Rapport TP n°2 IA01

Damien MARIÉ et Antoine POUILLAUDE

5 novembre 2013

Table des matières

Remarques générales

Ce TP, bien que pas facile, nous a ammener à implémenter des fonctions de parcours d'états en fonction de conditions fixées au départ. C'est-à-dire que nous devions à chaque fois que l'on changeait d'état vérifier si l'état était cohérent par rapport à l'ennoncé qui nous été donné. C'est, au final, un TP très instructif et amusant même.

Chapitre 1

Préparation du problème

1.1 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 :

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

1.2 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.

1.3 Question n°3

On donne l'état initial comme ceci :

(3 3 1)

On a également l'état final :

(0 0 0)

1.4 Question n°4

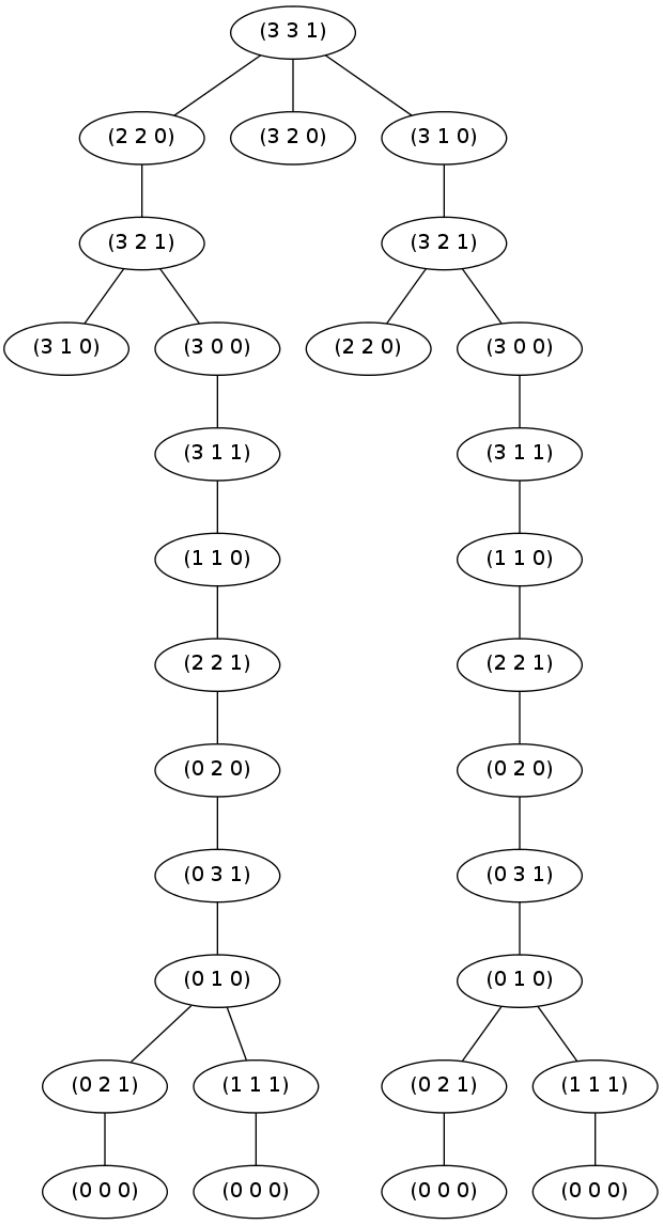

FIGURE 1.1 – Arbre de résolution partiel

Chapitre 2

Parcours de l'arbre pour trouver les solutions

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.

2.1 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 2.1– 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)))
```

2.2 Vérifier les états

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

Listing 2.2– 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

2.3 La fonction d'inclusion

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

Listing 2.3– 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))))
)
```

2.4 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 2.4– 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))
)
```

2.5 Parcours en largeur d'abord

2.5.1 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 2.5– enqueue (x Q)

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

Défiler

Listing 2.6– dequeue (Q)

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

2.5.2 Fonction de parcours

Voici donc la tant attendue fonction de parcours qui au terme de son execution doit donner les manières de résoudre le problème donné.

Algorithme

Algorithm 1 Algorithme de parcours en largeur d'abord

while Q is not empty **do**
  node := Q[1]
  dequeue(Q)
  if $car(x) = (0\ 0\ 0)$ **then**
    retourner $cadr(node) + (0\ 0\ 0)$
  else if $verify(car\ node)$ **then**
    for all $x \in \begin{cases} *R - ops* & \text{si } cadr(car(node)) = 0 \\ *L - ops* & \text{si } cadr(car(node)) = 1 \end{cases}$ **do**
      res := $apply - op(car(node), op)$
      if $not\ in(res, cadr(node))$ and $verify(res)$ **then**
        $Q := \begin{cases} enqueue(((res)\ (car(node))), Q) & \text{si } cadr(node) = NIL \\ enqueue(((res)\ (cadr(node) + car(node))), Q) & \text{sinon} \end{cases}$
      end if
    end for
  end if
end while

Implémentation en lisp

Listing 2.7– 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 "Solution: ")
              (print (append (cadr node) (list '(0 0 0)))))
            ((verify (car node))
              (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" (BFS-solve Q))
)
```

Après l'execution de cette fonction nous obtenons bien les 4 solutions que nous attendions selon l'arbre dessiné plus haut.

Listing 2.8– Résultat d'execution de la fonction de parcours en largeur d'abord.

```
"Parcours en largeur"
"Solution:"
((3 3 1) (2 2 0) (3 2 1) (3 0 0) (3 1 1) (1 1 0) (2 2 1) (0 2 0) (0 3 1)
 (0 1 0) (0 2 1) (0 0 0))
"Solution:"
((3 3 1) (2 2 0) (3 2 1) (3 0 0) (3 1 1) (1 1 0) (2 2 1) (0 2 0) (0 3 1)
 (0 1 0) (1 1 1) (0 0 0))
"Solution:"
((3 3 1) (3 1 0) (3 2 1) (3 0 0) (3 1 1) (1 1 0) (2 2 1) (0 2 0) (0 3 1)
 (0 1 0) (0 2 1) (0 0 0))
"Solution:"
((3 3 1) (3 1 0) (3 2 1) (3 0 0) (3 1 1) (1 1 0) (2 2 1) (0 2 0) (0 3 1)
 (0 1 0) (1 1 1) (0 0 0))
```

2.6 Parcours en profondeur d'abord

La fonction suivante sert à parcourir les états mais cette fois-ci en profondeur d'abord. En revanche, la fonction ne renvoi le résultat attendu et nous ne savons pas pourquoi.

Algorithmique

Algorithm 2 Algorithme de parcours en profondeur d'abord

if $car(x) = (0\ 0\ 0)$ **then**
  retourner $previous - states + (0\ 0\ 0)$
else if $not\ verify(state)$ **then**
  retourner NIL
else
  for all $x \in \begin{cases} *R - ops* & \text{si } cadr(car(node)) = 0 \\ *L - ops* & \text{si } cadr(car(node)) = 1 \end{cases}$ **do**
    res := $apply - op(car(node), op)$
    if $not\ in(res, cadr(node))$ and $verify(res)$ **then**
      retourner $\begin{cases} DFS - solve(res, (state)) & \text{si } previous - states = NIL \\ DFS - solve(res, previous - states + state) & \text{sinon} \end{cases}$
    end if
  end for
end if

Implémentation en lisp

Listing 2.9– DFS-solve (state previous-states) version profondeur d'abord

```
(defun DFS-solve (state previous\-{}states)
  (cond
    ((equal state '(0 0 0))
     (print "Solution:")
     (print (nconc previous\-{}states (list '(0 0 0)))))
    ((not (verify state)) nil)
    (T
      (dolist
        (op (if (eq (caddr state) 0) *R-ops* *L-ops*))
        (setq res (apply\-{}op state op))
        (if (and (not (in res previous\-{}states)) (verify res))
          (if (null previous-states)
            (DFS-solve res (list state))
            (DFS-solve res (nconc previous-states (list state)))
          )
        )
      )
    )
  )
)
```

$previous - states$ reprèsente les états déjà visités durant le parcours et *state* reprèsente l'état courant.

Après l'excution de la fonction on obtiens le résultat suivant.

Listing 2.10– Résultat d'execution de la fonction de parcours en profondeur d'abord.

```
"Solution:"
((3 3 1) (2 2 0) (3 2 1) (3 2 1) (3 0 0) (3 1 1) (1 1 0) (2 2 1) (0 2 0)
 (0 3 1) (0 1 0) (0 2 1) (0 0 0))
"Solution:"
((3 3 1) (3 1 0) (3 2 1) (3 2 1) (3 0 0) (3 1 1) (1 1 0) (2 2 1) (0 2 0)
 (0 3 1) (0 1 0) (0 2 1) (0 0 0))
```