IA01 Projet
TP2

Jeu des Allumettes

Remarque:

Les règles de ce jeu varient. On a donc choisi pour ce TP de prendre la règle suivante:

"Le Joueur gagnant est celui qui prend la dernière allumette"

1) États possibles dans un jeu de 5 allumettes

Format : Joueur - allumettes restantes

J1-5	J1-1	J1-0
J2-4	J1-2	
J2-3	J1-3	
J2-2	J2-0	

2) Liste des opérateurs

Les opérateurs possibles dans ce jeu sont:

- J1 - Retirer 1 allumette
- J1 - Retirer 2 allumettes
- J1 - Retirer 3 allumettes
- J2 - Retirer 1 allumette
- J2 - Retirer 2 allumettes
- J2 - Retirer 3 allumettes

On peut les classer selon leur possibilité a être affectés, c'est-à-dire, en précisant des règles de jeu.

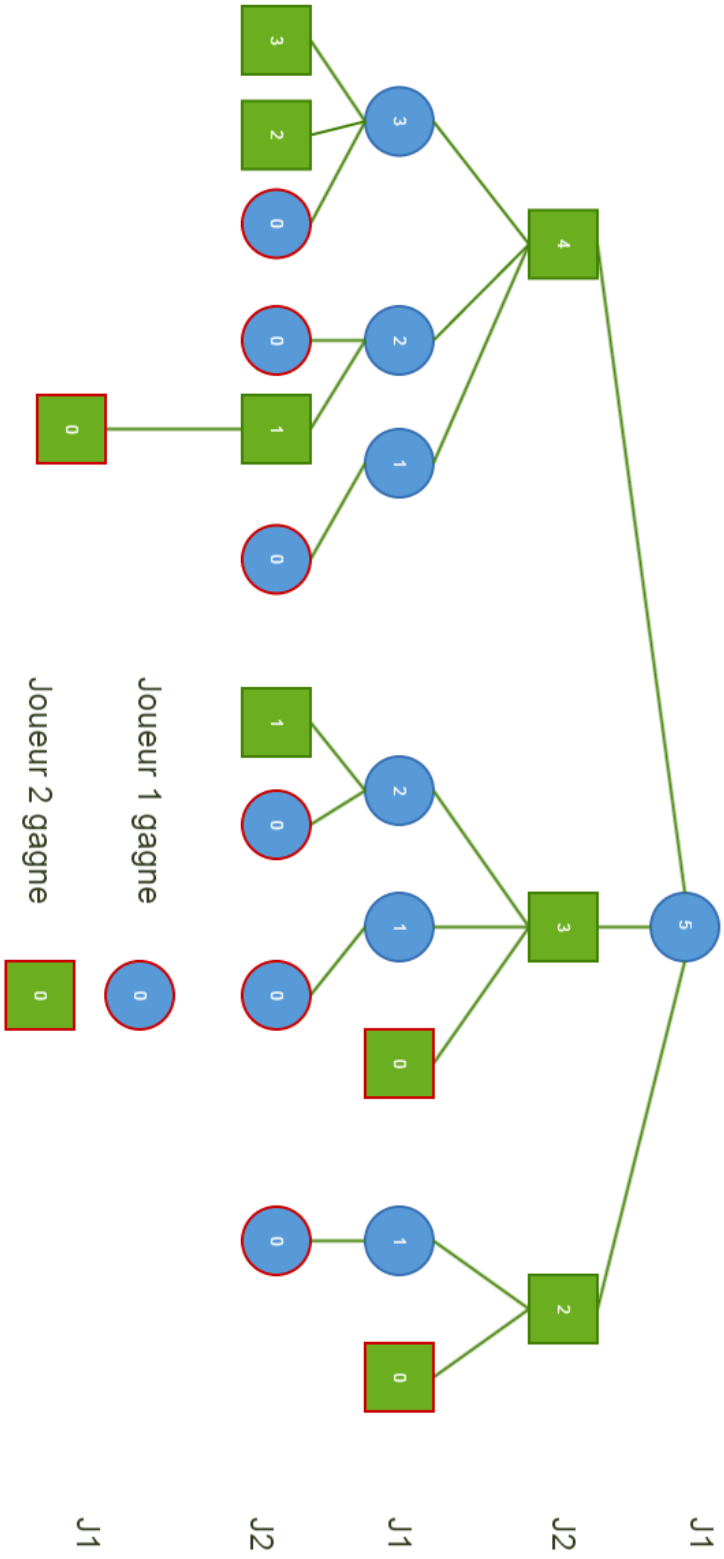
On a donc par exemple l'état (J2 - 2) auquel on ne peut pas affecter les opérateurs [J1 - R1], [J1 - R2], [J1 - R3] (étant donné que c'est le tour de J2), ni l'opérateur [J2 - R3] car il n'y a que deux allumettes disponibles.

3) États initiaux et finaux

Si la partie commence avec le Joueur 1 , l'état initial est (J1 - X), ou X représente le nombre initial d'allumettes, dans le cas de la question 1, 5 allumettes.

Les états finaux peuvent donc être (J2 - 1) et (J1 - 1) en dépendant des mouvements effectués par chacun, sachant que s'il reste une seule allumette, c'est le joueur qui a le tour qui gagne.

4)Arbre de recherche



5) Fonctions Lisp

a) Exploration en profondeur

Choix :

Notre programme est une “intelligence artificielle” (que nous appellerons J1 ou IA) contre laquelle joue un joueur réel. Ce programme choisi quel coup jouer au tour présent.

Nous avons choisi de faire parcourir à notre programme la liste complète des états possibles dès le lancement du jeux. La fonction `creerListe` explore le graphe des possibilités pour génèrer une liste contenant toutes les informations voulues selon la représentation suivante :

```
((sommetDuSousArbre joueurActuel conclusionDuSousGraphe)
(sousArbreSi1AllumetteRetirée)(sousArbreSi2AllumettesRetirées)(sousArb
reSi3AllumettesRetirées))
```

Avec `conclusionDuSousGraphe` qui est :

- 2 : IA gagne forcément dans ce sous-arbre. C’est à dire que si l’IA joue ce qui lui est conseillé et quelque soit ce que l’adversaire fait, IA gagnera.
- ou bien 1 : il est possible de gagner ou de perdre dans ce sous-arbre. Tout dépend de ce que fait l’adversaire mais il existe tout de même un sous-arbre dans celui-ci qui est de type 2.
- ou alors 0 : sauf erreur de la part de l’adversaire, IA perd cette partie.

Par exemple, si la partie commence avec 20 allumettes et que c’est au joueur 1 de jouer, on aura pour liste d’états :
((20 J1 1) (ssArbre19J2) (ssArbre18J2) (ssArbre17J2))

À la fin, on aura un arbre comme ceci :
((2 J2 1) ((1 J1 2) (0 J1 2)) (0 J2 0))

Après réflexion, on va simplifier la fin : dès qu’il reste moins de trois allumettes, on conclut (car le joueur qui va jouer va retirer toutes les allumettes).

On a fait une fonction `conclusion` qui prend en paramètre le joueur qui doit prendre une décision dans l’état actuel ainsi que la liste des conclusions des 3 sous-arbres maximum. Cette fonction retourne la conclusion tirée. Par exemple :
“Si 2 des sous arbres permettent à l’IA de gagner (i.e. ils ont tous une conclusion 2) et que le troisième sous arbre a une conclusion incertaine (i.e. il a pour conclusion 1) on a deux cas : soit la décision revient à l’IA, dans ce cas il va choisir un arbre 2, donc on

retourne un conclusion 2. Si la décision revient à l'adversaire, on ne peut pas savoir ce qu'il va jouer donc on retourne 1."
Cet exemple est dans la ligne 2 du tableau récapitulatif suivant :

Etats ↓ Décision→	J1 / IA	J2
2 2 2	2	2
2 2 1	2	1
2 2 0	2	0
2 1 1	2	1
2 1 0	2	0
2 0 0	2	0
1 1 1	1	1
1 1 0	1	0
1 0 0	1	0
0 0 0	0	0

Tableau de la conclusion à donner pour un certain état en fonction des conclusion des sous-arbres

Fonctions :

(chaque fonction dépend des précédentes)

- `switchJ (joueur)` : fonction très simple retournant J1 si joueur=J2 et J2 si joueur=J1
- `conclusion (joueur liste1 liste2 liste3)` qui a déjà été expliquée.
- `creerListe (joueur nbAllu)` : cette fonction retourne une liste contenant l'analyse complète de l'arbre des possibilités en profondeur. Attention, cette fonction a une complexité en $O(3^n)$ donc très mauvaise ! Nous avons donc limité le nombre d'allumettes possibles à 20. On aurait pu mettre un peu plus (25 par exemple) mais par exemple à 50 il faudrait 9 100 millénaires pour calculer les possibilités à 2.5 Ghz ! De plus il faudrai considérer la complexité spatiale pour

un nombre aussi grand. Avec 20 allumettes (en donc complexité en 3^{17}, vu qu'on gère les 3 dernières allumettes à la main), on fait le calcul en 0.05 s.

- `afficherListe (liste)` : fonction d'affichage de la liste retournée par `creerliste`, avec indentation et analyse de l'état final. Elle utilise la fonction `printTree (liste prefixe)` pour la partie récursive.
- `allumettesRestantes (possibilite)` : cette petite fonction retourne le nombre d'allumettes actuellement disponibles. Elle ne calcule rien en réalité, `creerliste` l'a déjà fait donc on retourne le `car` du `car` de l'arbre de possibilités où ce nombre est enregistré.
- `getSsArbre (possibilite conclusion)` : retourne le sous-arbre de l'état actuel ayant la conclusion voulue.
- `jeuAllumettes ()` : fonction de jeu, qui permet à l'utilisateur de choisir qui commence et le nombre d'allumettes qu'il y a au départ. Dans un second temps, on calcule l'arbre des possibilités avec creerliste. Ensuite, une boucle s'occupe de faire jouer l'utilisateur contre l'ordinateur et à la fin on donne le gagnant.

Code :

```
(defun switchJ (joueur) ; retourne l'autre joueur que joueur
	(cond
		((eq joueur 'J1) (return-from switchJ 'J2))
		((eq joueur 'J2) (return-from switchJ 'J1))
	)
)

(defun conclusion (joueur liste1 liste2 liste3)
	(let
		((c1 ()) (c2 ()) (c3 ()) )

		; on récupère les conclusions des sous-listes
		(if (listp (car liste1))
			(setq c1 (caddr (car liste1)))
			(setq c1 (caddr liste1))
			)
		(if (listp (car liste2))
			(setq c2 (caddr (car liste2)))
			(setq c2 (caddr liste2))
			)
		(if (listp (car liste3))
			(setq c3 (caddr (car liste3)))
			(setq c3 (caddr liste3))
			)
```

```
            ; on conclut en fonction des conclusions des sous arbres
            (cond
                  ((eq 0 (+ c1 c2 c3)) (return-from conclusion 0))
                  ((eq 6 (+ c1 c2 c3)) (return-from conclusion 2))
                  ((and (eq joueur 'J1) (>= (+ c1 c2 c3) 3) (not (eq 1 (* c1 c2
c3)))) (return-from conclusion 2))
                  ((and (eq joueur 'J1) (eq 0 (* c1 c2 c3)) (eq 2 (+ c1 c2 c3)) (or
(eq 2 c1) (eq 2 c2) (eq 2 c3))) (return-from conclusion 2))
                  ((and (eq joueur 'J2) (eq 0 (* c1 c2 c3))) (return-from conclusion
0))
                  (T (return-from conclusion 1))
            )
      )
)


(defun creerListe (joueur nbAllu)
        (let (
            (concl NIL)
            (liste1 NIL)
            (liste2 NIL)
            (liste3 NIL)
            (nbAlluIni nbAllu)
            (joueurSuivant (switchJ joueur))
            );fin var let

        (cond
            ((and (eq joueur 'J1)(<= nbAllu 3)(>= nbAllu 1)) ; si on est dans un état
final : l'IA gagne
                  (setq concl 2) ; c'est un cas favorable du point de vue de
l'ordinateur !
                  (setq nbAllu 0)
                  (return-from creerListe
                      (list nbAllu joueurSuivant concl) ; on retourne (0 J2 2)
                  )
            )

            ((and (eq joueur 'J2)(<= nbAllu 3)(>= nbAllu 1))  ; si on est dans un état
final : le joueur gagne
                  (setq concl 0) ; c'est un cas défavorable du point de vue de
l'ordinateur
                  (setq nbAllu 0)
                  (return-from creerListe
                      (list nbAllu joueurSuivant concl) ; on retourne (0 J1 0)
                  )
            )

            (T (let ( ;si on est dans le cas normal (il reste plus de 3 allumettes)
```

```
                    (liste1 (creerListe joueurSuivant (- nbAllu 1)))
                    (liste2 (creerListe joueurSuivant (- nbAllu 2)))
                    (liste3 (creerListe joueurSuivant (- nbAllu 3)))
                    (concl ())
                    )
                    ; on tire la conclusion des 3 sous arbres qu'on vient d'explorer
                    (setq concl (conclusion joueur liste1 liste2 liste3))
                    (return-from creerListe
                        (list (list nbAlluIni joueur concl) liste1 liste2 liste3)
                    )
                ) ; fin let
            );fin default
        );fin cond
    ) ; fin let
) ; fin fonction


(defun allumettesRestantes (possibilite)
    (if (listp (car possibilite)) ; si on nous donne une liste de possibilités
        (return-from allumettesRestantes (car (car possibilite)))
        (return-from allumettesRestantes (car possibilite)) ; si on nous donne juste un
état (final)
    )
)

(defun getSsArbre (possibilite conclusion)
    (let
        (
            (nbAllumALaisser NIL) ; car du sous-arbre avec la conclusion voulue
        )
        (dolist (ssArbre (cdr possibilite)) ; pour chaque sous arbre possible, on
regarde la conclusion
            (if (not (LISTP (car ssArbre))) ;
                (if (eq (car ssArbre) conclusion)
                    (setq nbAllumALaisser (car ssArbre)) ; si c'est la conclusion
attendue, on retourne ce sous-arbre
                )
                (if (eq (caddr (car ssArbre)) conclusion)
                    (setq nbAllumALaisser (car ssArbre)) ; si c'est la conclusion
attendue, on retourne ce sous-arbre
                )
            )
        )
        (return-from getSsArbre (assoc nbAllumALaisser possibilite))
    )
)
```

```
(defun jeuAllumettes ()
   (let ((reponse 'J0) ; le joueur qui commence (ORDI / MOI)
         (aQui2Jouer 'J0) ; la "personne" a qui c'est de jouer (J1 / J2)
         (nbAllum 5) ; le nombre d'allumettes au départ
         (possibilite ()) ; l'arbre des possibilités
         (repAllum 0) ; le nb d'allumettes que l'utilisateur veut retirer
         (tempsDattente 2) ; durée des sleep
         (onSestDejaMoque NIL) ; pour ne se moquer de l'utilisateur qu'une seule fois
         )

         (print "Jeu des allumettes")
         (print "Qui commence ? ORDI/MOI")
         (setq reponse (read-line))

         (loop while (and (not (equal reponse "ORDI")) (not (equal reponse "MOI"))) ;
blindage
             do (print "Mauvaise réponse ! ORDI/MOI")
             (setq reponse (read-line))
         )

         (print "Combien d'allumettes ? (entre 5 et 20)")
         (setq nbAllum (parse-integer (read-line)))
         (loop while (or (> nbAllum 20) (< nbAllum 5)) ; blindage
             do (setq nbAllum (parse-integer (read-line)))
         )

         (print "Calcul de l'arbre possibilités...")
         (if (equal reponse "ORDI")
             (setq possibilite (creerListe 'J1 nbAllum))
             (setq possibilite (creerListe 'J2 nbAllum))
         )

         (sleep tempsDattente)
         ; (print possibilite)

         (cond
            ((equal reponse "ORDI") (setq aQui2Jouer 'J1))
            ((equal reponse "MOI") (setq aQui2Jouer 'J2))
            )

         (print "Début du jeu !")
         (sleep tempsDattente)

         (loop while (not (eq (allumettesRestantes possibilite) 0)) ; tant qu'il reste
des allumettes en jeu
             do (progn
                 (print "Nombre d'allumettes restantes :")
                 (print (allumettesRestantes possibilite))
```

```
                (sleep tempsDattente)
                (if (eq aQui2Jouer 'J1)
                    (progn ; dans le cas où c'est à l'ordi de jouer
                        ; (print "Ordi de jouer")
                        (cond ; l'ordinateur prend sa décision (nombre d'allumettes à
retirer)
                            ((equal (cdar possibilite) (list 'J1 2)) ; si on sait qu'on
va gagner
                                (if (not onSestDejaMoque) ; on affiche ce message une
seule fois
                                    (progn
                                        (print "Je vais gagner ! Héhéhé ^^")
                                        (sleep tempsDattente)
                                        (setq onSestDejaMoque T)
                                    )
                                )
                                ; on choisit le sous-arbre où on est sûr de gagner
                                ; donc on retire (nbd'allumettesrestantes
-nbd'alumetteesàlaisserpourquelecassoitfavorable)
                                (setq repAllum (- (allumettesRestantes possibilite)
(allumettesRestantes (getSsArbre possibilite 2))))
                            )
                            ((equal (cdar possibilite) (list 'J1 1))
                                ; (print "Bon bah on va bien voir...")
                                ; si il y a un sous arbre où on gagne on le prend,
sinon on joue un pour gagner du temps
                                (if (getSsArbre possibilite 2)
                                    (setq repAllum (- (allumettesRestantes possibilite)
(allumettesRestantes (getSsArbre possibilite 2))))
                                    (setq repAllum 1)
                                )
                            )
                            ((equal (cdar possibilite) (list 'J1 0))
                                ; (print "Je vais perdre ! Noooooooon !!!")
                                (setq repAllum 1)
                            )
                        )
                        (if (or (> repAllum 3) (< repAllum 1)) ; si le nombre
d'allumettes choisi par l'ordi n'est pas correct
                            (print "Erreur de fonctionnement, l'ordinateur essaye de
tricher !")
                            )

                        ; on va éliminer les parties de l'arbre de possibilité qui ne
sont plus utiles
                        (pop possibilite)
                        (cond ; selon le nombre d'allumettes retirées
                            ((eq repAllum 1)
```

```
                                (print "Je retire une allumette")
                                (setq possibilite (car possibilite)))
                            ((eq repAllum 2)
                                (print "Je retire deux allumettes")
                                (setq possibilite (cadr possibilite)))
                            ((eq repAllum 3)
                                (print "Je retire trois allumettes")
                                (setq possibilite (caddr possibilite)))
                        )
                        (setq aQui2Jouer (switchJ aQui2Jouer)) ; on change le joueur
                    )
                    ( ; dans le cas où c'est à l'utilisateur de jouer
                        if (> (allumettesRestantes possibilite) 3)
                            (progn
                                ; (print "utilisateur de jouer")

                                (print "Combien d'allumettes voulez vous retirer ?
(entre 1 et 3)")
                                (setq repAllum (parse-integer (read-line)))
                                (when (or (> repAllum 3) (< repAllum 1)) ; blindage
                                    (setq repAllum (parse-integer (read-line)))
                                )

                                ; on va éliminer les parties de l'arbre de possibilité
qui ne sont plus utiles
                                (pop possibilite)
                                (cond ; sur le nombre d'allumettes retirées
                                    ((eq repAllum 1) (setq possibilite (car
possibilite)))
                                    ((eq repAllum 2) (setq possibilite (cadr
possibilite)))
                                    ((eq repAllum 3) (setq possibilite (caddr
possibilite)))
                                )
                                (setq aQui2Jouer (switchJ aQui2Jouer))
                            )
                    )
                ) ; fin if des utilisateurs
            ) ; fin progn
            ; (print possibilite)
        ) ; fin loop

        ; le jeu est terminé
        (cond ; sur le résultat à la fin
            ((equal possibilite '(0 J1 0))
                (print "Il reste moins de 3 allumettes !")
                (sleep tempsDattente)
                (print "On considère que vous les retirez toutes")
```

```
                (print "donc vous avez gagné, bravo !")
                )
            ((equal possibilite '(0 J2 2))
                (print "Il reste moins de 3 allumettes et c'est à l'ordinateur de
jouer...")
                (sleep tempsDattente)
                (print "Il vient de toutes les prendre")
                (print "donc vous avez perdu, désolé...")
                )
        )
   )
)


(defun afficherListe (liste)
    (let ((prefixe 0))
        (format t "~S~%" (car liste)) ; on affiche le premier élément de la liste, qui
est un cas particulier
        (printTree (cdr liste) (+ prefixe 1)) ; on afficher le reste de la liste
    )
)



(defun printTree (liste prefixe)
    (dolist (elemListe liste) ; pour chaque élément de la liste
        (if (and (listp elemListe) (listp (car elemListe))) ; si on est avec un état
possédant des successeurs
            (progn
                (dotimes (i prefixe)
                    (format t "|~T~T~T") ; on affiche le nombre de préfixes qu'il faut
                )
                (format t "~S~%" (car elemListe)) ; on affiche l'état actuel
                (printTree (cdr elemListe) (+ prefixe 1)) ; on affiche les suivant avec
une indentation de plus
            )
            (progn ; si on est avec un état sans successeur
                (dotimes (i prefixe)
                    (format t "|~T~T~T")
                )
                (format t "~S" elemListe) ; on l'affiche avec le nb de préfixe utiles
                (cond
                    ((equal elemListe '(0 J1 0)) (format t " l'ordinateur perd~%"
elemListe))
                    ((equal elemListe '(0 J2 2)) (format t " l'ordinateur gagne~%"
elemListe))
                )
            )
```

```
      )
   )
)
```

Exemples :

Voici un exemple d'exécution des 3 fonctions précédentes à retenir :

- creerListe :

```
(creerListe 'J1 6) ; signifie donc : calculer les possibilités avec J1 qui joue en premier avec 6 allumettes sur le plateau
((6 J1 2) ((5 J2 0) ((4 J1 0) (0 J1 0) (0 J1 0) (0 J1 0)) (0 J2 2) (0 J2 2)) ((4 J2 2) (0 J2 2) (0 J2 2) (0 J2 2)) (0 J1 0))
```

- jeuAllumettes :

```
> (jeuAllumettes)

"Jeu des allumettes"
"Qui commence ? ORDI/MOI" MOI

"Combien d'allumettes ? (entre 5 et 20)" 5

"Calcul de l'arbre possibilités..."
"Début du jeu !"
"Nombre d'allumettes restantes :"
5
"Combien d'allumettes voulez vous retirer ? (entre 1 et 3)" 1

"Nombre d'allumettes restantes :"
4
"Je retire une allumette"
"Il reste moins de 3 allumettes !"
"On considère que vous les retirez toutes"
"donc vous avez gagné, bravo !"
```

- afficherListe :

```
> (afficherliste (creerliste 'J1 6))
(6 J1 2)
|   (5 J2 0)
|   |   (4 J1 0)
|   |   |   (0 J1 0) l'ordinateur perd
```

```
|   |   |   (0 J1 0) l'ordinateur perd
|   |   |   (0 J1 0) l'ordinateur perd
|   |   (0 J2 2) l'ordinateur gagne
|   |   (0 J2 2) l'ordinateur gagne
|   (4 J2 2)
|   |   (0 J2 2) l'ordinateur gagne
|   |   (0 J2 2) l'ordinateur gagne
|   |   (0 J2 2) l'ordinateur gagne
|   (0 J1 0) l'ordinateur perd
NIL
```

b) Exploration en largeur

La fonction exploreLargeur crée la liste des successeurs selon le modèle qu'on a choisi préalablement:

(
((etatInit) (succ1) (succ2) (succ3))
((succ1) (succ1.1) (succ2.1) (succ3.1))
...
)

Dans ce modèle chaque état correspond a un numéro de tour (en partant de 0) a un nombre représentant le nombre d'allumettes restantes et la désignation du Joueur a qui appartient le tour : (0 15 J1) *exemple d'état initial.*

Une fois la liste créée, on la traite grâce a une fonction d'affichage pour trouver les solutions ou l'IA gagne. On utilise donc un algorithme similaire a celui utilise pour le labyrinthe. Etant donne ceci on peut utiliser une fonction d'association pour simplifier l'exploitation du résultat.

L'avantage de la fonction exploreLargeur est qu'elle nous permet de créer, si besoin, une IA capable de jouer (comme pour la profondeur) contre un joueur humain, étant donne qu'on lui passe en paramètre le nombre d'allumettes et le joueur qui commence la partie.

Le désavantage est que la liste créée est très grande, et demande donc a la fonction beaucoup de temps d'exécution.

Fonctions :

de la même manière que dans la partie a), ces fonctions dépendent des précédentes et de la fonction switchJ du a)

- `exploreLargeur (nbAllu joueur)` : première fonction d'exploration en largeur de notre arbre des possibilités. Analogue à la fonction `creerListe (joueur nbAllu)` de la partie a)
- `assocList (listCle listeRecherche)` : fonction analogue à assoc mais pour détecter des listes en tête de liste.
- `afficherListeLargeur (liste)` : afficheur du retour d'`exploreLargeur`. Initialise la variable de préfixe puis utilise la fonction récursive `printTreel`.
- `printTreel (pere liste prefixe)` : fonction récursive d'affichage, à ne pas utiliser sans `afficherListeLargeur.`
- `exploreBDF` : cherche le chemin le plus cours dans l'arbre de possibilité. Fonction très peu réaliste (on ne gère pas ce que l'adversaire joue) mais pédagogique.

Codes :

```
1  (defun assocList (listCle listeRecherche)(
2      if (listp listeRecherche) (progn
3
4          (dolist (elRecherche listeRecherche)
5              (if (equal (car elRecherche) listCle)(
6
7                  return-from assocList elRecherche
8
9                  ))
10
11             )
12
13         )(progn
14
15         (print "listeRecherche n'est pas une liste"
16         (return-from assocList)
17         )
18     ))
19
```

```
(defun exploreLargeur (nbAllu joueur)
	(setq etatInit (list 0 nbAllu joueur))
	(setq flagFirst 0)
	(setq listeComplete '())
	(setq listeActuelle '())
	(setq etatActuel (list 0 nbAllu joueur)) ;on affecte l'etat initial a etatActuel vu qu'on commencera par
lui
	(setq a_visiter '())

	(setq a_visiter  (list etatActuel))


	(loop while (not (eq a_visiter NIL)) do

		(setq etatActuel (car a_visiter)) ;on affecte le premier etat de a_visiter a l'etatActuel
		(setq listeActuelle (list etatActuel))

			(dotimes (i 3)
				(if (> (- (cadr etatActuel) i) 0 )(progn

					(setq listeAppend (list (+ (car etatActuel) 1) (-
(cadr etatActuel) (+ 1 i)) (setq joueur (switchJ joueur)) ))

					(if (and (= 3 (list-length listeActuelle)) (not (listp
(car listeActuelle)))) (progn ;on verifie pour la premiere insertion
						;(format t "Entre en premiere
insertion~%")
						(setq listeActuelle (list
listeActuelle listeAppend))

						(if (not (= (cadr listeAppend)
0))(
							setq a_visiter (list
a_visiter listeAppend)
						))

					)(progn ;else pour les insertions suivantes

						;(format t "Entre en insertion
2~%")
						(setq listeActuelle (append
listeActuelle (list listeAppend)))

						(if (not (= (cadr listeAppend)
0))(progn ;par-ce qu'il est inutile de visiter un etat qui a 0 allumettes

							;(format t
"Entre en a_visiter:~S~%" a_visiter)
```

```
                                                                                (setq a_visiter
(append a_visiter (list listeAppend)))
                                                                                    ))
                                                                            ))

                                        )( progn;else
                                                (setq listeAppend (list (+ (car etatActuel) 1) 0 (setq joueur
(switchJ joueur)) )) ; on met le nombre d'allumettes a 0
                                                (setq listeActuelle (append listeActuelle (list
listeAppend)))
                                                )
                                        )

                                        ;(format t "ListeActuelle:~S~%~%" listeActuelle)
                                        ;(format t "Liste a_visiter:~S~%~%" a_visiter)
                                )

                                (if (= flagFirst 0) (progn
                                        ;(format t "Liste Complete NIL~%")

                                        (setq listeComplete (list listeActuelle))
                                        (setq flagFirst 1)
                                )(progn
                                        ;(format t "Liste Complete !NIL~%")
                                        ;(format t "Liste Complete Pre-aff: ~S~%" listeComplete)
                                        (setq listeComplete (append listeComplete (list
listeActuelle)))
                                        ;(format t "Liste Complete Post-aff: ~S~%" listeComplete)
                                )
                                )

                                ;(format t "Liste Complete fin dotimes:~S~%" listeComplete)

                                (setq listeActuelle '()) ;on reinit listeActuelle
                                (pop a_visiter) ;on retire l'etat qu'on visite actuellement de a_visiter

                                ;(format t "A_visiter:~S~%~%----------~%" a_visiter)
                                )

        (return-from exploreLargeur listeComplete)
)
```

```
(defun afficherListeLargeur (liste)
  (let ((prefixe 0))
    (printTreel (car liste) liste prefixe) ; on afficher le reste de la liste
  )
)



(defun printTreel (pere liste prefixe)
  (if (listp (car pere))
    (progn ; si on est avec un état possédant des successeurs
      (dotimes (i prefixe)
        (format t "|~T~T~T") ; on affiche le nombre de préfixes qu'il faut
      )
      (format t "~S~%" (cdar pere)) ; on affiche l'état actuel (sans écrire le car qui est le numéro de tour)

      ; on va afficher les états suivants
      (if (eq (cadr (cadr pere)) 0) ; si le premier successeur est un EF, l'assoc n'existe pas
        (printTreel (cadr pere) liste (+ prefixe 1))
            (printTreel (assocList (cadr pere) liste) liste (+ prefixe 1))
      )
      (if (eq (cadr (caddr pere)) 0) ; idem, soit on affiche juste l'EF, soit on affiche le successeur et des
successeurs
        (printTreel (caddr pere) liste (+ prefixe 1))
            (printTreel (assocList (caddr pere) liste) liste (+ prefixe 1))
      )
      (if (eq (cadr (cadddr pere)) 0) ; idem
        (printTreel (caddr pere) liste (+ prefixe 1))
            (printTreel (assocList (cadddr pere) liste) liste (+ prefixe 1))
      )
    )
    (progn ; si on est avec un état sans successeur (Etat Final)
      (dotimes (i prefixe) ; on affiche le nombre de préfixes nécessaires
        (format t "|~T~T~T")
      )
      (format t "~S" (cdar pere)) ; on affiche l'état final sans le numéro de tour
      (cond
        ((equal (cdr pere) '(0 J1)) (format t " l'ordinateur perd~%" pere))
        ((equal (cdr pere) '(0 J2)) (format t " l'ordinateur gagne~%" pere))
        (T (format t "~%" pere))
      )
    )
  )
)
```

```
(defun exploreBDF (etatInit liste_possibilites)
	(setq a_visit (list etatInit)); liste des visites à faire, forme :(sommet parent grand-parent ...)
	(setq deja_visite ())


	(print "Debut de l'exploration de l'arbre")
	(loop
		(if (not (eq a_visit NIL))

		(progn
			(format t "Entre dans le if~%")
			(setq actuel (car a_visit))
			(format t "Actuel:~S~% " actuel)
			(print actuel)
			(push actuel deja_visite)
			(dolist (fils (cdr (assocList actuel liste_possibilites)))
				(if (not (member fils deja_visite))
					(if (not (eq (cadr fils) 0))
						(progn
							(setq a_visit (append a_visit (list (append (list
fils) (list actuel)))))
							(format t "Nouveau a_visit ~S~%" a_visit)
							)
						; on ajoute (fils acutel) en fin de liste
						(progn ;else
						(setq a_visit ()) ; on vide la liste
						(setq resultat (append (list actuel)))
						)
					)
				)
			)
			(pop a_visit) ; on enlève le premier élément de la liste
		)
		(progn
			(print "Fini ! Voilà le résultat : ")
			(if resultat
				(print (reverse resultat)); on inverse le chemin, et on l'affiche
				NIL ; on retourne NIL si on a pas trouvé la sortie
			)
			(return)
			)
		)
	)
)
```

Exemples :

```
> (exploreLargeur 5 'J1)
(((0 5 J1) (1 4 J2) (1 3 J1) (1 2 J2)) ((1 4 J2) (2 3 J1) (2 2 J2) (2 1 J1)) ((1 3 J1)
(2 2 J2) (2 1 J1) (2 0 J2))
 ((1 2 J2) (2 1 J1) (2 0 J2) (2 0 J1)) ((2 3 J1) (3 2 J2) (3 1 J1) (3 0 J2)) ((2 2 J2)
(3 1 J1) (3 0 J2) (3 0 J1))
 ((2 1 J1) (3 0 J2) (3 0 J1) (3 0 J2)) ((2 2 J2) (3 1 J1) (3 0 J2) (3 0 J1)) ((2 1 J1)
(3 0 J2) (3 0 J1) (3 0 J2))
 ((2 1 J1) (3 0 J1) (3 0 J2) (3 0 J1)) ((3 2 J2) (4 1 J2) (4 0 J1) (4 0 J2)) ((3 1 J1)
(4 0 J1) (4 0 J2) (4 0 J1))
 ((3 1 J1) (4 0 J2) (4 0 J1) (4 0 J2)) ((3 1 J1) (4 0 J1) (4 0 J2) (4 0 J1)) ((4 1 J2)
(5 0 J2) (5 0 J1) (5 0 J2)))

> (exploreBDF '(0 4 J1) (exploreLargeur 4 'J1))

> (afficherListeLargeur (exploreLargeur 5 'J1))
(5 J1)
|   (4 J2)
|   |   (3 J1)
|   |   |   (2 J2)
|   |   |   |   (1 J2)
|   |   |   |   |   (5 0 J2) l'ordinateur gagne
|   |   |   |   |   (5 0 J1) l'ordinateur perd
|   |   |   |   |   (5 0 J1) l'ordinateur perd
|   |   |   |   (4 0 J1) l'ordinateur perd
|   |   |   |   (4 0 J1) l'ordinateur perd
|   |   |   (1 J1)
|   |   |   |   (4 0 J1) l'ordinateur perd
|   |   |   |   (4 0 J2) l'ordinateur gagne
|   |   |   |   (4 0 J2) l'ordinateur gagne
|   |   |   (3 1 J1)
|   |   (2 J2)
|   |   |   (1 J1)
|   |   |   |   (4 0 J1) l'ordinateur perd
|   |   |   |   (4 0 J2) l'ordinateur gagne
|   |   |   |   (4 0 J2) l'ordinateur gagne
|   |   |   (3 0 J2) l'ordinateur gagne
|   |   |   (3 0 J2) l'ordinateur gagne
|   |   (1 J1)
|   |   |   (3 0 J2) l'ordinateur gagne
|   |   |   (3 0 J1) l'ordinateur perd
|   |   |   (3 0 J1) l'ordinateur perd
```

```
|   (3 J1)
|   |   (2 J2)
|   |   |   (1 J1)
|   |   |   |   (4 0 J1) l'ordinateur perd
|   |   |   |   (4 0 J2) l'ordinateur gagne
|   |   |   |   (4 0 J2) l'ordinateur gagne
|   |   |   (3 0 J2) l'ordinateur gagne
|   |   |   (3 0 J2) l'ordinateur gagne
|   |   (1 J1)
|   |   |   (3 0 J2) l'ordinateur gagne
|   |   |   (3 0 J1) l'ordinateur perd
|   |   |   (3 0 J1) l'ordinateur perd
|   |   (2 1 J1)
|   (2 J2)
|   |   (1 J1)
|   |   |   (3 0 J2) l'ordinateur gagne
|   |   |   (3 0 J1) l'ordinateur perd
|   |   |   (3 0 J1) l'ordinateur perd
|   |   (2 0 J2) l'ordinateur gagne
|   |   (2 0 J2) l'ordinateur gagne
NIL
```

Conclusion:

La fonction exploreBDF présente un problème d'exécution, étant donne la difficulté de la coder et le temps imparti il nous a été impossible de debug et réaliser cette fonction. Elle pourrait permettre a l'AI de jouer contre le joueur.

Après avoir tapé les deux fonctionnement, et analysé le résultat et temps d'exécution, on peut conclure que l'algorithme en profondeur est plus efficace au niveau de la création du modèle de l'arbre, ainsi que dans la recherche d'états puisqu'on peut facilement détecter les noeuds qui permettent a l'IA d'avancer dans la bonne direction. Elle arrive donc a respecter des stratégies pour gagner (se ramener a un multiple de `nbAllum_Ini + 1` par exemple) sans les avoir codées.

Par contre l'algorithme en largeur nous permet de trouver le chemin le plus rapide à la fin, mais pas le plus probable. Il est donc possible, mais difficile de coder une IA capable de jouer et gagner sans faute. On affiche donc l'arbre génèré par l'algorithme en montrant tous les chemins possibles jusqu'à la fin.