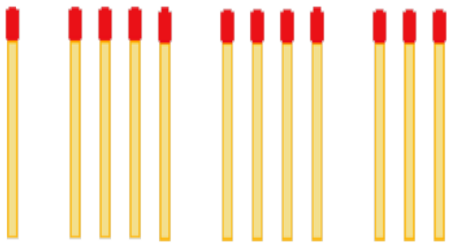
Jianghan Li

Laviolette Etienne

TP n°2 :

INTELLIGENCE ARTIFICIELLE

JEU DES ALLUMETTES



I. Objectifs du TP :

Le but ici est de modéliser un jeu bien connu du grand public : le jeu des allumettes. Notre manière d'étudier ce problème lors de ce TP illustre parfaitement la représentation logique d'un problème, son énonce et sa formalisation. Nous sommes donc au cœur de la représentation des connaissances, étape préalable obligatoire à la manipulation de celles-ci afin d'arriver à nos fins. Les cours d'IA01 enseignés par Mme. Abel et Mr. Fontaine constituent la combinaison parfaite à la résolution de ce genre de problème.

Notre cadre d'étude est le suivant :

Ce jeu se joue à deux. La situation de départ est un nombre donné d'allumettes, associé à l'indication du joueur qui doit commencer le jeu. Ensuite, libre au joueur de retirer 1, 2 ou 3 allumettes parmi celles encore restantes. Le joueur qui gagne est celui qui retire là où les dernières allumettes.

II. Formalisation du problème :

Nous allons dans tout le cadre de ce TP simuler le jeu précédemment présenté avec 5 allumettes.

a.) Liste des états possibles :

(5 1) ; (5 2) ; (4 2) ; (4 1) ; (3 2) ; (2 2) ; (3 1) ; (2 1) ; (1 1) ; (1 2) ; (0 2) ; (0 1)

Le premier nombre entre parenthèses représente le nombre d'allumettes restantes.

Le second est le joueur courant. Celui qui joue.

Par exemple : (5 1) signifie qu'il reste 5 allumettes à jouer et que c'est au joueur numéro 1 de jouer.

A noter que pour l'instant on ne sait pas quel joueur commence, donc la liste des états possibles est la plus complète possible. On restreindra ensuite celle-ci puisque dans la suite de l'étude on sait que le joueur 1 est toujours celui qui commence. A ce moment ci, les états (5 2) et (4 2) n'existeront plus.

b.) <u>Liste des opérateurs selon l'état courant :</u>

(3 1) ; (2 1) ; (1 1)

(3 2) ; (2 2) ; (2 1)

Le premier nombre est parenthèse constitue le nombre d'allumettes retirées.

Le second nous informe sur le joueur qui retire les allumettes.

On compte trois transformations possibles par joueur. En effet, les règles du jeu imposent certaines contraintes lors de la définition de ces états.

Chaque joueur doit au moins retirer une allumette lorsqu'il joue. Donc les états (2 0) et (1 0) sont à proscrire.

Ensuite chaque joueur peut retirer au maximum 3 allumettes parmi les restants. C'est pourquoi les états ayant un premier nombre entre parenthèse supérieur à 3 sont à proscrire dans la définition des transformations possibles.

A noter que l'on peut associer un état à une transformation :

(5 1) x (3 1) → (2 2).

Lorsqu'on se situe dans l'état (5 1) : il reste 5 allumettes et c'est au joueur un de jouer et qu'on applique la transformation (3 1) : retirer 3 allumettes par le joueur 1. On aboutit à l'état suivant (2 2) : il reste deux allumettes et c'est au joueur 2 de jouer.

c.) <u>Etat initial et états finals :</u>

D'après les contraintes imposées par l'énoncé notre état initial est (et ne peut être que celui-ci) :

(5 1) : il reste 5 allumettes à jouer (toutes les allumettes) et c'est au joueur 1 de jouer (c'est toujours le joueur un qui commence).

Il existe cependant deux états finals dans ce jeu : soit c'est le joueur un qui gagne, soit c'est le joueur deux.

Nous avons traduit cela de la manière suivante :

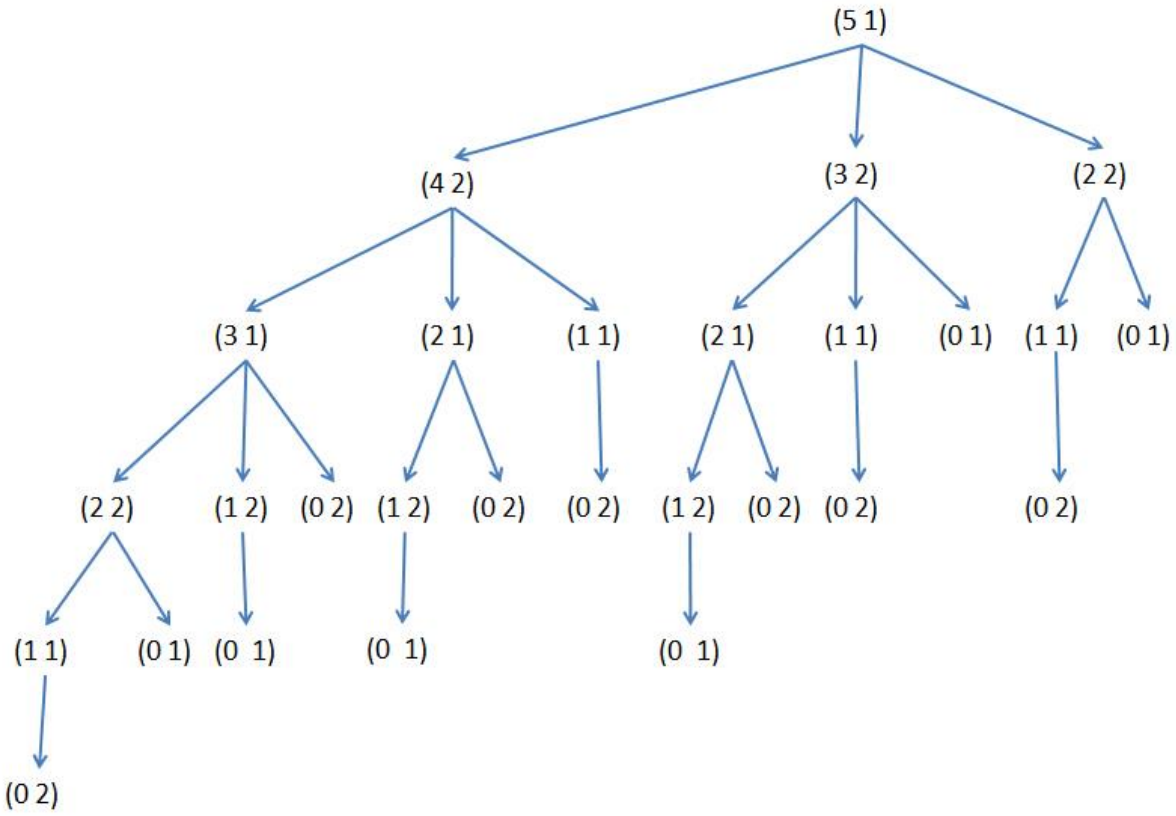
(0 1) : il reste aucune allumette et c'est au joueur numéro un de jouer, c'est donc le 2 qui a gagné.

(0 2) : il reste aucune allumette à retirer et c'est au joueur numéro deux de jouer, c'est donc le 1 qui a gagné cette partie.

d.) *Arbre de recherche à partir de l'état initial*

Voici ci-dessous, l'arbre de recherche à partir de notre état initial. Chaque nœud est un état quand chaque arc supporte la transformation permettant d'aboutir à l'état de destination de l'arc en partant depuis l'état origine de l'arc.

On peut donc écrire sous la forme d'un produit de deux éléments de R^2 nous donnant une solution dans R^2. On peut donc décrire mathématiquement ce problème comme une application de R^2*R^2 dans R^2 :



III. Fonctions lisp de recherche en profondeur et en largeur :

a.) Fonctions de service :

- La première fonction de service que nous avons jugé utile de déclarer est la suivant :

```
(defun rival (joueur)
        (cond
         (( eq joueur 1) 2)
         (( eq joueur 2) 1)
         (t nil)
        )
)
```

Cette fonction nommée « rival » permettant retourner le numéro du joueur qui sera le rival de celui entré en paramètre.

Dans notre cas le rival du joueur 1 est le numéro 2 et inversement.

Nous utiliserons cette fonction dans la fonction successeur afin d'automatiser le remplissage du numéro du joueur jouant après le joueur courant.

- La seconde fonction de service est « not-dernier » :

```
(defun not-dernier (L)
        (setq final '())
        (setq i 0)
        (while (< i (- (length L) 1) )
                (progn
                        (setq final (append final (list (nth i L))))
                        (setq i (+ 1 i))
                )
         )
        final
        )
```

Cette fonction prend une liste en paramètre et renvoie la liste entrée en paramètre sans son dernier élément.

Par exemple :

Si on entre la liste : ((5 0) (3 0) (2 0)) en paramètre, la fonction nous renvoie : ((5 0) (3 0)).

Cette fonction utilisée dans le parcours en profondeur est utile afin de « remonter » les noeuds de l'arbre. Le détail précis de son rôle se fera dans la fonction profondeur illustrée ci-après.

- La troisième est la fonction « successeur » :

```
(defun successeur (nb joueur)
        (let ((list_suc nil) (joueur2 (rival joueur)))
         (if (> nb 2) (push (list (- nb 3) joueur2) list_suc))
         (if (> nb 1) (push (list (- nb 2) joueur2) list_suc))
         (if (> nb 0) (push (list (- nb 1) joueur2) list_suc))
         list_suc)
)
```

Cette fonction prend en paramètre le nombre d'allumettes restantes et le numéro du joueur courant. Elle retourne une liste constituée de tous les successeurs possible à l'état courant.

La variable locale list_suc permettra de stocker la liste des successeurs que nous renverront à la fin de cette fonction.

Aussi les « if » sont tout à fait cumulables puisque qu'un état peut avoir plusieurs états successeurs (et au maximum trois).

Nous utilisons ici la fonction « rival » qui permet d'automatiser le changement suivant :

Lorsque l'état courant se situe sur le joueur 1 alors les successeurs auront forcément un numéro de joueurs égal à 2. Et inversement.

Par exemple :

L'appel de : (successeur 5 1) permet de renvoyer la liste des successeurs possibles de l'état initial :

((4 2) (3 2) (2 2)).

Nous avons fait le choix de passer en paramètre de cette fonction des nombre et non pas une liste comme nous aurions pu le faire. (Par exemple créer une fonction prenant en paramètre '(5 1)).

- La dernière fonction de service que nous avons écrit n'est pas à proprement parlé une fonction de service puisqu'elle n'est pas utilisé dans les fonctions profondeur ou largeur décrites ci-dessous.

Cependant elle nous a été d'une certaines utilité pour savoir s'il existait des solutions (parfaites) correspondant à un certain état initial.

En effet certaines configurations de jeu (4 allumettes et le joueur 1 qui commence.. ne permet pas au joueur 1 de gagner.. Au contraire il est sur de perdre). La fonction test_etat permet dans de savoir si dans la configuration que l'on entre en paramètre il existe une solution parfaite pour le joueur.

```
(defun test_etat (nb joueur)

  (cond
   ((eq nb 0) (rival joueur))
   ((> nb 0)  (let* ((joueur2 (rival joueur))
                     (e1 (test_etat (- nb 1) joueur2))
                     (e2 (test_etat (- nb 2) joueur2))
                     (e3 (test_etat (- nb 3) joueur2)))
              (if (or (eq joueur e1) (eq joueur e2) (eq joueur e3)) joueur joueur2)))
   (t nil)
  )
)
```

Si nous entrons : (test_etat 5 1), la fonction nous renvoie 1, il existe donc un chemin parfait pour le joueur 1 dans cette configuration de 5 allumettes au départ.

b.) Parcours en profondeur :

La première fonction qui nous était demandée était celle du parcours en profondeur de notre arbre précédemment décrit.

Pour cela nous allons nous servir des différentes fonctions de service décrites dans la sous-partie précédente.

Voici le code LISP de notre fonction de parcours en profondeur :

```
(defparameter solution nil)

(defun profondeur (etat)
        (if (equal (car (last solution)) '(0 2))
                    (progn
                       (print 'gagne)
                       (setq solution (cons '(5 1) solution))
                       (print solution)
                       (setq solution (cdr solution))
                            )
         (if (equal (car (last solution)) '(0 1)) ()
            (dolist (x (successeur (car etat) (cadr etat)) 'fin)
              (progn
                (setq solution (append solution (list x)))
                (profondeur x)
                (setq solution (not-dernier solution))
                )
            )
         )
        )
```

Ce parcours en profondeur prend en paramètre l'état initial de notre espace d'états sous la forme d'une liste : (profondeur '(5 1)) sera l'appel de la fonction.

La particularité de cette fonction est qu'elle utilise une variable speciale initialisée à nil par l'intermédiaire d'un defparameter. Cette variable speciale va permettre de stocker et d'afficher successivement les différentes solutions obtenues par le parcours en profondeur de l'arbre.

La première partie est la condition de sortie des appels récursifs lorsqu'on rencontre une solution (c.f l'état final est (0 2). Ce cas constitue le premier de parcours en profondeur « total ». On arrive au bout d'une branche en ayant gagné.

La seconde partie est le cas où on arrive au bout d'une branche en ayant perdu. Rien ne se passe dans ce cas.

Dans les cas restants on parcourt avec x les successeurs de l'état courant. On les parcourt un par un grâce à l'appel récursif sur chacun d'entre eux.

Cette fonction nous permet de trouver toutes les solutions du parcourt en profondeur.

On notera que c'est le code de la fonction « successeur » qui détermine comment on parcourt les nœuds de l'arbre en profondeur.

On note aussi que si notre joueur 1 n'enlève qu'une allumette. Quel que soit l'action de l'adversaire, le joueur un gagnera obligatoirement (avec un minimum de réflexion).

Le point critique pour afficher toutes les solutions fut d'ajouter cette ligne dans le dolist :

(setq solution (not-dernier solution))

Afin de continuer notre parcours sur la branche précédente. (cette ligne nous permet de remonter aussi haut dans l'arbre une fois qu'on est arrivé au bout d'une branche).

Cette deuxième version permet de savoir s'il existe un chemin en profondeur pour l'état initial rentré sans l'afficher :

```
(defun explore-depth (nb joueur)
  (push (list nb joueur) states-passes)
  (if (equal (list nb joueur) state-final)
      t
    (let ((ens-states (successor nb joueur)))
      (if (null ens-states)
          nil
        (loop
          (setq etat_courant (pop ens-states))
          (if (explore-depth (car etat_courant) (cadr etat_courant))
                  (return t )
            (if (null ens-states) (return nil)))
          )))))
```

Ici les variables spéciales « state-final » et « ens-state » doivent être initialisées respectivement à '(0 2) et nil. La fonction renvoit T s'il existe une solution en parcourt en profondeur.

c.) Parcours en largeur :

Le parcours en largeur dans cet espace d'état prend en paramètre l'état initial sous forme de liste, tout comme le parcours en profondeur.

La gestion de ce parcours se fait grâce à une file.

A chaque nœud parcouru, on enfile la liste de ses successeurs dans la file que constitue solution.

Si le dernier élément de la liste solution est la sous liste (0 2) alors on est arrivé à une solution grâce à a ce parcours en largeur.

L'enfilage des successeurs du nœud courant se fait grâce à un dolist sur les successeurs du nœud courant. On défile ensuite le nœud qui vient d'être traité grâce à un pop sur solution et on appel récursivement la fonction largeur sur le premier élément de la file.

Nous nous sommes heurtés à un problème : comment se souvenir des nœuds parcourus dans la file pour afficher la solution. Nous avons essayés de créer une liste intermédiaire qui sera construite de sorte de dénombre le nombre de successeurs de chaque nœuds.

(3 3 3 2 3)

Le premier nœud (état initial) contient trois successeurs, d'où le premier 3. Puis le premier successeur de l'état initial a aussi 3 successeurs, tout comme le second (donc 3 3 3). Par contre le troisième successeur n'a que deux successeurs (3 3 3 2 3). Le dernier trois nous indique que le premier successeur du premier successeur de l'état initial a trois successeurs.

Malheureusement cette méthode s'est révélée infructueuse et nous n'avons pas été capables d'afficher les solutions du parcours en largeur.

Voici le code de notre fonction

```
(defun largeur (etat)
        (if (equal (car (last solution)) '(0 1))
          (progn
            (print 'gagne))
         (progn
          (dolist (x (successeur1 (car etat) (cadr etat)) 'fin)
            (setq solution (append solution (list x))))
          (pop solution)
```

```
    (largeur (car solution))
  )
  )
 )
```

Conclusion :

Ce TP nous a permis de modéliser et de formaliser un problème donné. Nous avons donc suivi le processus indiqué dans le TP : comment modéliser un état, dresser la liste des états possibles, les actions, l'état initial, les états finals… Pour ensuite déterminer les fonctions de services nécessaires à la résolution du problème à proprement parlé. Ce TP a mis en lumière l'importance de la phase préliminaire d'étude du problème. Il ne faut pas négliger la recherche en algorithmes. Bien qu'effectuée au brouillon et non présente dans ce rapport, elle a eu une importance capitale dans notre travail.