Forest Thomas

Quintaneiro Raphaël

20/11/2022

TP02

IA01

utc
Université de Technologie
Compiègne

Sommaire

Introduction

Nous abordons dans ce TP la modélisation d'une IA pour le jeu de Nim. C'est un jeu simple qui peut être représenté par une succession d'états et dans lequel il y a toujours un gagnant et un perdant. Afin de proposer une IA pour ce jeu, nous allons étudier deux résolutions possibles. La première est une IA indépendante du joueur et fixe : à un moment donné de la partie et pour un mouvement donné du joueur, l'IA choisira toujours le même nombre d'allumette à enlever. Une partie est donc facilement reproductible. Dans la seconde résolution l'IA réagit au hasard avec un modèle probabiliste qui change au fur et à mesure des parties et en fonction du joueur.

Lors de la première partie de ce TP, nous analyserons les différents états ainsi qu'un algorithme qui nous a été donné. Pour la seconde partie, nous nous inspirerons de cette analyse pour proposer un modèle différent plus agréable à jouer pour un joueur humain.

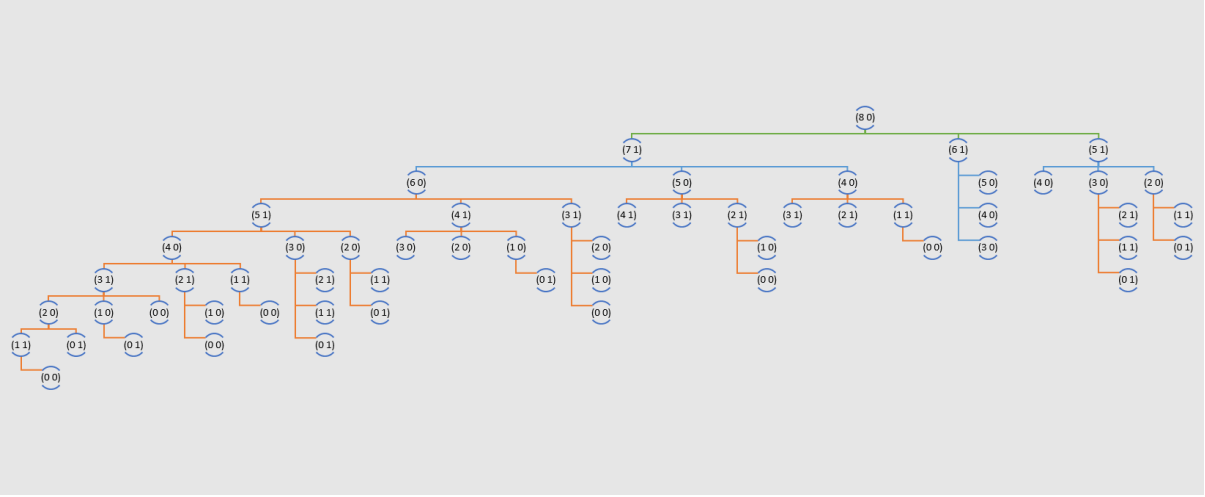
Résolution 1

1. Pour la suite de l'exercice, on décidera de noter 0 lorsque l'IA à la main et 1 lorsque l'humain à la main. Ainsi, on obtient alors :

Etat Initial : (16 0)

Etat finaux : (0 0) OU (0 1)

2. Ci-dessous, l'arbre de recherche en partant de l'état (8,0) :



3. Le code de la fonction explore commence par vérifier les conditions de fin de la fonction :

- Si le joueur est l'humain et que le nombre d'allumettes est égal à 0 alors on renvoie **NIL**
- Si le joueur est l'IA et que le nombre d'allumettes est égal à 0 alors on renvoie **T**

Sinon, tant qu'il existe des successeurs à l'état actuel, on les explore récursivement. Or, étant donné que les coups « décisifs » se jouent lorsqu'il reste moins de 4 allumettes, tous les coups précédents seront de retirer 3 allumettes (car c'est le premier successeur pour chaque action). Autrement dit, l'IA jouera naïvement 3 allumettes à chaque tour tant qu'il en reste au moins 4 car cela n'influence pas dans ce programme la victoire de l'IA.

Le code cherche alors une issue dans laquelle l'IA gagne (**sol = t**). Le code explore toutes les possibilités (dans l'ordre des successeurs) tant qu'il ne trouve pas une solution gagnante pour l'IA

On est donc ici en présence d'une recherche en **profondeur**.

4. En s'appuyant sur l'arbre de la question 2, on peut remarquer que la recherche en profondeur semblerait plus adaptée. En effet, la recherche en largeur impliquerait de considérer un nombre d'état beaucoup plus grand que la recherche en profondeur. Comme expliqué dans la question précédente, la fonction en profondeur se dirigera plus rapidement vers un état gagnant étant donné que l'IA trouvera toujours une solution gagnante lorsqu'il reste au moins 4 allumettes.

5. Pour améliorer le parcours de la fonction explore, on peut simplement éviter que l'IA ne choisisse des solutions qui ne font pas sens. Par exemple, lorsque l'IA à la main et qu'il reste 3 allumettes, son premier choix sera de retirer 3 allumettes (de même lorsqu'il reste 2 allumettes). Or, dans ce cas l'IA perd instantanément alors qu'elle pourrait gagner en faisant un choix plus judicieux. Ainsi, en rajoutant une condition pour éviter les deux cas cités précédemment on évite à l'IA de devoir revenir sur son chemin pour trouver une solution gagnante. Ci-dessous le code lisp correspondant à cette solution.

```
(defun explore (allumettes actions joueur i)
  (cond
   ((and (eq joueur 'humain) (eq allumettes 0)) nil)
   ((and (eq joueur 'IA) (eq allumettes 0)) t)
    (t (progn
         (let ((sol nil) (coups (successeurs allumettes actions)))
           (cond
              ((and (eq joueur 'IA) (eq allumettes 3)) (pop coups))
              ((and (eq joueur 'IA) (eq allumettes 2)) (pop coups)))
            (while (and coups (not sol))
              (progn
                (format t "~%~V@tJoueur ~s joue ~s allumettes - il reste ~s allumette(s) " i joueur (car coups) (- allumettes (car coups)))
                (setq sol (explore (- allumettes (car coups)) actions (if (eq joueur 'IA) 'humain 'IA) (+ i 3)))
                (if sol
                    (setq sol (car coups)))
                (format t "~%~V@t sol = ~s~%" i sol)
                (pop coups)
                )
              )
            sol)))))
```

6. Notes supplémentaires sur cette résolution :

En gardant l'esprit de cette résolution mais en changeant sa modélisation, nous pourrions arriver à un modèle d'IA imbattable et facile à généraliser pour d'autres jeux. En effet en représentant n'importe quel jeu sous forme d'états successif, il est possible de construire un graphe représentant le déroulement de toutes les parties possibles.

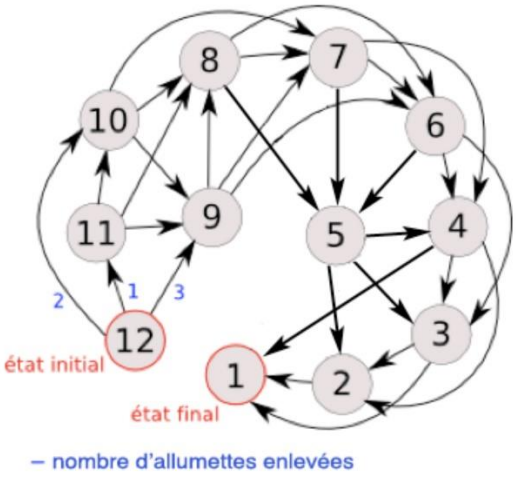


Figure 1 : Graphe associé au jeu de Nim avec 12 allumettes de départ (source interstices.info)

Dès lors, au lieu de s'intéresser à des états précis, l'IA peut s'intéresser à des ensembles. Si le jeu à l'instar du jeu de Nim peut se représenter par un graphe sans cycle, alors il possède un noyau unique. Ce noyau est par définition stable et absorbant : aucun arc ne relie deux sommets du noyau entre eux et chaque sommet est à un arc de distance maximum d'un sommet appartenant au noyau. Pour gagner, il suffit alors à l'IA de rester dans le noyau tout au long de la partie (car le joueur sera obligé de ne pas y rentrer).

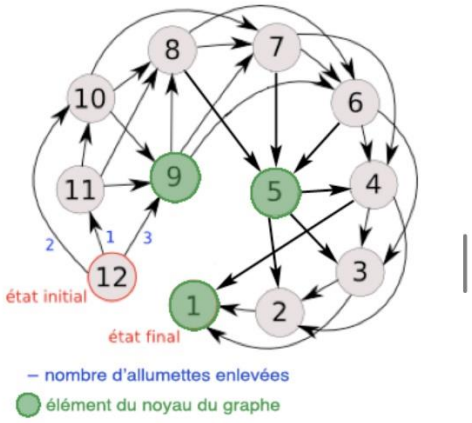


Figure 2 : Représentation du noyau du jeu de Nim (source interstices.info)

Sachant que rechercher un noyau dans un graphe sans circuit se réalise en temps polynomial, il est facile d'implémenter une IA qui choisira de rester dans le noyau à tout moment de la partie.

On aurait alors une IA qui ne peut être battu et qui est adaptable à n'importe quel jeu similaire sans changer le code LISP.

Résolution 2

2.

- La fonction *JeuJoueur* permet de récupérer le coup que l'utilisateur veut jouer.

```
;;; Question 2:
(defun JeuJoueur (allumettes actions)
  (format t "~%Il reste ~s allumettes. Combien voulez-vous en retirez ? " allumettes)
  (let ((stdin (read)) (succ (successeurs allumettes actions)))
    (while (not (member stdin succ))
      (format t "Impossible, veuillez rentrer un nouveau nombre : ")
      (setq stdin (read))
      )
    ;(format t "Vous avez décidé de retirer ~s allumetes" stdin)
    stdin
    )
  )
```

⇨ On définit deux variables :

- ***stdin*** qui représente l'entrée de l'utilisateur
- ***succ*** qui représente les successeurs du nombre d'allumettes restant.

On vérifie l'entrée de l'utilisateur à l'aide d'une boucle **while**. Tant que l'entrée de l'utilisateur ne fait pas partie des successeurs possibles, alors ce dernier doit rentrer un nouveau nombre. Une fois que cela est vérifié, on renvoie le coup joué.

3.

- La fonction *explore-renf* permet l'interaction de l'utilisateur dans le jeu. Nous avons pour cela repris la même structure que la fonction *explore* de l'exercice 1.

```
;;; Question 3:
(defun explore-renf (allumettes actions joueur)
  (cond
   ((and (eq joueur 'humain) (eq allumettes 0))
    (format t "~%L'IA a perdu")
    nil)
   ((and (eq joueur 'IA) (eq allumettes 0))
    (format t "~%L'IA a gagné")
    actions)
   (t (progn
        (let ((coup NIL))
          (if (eq joueur 'humain)
              (setq coup (JeuJoueur allumettes actions))
            (setq coup (Randomsuccesseurs (successeurs allumettes actions))))
          (format t "~%Il y a ~s allumette(s), Joueur ~s tire ~s allumette(s) => il reste ~s allumette(s) " allumettes joueur coup (- allumettes coup))
          (explore-renf (- allumettes coup) actions (if (eq joueur 'IA) 'humain 'IA))
          )))))
```

⇨ Les deux premières conditions sont globalement les mêmes que dans la fonction *explore*. Si aucune de ces deux conditions n'est valide, on passe dans la partie principale. Dans cette partie, on vérifie qui est en train de jouer. Si c'est l'humain alors on lui demande son coup grâce à la fonction *JeuJoueur*, sinon on prend un successeur aléatoire grâce à la fonction *Randomsuccesseurs*. Ensuite, on passe simplement au coup suivant.

4. Nous avons décidé de sauter cette question étant donné que la fonction *renforcement* de la question 5 sera directement implémenté pour tous les coups de l'IA (et non uniquement le dernier) dans le code final de la question 6.

5.

- La fonction *renforcement* permet d'ajouter le coup gagnant aux coups possible pour le nombre d'allumettes en jeu dans la liste d'actions.

```
(defun renforcement (allumettes coup actions)
  (push coup (cdr (assoc allumettes actions)))
  actions)
```

⇨ Le principe est simple, on veut simplement ajouter *coup* à la liste des successeurs de *allumettes* dans *actions*. On utilise donc pour cela la fonction *push*.

6.

- La fonction *explore-renf-rec* reprend le même principe que la fonction *explore-renf* de la question 3 mais en y ajoutant le renforcement dans le cas où l'IA gagne.

```
;;; Question 6 (Fonction finale):
(defun explore-renf-rec (allumettes actions joueur chemin)
  (cond
   ((and (eq joueur 'humain) (eq allumettes 0))
    (format t "~%L'IA a perdu")
    nil)
   ((and (eq joueur 'IA) (eq allumettes 0))
    (progn
      (format t "~%L'IA a gagné")
      (dolist (x chemin) ; Pour chaque coup joué par l'IA, on l'ajoute à la liste d'actions
        (renforcement (car x) (cadr x) actions))) ; (car x) = nb d'allumettes au moment du coup, (cadr x) = coup joué
     actions)
   (t (progn
        (let ((coup NIL))
          (if (eq joueur 'IA)
              (progn
                (setq coup (Randomsuccesseurs (successeurs allumettes actions))) ; successeur aléatoire dans la liste d'actions
                (push (list allumettes coup) chemin)) ; ajout du coup joué par l'IA
            (setq coup (JeuJoueur allumettes actions))) ; lecture du coup de l'utilisateur
          (format t "~%Il y a ~s allumette(s), Joueur ~s tire ~s allumette(s) => il reste ~s allumette(s) " allumettes joueur coup (- allumettes coup))
          (explore-renf-rec (- allumettes coup) actions (if (eq joueur 'IA) 'humain 'IA) chemin)
          )))))
```

⇨ Pour garder une trace des coups joués par l'IA nous avons décidé d'ajouter une variable *chemin* dans les paramètres de la fonction. Ainsi, à chaque fois que l'IA joue, on ajoute à *chemin* une liste comprenant le nombre d'allumettes restantes avant qu'elle joue ainsi que le coup joué. Ainsi, lorsque l'IA gagne, on appelle la fonction *renforcement* pour tous les coups joués par l'IA (→ pour tous les coups présents dans *chemin*).

```
CG-USER(81):
((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) ...)
CG-USER(82):
Il y a 16 allumette(s), Joueur IA tire 3 allumette(s) => il reste 13 allumette(s)
Il reste 13 allumettes. Combien voulez-vous en retirez ? 3

Il y a 13 allumette(s), Joueur HUMAIN tire 3 allumette(s) => il reste 10 allumette(s)
Il y a 10 allumette(s), Joueur IA tire 1 allumette(s) => il reste 9 allumette(s)
Il reste 9 allumettes. Combien voulez-vous en retirez ? 2

Il y a 9 allumette(s), Joueur HUMAIN tire 2 allumette(s) => il reste 7 allumette(s)
Il y a 7 allumette(s), Joueur IA tire 1 allumette(s) => il reste 6 allumette(s)
Il reste 6 allumettes. Combien voulez-vous en retirez ? 1

Il y a 6 allumette(s), Joueur HUMAIN tire 1 allumette(s) => il reste 5 allumette(s)
Il y a 5 allumette(s), Joueur IA tire 3 allumette(s) => il reste 2 allumette(s)
Il reste 2 allumettes. Combien voulez-vous en retirez ? 2

Il y a 2 allumette(s), Joueur HUMAIN tire 2 allumette(s) => il reste 0 allumette(s)
L'IA a gagné
((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 1 3 2 1) (9 3 2 1) (8 3 2 1) (7 1 3 2 1) ...)
```

Sur cet exemple, on voit bien qu'après avoir gagné, l'IA ajoute ses différents coups joués à la liste possible des successeurs de chaque actions effectuée (ici pour les actions 16, 10, 7, 5).

Conclusion

Dans la première résolution, nous utilisons principalement les règles du jeu et nous implémentons l'IA de manière à ce qu'elle réagisse au mieux à chaque état. Cette implémentation est intéressante pour ce jeu car le nombre d'état est assez limité. On peut alors facilement les explorer et les analyser afin de faire le meilleur coup quel que soit le niveau du joueur en face.

Néanmoins, cette résolution a plusieurs inconvénients. Tout d'abord, celle-ci n'est pas particulièrement ludique pour le joueur. En effet comme mentionné précédemment dans ce rapport, l'IA implémentée ici ne « réagit » finalement qu'à la fin de la partie. Elle se contente de prendre 3 allumettes jusqu'à ce qu'il ne reste que 4 allumettes. Même avec les améliorations proposées, il sera difficile d'offrir au joueur des parties différentes.

Un autre inconvénient dans cette solution apparait si l'on considère son fonctionnement plus global. En effet, si une implémentation de cette résolution à d'autres jeux est considérée. Celle-ci peut se retrouver très rapidement impossible si le nombre d'état devient très grand. Par exemple, aux échecs, le nombre de parties jouables est de l'ordre de 10^{120}, il devient donc logiquement impossible de prévoir tous les états.

Analysons à présent la seconde résolution. Celle-ci peut s'apparenter à du machine learning. En effet, nous mettons ici en place un modèle statistique simple pour s'adapter au joueur. Ce modèle est moins efficace durant les premières utilisations mais plus il évolue, plus il est précis et adapté au joueur. Ainsi, ce modèle peut être considéré comme plus ludique car la difficulté est croissante. De plus, en modifiant ce modèle, il peut s'adapter à tout type de jeu. C'est par exemple ce concept qui est utilisé par le moteur LeelaChessZero pour proposer une IA aux échecs. On peut en effet considérer uniquement l'état en cours pour prédire le prochain coup avec un modèle statistique cohérent.

L'inconvénient principal de cette solution est bien évidement son temps d'adaptation. Tant qu'un certain nombre de parties n'a pas été joué, L'IA est beaucoup trop aléatoire pour jouer correctement. Une solution pour palier à ce problème peut être de faire jouer deux IA similaires ensemble afin de se « former » avant de la confronter à un joueur humain. Plus cette IA sera entrainée, plus elle sera dure à battre.