Rapport TP n°2 IA01

Damien MARIÉ et Antoine POUILLAUDE

5 novembre 2013

Table des matières

Remarques générales

Chapitre 1

Question n°1

Suivons le formalisme donné dans l'ennoncé. Un état est ainsi représenté : (État rive gauche, État rive droite) sachant qu'un état d'une rive est donné par (nombre de Missionnaire, nombre de Cannibale, 1 si le bâteau est amarré à la rive 0 sinon). Ainsi nous obtenons l'ensemble des états :

$$\text{Ensemble d'état} = \{((3\ 3\ 1)(0\ 0\ 0)), ((2\ 2\ 0)(1\ 1\ 1)), ((3\ 2\ 1)(0\ 1\ 0)), ((0\ 1\ 0)(3\ 2\ 1)), ...\}$$

Cela dit on remarque très vite que la notation est lourde et futile puisque dans tous les cas on doit avoir :

$$\begin{cases} \sum_{\{Rivegauche, Rivedroite\}} Cannibale = 3 \\ et \\ \sum_{\{Rivegauche, Rivedroite\}} Missionaire = 3 \end{cases} \quad \forall \text{ etat } \in \text{Ensemble d'état}$$

Alors on pourrait simplifier la liste des états. Du moment que l'on connait l'état d'une on peut alors connaitre l'état de l'autre rive par déduction. Par conséquent, nous simplifierons en affichant l'état que d'un seul côté. Ainsi on aura l'ensemble des états suivant :

$$\text{Ensemble d'état} = \{(3\ 3\ 1), (2\ 2\ 0), (3\ 2\ 1), (3\ 2\ 0), (0\ 1\ 0), (2\ 3\ 1), (2\ 3\ 0), (1\ 1\ 1), ...\}$$

Chapitre 2

Question n°2

On donne l'ensemble des opérateurs comme suit :

$$\begin{cases} *R - ops* <= ((0\ 1)\ (1\ 0)\ (1\ 1)\ (2\ 0)\ (02)) \\ *L - ops* <= ((0\ -1)\ (-1\ 0)\ (-1\ -1)\ (-2\ 0)\ (0\ -2)) \end{cases}$$

Bien sûr on a fait la distinction entre les opérations que l'ont peut effectué sur la rive gauche et droite. Les opérations dangereuses pour les missionaires sont, d'une manière générale celles durant lesquels des missionnaires sont déplacés d'un bord à l'autre sans pour autant que des cannibales soient déplacés.

Chapitre 3

Question n°3

On donne l'état initial comme ceci :

(3 3 1)

On a également l'état final :

(0 0 0)

Chapitre 4

Question n°4

Chapitre 5

Question n°5

On donne à présent les algorithmes pour la résolution de ce problème. Nous avons tenté de programmer l'algorithme des deux façons qui nous étaient proposés : parcours en largeur d'abord et en profondeur d'abord.

Mais définissons tout d'abord les fonctions de services et les variables associés aux problèmes.

5.1 Les opérations

On donne la déclaration en lisp des listes correspondantes aux opérations que l'ont peut effectuer sur les états en fonction de la rive sur laquelle se trouve le bâteau.

Listing 5.1– Déclaration des opérations

```
(setq *R-ops* '((0 1) (1 0) (1 1) (2 0) (0 2)))
(setq *L-ops* '((0 -1) (-1 0) (-1 -1) (-2 0) (0 -2)))
```

5.2 Vérifier les états

Voici la fonction qui vérifie si un états est correct ou non.

Listing 5.2– verify (state)

```
(defun verify (state)
  (cond
    ((or (< (car state) 0) (< (cadr state) 0)) nil)
    ((or (> (car state) 3) (> (cadr state) 3)) nil)
    (
      (and
        (and (not (= (car state) 3)) (not (= (car state) 0)))
        (not (= (car state) (cadr state)))
      )
      nil
    )
    (T T)
  )
)
```

Ces conditions ont été déduites du tableau suivant

Nombre de Missionnaire	Nombre de Cannibale	État
0	0	OK
1	0	Incorrect
2	0	Incorrect
3	0	OK
0	1	OK
1	1	OK
2	1	Incorrect
3	1	OK
0	2	OK
1	2	Incorrect
2	2	OK
3	2	OK
0	3	OK
1	3	Incorrect
2	3	Incorrect
3	3	OK

5.3 La fonction d'inclusion

Cette fonction permet de savoir si un état appartient à l'ensemble des états déjà visités.

Listing 5.3– in (x L) version récursive

```
(defun in (x L)
  (if (or (eq x nil) (eq L nil)) NIL (if (equal (car L) x) T (in x (cdr L))))
)
```

5.4 Applications des opérations à l'état courant

Cette fonction va permettre d'avoir le résultat de l'application des opérations choisies sur l'état courant.

Listing 5.4– apply-op (state op)

```
(defun apply-op (state op)
  (list
    (+ (car state) (car op))
    (+ (cadr state) (cadr op))
    (if (eq (caddr state) 1) 0 1))
)
```

5.5 Parcours en largeur d'abord

5.5.1 Fonctions pour enfiler et défiler

Les deux fonctions suivantes vont servirent à enfiler et à défiler des éléments dans une file FIFO afin de pouvoir par la suite implémenter la fonction de parcours en largeur d'abord. Cependant, on aurait, après réflexion, utiliser utiliser les fonctions de bases de lisp pour arriver à nos fins.

Enfiler

Listing 5.5– enqueue (x Q)

```
(defun enqueue (x Q)
  (if (null Q)
    (list x)
        (nconc Q (list x))
  )
)
```

Défiler

Listing 5.6– dequeue (Q)

```
(defun dequeue (Q)
  (cdr Q)
)
```

5.5.2 Fonction de parcours

Algorithme

Algorithm 1 Algorithme de parcours en largeur d'abord

if $car(x) = (0\ 0\ 0)$ **then**
  retourner T
else if $L[1] \in M$ **then**
  retourner $(L[1]F4(L/L[1]))$
else
  retourner $F4(L\ L[1])$
end if

Implémentation en lisp

Listing 5.7– BFS-solve (N L) version largeur d'abord

```
(defun BFS-solve (Q)
  (dolist (node Q)
    (setq Q (dequeue Q))
      (cond
            ((equal (car node) '(0 0 0))
              (print "One of the solution is ")
              (print (append (cadr node) (list '(0 0 0)))))
```

```
                ((verify (car node))
                  (let ((res '()))
                  (dolist
                    (op (if (eq (caddr (car node)) 0) *R-ops* *L-ops*))
                    (setq res (apply-op (car node) op))
                    (if (and (not (in res (cadr node))) (verify res))
                  (if (eq (cadr node) NIL)
                        (setq Q (enqueue (list res (list (car node))) Q))
                        (setq Q
                          (enqueue (list res
                          (append (cadr node)
                          (list (car node))))
                          Q
                        )
                      )
                    )
                  )
                )
              )
            )
          )
        )
  )
)
(if (null Q) "END" (solve-larg Q))
)
```

Listing 5.8– F2 (N L) version itérative

```
                (defun F2Iter (N L)
                      (if (< N (length L))
                            (let ((out NIL))
                                  (dotimes
                                    (x (+ N 1)
                                  out)
                              (cons out (nth x L))
                                  )
                              )
                              (print "The number you chose is not correct")
                       )
                 )
```

F2rec et F2iter retourne la tête de la liste L formée par les N premiers éléments de la liste L.

5.6 Retourner la concaténation de deux listes

Listing 5.9– F3 (L M) version recursive

```
                (defun F3rec (L M)
                  (if (null L) M
                    (F3rec (rest L) (cons (first L) M))
                  )
                )
```

Listing 5.10– F3 (L M) version itérative

```
(defun F3iter (L M)
  (append L M)
)
```

F3rec et F3iter retournent la concaténation des deux listes L et M.

5.7 Intersection de deux listes

Algorithm 2 Algorithme pour l'intersection de deux listes

if $L = NIL$ **then**
  retourner T
else if $L[1] \in M$ **then**
  retourner $(L[1]F4(L/L[1]))$
else
  retourner $F4(L\ L[1])$
end if

Listing 5.11– F4 (L M) version recursive

```
(defun F4rec (L M)
      (if (NULL L) L
            (if (member (first L) M)
              (cons (first L) (F4rec (rest L) M))
              (F4rec (rest L) M)
            )
      )
)
```

Listing 5.12– F4 (L M) version itérative

```
(defun F4iter (L M)
      (let ((out '()))
            (if (< (length L) (length M))
                  (dolist (x L out)
                        (if (member x M)
                              (cons out x)
                        )
                  )
                  (dolist (x M out)
                        (if (member x L)
                              (cons out x)
                        )
                  )
            )
      )
)
```

F4rec et F4iter retournent l'intersection des deux listes. Attention, cependant, après test la fonction F4iter ne retourne pas ce que l'on souhaite.

5.8 Inclusion de deux listes

La première fonction a été produite avant la réception des spécifications que Mme Abel nous a envoyé en plus. Elle vérifie simplement que tous les éléments de la première liste sont dans la seconde, et ce, indépendamment de l'ordre dans lequel les éléments sont.

Algorithm 3 Algorithme pour l'inclusion de deux listes version 1

if $L = NIL$ **then**
  retourner T
else if $L[1] \in M$ **then**
  retourner $F4(L\ L[1])$
else
  retourner NIL
end if

Listing 5.13– F5 (N L) version recursive 1

```
(defun F5 (L M)
        (if (NULL L) T
                (if (member (first L) M) (F5 (rest L) M)
                        NIL
                )
        )
)
```

Cet seconde fonction tient, cette fois-ci, compte de l'ordre des éléments de la première liste. Par exemple si la liste L est définie comme L=(1 2) et M comme M=(1 3 5 2) alors la fonction retournera NIL alors qu'avec M=(1 2 5 3) elle renverra T. La composition de ces deux fonctions, en revanche, nous donne une compléxité plutôt très mauvaise puisque dans le pire des cas on a une complexité en $O(card(L) * card(M))$ c'est à dire, en simplifiant, $O(n^2)$.

Algorithm 4 Algorithme pour l'inclusion de deux listes version 2

if $L = NIL$ **then**
  retourner T
else if $M = NIL$ **then**
  retourner T
else if $verifF5(L, M)$ **then**
  retourner T
else
  retourner NIL
end if

Listing 5.14– F5 (L M) version récusive 2

```
(defun F5 (L M)
        (cond
                ((NULL L) T)
                ((NULL M) NIL)
                ((verifF5 L M) T)
                (T (F5 L (rest M)))
        )
)
```

La fonction suivante vérifie les termes consécutifs.

Algorithm 5 Algorithme de vérification

if $L = NIL$ **then**
  retourner T
else if $M = NIL$ **then**
  retourner T
else if $L[1] = M[1]$ **then**
  retourner $verifF5(rest(L), rest(M))$
else
  retourner NIL
end if

Listing 5.15– verifF5 (L M) version récusive 2

```
(defun verifF5 (L M)
        (cond
                ((NULL L) T)
                ((NULL M) NIL)
                ((eq (first L) (first M)) (verifF5 (rest L) (rest
                (T NIL)
        )
)
```

Chapitre 6

Éxercice 2

Cet exercice consistant purement à la compréhension de la fonction mapcar dans un exemple plutôt simple, il n'y pas de commentaires dessus.

Listing 6.1– list-carres (L) avec fonction lambda

```
(defun list-carres (L)
  (mapcar (lambda (x)(* x x)) L)
)
```

Chapitre 7

Éxercice 3

My-assoc est une fonction qui recherche si la clé donner en argument se trouve dans la liste.

7.1 Algorithmique

Ci-dessous se trouve l'algorithme fait en pseudo-code de la fonction.

Algorithm 6 Algorithme de My-Assoc

if $cle! = clede L[1]$ **then**
  retourner $my - assoc(cle, rest(L))$
else
  retourner la paire
end if

7.2 Code source

Et voici le code source de ladite fonction.

Listing 7.1– My_Assoc (cle a-liste) version récusive

```
(defun my-assoc (cle a-liste)
  (if (null a-liste)
    NIL
    (if (equal (first (first a-liste)) cle)
      (first a-liste)
      (my-assoc cle (rest a-liste))
     )
   )
)
```

Chapitre 8

Éxercice 4

8.1 Petites fonctions pour commencer

Pour retourner le nom il suffit de logiquement renvoyer le premier élément de la liste.

Listing 8.1– nom(Personne)

```
(defun nom (pers) (first pers))
```

Pour retourner le prénom il suffit de logiquement renvoyer le second élément de la liste.

Listing 8.2– prenom(Personne)

```
(defun prenom (pers) (cadr pers))
```

Pour retourner la ville il suffit de logiquement renvoyer le troisième élément de la liste.

Listing 8.3– ville(Personne)

```
(defun ville (pers) (caddr pers))
```

Pour retourner l'âge il suffit de logiquement renvoyer l'avant avant dernier élément de la liste.

Listing 8.4– age(Personne)

```
(defun age (pers) (cadddr pers))
```

Pour retourner le nombre de livre il suffit de logiquement retourner le dernier élément de la liste.

Listing 8.5– nom(Personne)

```
(defun nom (pers) (first pers))
```

Pour retourner le nom il suffit de logiquement renvoyer le premier élément de la liste.

Listing 8.6– nombre-livre(Personne)

```
(defun nombre-livre (pers) (car (last pers)))
```

Pour les questions les premières fonctions sont triviales cependant la dernière fonction (FB6) à été difficilement réalisable en version récursive de par la nature même de la question (en récursif pur, celà reviendrai à faire une moyenne progressive).

8.2 Personnes contenues dans la base de données.

Je vais tout simplement retourner toute la base données.

Listing 8.7– FB1(data)

```
(defun FB1 (data)
  (if (null (rest data))
    data
    (cons (first data) (FB1 (rest data)))
  )
)
```

8.3 Les personnes qui s'appellent Pierrot.

Ci-dessous se trouve l'algorithme fait en pseudo-code de la fonction.

Algorithm 7 Algorithme de Pierrot

if $data = NIL$ **then**
  retourner $data$
else if $nom(data[1])$ ="Pierrot" **then**
  retourner $(data[1]FB2(rest(data)))$
else
  retourner $FB2(rest(data))$
end if

Listing 8.8– FB2(data)

```
(defun FB2 (data)
  (if (null data)
    data
        (if (eq (nom (first data)) 'Pierrot)
          (cons (first data) (FB2 (rest data)))
              (FB2 (rest data))
            )
      )
)
```

8.4 Les personnes avec le nom donné en argument.

Ci-dessous se trouve l'algorithme fait en pseudo-code de la fonction.

Algorithm 8 Algorithme de nom

if $data = NIL$ **then**
  retourner $data$
else if $nom(data[1]) = name$ **then**
  retourner $(data[1]FB2(rest(data), name))$
else
  retourner $FB2(rest(data), name)$
end if

Listing 8.9– FB3(data name)

```
(defun FB3 (data name)
  (if (null data)
    data
        (if (eq (nom (first data)) name)
          (cons (first data) (FB2 (rest data) name))
              (FB2 (rest data) name)
          )
    )
)
```

8.5 Les personnes avec X années.

Ci-dessous se trouve l'algorithme fait en pseudo-code de la fonction.

Algorithm 9 Algorithme de l'âge

if $data = NIL$ **then**
  retourner NIL
else if $age(data[1]) = X$ **then**
  retourner $data[1]$
else
  retourner $FB4(rest(data))$
end if

Listing 8.10– FB4(data X)

```
(defun FB4 (data x)
  (if (null data)
    NIL
        (if (= (age (first data)) x)
          (first data)
          (FB4 (rest data) x)
        )
    )
)
```

8.6 Les personnes ayant moins de X livres.

Ci-dessous se trouve l'algorithme fait en pseudo-code de la fonction.

Algorithm 10 Algorithme de l'âge

if $data = NIL$ **then**
  retourner NIL
else if $nombre - livre(data[1]) \leq X$ **then**
  retourner $data[1]$
else
  retourner $FB5(rest(data))$
end if

Listing 8.11– FB5(data X)

```
(defun FB5 (data x)
  (if (null data)
    NIL
        (if (< (nombre-livre (first data)) x)
          (first data)
          (FB5 (rest data) x)
        )
      )
)
```

8.7 Moyenne des ages de la famille X.

Listing 8.12– FB6(data X)

```
(defun FB6 (data x) (
    (setq c 0)
    (setq sum 0)
    (dolist (el data) (
        (if (eq (name el) x) (
            (setq sum (+ sum (age el)))
            (setq c (+ c 1))
        ))
    ))
    (if (eq c 0) 0 (
        (/ sum c)
    ))
))
```