ROB316 - TP5 : Planification d'actions

Alice PHE & Dajing GU

Janvier 2020

1 Introduction

1.1 Résumé du Cours

Étant donnés des actions génériques possibles, on veut trouver une séquence d'actions instanciées (un plan) qui amène le système à un état final contenant les buts. La **Planification d'action** est donc une activité de construction d'un plan.

Des hypothèses de simplification sont faites à cause de l'incapacité d'expliciter toute la combinatoire. Les opérateurs peuvent être exprimés en STRIPS / ADL / PDDL. Dans **Planning Domain Definition Language (PDDL)**, un opérateur est composé de pré-conditions et post-conditions.

En utilisant différent types d'algorithmes, un planificateur d'actions peut enfin être construit.

1.2 Introduction du TP

En utilisant PDDL (Planning Domain Definition Language), on va traiter des problèmes en récupérant le planificateur d'actions CPT version 2 (cpt.exe).

2 Exercice 1

La commande pour récupérer le planificateur est :

```
cpt.exe -o domain-blocksaips.pddl -f blocksaips01.pddl
```

2.1 Signification des quatre opérateurs

Il y a quatre opérateurs dans le fichier de domaine: `pick-up`, `put-down`, `stack` et `unstack`. Ce sont les actions génériques qui amène le système au but final.

- `pick-up` : C'est l'action de récupérer une chose à la main
- `put-down` : l'action de poser l'objet qu'on a à la main sur la table
- `stack` : l'action de poser l'objet qu'on a sur la main sur un bloc qui est disponible
- `unstack` : l'action de récupérer un bloc posé sur un autre

2.2 Différence entre `put-down` et `stack`.

Dans l'un des cas `put-down`, on pose l'objet sur la table, et dans l'autre - stack -, on pose le block sur un autre

2.3 Fonctionnement du fluent `holding ?x`

Le fluent `holding ?x` signifie si le bloc est dans la pince : c'est une description de la **position intermédiaire** du cube qui n'est pas sur la table ou sur un bloc. S'il n'était pas présent, on aurait pu passer par des opérateurs qui agissent sur les cubes directement.

3 Exercice 2

La commande pour récupérer le planificateur est :

```
cpt.exe -o domain-blocksaips.pddl -f blocksaips01.pddl
```

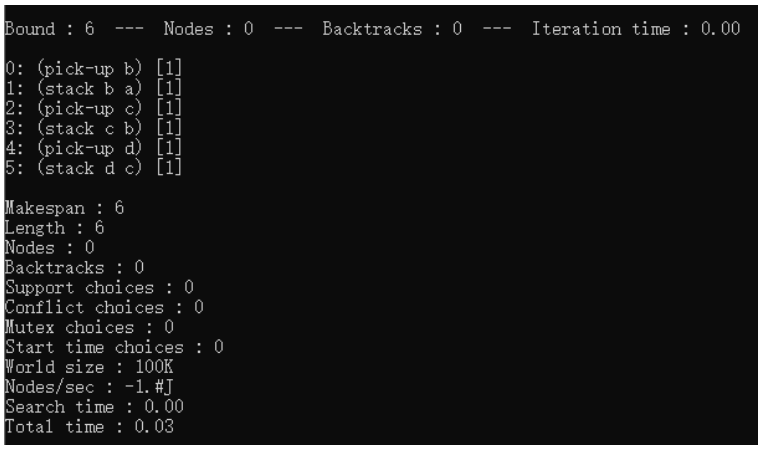

Figure 3.1: Résultat de d'exercice 2

Selon le résultat de cet exercice présenté dans la figure 3.1, on peut obtenir les résultats ci-dessous:

- Longueur du plan-solution : 6
- Temps de computation : 0.00 (search time)
- Nombre d'itération : 1
- Temps de chaque action : 1

On peut insérer une boucle de `(pick-up b)`, `(put-down b)`, `(pick-up b)`, `(put-down b)` au début du plan-solution fourni. Un tel plan-solution peut aussi nous aider à réaliser notre but. Ces autres plans-solutions ne sont pas fournis car ce planificateur nous donne seulement le plan-solution **le plus optimal**, c'est à que que le plan dont la longueur est la plus petite.

4 Exercice 3

La commande pour récupérer le planificateur est :

```
cpt.exe -o domain-blocksaips.pddl -f Exercice3.pddl
```

Le fichier `pddl` associé est présenté ci-dessous:

Code Listing 1: Fichier pour Exercice 3

```
1 (define (problem BLOCKS-4-0)
2 (:domain BLOCKS)
3 (:objects A B C D)
```

```
4  (:INIT (CLEAR B) (ON B C) (ON C A) (ON A D) (ONTABLE D) (HANDEMPTY))
5  (:goal (AND (CLEAR D) (ON D C) (ON C A) (ON A B) (ONTABLE B)))
6  )
```

```
Problem : 34 actions, 25 fluents, 81 causals
          6 init facts, 5 goals

Bound : 10  ---  Nodes : 0  ---  Backtracks : 0  ---  Iteration time : 0.00

0: (unstack b c) [1]
1: (put-down b) [1]
2: (unstack c a) [1]
3: (put-down c) [1]
4: (unstack a d) [1]
5: (stack a b) [1]
6: (pick-up c) [1]
7: (stack c a) [1]
8: (pick-up d) [1]
9: (stack d c) [1]

Makespan : 10
Length : 10
Nodes : 0
Backtracks : 0
Support choices : 0
Conflict choices : 0
Mutex choices : 0
Start time choices : 0
World size : 100K
Nodes/sec : 0.00
Search time : 0.02
Total time : 0.07
```

Figure 4.1: Résultat de d'exercice 3

Selon le résultat de cet exercice présenté dans la table 4.1, on peut obtenir les résultats ci-dessous:

Longueur du plan-solution	**Temps de computation (s)**	**Nombre d'itération**	**Temps de chaque action**
10	0.02	1	1

Table 4.1: Table de résultats

5 Exercice 4

La commande pour récupérer le planificateur est :

```
cpt.exe -o domain-blocksaips.pddl -f Exercice4.pddl
```

Le fichier `pddl` associé est présenté ci-dessous:

Code Listing 2: Fichier pour Exercice 4

```
1  (define (problem BLOCKS-10-0)
2  (:domain BLOCKS)
3  (:objects A B C D E F G H I J)
4  (:INIT (CLEAR C) (ON C G) (ON G E) (ON E I) (ON I J) (ON J A) (ON A B) (ONTABLE B)
5          (CLEAR F) (ON F D) (ON D H) (ONTABLE H) (HANDEMPTY))
6  (:goal (AND (CLEAR C) (ON C B) (ON B D) (ON D F) (ON F I) (ON I A) (ON A E) (ON E H)
7          (ON H G) (ON G J) (ONTABLE J)))
8  )
```

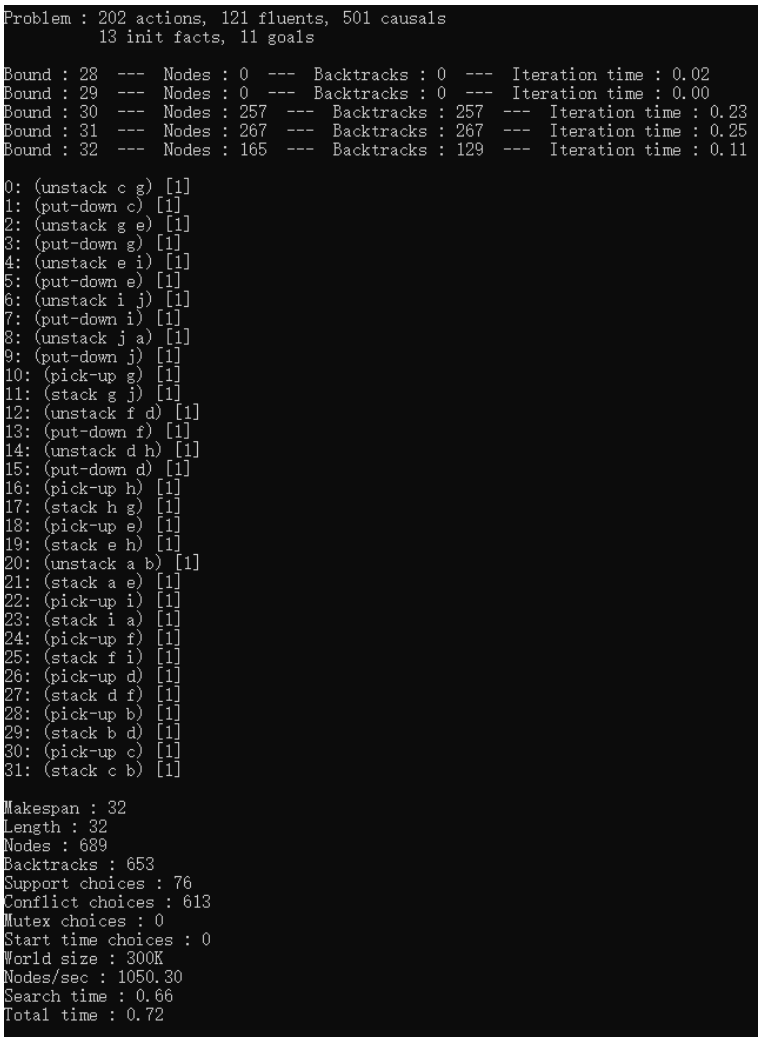

Figure 5.1: Résultat de d'exercice 4

Selon le résultat de cet exercice présenté dans la table 5.1, on peut obtenir les résultats ci-dessous:

Longueur du plan-solution	**Temps de computation (s)**	**Nombre d'itération**	**Temps de chaque action**
32	0.66	5	1

Table 5.1: Table de résultats

6 Exercice 5

La commande pour récupérer le planificateur est :

```
cpt.exe -o domain-noeuds.pddl -f Exercice5.pddl
```

6.1 Écriture du problème

Selon l'énoncé de l'exercice, on a réécrit le problème, dont le fichier `pddl` est présenté ci-dessous:

Code Listing 3: Définition du domaine NOEUDS

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Graph acyclique
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (domain NOEUDS)
  (:requirements :strips)
  (:predicates ((on ?x)
             arc ?x ?y)
            )

  (:action go
         :parameters (?x ?y)
         :precondition (and (on ?x) (arc ?x ?y))
         :effect
         (and (not (on ?x)) (on ?y))))
```

6.2 Test du modèle

Afin de tester notre modèle, on a crée un graphe acyclique comme présenté dans la figure 6.1.

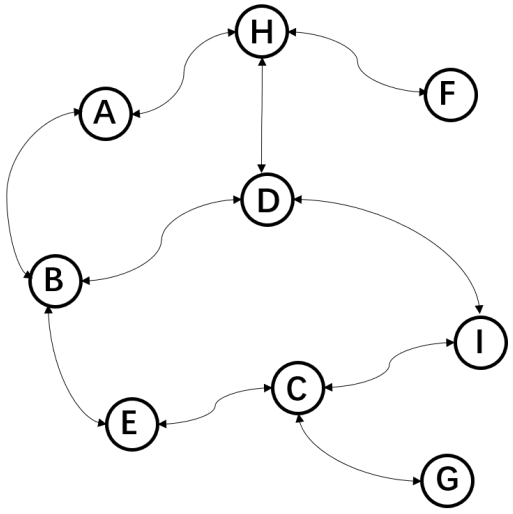

Figure 6.1: Le graphe acyclique de l'exercice 5

A est le noeud de commencement et **F** est le but.

Code Listing 4: Code du Test

```
(define (problem GRAPH-NOEUDs-9-0)
(:domain NOEUDS)
(:objects A B C D E F G H I)
(:INIT (ON A) (arc A B) (arc A H) (arc B A) (arc B D) (arc B E) (arc C E) (arc C I)
         (arc C G) (arc D H) (arc D B) (arc D I) (arc E B) (arc E C) (arc F H)
         (arc G C) (arc H F))
(:goal (AND(ON F))))
```

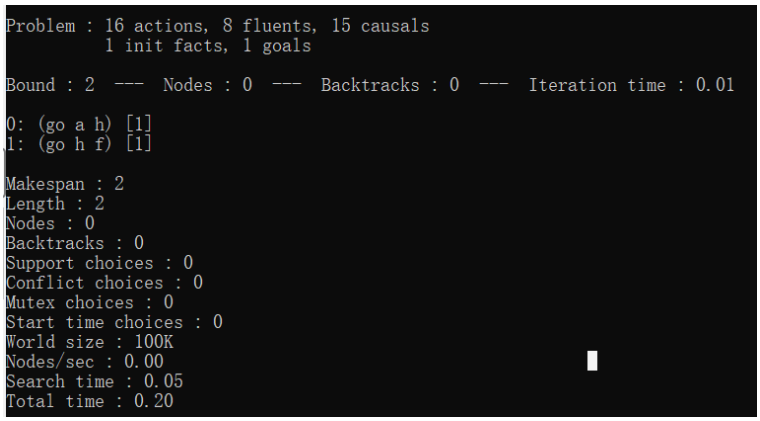

Figure 6.2: Le graphe acyclique de l'exercice 5

Selon le résultat présenté dans la figure 6.2, le chemin optimal est **A** $\rightarrow$ **H** $\rightarrow$ **F**, qui est sans doute la séquence la plus courte. Cette méthode de planification est assez efficace puisqu'il a essayé beaucoup de séquence d'action avant de trouver la plus optimale. Cependant comparé à d'autre algorithmes de recherche de chemin - A*, D* - qui permettent de mettre des poids et une direction de recherche pour optimiser l'efficacité du temps de recherche du chemin, on peut inférer que cette méthode est moins bonne.

7 Exercice 6

La commande pour récupérer le planificateur est :

```
cpt.exe -o domain-singe.pddl -f singe-bananes01.pddl
```

7.1 Problème du singe

Le fichier `pddl` du problème associé est présenté ci-dessous avec comme instanciation les objets singe, boîte et banane et comme objectif pour le singe de récupérer les bananes, - et le fichier du domaine avec les 6 opérateurs étant dans l'archive envoyée :

Code Listing 5: Fichier pour l'exercice 6

```
(define (problem MONKEY-1)
    (:domain MONKEY)
  (:objects A B C MyMonkey MyBox MyBananas )
  (:init (monkey MyMonkey)
         (box MyBox)
         (bananas MyBananas)
         (onfloor)
         (at MyMonkey A)
         (at MyBox B)
         (at MyBananas C)
)
  (:goal (and (hasbananas)))
)
```

```
(define (domain MONKEY)
  (:requirements :strips)
  (:predicates (at ?x ?y)
    (onfloor)
    (onbox ?x)
    (hasbananas)
    (monkey ?x)
```

```
      (box ?x)
      (bananas ?x)
      )

   (:action aller
      :parameters (?m ?x ?y)
      :precondition ( and (monkey ?m) (onfloor) (at ?m ?x))
      :effect
        (and (at ?m ?y)
        (not (at ?m ?x))
         )
   )

   (:action pousser
      :parameters (?m ?b ?x ?y)
      :precondition ( and (monkey ?m) (box ?b) (at ?b ?x) (at ?m ?x) (onfloor))
      :effect
        (and (at ?m ?y)
        (not (at ?m ?x))
        (at ?b ?y)
        (not (at ?b ?x))
        )
   )

   (:action monter
      :parameters (?m ?b ?x)
      :precondition ( and (monkey ?m) (box ?b) (at ?b ?x) (at ?m ?x) (onfloor) )
      :effect
        (and (not (onfloor))
        (onbox ?x)
        )
   )

   (:action descendre
      :parameters (?m ?b ?x)
      :precondition ( and (monkey ?m) (box ?b) (at ?b ?x) (at ?m ?x) (onbox ?x) )
      :effect
        (and (onfloor)
        (not (onbox ?x))
        )
   )

   (:action attraper
      :parameters (?m ?x)
      :precondition ( and (monkey ?m) (bananas ?x) (onbox ?x) (at ?m ?x))
      :effect
        ( and (hasbananas)
        )
   )

   (:action lacher
      :parameters (?m ?x ?b)
      :precondition ( and (monkey ?m) (bananas ?b) (hasbananas) (at ?m ?x))
      :effect
        ( and (not(hasbananas))
          (at ?b ?x)
        )
   )
)
```

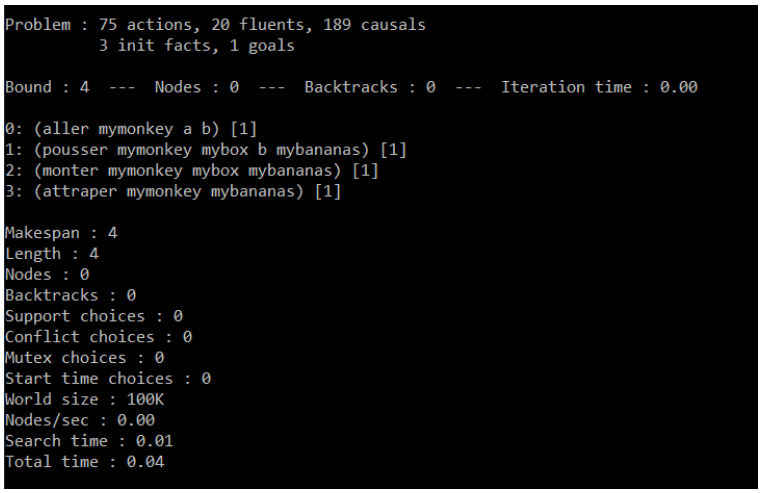

Figure 7.1: Résultat de d'exercice 6

Selon le résultat de cet exercice présenté dans la table 7.1, on peut obtenir les résultats ci-dessous:

Longueur du plan-solution	**Temps de computation (s)**	**Nombre d'itération**	**Temps de chaque action**
4	0.01	1	0.00

Table 7.1: Table de résultats

La solution ainsi obtenue en 1 itération est que : le singe aille à la boîte, il pousse la boîte à la position sous les bananes, monte sur la boîte, puis récupère les bananes.

L'opérateur 'pousser' souffre d'un problème de **qualification** : on a pas pu lister la **précondition** de poids et de force associée au singe.

8 Exercice 7

La commande pour récupérer le planificateur est :

```
cpt.exe -o domain-hanoi.pddl -f Exercice7-5.pddl
```

8.1 Initialisation du problème

On a défini le problème des tours de Hanoi avec le nombre de disques entre 1 et 5, le fichier `pddl` du problème présenté ci-dessous appartient au cas de 5 disques.

Code Listing 6: Écriture du problème pour l'exercice 7 (5 disques)

```
(define (problem TOUR-HANOI)
(:domain HANOI)
(:objects p1 p2 p3 d1 d2 d3 d4 d5)
(:init

    (peg p1) (peg p2)   (peg p3)
    (disk d1) (disk d2) (disk d3) (disk d4) (disk d5)

    (smaller d1 p1) (smaller d2 p1) (smaller d3 p1) (smaller d4 p1) (smaller d5 p1)
    (smaller d1 p2) (smaller d2 p2) (smaller d3 p2) (smaller d4 p2) (smaller d5 p2)
    (smaller d1 p3) (smaller d2 p3) (smaller d3 p3) (smaller d4 p3) (smaller d5 p3)
    (smaller d1 d2) (smaller d1 d3) (smaller d1 d4) (smaller d1 d5)
    (smaller d2 d3) (smaller d2 d4) (smaller d2 d5)
    (smaller d3 d4) (smaller d3 d5)
```

```
    (smaller d4 d5)

    (on d1 d2) (on d2 d3)(on d3 d4) (on d4 d5) (on d5 p1)
    (clear d1)  (clear p2) (clear p3)

    (PinceEmpty)
)
(:goal  (and (on d1 d2) (on d2 d3) (on d3 d4) (on d4 d5) (on d5 p3))
)
)
```

8.2 Définition du domaine

La définition des quatre opérateurs sont présenté ci-dessous.

Code Listing 7: Définition du domaine de l'exercice 7

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Tour de hanoi
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (domain HANOI)
  (:requirements :strips)
  (:predicates (peg ?x)
               (disk ?x)
               (smaller ?x ?y)
               (on ?x ?y)
               (clear ?x)
               (PinceEmpty)
               (onPince ?x)
               (PinceFull)
              )
  (:action remove_from_peg
         :parameters (?x ?y)
         :precondition (and (disk ?x) (peg ?y) (on ?x ?y) (clear ?x) (PinceEmpty) )
         :effect
         (and (not(PinceEmpty)) (onPince ?x) (PinceFull) (clear ?y) (not(on ?x ?y)) ))
  (:action put_on_peg
         :parameters (?x ?y)
         :precondition (and (disk ?x) (peg ?y) (onPince ?x) (clear ?x) (clear ?y) (smaller ...
             ?x ?y) )
         :effect
         (and (PinceEmpty) (not(onPince ?x)) (not(PinceFull)) (on ?x ?y) (not(clear ?y))))
  (:action remove_from_disk
         :parameters (?x ?y)
         :precondition (and (disk ?x) (disk ?y) (on ?x ?y) (clear ?x) (PinceEmpty))
         :effect
         (and (not(PinceEmpty)) (onPince ?x)(PinceFull) (clear?y) (not(on ?x ?y))))
  (:action put_on_disk
         :parameters (?x ?y)
         :precondition (and (disk ?x) (disk ?y) (onPince ?x) (clear ?x)  (clear ?y) ...
             (smaller ?x ?y) )
         :effect
         (and (PinceEmpty) (not(onPince ?x)) (not(PinceFull)) (on ?x ?y) (not(clear ?y))))
)
```

8.3 Résultat

Selon le résultat de cet exercice présenté dans la table 8.1, on peut obtenir les résultats ci-dessous:

Nombre de risque	Longueur du plan-solution	Temps de computation (s)	Nombre d'itération
1	2	0.02	1
2	6	0.03	1
3	14	0.09	5
4	30	54.12	17
5	NOT FOUND	NOT FOUND	NOT FOUND

Table 8.1: Table de résultats

Pour les cas dont le nombre de disque est entre 1 à 4, on peut obtenir un plan-solution optimal. Mais pour le cas de 5 disques, le temps d'exécution est trop long donc il ne trouve pas la solution.

On peut déduire que $\boldsymbol{L = 2^{n+1} - 2}$, avec **L** la longueur du plan-solution et **n** le nombre des disques, qui peut être expliqué par le fait que les actions sont séparés en 2 types: l'action de `remove` et l'action de `put_on`.

Plus le nombre des disques **n** est grand, plus la longueur du plan-solution **L** est grande, et plus de méthodes il faut essayer pour trouver celle d'optimale.

8.4 Analyse de pseudo-code

Le pseudo-code fourni est une méthode récursive. La complexité temporelle du problème de la tour de Hanoi est $O(2^n)$ puisque le déplacement de **n** disques est séparé en deux déplacements de **n-1** disques: $T(n) = 2 \cdot T(n-1) + 1$, qui correspond à ce que l'on a trouvé dans l'exercice 3.

Code Listing 8: Pseudo-codede l'exercice 7

```
1 PROCEDURE Hanoi(NDisques, PicDepart, PicIntermediaire, PicArrivee)
2     SI NDisques different de 0 ALORS
3         Hanoi(NDisques - 1, PicDepart, PicArrivee, PicIntermediaire);
4         Afficher("Deplacer le disque numero" + NDisques + <du pic > + PicDepart + <au pic> ...
              + PicArrivee + < \n >);
5         Hanoi(NDisques - 1, PicIntermediaire, PicDepart, PicArrivee);
6     FINSI
7 FINPROCEDURE
```

- On déplace tout d'abord n-1 disques de `PicDepart` à `PicIntermediaire`;
- On déplace ensuite le n^{ieme} disque de `PicDepart` à `PicArrivee`;
- On déplace enfin n-1 disques de `PicIntermediaire` à `PicArrivee`

La différence entre **CPT** et **Récursive** est que **CPT** utilise la méthode de retour arrière(Backtrack) pour trouver la solution finale et pert donc beaucoup de temps à tester des solutions incorrectes. Alors que **Récursive** sait déjà comment obtenir la solution optimale, et va ainsi prendre moins de temps pour le calcul une fois qu'on a implémenté le modèle.

En conclusion, la méthode récursive est meilleure que la méthode backtrack.