Compte rendu TP2

Problème

L'objectif de ce TP est de résoudre le jeu des allumettes grâce à une recherche de parcours dans un ensemble d'états.

Etats possibles dans une partie de 11 allumettes

On choisit la représentation suivante pour les états : (Nombre d'allumettes restantes / Prochain joueur à jouer)

On suppose que le joueur 1 commence et que le jeu se joue avec 11 allumettes initiales.

Liste des états possibles pour 11 allumettes (le joueur 0 commence) :

(11 1)	(7 1)	(4 1)	(1 0)
(10 0)	(6 1)	(3 1)	(0 0)
(9 0)	(5 1)	(2 1)	(1 1)
(8 0)	(7 0)	(4 0)	(0 1)
(9 1)	(6 0)	(3 0)	
(8 1)	(5 0)	(2 0)	

Liste des opérateurs

On choisit la représentation suivante pour les opérateurs : (Joueur / Nombre d'allumettes prélevées).

Liste des actions possibles :

(1 1) (1 2) (1 3) (0 1) (0 2) (0 3)

Actions possibles selon l'état:

Si l'état est (x 0) avec x >= 3, les actions possibles sont (0 1) (0 2) (0 3)

Si l'état est (2 0), les actions possibles sont (0 1) (0 2)

Si l'état est (1 0), la seule action possible est (0 1)

Si l'état est (x 1) avec x >= 3, les actions possibles sont (1 1) (1 2) (1 3)

Si l'état est (2 1), les actions possibles sont (1 1) (1 2)

Si l'état est (1 1), la seule action possible est (1 1)

Si l'état est (0 0) ou (1 0), il n'y a aucune action possible

Etats initial et finaux

Sachant que le joueur 1 commence, l'état initial est : (11 1)

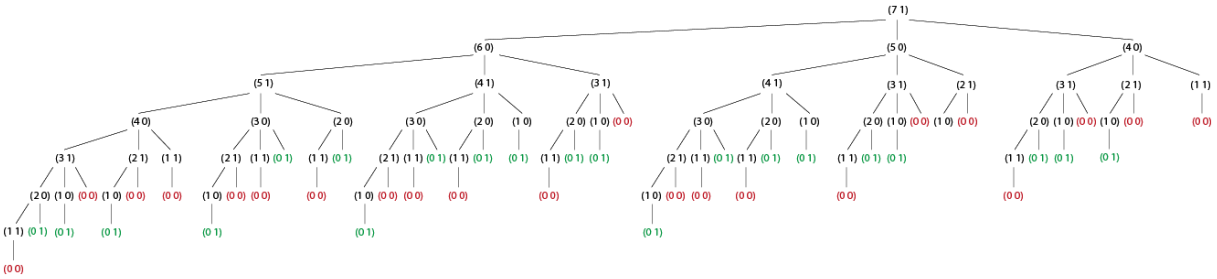
Etats finaux :

(0 0) : il ne reste plus d'allumettes, le joueur 1 a gagné

(0 1) : il ne reste plus d'allumettes, le joueur 0 a gagné

Arbre de recherche

Pour plus de facilité, on représentera l'arbre de recherche pour une partie disposant de 7 allumettes, l'état initial est donc (7 1).



Fonctions de service

On définit une liste *actions* comprenant toutes les actions détaillées précédemment.

On définit d'abord des fonctions de services permettant de renvoyer le prochain joueur selon l'état, le nombre d'allumettes qu'il reste selon l'état, le joueur qui joue selon l'action, le nombre d'allumettes à prélever selon l'action. Ces fonctions permettront un code plus lisible pour les fonctions suivantes.

La fonction `action_possibles` parcourt une à une toutes les actions de la liste *actions* et renvoie la liste des actions applicables sur un état donné en fonction des règles que nous avons évoquées précédemment.

La fonction `etat_gagnant` renvoie l'état final pour que le joueur x passé en paramètres gagne.

Enfin la fonction `successeurs_possibles` prend en argument un état et, grâce aux fonctions précédentes, renvoie la liste de tous les états qui découlent de l'état donné en appliquant chacune des actions disponibles.

La fonction `jouer` applique simplement une action donnée à un état donné. On revérifie d'abord que l'action est bien applicable, puis on soustrait le nombre d'allumettes voulu et on change de joueur. On renvoie l'état ainsi obtenu.

La fonction `affichage` nous permet d'afficher les allumettes enlevées à chaque moment du parcours passé en paramètres.

On vérifie bien que chaque fonction renvoie correctement ce que l'on souhaite.

Pour les différents parcours, nous avons réalisé plusieurs fonctions : un parcours en profondeur qui renvoie l'ensemble de tous les chemins menant au gagnant recherché, un second parcours en profondeur qui renvoie le premier chemin trouvé menant au gagnant recherché et enfin un parcours en largeur qui renvoie lui-aussi le premier parcours menant au gagnant recherché.

Parcours en profondeur

parcours_profondeur_tous

Cette fonction parcourt l'arbre en profondeur et renvoie tous les chemins trouvé menant au gagnant passé en paramètre.

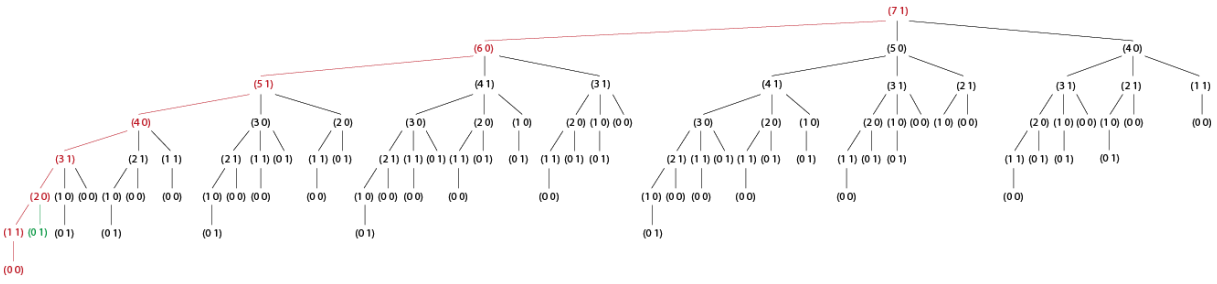
```
(defun parcours_profondeur_tous(etat parcours gagnant)
	(let ((g (etat_gagnant gagnant))) ;; on affecte a g l'etat final pour que le gagnant gagne
		(push etat parcours) ;; on push dans parcours l'etat actuel comme on vient de le visiter
		(cond
			((EQUAL etat g)(print (reverse parcours))) ;; si l'etat actuel correspond a notre etat final souhaite on affiche le parcours
			(T
				(let ((succ (successeurs_possibles etat))) ;; dans tous les cas, on continue d'etudier les successeurs de l'etat
					(dolist (xx succ)
						(parcours_profondeur_tous xx parcours gagnant) ;; on appelle pour chaque successeur la fonction
					)
				)
			)
		)
	)
)
```

La fonction renvoie bien ce qu'on attend, pour ne pas avoir trop de ligne, on a testé pour l'état initial (7 1) :

```
CG-USER(59): (parcours_profondeur_tous '(7 1) NIL 1)

((7 1) (6 0) (5 1) (4 0) (3 1) (2 0) (1 1) (0 0))
((7 1) (6 0) (5 1) (4 0) (3 1) (0 0))
((7 1) (6 0) (5 1) (4 0) (2 1) (0 0))
((7 1) (6 0) (5 1) (4 0) (1 1) (0 0))
((7 1) (6 0) (5 1) (3 0) (2 1) (0 0))
((7 1) (6 0) (5 1) (3 0) (1 1) (0 0))
((7 1) (6 0) (5 1) (2 0) (1 1) (0 0))
((7 1) (6 0) (4 1) (3 0) (2 1) (0 0))
((7 1) (6 0) (4 1) (3 0) (1 1) (0 0))
((7 1) (6 0) (4 1) (2 0) (1 1) (0 0))
((7 1) (6 0) (3 1) (2 0) (1 1) (0 0))
((7 1) (6 0) (3 1) (0 0))
((7 1) (5 0) (4 1) (3 0) (2 1) (0 0))
((7 1) (5 0) (4 1) (3 0) (1 1) (0 0))
((7 1) (5 0) (4 1) (2 0) (1 1) (0 0))
((7 1) (5 0) (3 1) (2 0) (1 1) (0 0))
((7 1) (5 0) (3 1) (0 0))
((7 1) (5 0) (2 1) (0 0))
((7 1) (4 0) (3 1) (2 0) (1 1) (0 0))
((7 1) (4 0) (3 1) (0 0))
((7 1) (4 0) (2 1) (0 0))
((7 1) (4 0) (1 1) (0 0))
```

parcours_profondeur_premier

En fonction du gagnant passé en paramètre le parcours renvoyé est :




```
(defparameter sol 0) ;; on definit un parametre sol a initialiser des que l'on veut appeler la fonction parcours_profondeur_premier

(defun parcours_profondeur_premier(etat parcours gagnant)
	(let ((g (etat_gagnant gagnant))) ;; on affecte a g l'etat final pour que le gagnant gagne
		(push etat parcours) ;; on push dans parcours l'etat actuel comme on vient de le visiter
		(cond
			((= sol 1) NIL) ;; si sol est a 1 on arrete car le premier parcours a deja ete trouve
			((EQUAL etat g) (affichage (reverse parcours)) (setq sol 1))
			;; si l'etat actuel correspond a notre etat final souhaite on affiche le parcours et on modifie le parametre sol a 1
			(T
				(let ((succ (successeurs_possibles etat))) ;; sinon on observe les successeurs de l'etat
					(dolist (xx succ)
						(parcours_profondeur_premier xx parcours gagnant) ;; et on appelle la fonction sur les successeurs
					)
				)
			)
		)
	)
)
```

La fonction renvoie bien ce qu'on attend pour l'état initial (7 1) :

```
CG-USER(89): (parcours_profondeur_premier '(7 1) NIL 1)
Il y a 7 allumettes, c'est au joueur 1 de jouer
| | | | | | |
o o o o o o o
Il a enleve 1 allumettes

Il y a 6 allumettes, c'est au joueur 0 de jouer
| | | | | |
o o o o o o
Il a enleve 1 allumettes

Il y a 5 allumettes, c'est au joueur 1 de jouer
| | | | |
o o o o o
Il a enleve 1 allumettes

Il y a 4 allumettes, c'est au joueur 0 de jouer
| | | |
o o o o
Il a enleve 1 allumettes

Il y a 3 allumettes, c'est au joueur 1 de jouer
| | |
o o o
Il a enleve 1 allumettes

Il y a 2 allumettes, c'est au joueur 0 de jouer
| |
o o
Il a enleve 1 allumettes

Il y a 1 allumettes, c'est au joueur 1 de jouer
|
o

Le joueur 1 a enleve les dernieres allumettes, il a gagne
```

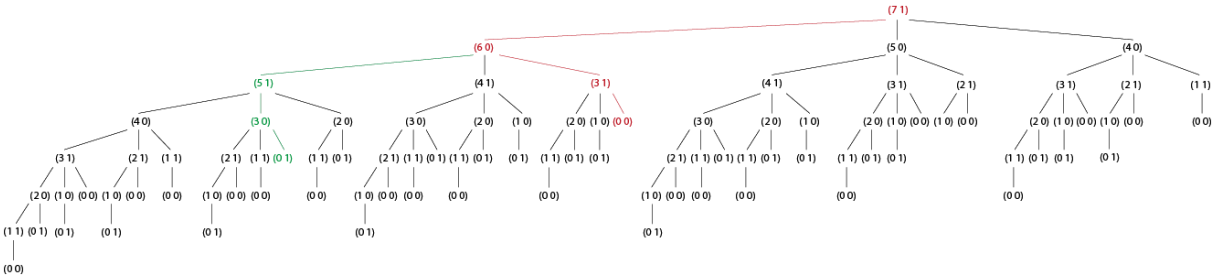
La fonction renvoie bien ce qu'on pourrait attendre pour l'état initial (11 1) :

```
CG-USER(58): (parcours_profondeur_premier '(11 1) NIL 1)
Il y a 11 allumettes, c'est au joueur 1 de jouer
| | | | | | | | | | |
o o o o o o o o o o o
Il a enleve 1 allumettes

Il y a 10 allumettes, c'est au joueur 0 de jouer
| | | | | | | | | |
o o o o o o o o o o
Il a enleve 1 allumettes

Il y a 9 allumettes, c'est au joueur 1 de jouer
| | | | | | | | |
o o o o o o o o o
Il a enleve 1 allumettes

Il y a 8 allumettes, c'est au joueur 0 de jouer
| | | | | | | |
o o o o o o o o
Il a enleve 1 allumettes

Il y a 7 allumettes, c'est au joueur 1 de jouer
| | | | | | |
o o o o o o o
Il a enleve 1 allumettes

Il y a 6 allumettes, c'est au joueur 0 de jouer
| | | | | |
o o o o o o
Il a enleve 1 allumettes

Il y a 5 allumettes, c'est au joueur 1 de jouer
| | | | |
o o o o o
Il a enleve 1 allumettes

Il y a 4 allumettes, c'est au joueur 0 de jouer
| | | |
o o o o
```

```
Il a enleve 1 allumettes

Il y a 3 allumettes, c'est au joueur 1 de jouer
| | |
0 0 0
Il a enleve 1 allumettes

Il y a 2 allumettes, c'est au joueur 0 de jouer
| |
0 0
Il a enleve 1 allumettes

Il y a 1 allumettes, c'est au joueur 1 de jouer
|
0

Le joueur 1 a enleve les dernieres allumettes, il a gagne
```

Parcours en largeur

En fonction du gagnant passé en paramètres, le parcours renvoyé est :



Pour faciliter la fonction `parcours_largeur`, on crée deux nouvelles fonctions de services. L'une pour vérifier si l'état final fait partie de la liste des chemins. L'autre pour créer une liste de chemins découlant du chemin passé en paramètre. En effet, on gérera une file pour parcourir l'arbre en largeur, tant que l'état final ne fait pas parti des chemins de la file, on remplace chaque chemin par les chemins d'un niveau supérieur. Par exemple, pour l'état initial (7 1), les premiers passages dans la boucle réalisent :

((7 1))

(((6 0) (7 1)) ((5 0) (7 1)) (4 0) (7 1)))

Et ainsi de suite…

```
(defun end_in_file (end file)
        (dolist (xx file) ;; pour chaque chemin
                (dolist (yy xx) ;; pour chaque etat
                    (if (equal yy end) (return-from end_in_file xx))
                    ;; si l'etat est egal a l'etat final, on retourne le chemin contenant cet etat final
                )
        )
)

(defun succ_list (chemin)
        (let ((result NIL) (succs (successeurs_possibles (car chemin)))) ;; on recupere les successeurs du premier etat du chemin
                (dolist (xx succs) ;; pour chaque successeurs
                        (push (append (list xx) chemin) result) ;; on push un nouveau chemin auquel on ajoute le successeur
                )
                (reverse result) ;; on renvoie la liste des chemins avec les successeurs ajoutes mais en reverse pour correspondre a l'arbre
        )
)

(defun parcours_largeur (start gagnant)
        (let ((file (list (list start))) (g (etat_gagnant gagnant))) ;; on affecte a g l'etat final pour que le gagnant gagne
                (loop while (null (END_IN_FILE g file)) ;; tant que l'etat final n'est pas dans la file
                        do
                                (let ((new_file NIL)) ;; on cree une nouvelle file contenant tous les chemins decoulant des chemins de l'ancienne file
                                        (dolist (xx file) (setq new_file (append (SUCC_LIST xx) new_file)))
                                        (setq file new_file)
                                )
                )
                (affichage (reverse (END_IN_FILE g file))) ;; on affiche le chemin trouve contenant l'etat final
        )
)
```

La fonction renvoie bien ce qu'on attend pour l'état initial (7 1) :

```
CG-USER(84): (parcours_largeur '(7 1) 1)
Il y a 7 allumettes, c'est au joueur 1 de jouer
| | | | | | |
0 0 0 0 0 0 0
Il a enleve 1 allumettes

Il y a 6 allumettes, c'est au joueur 0 de jouer
| | | | | |
0 0 0 0 0 0
Il a enleve 3 allumettes

Il y a 3 allumettes, c'est au joueur 1 de jouer
| | |
0 0 0

Le joueur 1 a enleve les dernieres allumettes, il a gagne
```

La fonction renvoie aussi bien ce qu'on attendrait pour l'état initial (11 1) :

```
CG-USER(85): (parcours_largeur '(11 1) 1)
Il y a 11 allumettes, c'est au joueur 1 de jouer
| | | | | | | | | | |
0 0 0 0 0 0 0 0 0 0 0
Il a enleve 1 allumettes

Il y a 10 allumettes, c'est au joueur 0 de jouer
| | | | | | | | | |
0 0 0 0 0 0 0 0 0 0
Il a enleve 1 allumettes

Il y a 9 allumettes, c'est au joueur 1 de jouer
| | | | | | | | |
0 0 0 0 0 0 0 0 0
Il a enleve 3 allumettes

Il y a 6 allumettes, c'est au joueur 0 de jouer
| | | | | |
0 0 0 0 0 0
Il a enleve 3 allumettes

Il y a 3 allumettes, c'est au joueur 1 de jouer
| | |
0 0 0

Le joueur 1 a enleve les dernieres allumettes, il a gagne
```

Conclusion

Par choix, nous avons souhaité réaliser à la fois le parcours en profondeur et le parcours en largeur de l'arbre de recherche. Cependant, on observe bien que le parcours en largeur mène plus rapidement à la victoire que le parcours en profondeur.