Université de Technologie de Compiègne

IA01

Intelligence Artificielle : Représentation

Rapport TP2
Problème des deux récipients

Guillaume GALASSO GI01
Alexandre MAZGAJ GI01

13 novembre 2016

Table des matières

Introduction

Dans ce second TP de l'UV IA01, on nous propose de résoudre un problème de recherche dans un espace d'états au travers du problème des deux récipients. Ainsi, nous avons d'abord travaillé sur une modélisation du problème avant de l'implémenter en langage Lisp.

Enfin, nous avons rédigé 2 fonctions pour résoudre le problème selon le mode de recherche en profondeur puis en largeur d'abord ainsi que plusieurs fonctions de services nécessaires à leur bon fonctionnement.

1 Définition formelle du problème de recherche

1.1 Définition des ensembles d'états

Soit Q l' ensemble non vide des états représentables pour le problème donné.
On a : $Q = \{x \in \mathbb{N}, y \in \mathbb{N}/(x, y) \in \mathbb{N} \times \mathbb{N}, \forall x \leq 4 \ \forall y \leq 3\}$
Ainsi, il peut y avoir au plus 20 états dont :

— l'ensemble des états initiaux avec $S = \{(0, 0)\} \subset Q$

— l'ensemble des états-solutions $G = \{(2, 0), (2, 1), (2, 2), (2, 3)\} \subset Q$

1.2 L'ensemble d'actions

Soit A l'ensemble d'action du problème de recherche. Il y a 8 actions possibles :

1. **(x<4, y) → (4,y) :**
   Si le contenu de R1 est inférieur à 4 litres, on peut le remplir. Il contiendra alors 4 litres le contenu y de R2 restant inchangé.
2. **(x, y<3) → (x,3) :**
   Si le contenu de R2 est inférieur à 3 litres, on peut le remplir. Il contiendra alors 3 litres le contenu y de R1 restant inchangé.
3. **(x, y>0) → (x,0) :**
   Si R2 n'est pas vide, on peut le vider, le contenu de R1 restant inchangé.
4. **(x>0, y) → (0,y) :**
   Si R1 n'est pas vide, on peut le vider, le contenu de R2 restant inchangé.

5. **(x>0, y) avec x+y $\leq$ 3 $\rightarrow$ (0, y+x) :**
   Si R1 n'est pas vide et que la somme des quantités de R1 et R2 est inférieur ou égale à 3, on peut vider totalement R1 dans R2.
6. **(x, y>0) avec x+y $\leq$ 4 $\rightarrow$ (x+y, 0) :**
   Si R2 n'est pas et que la somme des quantités de R1 et R2 est inférieur ou égale à 4 litres, on peut vider totalement R2 dans R1.
7. **(x<4, y) avec x+y $>$ 3 $\rightarrow$ (x+y-3, 3) :**
   Si R1 et R2 ne sont pas vides, que R1 n'est pas rempli et que la somme de leur quantité d'eau est supérieur à 3 litres, on peut verser un partie du contenu de R1 pour remplir totalement R2. R1 n'est pas vide.
8. **(x, y<3) avec x+y $>$ 4 $\rightarrow$ (4, y+x-4) :**
   Si R1 et R2 ne sont pas vides, que R2 n'est pas rempli et que la somme de leur quantité d'eau est supérieur à 4 litres, on peut verser un partie du contenu de R2 pour remplir totalement R1. R2 n'est pas vide.

2 Définition des fonctions de service

2.1 La fonction `actions(etat)`

Pour la fonction `actions(etat)`, nous avons défini les actions possibles en fonction de l'état des deux réservoirs. A partir d'un état donné, la fonction va retourner une liste des numéros des actions possibles à partir de cet état.

Listing 2.1 – Fonction `actions(etat)`

```
1  (defun actions (etat)
2    (let ((action nil))
3     (if(<(car etat) 4) (push 1 action))
4     (if (< (cadr etat) 3) (push 2 action))
5     (if (> (car etat) 0) (push 3 action))
6     (if (> (cadr etat) 0) (push 4 action))
7     (if (and (<= (+ (car etat) (cadr etat)) 4) (> (cadr etat) 0)) (push 5 action))
8     (if (and (<= (+ (car etat) (cadr etat)) 3) (> (car etat) 0)) (push 6 action))
9     (if (and (> (+ (car etat) (cadr etat)) 4) (< (car etat) 4)) (push 7 action))
10    (if (and (> (+ (car etat) (cadr etat)) 3) (< (cadr etat) 3)) (push 8 action))
11    action))
```

La fonction va vérifier pour chacune des 8 actions si l'état passé en paramètre répond aux conditions, que nous avons défini. Si la condition est vérifiée, la fonction va mettre le numéro de cette action possible dans une liste définie comme variable locale.

2.2 La fonction `successeurs(etat etatsVisites)`

2.2.1 Définition de la fonction

La fonction `successeurs(etat etatsVisites)` va, pour chaque numéro d'action donné par la fonction action à partir d'un état donné q_0, affecter à la variable locale `temp` la valeur de l'état que nous pouvons atteindre à partir de q_0.

On effectue ensuite un test avec l'ensemble des états visités `etatsVisites`

- Si cet état à déjà était atteint et donc appartient à la liste des états visités, alors on ne le met pas dans la liste `lbis` (qui est une variable locale).
- Si l'état n'a pas encore été visité, alors on le place dans `lbis`.

Une fois ces tests effectues, la fonction retourne le contenu de `lbis`.

Listing 2.2 – Fonction `successeurs(etat etatsVisites)`

```
1  (defun successeurs (etat etatsVisites)
2    (let ((lbis NIL))
3    (dolist (x (action etat) lbis)
4      (when (eq 1 x) (setq temp (append '(0) (list (R2 etat))))
5        (if (not (member temp etatsVisites :test #'equal))
6          (setq lbis (append lbis (list temp)))))
7
8      (when (eq 2 x) (setq temp (append (list (R1 etat)) '(0)))
9        (if (not (member temp etatsVisites :test #'equal))
10         (setq lbis (append lbis (list temp)))))
11
12     (when (eq 3 x) (setq temp (append '(4) (list (R2 etat))))
13       (if (not (member temp etatsVisites :test #'equal))
14         (setq lbis (append lbis (list temp)))))
15
16     (when (eq 4 x) (setq temp (append (list (R1 etat)) '(3)))
17       (if (not (member temp etatsVisites :test #'equal))
18         (setq lbis (append lbis (list temp)))))
19
20     (when (eq 5 x) (setq temp(append(list(-(+(R1 etat)(R2 etat))3))'(3)))
21       (if (not (member temp etatsVisites :test #'equal))
22         (setq lbis (append lbis (list temp)))))
23
24     (when (eq 6 x) (setq temp (append '(0) (list (+ (R1 etat) (R2 etat)))))
```

```
25        (if (not (member temp etatsVisites :test #'equal))
26          (setq lbis (append lbis (list temp)))))
27
28      (when (eq 7 x) (setq temp(append '(4)(list(-(+(R1 etat)(R2 etat))4))))
29        (if (not (member temp etatsVisites :test #'equal))
30          (setq lbis (append lbis (list temp)))))
31
32      (when (eq 8 x) (setq temp (append (list (+ (R1 etat) (R2 etat))) '(0)))
33        (if (not (member temp etatsVisites :test #'equal))
34          (setq lbis (append lbis (list temp)))))
35    )))
```

A noter que si aucune action n'est possible compte tenu de l'état de départ et des états visités précédemment, la fonction retourne `nil`.

2.2.2 Autre méthode : passage par une autre fonction outil

Une solution alternative serait de définir une autre fonction de service `resultat-action (etat action)` qui a partir d'un état q_0 donné et de la liste des actions possibles, revoie les états correspondants aux numéros des actions à partir de q_0.

Listing 2.3 – Fonction `resultat-action (etat action)`

```
1  (defun resultat-action (etat action)
2    (cond
3     ((= action 1) (list 4 (cadr etat)))
4     ((= action 2) (list (car etat) 3))
5     ((= action 3) (list 0 (cadr etat)))
6     ((= action 4) (list (car etat) 0))
7     ((= action 5) (list (+ (car etat) (cadr etat)) 0))
8     ((= action 6) (list 0 (+ (car etat) (cadr etat))))
9     ((= action 7) (list 4 (- (cadr etat) (- 4 (car etat)))))
10    ((= action 8) (list (- (car etat) (- 3 (cadr etat))) 3))
11     ))
```

Ainsi, la fonction `successeurs(etat etatsVisites)` suit le fonctionnement suivant :

- Une variable locale `listeAction` permet d'avoir les états correspondants aux numéros des actions possibles. On utilise une opération de mapping avec la fonction `resultat-action` définie précédemment sur la liste contenant le numéro des actions possibles à partir de l'état donné en entrée (fonction `action`).
- Pour chaque état de `listeAction`, si il n'est pas dans la liste des états visités, on le met dans la liste `listeEtatsSuivants` définie localement.

Enfin, on revoie `listeEtatsSuivants` (valeur de retour de la boucle dolist).

Listing 2.4 – Fonction `successeursBis(etat etatsVisites)`

```
1 (defun successeursBis(etat etatsVisites)
2   (let ((listeEtatsSuivants nil)
3     (listeAction (mapcar #' (lambda(x) (resultat-action etat x)) (actions etat))))
4
5     (dolist (elem listeAction listeEtatsSuivants)
6       (if (not (member elem etatsVisites :test #'equal))
7           (push elem listeEtatsSuivants)
8         ))
9     ))
```

3

Recherche dans l'espace d'états en profondeur d'abord

3.1 Principe général de la recherche en profondeur

Pour rechercher dans l'espace d'états en profondeur d'abord, il faut parcourir l'arbre de recherche depuis la racine en prenant systématiquement la branche la plus à gauche possible jusqu'à un noeud terminal.

Ensuite, on remonte jusqu'à ce qu'il y ait au moins une autre branche à parcourir et on réitère l'opération. Quand il n'y a plus de branche à parcourir (on est revenu à la racine), la recherche est terminée.

3.2 Algorithme

Soit l'algorithme `rech-prof(etat etatsVisites)`, avec `etat` l'état courant et `etatsVisites` la liste des états visités, procédant comme suit :

Initialisation : `etatsVisites` ← `nil`

Étape 1 : `etatsSuivants` ← `successeurs (etat etatsVisites)`

/* Pour un état donné, on associe une liste d'états successeurs `etatsSuivants` à partir des états visités précédemment.*/

`etatsVisites` ← `etatsVisites` ∪ `etat`

Étape 2 : Si `etatsSuivants` = `nil` /* les états possibles ont tous été visités */
alors retourner `nil`.
Si `(car etat)` = 2 /* l'état actuel est un état-solution */
alors on affiche la liste des états visités.

Étape 3 : Tant que `etatsSuivants` ≠ `nil` faire :
`rech-prof((car etatsSuivants) etatsVisites)`
`etatsSuivants` ← `etatsSuivants - (car etatsSuivants)`
Fin tant que

3.3 Implémentation en Lisp

Listing 3.1 – Fonction `rech-prof2(etat etatsVisites)`

```
1  (defun rech-prof2(etat etatsVisites)
2    (let ((etatsSuivants (successeurs etat etatsVisites)))
3      (push etat etatsVisites)
4      (cond
5       ((equal etatsSuivants nil) nil)
6       ((= (car etat) 2) (format t "~&solution : ~a" (reverse etatsVisites)))
7       (T
8        (while (not (equal etatsSuivants nil))
9          (rech-prof2 (car etatsSuivants) etatsVisites)
10         (pop etatsSuivants)
11 )))))
```

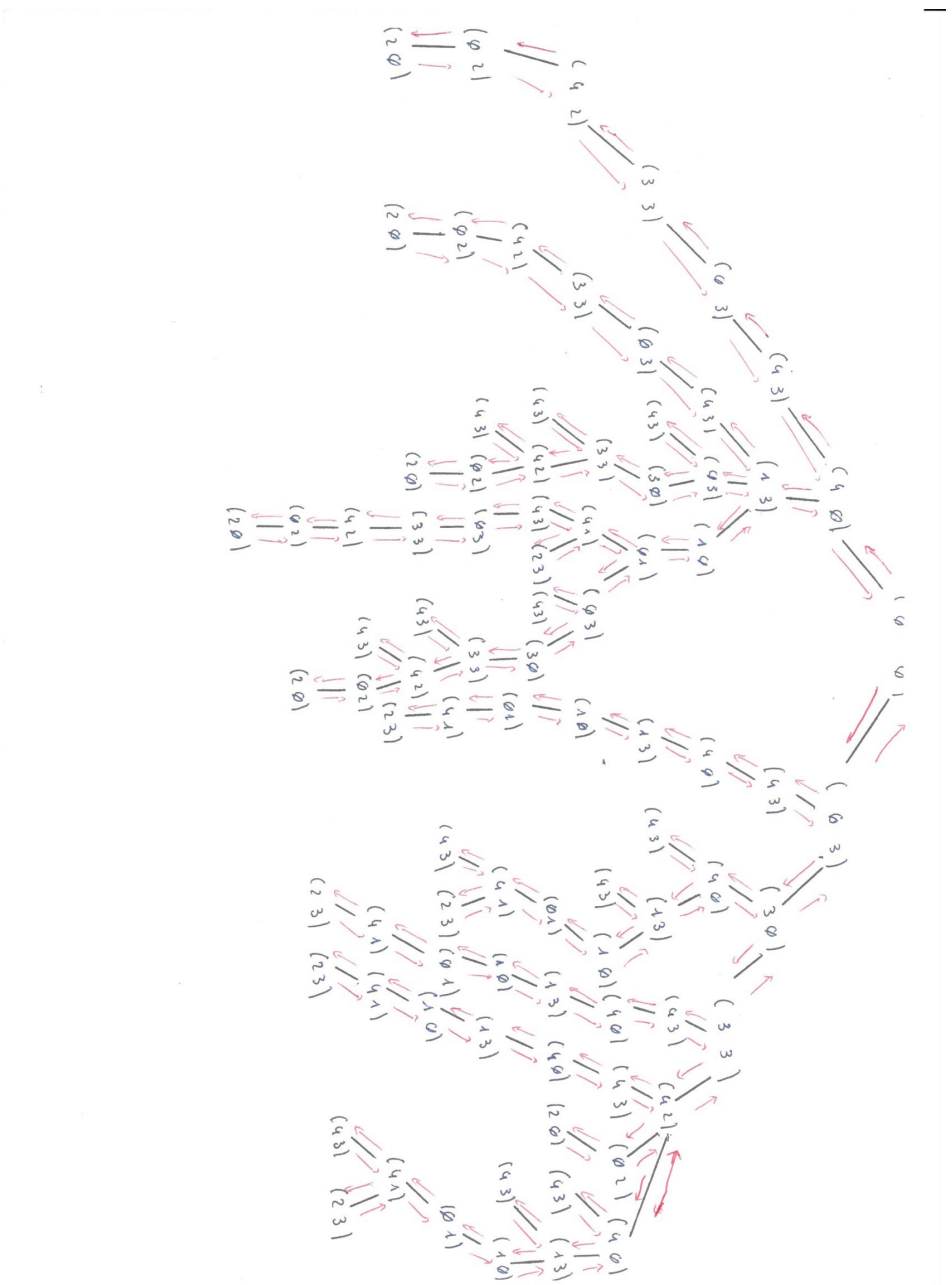
On définit une autre fonction qui comprend comme variable `etatsVisites` qui utilise la fonction ci-dessus.

Listing 3.2 – Fonction `rech-prof(etat)`

```
1  (defun rech-prof(etat)
2    (defparameter etatsVisites nil)
3    (rech-prof2 etat etatsVisites))
```

3.4 Arbre de recherche correspondant

L'arbre associé à la recherche en profondeur d'abord est donné ci-dessous. Le parcours est similaire au principe général décrit plus haut : on parcours les branches de l'arbre les plus à gauche possible avant de remonter à partir d'un noeud terminal.



4 Recherche dans l'espace d'états en largeur d'abord

4.1 Principe général de la recherche en largeur

A partir de la racine R, on liste les fils de R pour ensuite les explorer un par un. On réitère l'opération avec chaque fils jusqu'à ce qu'on ait attend tous les noeuds terminaux. Ce mode de fonctionnement utilise une file.

4.2 Algorithme

Soit l'algorithme `rech-larg(etat etatsVisites)`, avec `etat` l'état courant, `etatsVisites` la liste des états visités et `file` une file. Il procède comme suit :

Initialisation : `etatsVisites` ← `etat`
Enfiler `(list (list etat (list (etatsVisites)))`

Étape 1 : `temp` ← Défiler (`file`)
`etat` ← `(car temp)`
`etatVisites` ← `(cadr temp)`
`etatsSuivants` ← `successeurs (etat etatsVisites)`
Si `etat` est solution, alors on affiche les états visités

Étape 2 : Pour chaque élément `elem` de `etatsSuivants` faire
Enfiler `(list (list elem (list (etatsVisites)))`
Fin pour

Étape 3 : Si la file n'est pas vide reprendre à l'étape 2.

4.3 Implémentation en Lisp

Listing 4.1 – Fonction `rech-prof(etat)`

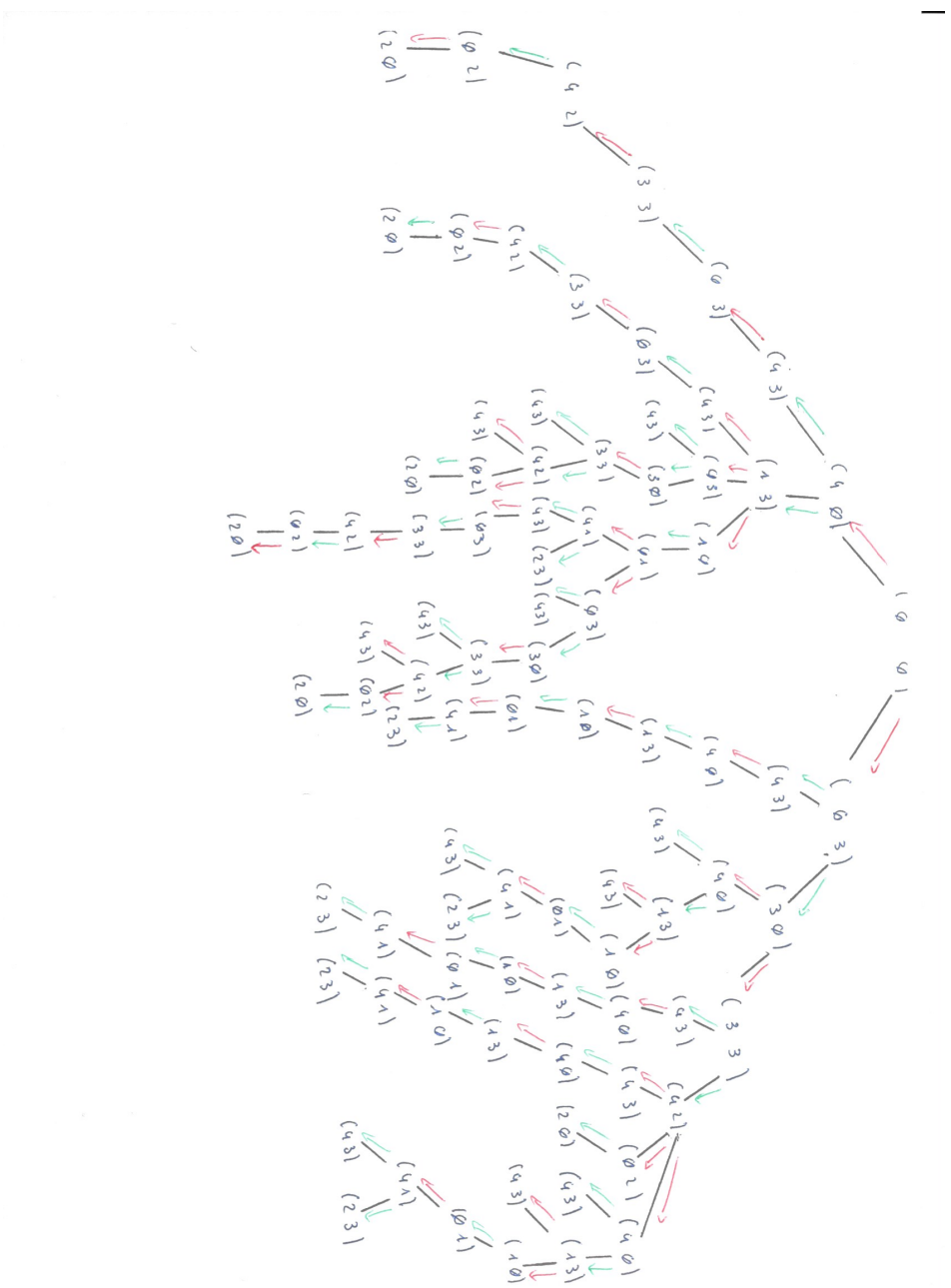
```
(defun rech-larg (etat)
  (let (
        (file (list (list etat (list etat))))
        (temp nil) (etatsVisites nil)
        (etatsSiuvants nil))

  (while (not (null file))
    (progn
      (setq temp (pop file))
      (setq etat (car temp))
      (setq etatsVisites (cadr temp))
      (setq etatsSiuvants (successeurs etat etatsVisites))

      (if (equal (car etat) 2)
        (format t "~%Solution : ~a" etatsVisites)
        (dolist (elem etatsSiuvants)
        (setq file (append file (list (list elem (append (list elem) etatsVisites)
      ))))
))))))
```

La file contient à l'itération i de la boucle l'ensemble des chemins parcourus d'une profondeur i. A chaque itération i, on défile le chemin courant `temp` et pour chaque nouvel état possible `etatsSiuvants` à partir du dernier état de `temp` (le dernier noeuds parcourus, `etat` = `(car temp)`), on enfile un nouveau chemin qui contient en plus ce nouvel état. Il y a donc autant de nouveaux chemins créés que de nouveaux états possibles.

Ce processus de construction de la file est un moyen de conserver l'ensemble des chemins parcourus.

4.4 Arbre de recherche correspondant

L'arbre associé à la recherche en largeur d'abord est donné ci-dessous. Le parcours est similaire au principe général décrit plus haut : on parcours simultanément tous les successeurs d'un noeud depuis la racine (flèche de la même couleur).



5 Comparaison des deux méthodes de recherche

5.1 Principe général

D'un point de vue général, il apparait que la recherche en profondeur d'abord est plus simple et moins couteuse à mettre en place que la recherche en largeur d'abord (notamment par rapport à la conservation de tous les chemins parcourus dans la file)

Pour une étude plus approfondie, nous avons analysé pour les deux méthodes les critères suivants :

- La complétude
- La complexité temporelle et spatiale mesurée par :
  - **b** : branchement maximal de l'arbre de recherche
  - **d** : profondeur de la meilleur solution
  - **m** : profondeur maximale de l'espace d'états

5.2 Comparatif détaillé

Par rapport au problème posé qui consiste à trouver toutes les solution (donc à parcourir l'ensemble de l'arbre, on a :

- Pour la recherche en profondeur d'abord :
  - Complète car b est fini

- Complexité en temps : $O(b^m)$
- Complexité en espace : $O(b \times m)$

— Pour la recherche en largeur d'abord :

- Complète car b est fini
- Complexité en temps : $O(b^m)$
- Complexité en espace : $O(b^m)$ car au pire cas, on doit stoker dans la file l'ensemble des chemins de l'arbre

Ainsi, on se rend compte que si la complexité temporelle est identique pour les deux méthodes, la complexité spatiale est clairement à l'avantage de la recherche en profondeur. Cela est du au fait que l'on veuille trouver toutes les solutions : la recherche en largeur n'est pas adaptée.

Par contre, si l'on voulait trouver la ou les solutions optimales, la recherche en largeur d'abord serait beaucoup plus efficace que celle en profondeur d'abord. Les complexités spatiales et temporelles de la recherche en profondeur ne changerait pas et on serait obliger le stoker dans une liste l'ensemble des solution avant de sélectionner la plus courte avec un processus itératif par exemple.

Pour la recherche en largeur, la complexité spatiale serait en $O(b^d)$ comme la complexité temporelle et la solution optimale serait trouvée directement.

Conclusion

Pour conclure, ce second projet nous a permis de mettre en application les méthodes de recherche dans un espace d'états vues en cours et lors des TDs sur des exemples concrets.

Cela nous a permis de revenir sur les différentes étapes de résolution de ce type de problèmes, que ce soit par rapport à la modélisation du problème, l'implémentation en langage Lisp (fonctions de service, fonctions de recherche en largeur et en profondeur) ou l'analyse des résultats.