm 1: Algorithme explore-actions

Data: $0 \leq nb\_allumettes \leq 16$
$joueur = IA$ ou $joueur = humain$
$actions =$ liste des actions possibles
Result: *nil* ou *actions*

explore-actions ($nb\_allumettes$, $joueur$, $actions$)
 if $j == IA$ **and** $nb\_allumettes == 0$ **then**
 | $sol \leftarrow$ **nil**
else
 | **if** $j == humain$ **and** $nb\_allumettes == 0$ **then**
 | | $sol \leftarrow actions$
 | **else if** $j == IA$ **then**
 | | $coup \leftarrow$ **Randomsucceseurs** ($nb\_allumettes$, $actions$)
 | | $sol \leftarrow$ **explore-actions**($nb\_allumettes$ - $coup$, $humain$, $actions$)
 | **else**
 | | $coup \leftarrow$ **JeuJoueur** ($nb\_allumettes$, $actions$)
 | | $sol \leftarrow$ **explore-actions**($nb\_allumettes$ - $coup$, IA, $actions$)
 | **end**
end
return : sol

Pour la première version de la fonction, nous avons choisi une version récursive de la fonction et avons ainsi 4 cas à étudier :

— (0 IA) : l'IA a gagné puisque le joueur a saisi la dernière allumette
— (0 humain) : l'humain a gagné puisque l'IA a saisi la dernière allumette
— (n IA), n $\neq$ 0 : C'est à l'IA de jouer
— (n humain), n $\neq$ 0 : C'est à l'humain de jouer

Dans le cas où c'est à l'humain de jouer, nous récupérons sa saisie à l'aide de la fonction JeuJoueur codée plus haut. Cependant, nous avons commenté cette ligne et générons un choix aléatoire pour l'humain afin de simplifier les tests de la fonction. Pour choisir à nouveau le comportement de l'humain, il suffit de décommenter la ligne appelant JeuJoueur et de commenter la ligne faisant appel à Randomsucceseurs pour le choix de l'humain. La fonction va retourner la liste des actions si l'IA gagne et nil sinon. Pour cela, nous ne retournons pas t ou nil mais la liste des actions, nous remplaçons alors t par actions.

```
(defun explore-renf (nb_allu actions joueur i)
  (cond
   ((and (eq joueur 'humain) (eq nb_allu 0)) nil)
   ((and (eq joueur 'IA) (eq nb_allu 0)) actions)
   (t
    (let  ((sol nil) (coup nil))
      (if (eq joueur 'humain)
        (progn
          ;; (setq coup (JeuJoueur nb_allu *actions*))
          (setq coup (Randomsuccesseurs (cdr (assoc nb_allu actions))
 ))
          (format t "~%~V@tJoueur ~s joue ~s allumettes - il reste ~s
  allumette(s) " i joueur coup (- nb_allu coup))
          (setq sol (explore-renf (- nb_allu coup) actions 'IA (+ i
 3)))
        )
        (progn
          (setq coup (Randomsuccesseurs (cdr (assoc nb_allu actions))
 ))
          (format t "~%~V@tJoueur ~s joue ~s allumettes - il reste ~s
  allumette(s) " i joueur coup (- nb_allu coup))
          (setq sol (explore-renf (- nb_allu coup) actions 'humain (+
  i 3)))
        )
      )
    sol))
 )
)
```

FIGURE 5 – explore-renf : version 1

La seconde version est également récursive et fait donc jouer l'IA et l'humain dans le même appel récursif.

Le déroulé de la fonction est le suivant :
- On fait jouer l'IA
- On vérifie si elle a perdu suite à son coup
- Si oui, on renvoie NIL
- Sinon, on fait jouer le joueur
- On vérifie si le joueur a perdu suite à son coup
- Si oui, on renvoie une liste contenant la dernière action de l'IA
- Le cas échéant, fait un appel récursif et on renvoie NIL si cet appel mène à un échec de l'IA, et une concaténation de la dernière action de l'IA avec la liste de ses actions renvoyée par l'appel récursif dans le cas ou ce dernier mène à une victoire de l'IA

Le choix de l'humain se fait de la même façon que la version précédente de cette fonction. Tout comme cette dernière, cette deuxième version de la fonction va retourner la liste des actions si l'IA gagne et nil sinon.

```
(defun explore-renf (allumettes actions)
  (let ((choix_possibles (successeurs allumettes actions))(
action_IA NIL)(action_humain NIL))

    (format t "Restant IA : ~s ~s~&" allumettes choix_possibles)
    (setq action_IA (Randomsuccesseurs choix_possibles))
    (format t "choisi IA : ~s~2&" action_IA)

    (if (equal (- allumettes action_IA) 0)
      NIL

      (progn

        (format t "Restant humain : ~s ~s~&" (- allumettes
action_IA) (successeurs (- allumettes action_IA) actions))
        ;;(setq action_humain (JeuJoueur (- allumettes action_IA)
actions)) ;; choix reel de l'humain
        (setq action_humain (Randomsuccesseurs (successeurs (-
allumettes action_IA) actions))) ;; choix aleatoire de l'humain
        (format t "choisi humain : ~s~2&" action_humain)

        (if (equal (- allumettes action_IA action_humain) 0)
          actions
          (explore-renf (- allumettes action_IA action_humain)
actions)
        )
      )
    )
  )
)
```

FIGURE 6 – explore-renf : version 2

2.3 Exploration : avec renforcement

Dans cette partie, nous allons reprendre l'algorithme de l'exploration en utilisant simplement un appel à renforcement (cf. : algorithme question suivante).
Pour avoir une fonction qui renforce le dernier coup gagnant dans le cas où l'IA gagne, il suffit de reprendre la deuxième version de la fonction précédente et de rajouter :

(setq *actions* (renforcement allumettes action_IA *actions*))

Dans le cas où l'humain vient de perdre, on renforce le dernier coup de l'IA, vu la façon dont la fonction est construite, on a facilement accès au nombre d'allumettes qu'elle avait devant elle et au nombre d'allumettes qu'elle a choisi de prendre.

Algorithm 2: Algorithme explore-renf

Data: $0 \leq nb\_allumettes \leq 16$
$joueur = IA$ ou $joueur = humain$
$actions$ = liste des actions possibles
Result: *nil* ou *actions* renforcé

explore-renf (*nb_allumettes*, *actions*)
coup_IA ← **Randomsuccesseurs** (*nb_allumettes*, *actions*)
if *nb_allumettes - coup_IA == 0* **then**
  return : nil
else
  coup_h ← **JeuJoueur** (*nb_allumettes*, *actions*)
  if *nb_allumettes - coup_IA - coup_h == 0* **then**
    Renforcement(*nb_allumettes coup_IA actions*)
    Return : *actions*
  else
    sol ← **explore-renf**(*nb_allumettes - coup_IA - coup_h*, *actions*)
    Return : *sol*
  end
end

```
(defun explore-renf (allumettes actions)
    (let ((choix_possibles (successeurs allumettes actions))(
 action_IA NIL)(action_humain NIL))
      (format t "Restant IA : ~s ~s~&" allumettes choix_possibles)
      (setq action_IA (Randomsuccesseurs choix_possibles))
      (format t "choisi IA : ~s~2&" action_IA)
      (if (equal (- allumettes action_IA) 0)
        NIL
        (progn
          (format t "Restant humain : ~s ~s~&" (- allumettes
 action_IA) (successeurs (- allumettes action_IA) actions))
          ;;(setq action_humain (JeuJoueur (- allumettes action_IA)
 actions)) ;; choix reel de l'humain
          (setq action_humain (Randomsuccesseurs (successeurs (-
 allumettes action_IA) actions))) ;; choix aleatoire de l'humain
          (format t "choisi humain : ~s~2&" action_humain)

          (if (equal (- allumettes action_IA action_humain) 0)
            (progn
              (setq *actions* (renforcement allumettes action_IA *
 actions*))
              actions
            )
            (explore-renf (- allumettes action_IA action_humain)
 actions)
          )))))
```

FIGURE 7 – Exploration avec renforcement

2.4 Renforcement

L'objectif de cette fonction est de changer en mémoire la liste des actions qui est un paramètre global. Nous avons codé deux versions de cette fonction que vous pourrez retrouver après l'algorithme.

Algorithm 3: Algorithme renforcement

Data: $0 \leq nb\_allumettes \leq 16$
$1 \leq coup \leq 3$
$actions$ = liste des actions possibles
Result: Ne retourne rien, changement en mémoire

renforcement ($nb\_allumettes$, $coup$, $actions$)
 foreach $action$ **in** $actions$ **do**
  if $premier\_element(action) == nb\_allumettes$ **then**
   ajouter(action, coup)
end

La première version de cette fonction utilise la méthode dolist de parcours de liste. Cela nous permet d'itérer pour chaque élément de la liste. Nous n'avons alors plus que deux cas à traiter :

— Nous sommes au nombre d'allumettes à renforcer
— Nous n'y sommes pas

Dans le second cas, il suffit d'ajouter la liste contenant le nombre d'allumettes et les coups possibles, à la liste des actions possibles. Dans le premier cas, le traitement est bien différent. Dans ce cas, nous allons ajouter la liste contenant le nombre d'allumettes et les coups possibles auquel on a ajouté le coup passé en paramètre.

Pour ce faire, nous allons utiliser cons qui permet d'ajouter le coup en tête de liste, pour l'ajouter à la fin, nous utilisons la fonction reverse. Pour ainsi dire, nous l'ajoutons au début de la fin, soit au dernier élément. Pour remettre la liste dans le bon sens, nous utilisons de nouveau reverse. La liste finale étant inversée, nous l'inversons de nouveau. Enfin, nous pouvons affecter cette nouvelle liste à la variable globale.

```
(defun renforcement (nb_allu coup actions)
  (let ((new_liste nil))
    (format t "Renforcement de l'etat ~s avec l'action ~s~&"
nb_allu coup)
    (dolist (cp actions new_liste)
      (if (= (car cp) nb_allu)
        (setq new_liste (append new_liste (list (reverse (cons coup
(reverse cp))))))
        (setq new_liste (append new_liste (list cp)))
      )
    )
    (setq *actions* (reverse new_liste))
  )
)
```

FIGURE 8 – Renforcement avec un dolist

La seconde version de cette fonction se déroule en quatre étapes et ne nécessite pas de boucle :

— On récupère l'état à renforcer et une copie de la liste pour laquelle on a retiré l'état à renforcer
— On rajoute l'action souhaitée dans la liste des actions possibles de l'état
— On ajoute l'état à la copie de la liste
— On trie la copie de la liste en fonction du premier élément et on la renvoie

Trier la liste à la fin est une étape optionnelle, cela permet de garder les états dans l'ordre ce qui rend l'analyse plus simple.

```
(defun renforcement (allumettes coup_gagnant actions)
  (let* ((etat (assoc allumettes actions))(actions_nouv (remove
etat actions)))
    (format t "Renforcement de l'etat ~s avec l'action ~s~&"
allumettes coup_gagnant)
    (setq etat (append (list (car etat) coup_gagnant) (cdr etat)))
    (push etat actions_nouv)
    (sort actions_nouv #'> :key #'car)
  )
)
```

FIGURE 9 – Autre version du renforcement

2.5 Application du renforcement : explore-renf-rec

Pour la première version de l'apprentissage par renforcement, nous utilisons la même méthode que dans la question 2 explore-renf. Nous utilisons uniquement sol qui va contenir *actions* ou nil.

Si sol est différent de nil alors, nous modifions en mémoire la liste des actions (à l'aide de la fonction renforcement).

```
(defun explore-renf-rec (nb_allu actions joueur &optional (i 0))
    (cond
        ((and (eq joueur 'humain) (eq nb_allu 0)) nil)
        ((and (eq joueur 'IA) (eq nb_allu 0)) *actions*)
        (t
            (let ((sol nil) (coup nil))
                (if (equal joueur 'humain)
                    (progn
                        ;;(setq coup (JeuJoueur nb_allu *actions*))
                        (setq coup (Randomsuccesseurs (cdr (assoc
nb_allu *actions*))))
                        (setq sol (explore-renf-rec (- nb_allu coup
) *actions* 'IA (+ i 3)))
                    )

                    (progn
                        (setq coup (Randomsuccesseurs (cdr (assoc
nb_allu *actions*))))
                        (setq sol (explore-renf-rec (- nb_allu coup
) *actions* 'humain (+ i 3)))
                        (if sol
                            (renforcement nb_allu coup *actions*)
                        )
                    )
                )
            sol
            )
        )
    )
)
```

FIGURE 10 – Explore-renf-rec : version 1

Pour la seconde version de l'apprentissage par renforcement, nous reprenons le code de la question 3, mais en plus de renforcer le dernier coup de l'IA dans le cas où le joueur vient de perdre, on vérifie également le retour de l'appel récursif. Si l'appel récursif indique que la partie a été gagnée par l'IA, alors on renforce la liste d'actions avec le coup joué par l'IA. Ainsi, lorsque l'IA joue, les appels récursifs se cumulent tour après tour, et les coups joués sont en quelque sorte "en attente" d'être renforcé ou non. Dans le cas d'une victoire, le fait que l'IA ait gagné ou non "remonte" les appels récursifs grâce aux retours de la fonction, et on peut alors savoir si on doit renforcer ou non les coups joués en attente.

```
(defun explore-renf-rec (allumettes actions)
  (let ((choix_possibles (successeurs allumettes actions))(
action_IA NIL)(action_humain NIL))

    (format t "Restant IA : ~s ~s~&" allumettes choix_possibles)
    (setq action_IA (Randomsuccesseurs choix_possibles))
    (format t "choisi IA : ~s~2&" action_IA)

    (if (equal (- allumettes action_IA) 0)
      NIL

      (progn

        (format t "Restant humain : ~s ~s~&" (- allumettes
action_IA) (successeurs (- allumettes action_IA) actions))
        ;;(setq action_humain (JeuJoueur (- allumettes action_IA)
actions)) ;; choix reel de l'humain
        (setq action_humain (Randomsuccesseurs (successeurs (-
allumettes action_IA) actions))) ;; choix aleatoire de l'humain
        (format t "choisi humain : ~s~2&" action_humain)

        (if (equal (- allumettes action_IA action_humain) 0)
          (progn
            (setq *actions* (renforcement allumettes action_IA *
actions*))
            actions
          )
          (if (explore-renf-rec (- allumettes action_IA
action_humain) actions)
            (progn
              (setq *actions* (renforcement allumettes action_IA *
actions*))
              actions
            )
            NIL
          )
        )
      )
    )
  )
)
```

FIGURE 11 – Explore-renf-rec : version 2

3 Apprentissage

Afin de tester notre programme, nous avons fait jouer à l'IA 1000 parties. Pour que cela se fasse dans un temps raisonnable et comme expliqué précédemment, nous avons commenté la ligne demandant à l'humain de jouer et nous avons à la place fait jouer l'humain avec la même fonction que l'IA. À noter que les choix de l'humain ne seront alors pas complètement aléatoire, le jeu de l'humain évoluera de la même façon que le jeu de l'humain au fur et à mesure des renforcements. Nous aurions aussi pu faire jouer l'humain aléatoirement avec une base qui ne se renforce pas afin de conserver le caractère purement aléatoire.

La manière dont nous avons fait ce test revient donc à faire jouer l'IA contre elle-même 1000 fois.

Ce test a été réalisé avec la deuxième version de la fonction de la question 5, avec les instructions suivantes :

```
(dotimes (x 1000)(explore-renf-rec 16 *actions*))
(aff *actions*)
```

Les résultats de la liste d'actions suite à ce test sont disponibles dans le fichier *actions_test.txt*.

Le jeu de Nim étant un jeu résolu, on peut facilement interpréter les résultats. Dans ce TP la version du jeu de Nim étant la même que celle de "Fort Boyard", il faut pour gagner, laisser une allumette modulo 4 du nombre total d'allumettes (ainsi il faut laisser 1, 5, 9, 13). Ainsi, si l'adversaire prend trois allumettes on en prend une, s'il en prend deux on fait de même, et s'il en prend une on en prend 3, on est ainsi assurés de diminuer le nombre d'allumettes de 4 à chaque tour, jusqu'à ne laisser au joueur qu'une seule allumette.

Ainsi, si on a devant nous 12 allumettes, il faut donc en prendre 3 pour en laisser 9 à l'adversaire, et si on a devant nous 7 allumettes, il faut en prendre 2 pour en laisser 5 à l'adversaire, etc.

Lorsqu'on analyse la liste d'actions, on voit qu'on obtient exactement les mêmes résultats. Bien que les éléments qui composent la liste des actions possibles avec un nombre d'allumettes donné ne correspondent pas tous à la bonne réponse, on voit qu'on a tout de même la majorité d'entre eux qui sont corrects. Les états proche de 1 sont cependant plus précis que les états éloignés, (comme 16 ou 12), qui même s'ils ont une liste d'actions possibles qui corresponde majoritairement à la bonne réponse, le pourcentage de bonnes réponses reste plus faible que pour les faibles nombres d'allumettes.

Un point à noter est le cas de l'état avec 1 allumette et celui avec 15. Pour celui avec 1, on voit que cet état n'a jamais été renforcé, ce qui est logique étant donné que si on arrive dans cet état, c'est qu'on a perdu, et que le renforcement ne se fait qu'en cas de victoire. Pour ce qui est de l'état 15, ce n'est pas que l'IA ne peut

jamais gagner avec 15 allumettes, c'est juste que dans notre cas elle n'a jamais joué avec 15 allumettes. En effet étant donné qu'elle commence à 16, cette dernière prend au minimum 1 allumette et en laisse donc 15 au joueur. Pour renforcer l'état 15, on pourrait effectuer le même appel que celui fait précédemment, mais en changeant le nombre d'allumettes au départ, en le mettant soit à 15, soit à un nombre supérieur à 16.

Conclusion

Le second TP nous a permis d'approfondir nos connaissances sur les notions de représentation des connaissances vues en cours. Nous avons pu améliorer notre compréhension des espaces d'états et comprendre une nouvelle notion d'IA : l'apprentissage par renforcement. Nous avons testé notre programme en le faisant jouer de nombreuses fois, et en comparant les résultats aux résultats connus (le jeu de Nim étant un jeu résolu), nous constatons que l'IA arrive à apprendre quels sont les "bons" coups à jouer.

Le premier mode de parcours de liste nous permettait de trouver une solution, qui n'était pas toujours bonne car faisait de fortes hypothèses sur le comportement de l'humain. Cette méthode de résolution nous a toutefois permis de réfléchir aux différentes limites de certains algorithmes, et aussi comment les optimiser.

Le deuxième mode de résolution nous a permis d'apprendre un nouveau type de raisonnement : l'apprentissage par renforcement ne base pas la qualité de sa stratégie sur l'exploration, mais plutôt par l'expérience, les résultats empiriques. On part du principe que si statistiquement, tel choix dans telle situation mène plus souvent à la victoire, alors cela doit probablement être le bon.

Table des figures

Table des algorithmes