ROB316 - TD
Planification d'actions

BRAMBILLA Davide Luigi - GOMES DA SILVA Rafael

20 Janvier 2020

Introduction

Ce TD est consacré à la compréhension et l'écriture de domaines et des opérateurs en code *PDDL* afin de pouvoir réaliser une tache en utilisant la planification d'actions. Nous allons utiliser le *CPT = Constraint Programming Temporal Planner* est un compilateur temporel et permet d'explorer les différents actions et les mettre en série afin de pouvoir accomplir la tache désirée.

Afin de pouvoir exécuter le code nous devrons écrire:

./cpt.exe -o *nom-domain*.pddl -f *nom-probleme*.pddl

À l'interieur de cet document nous avons reporté les reponses aux questions des differents exercices.

1 Exercice 1

1.1 Que signifient les quatre opérateurs du fichier de domaine?

Le fichier de domaine a le contenu suivante:

Fichier 1: Contenu du ficher de domaine

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 4 Op-blocks world
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (domain BLOCKS)
  (:requirements :strips)
  (:predicates (on ?x ?y)
         (ontable ?x)
         (clear ?x)
         (handempty)
         (holding ?x))

  (:action pick-up
       :parameters (?x)
       :precondition (and (clear ?x) (ontable ?x) (handempty))
       :effect
       (and (not (ontable ?x))
        (not (clear ?x))
        (not (handempty))
        (holding ?x)))

  (:action put-down
       :parameters (?x)
       :precondition (holding ?x)
       :effect
       (and (not (holding ?x))
        (clear ?x)
        (handempty)
        (ontable ?x)))

  (:action stack
       :parameters (?x ?y)
       :precondition (and (holding ?x) (clear ?y))
       :effect
       (and (not (holding ?x))
        (not (clear ?y))
        (clear ?x)
        (handempty)
        (on ?x ?y)))

  (:action unstack
       :parameters (?x ?y)
       :precondition (and (on ?x ?y) (clear ?x) (handempty))
```

```
:effect
(and (holding ?x)
 (clear ?y)
 (not (clear ?x))
 (not (handempty))
 (not (on ?x ?y)))))
```

Les opérateurs indiquent les **paramètres** qui vont être affectés, la **pre-condition** qui doit être vérifiée afin que l'action puisse s'exécuter et, enfin, l'**effet** que l'action ramène sur le système. Les quatre opérateurs utilisés sont:

- **pick-up** qui soulève un *cube* depuis la *table*. Il a comme pre-condition le fait que l'élément à soulever est présent sur la table, que la main du robot soit libre et a comme effet le soulèvement de l'élément de la table.
- **put-down** qui génère l'ajout d'un *cube* sur la *table*. Il a comme pre-condition le fait que l'élément à ajouter soit déjà dans la main du robot et l'effet est l'ajout du *cube* à la *table* et la libération de la main occupée.
- **stack** qui ajoute un *cube* en dessus d'un autre. Comme condition il vérifie que un *cube* est déjà présent sur la *table* et que le deuxième *cube* est dans la main du robot. L'effet est la libération de la main et l'ajout du bloc en dessus duquel déjà sur la table.
- **unstack** qui enlève un *cube* qui est déjà en dessus d'un autre et présent dans la *table*. La pre-condition est le fait que la main du robot soit libre et que il y a effectivement deux *cubes* un en dessus de l'autre et l'effet est le déplacement du *cube* en haut.

1.2 Quelle est la différence entre l'opérateur *put-down* et l'opérateur *stack*? Pourquoi faut-il dissocier ces deux cas?

La différence entre l'opérateur *put-down* et l'operateur *stack* est dans le fait que le premier ajoute un *cube* sur la table et le deuxième ajoute un *cube* en dessus d'un autre *cube* spécifié.

Les deux cas doivent être dissociés car ils génèrent deux effets différents et, surtout, ils ont besoin d'un nombre diffèrent des paramètres et des conditions préalables différents pour s'exécuter.

1.3 Que veut dire le fluent *(holding ?x)*? A quoi sert-il? (Si ce fluent n'était pas là, comment faudrait-il changer l'écriture des opérateurs pour représenter le monde des cubes?)

Le fluent *holding* donne l'état de la main du robot: il dit si la main est en train de maintenir quelque chose. Dans le cas des opérateurs, dans leur effets, il communique si l'opérateur à généré un positionnement d'un *cube* et donc la libération du manipulateur où si l'opérateur a généré une suppression d'un *cube* de la *table* et donc l'occupation de la main du robot.

On devrait utiliser le mot *handempty* qui nous donne une information similaire: complémentaire par rapport à laquelle donnée par le robot. Le problème est donnée par le fait qu'on ne pourra plus dire quel objet on est en train de maintenir dans la main.
Pour pouvoir savoir quel objet est maintenu dans la main, on pourra définir l'état des autres objets du problème (s'ils sont où pas sur la *table*) et le seul objet qui ne sera pas defini on saura que il sera dans la main du *robot*.

2 Exercice 2

Afin d'exécuter le fichier nous allons écrire dans le terminal:

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

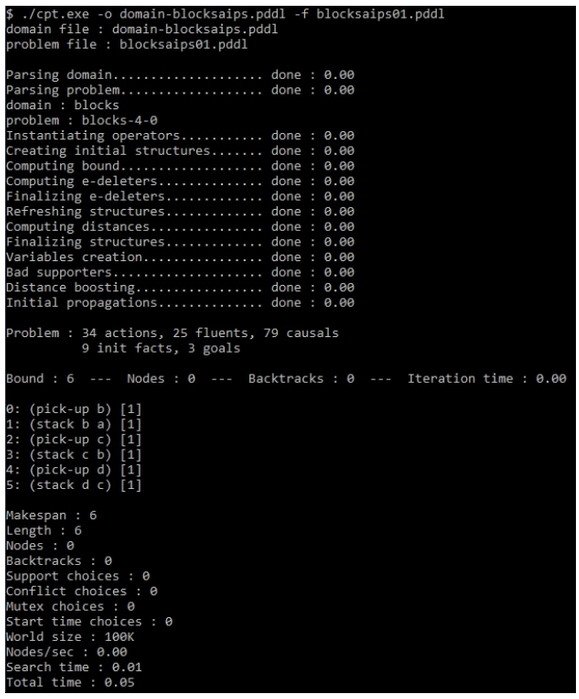

Figure 2.1: Résultat de l'exécution de l'exercice 2

2.1 Analyse de l'exécution du planificateur sur *blocksaips01.pddl*

Ci-dessous nous avons reporté les caractéristiques obtenues de l'exécution du code:

- Longueur du plan-solution: 6
- Temps écoulé: 0.05
- Itérations: 1
- Duré de chaque action: 1 (en durée d'opérations)

2.2 Quels plans-solutions plus longs peut-on imaginer pour résoudre ce problème de cubes? Pourquoi ne sont-ils pas fournis par ce planificateur?

On pourrait imaginer plein de solutions pour résoudre ce problème: il s'agit de toutes les solutions qui ne contiennent pas des actions qui amènent directement au but espéré. Par exemple, le fait de déplacer le *cube d* tout de suite sur le *cube a* nous fera perdre deux actions car le premier à mettre sur le *cube a* doit être le *cube a*.

Ces plans-solutions ne sont pas fournis par le planificateur parce que une fois que il trouve le parcours de longueur optimale il reporte seulement ce-dernier.

3 Exercice 3

3.1 L'écriture du problème

Nous avons ré-écrit le problème comme reporté ci-dessous:

Fichier 2: Exercice 3: l'écriture du problème

```
(define (problem BLOCKS-4-0)
(:domain BLOCKS)
(:objects D B A C )
(:INIT (CLEAR B) (ON A D)(ON C A)(ON B C) (ONTABLE D)(HANDEMPTY))
(:goal (AND (CLEAR D) (ONTABLE B) (ON A B)(ON C A) (ON D C))))
```

3.2 Quelle est la longueur du plan-solution? En combien de temps est-il trouvé? Combien y a-t-il eu d'itérations?

Ci-dessous nous avons reporté les caractéristiques obtenues de l'execution du code:

- Longueur du plan-solution: 10
- Temps écoulé: 0.06
- Itérations: 1
- Duré de chaque action: 1 (en durée d'opérations)

4 Exercice 4

4.1 L'écriture du problème

Nous avons ré-écrit le problème comme reporté ci-dessous:

Fichier 3: Exercice 4: l'écriture du problème

```
(define (problem BLOCKS-10-0)
(:domain BLOCKS)
(:objects A B C D E F G H I J )
(:INIT (CLEAR C) (CLEAR F) (ON A B)(ON J A)(ON I J)(ON E I)(ON G E)(ON C G) (ON D H) (ON F D)
    (ONTABLE B)(ONTABLE H)(HANDEMPTY))
(:goal (AND (CLEAR C) (ONTABLE J) (ON G J)(ON H G) (ON E H)(ON A E)(ON I A)(ON F I)(ON D F)(ON B
    D)(ON C B))))
```

4.2 Quelle est la longueur du plan-solution? En combien de temps est-il trouvé? Combien y a-t-il eu d'itérations?

Ci-dessous nous avons reporté les caractéristiques obtenues de l'execution du code:

- Longueur du plan-solution: 32
- Temps écoulé: 0.62
- Itérations: 5
- Duré de chaque action: 1 (en durée d'opérations)

5 Exercice 5

5.1 L'écriture du problème

Nous avons écrit un domaine *PDDL* contenant l'opérateur comme demandé. Nous avons écrit le domaine comme reporté ci-dessous:

Fichier 4: Exercice 5: l'écriture du problème

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Graphe Acyclique
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (domain graph)
  (:requirements :strips)
  (:predicates (arc ?from ?to)
         (on ?from)
         )

  (:action jump
       :parameters (?from ?to)
       :precondition (and (on ?from) (arc ?from ?to))
       :effect (and (on ?to)(not(on ?from))))
)
```

Nous avons, ensuite, testé notre code sur deux différents graphes acycliques que nous avons reportés ci-dessous:

5.1.1 Cas 1

La situation que nous avons voulu reporter dans le cas 1 est la suivante:

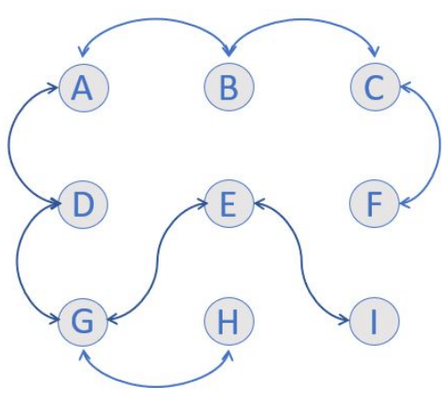

Figure 5.1: Exemple 1

Le code écrit est:

```
(define (problem graph)
(:domain graph)
(:objects A B C D E F G H I)
(:INIT (ON A) (ARC A B) (ARC B A) (ARC B C) (ARC C B) (ARC C F) (ARC F C) (ARC E I) (ARC I E) (ARC
    G H) (ARC H G) (ARC G E) (ARC E G) (ARC D G) (ARC G D) (ARC A D) (ARC D A))
(:goal (AND (ON I))))
```

Le résultat que nous avons obtenu est:

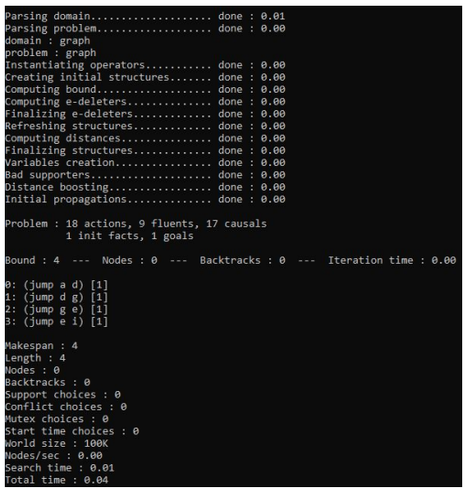

Figure 5.2: Résultat pour le premier exemple

où il est possible de voir que la séquence proposé est bien la correcte: *A-D-G-E-I.*

5.1.2 Cas 2

La situation que nous avons voulu reporter dans le cas 2 est la suivante:

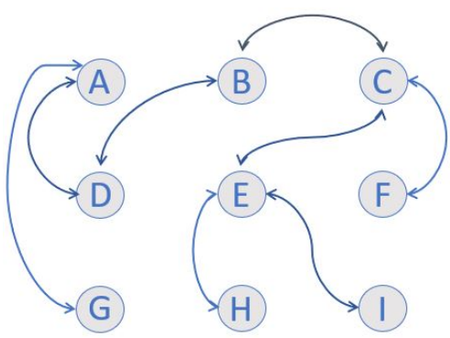

Figure 5.3: Exemple 2

Le code écrit est:

```
(define (problem graph)
(:domain graph)
(:objects A B C D E F G H I)
(:INIT (ON A) (ARC A D) (ARC D A) (ARC B D) (ARC D B) (ARC C B) (ARC B C) (ARC C F) (ARC F C) (ARC
    E C) (ARC C E) (ARC H E) (ARC E H) (ARC A G) (ARC G A) (ARC E I) (ARC I E))
(:goal (AND (ON I))))
```

Le résultat que nous avons obtenu est:

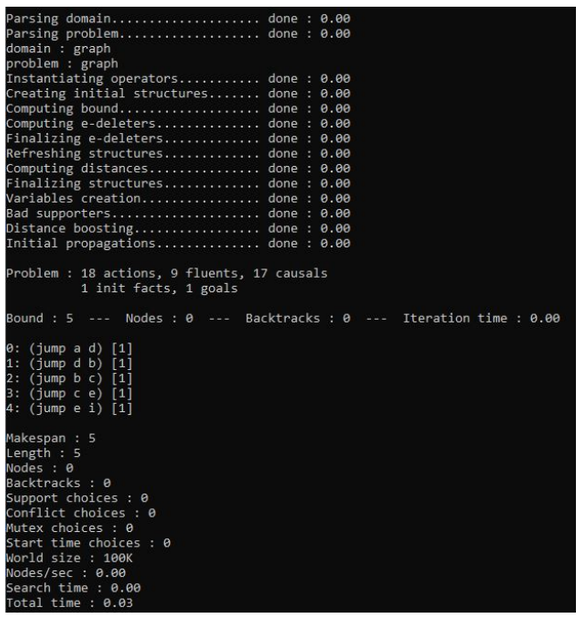

Figure 5.4: Résultat pour le deuxième exemple

où il est possible de voir que la séquence proposée est bien la correcte: *A-D-B-C-E-I*.

5.2 A votre avis, cette méthode de planification de chemin est-elle efficace? Pourquoi?

Cette méthode n'est pas très efficace pour le fait que elle vérifie une grande partie des actions possibles avant de trouver la meilleure.

6 Exercice 6

6.1 L'écriture du problème

Nous avons reporté ci-dessous le problème avec les conditions initiaux définies: un *singe* dans la position *A* sur le sol, une caisse sur le sol en position *B* et les *bananes* attachées au plafond en position *C*.

```
(  define (problem singe-bananes01)
   (:domain singe)
   (:objects A B C)
   (:INIT  (PosSing A)
         (PosCaisse B)
         (PosBanane C)
         (PosInit A)
         (Bas)
         (not(Haut))
         (not(HasBananas))
         (HandEmpty)
   )
   (:goal  (AND  (HasBananas)
         )
   )
)
```

Comme condition initiale nous avons défini les trois positions des trois acteurs (*singe*, *caisse*, *bananes*), nous avons défini **Bas** et *Haut* pour représenter la position du *singe*: respectivement s'il est sur le sol ou s'il est sur la caisse. De plus nous avons utilisé les deux prédicats *HasBananas* et *HandEmpty* afin de faire une distinction entre le moment quand le *singe* attrape les *bananes* et toutes les moments où il n'a rien dans ses mains.

Comme goal nous avons défini *seulement* ce que le signe doit faire: attraper les *bananes*.

6.2 L'écriture du domaine et des opérateurs

Dans le domaine nous avons defini les *predicates* que nous avons deja vu dans la section precedente.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Domain Singe
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(
   define (domain singe)
```

```
(:requirements :strips)
(:constants singe bananes caisse)
(:predicates  (PosSing ?x)
              (PosCaisse ?x)
        (PosBanane ?x)
        (PosInit ?x)
        (Bas)
        (Haut)
        (HasBananas)
        (HandEmpty)

)
```

De plus, nous avons défini les différents opérateurs demandés dans le texte et nous les avons reportés ci-dessous:

6.2.1 L'opérateur *Aller*

Cet opérateur nous permet de faire déplacer le *singe* d'une position initiale à une position finale.
On vérifie que le *singe* soit sur le sol et dans la *position de départ* et on génère le déplacement vers la *position de destination*.

```
(:action Aller
   :parameters (?from ?to)
   :precondition  (and (PosSing ?from)
              (Bas)
           )
     :effect  (and (PosSing ?to)
           (not(PosSing ?From))
        )
)
```

6.2.2 L'opérateur *Pousser*

Cet opérateur nous permet de donner la possibilité au *singe* de déplacer la *caisse* vers la position désirée.
On vérifie que le *singe* soit dans la même position de la *caisse* et on génère le déplacement de la *caisse* ver la *position de destination*.

```
(:action Pousser
   :parameters (?from ?to)
   :precondition  (and (PosSing ?from)
              (PosCaisse ?from)
              (Bas)
           )
     :effect  (and (PosCaisse ?to)
           (not(PosCaisse ?from))


     )
)
```

6.2.3 L'opérateur *Monter*

Cet opérateur nous permet de faire monter le singe sur la caisse. On vérifie que le *singe* soit en *Bas* et on la fait monter en passant à l'état *Haut*.

```
(:action Monter
   :parameters (?Pos)
   :precondition  (and (PosSing ?Pos)
              (PosCaisse ?Pos)
              (Bas)
           )
     :effect  (and (Haut)
           (not(Bas))
        )
)
```

6.2.4 L'opérateur *Descendre*

Cet opérateur nous permet de faire descendre le *singe* de la *caisse*. Son comportement est opposé par rapport à l'action de *monter*.

```
(:action Descendre
   :parameters (?Pos)
   :precondition  (and (PosSing ?Pos)
```

```
            (PosCaisse ?Pos)
            (Haut)
          )
     :effect  (and (Bas)
          (not(Haut))
        )
)
```

6.2.5 L'opérateur *Attraper*

Cet opérateur nous permet de terminer le jeu en faisant attraper les *bananes* au *singe*.
On vérifie que le *singe* soit monté sur la *caisse* et dans la position C où il y a les *bananes* et on lui les fait attraper.

```
(:action Attraper
   :parameters (?Pos)
   :precondition  (and (PosSing ?Pos)
            (PosCaisse ?Pos)
            (PosBanane ?Pos)
            (HandEmpty)
            (Haut)
          )
     :effect  (and (HasBananas)
          (not(HandEmpty))
        )
)
```

6.2.6 L'opérateur *Lâcher*

Cet opérateur nous permet de faire l'opération inverse à la précédente. Le *singe* posera les *bananes* qui tiens dans la main.
Nous avons choisi d'imposer que le *singe* peut libérer ses mains dans le cas où elle se trouve dans la position initiale (position A).

```
(:action Lacher
   :parameters (?Pos)
   :precondition  (and  (HasBananas)
            (PosInit ?Pos)
            (PosSing ?Pos)

          )
     :effect  (and (not(HasBananas))
          (PosBanane ?Pos)
          (HandEmpty)
        )
)
```

6.3 Quelle est le plan-solution ? En combien de temps est-il trouvé? Combien y a-t-il eu d'itérations?

Nous avons reporté ci-dessous les performances de notre algorithme.

- Longueur du plan-solution: 5
- Temps écoulé: 0.1
- Itérations: 1

Nous reportons aussi les resultats obtenus:

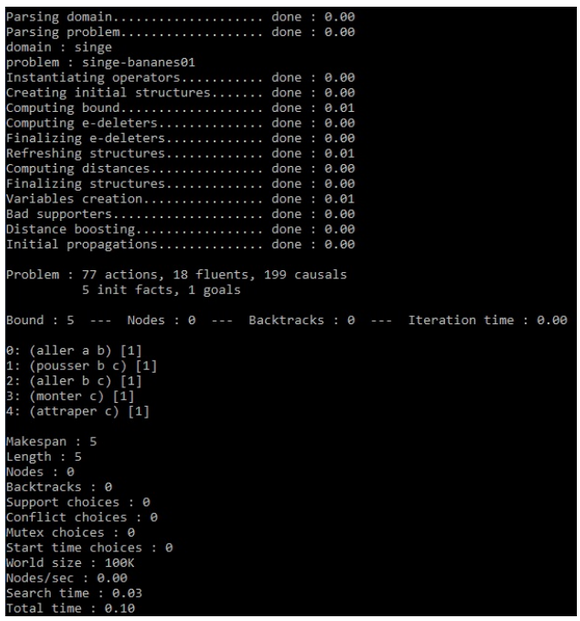

Figure 6.1: Résultat pour le problème de la singe et les bananes

Nous pouvons observer que le plan solution comprendre les instructions suivantes afin que la *singe* puisse attraper les *bananes*: *Aller-Pousser-Aller-Monter-Attraper*

6.4 Est-ce que l'opérateur *Pousser* écrit est un exemple du problème de la qualification ou de la ramification?

Le **Problème de la qualification** représente le cas où on ne peut pas lister toutes les *pré-conditions* dans un opérateur.

Le **Problème de la ramification** représente le cas où on ne peut pas lister toutes les *post-conditions* dans un opérateur.

L'opérateur **Pousser** est un exemple du problème de *qualification* parce que, comme remarqué dans l'énonce du TP, nous ne sommes pas en train de considérer le cas où la *caisse* est trop lourde: donc nous sommes en train de simplifier le problème à niveau de *pre-conditions* et donc on se retrouve à avoir le problème de *qualification*.

7 Exercice 7

7.1 Les problèmes

Il nous est demandé de définir le problème des tours de *Hanoi* dans des différents cas avec un nombre de disques entre 1 et 5.

Nous avons reporte ici le cas avec 4 disques.

```
(
  define (problem hanoi2)
  (:domain hanoi)
  (:objects p1 p2 p3 d1 d2 d3 d4)
  (:init
       (smaller d1 p1)
       (smaller d1 p2)
       (smaller d1 p3)

       (smaller d2 p1)
       (smaller d2 p2)
       (smaller d2 p3)

       (smaller d3 p1)
       (smaller d3 p2)
       (smaller d3 p3)

       (smaller d4 p1)
       (smaller d4 p2)
       (smaller d4 p3)

       (smaller d1 d2)(smaller d1 d3)(smaller d1 d4)
       (smaller d2 d3)(smaller d2 d4)
       (smaller d3 d4)
```

```
        (clearDisk d1)

        (clearPeg p2)
        (clearPeg p3)

        (onPeg d4 p1)

        (onDisk d1 d2)
        (onDisk d2 d3)
        (onDisk d3 d4)

        (PinceFree)

   )
   (:goal  (and (onDisk d1 d2)
           (onDisk d2 d3)
           (onDisk d3 d4)
           (onPeg d4 p3)
        )
   )
)
```

où nous pouvons observer les *prédicats* que nous avons utilisé et l'objectif que nous avons définie. Nous pouvons voir que le prédicat *smaller* nous permet de différencier les disques par leur dimension. Les prédicats *onDisk* et *onPeg* nous disent quand un disque est sur un autre ou sur la base. Le prédicat *PinceFree* nous dit si je suis en train de avoir dans la main un disque ou pas. Enfin les prédicats *clearPeg* et *clearDisk* nous disent quand un peg ou un disque n'ont pas des autres disques sur eux.

Enfin comme objectif nous avons défini la tour de *Hanoi* dans la position *p3*.

7.2 Le domaine et les opérateurs

Dans cette section nous avons pu écrire les prédicats que nous avons expliqué dans la section précédente

```
(
   define (domain hanoi)
   (:requirements :strips)
   (:predicates  (clearDisk ?x)
           (clearPeg ?x)
           (onDisk ?x ?y)
           (onPeg ?x ?y)
           (smaller ?x ?y) ;x<y
           (ReceiveDisk ?x)
           (PinceFree)
   )
```

et les quatre opérateurs que nous allons voir en détail.Nous les avons reporté ici:

7.2.1 L'opérateur *Take_from_peg*

Cet opérateur s'occupe de vérifier que je suis dans les conditions de pouvoir prendre un disque à partir de la base et reporte les conséquences de cette action: le fait de ne plus avoir la main libre, de libérer la base et du fait que le disque pris est maintenant dans ma main.

```
   (  :action Take_from_peg
      :parameters (?disk ?peg)
      :precondition  (and (onPeg ?disk ?peg)
                  (clearDisk ?disk)
                  (PinceFree)
               )
      :effect  (and (not(PinceFree))
               (ReceiveDisk ?disk)
               (clearPeg ?peg)
               (not(onPeg ?disk ?peg))
               (not(clearDisk ?disk))
            )
   )
```

7.2.2 L'opérateur *Leave_on_peg*

Cet opérateur permet de laisser un disque sur un peg dans le cas ce-dernier soit vide. Pour cet opérateur et les suivants nous n'allons pas expliquer le *pre-conditions* et *post-conditions* car elles sont facilement compréhensibles à partir du code.

```
   (  :action Leave_on_peg
      :parameters (?disk ?peg)
```

```
   :precondition  (and (clearPeg ?peg)
             (ReceiveDisk ?disk)

           )
   :effect  (and  (not(clearPeg ?peg))
           (not(ReceiveDisk ?disk))
           (onPeg ?disk ?peg)
           (PinceFree)
           (clearDisk ?disk)
        )
)
```

7.2.3 L'opérateur *Leave_on_disk*

Cet opérateur permet de laisser un disque sur un autre disque dans le cas ce-dernier soit vide.

```
(  :action Leave_on_disk
   :parameters (?disk1 ?disk2)
   :precondition  (and (smaller ?disk1 ?disk2)
             (clearDisk ?disk2)
             (ReceiveDisk ?disk1)
           )
   :effect  (and (not(ReceiveDisk ?disk1))
           (onDisk ?disk1 ?disk2)
           (PinceFree)
           (clearDisk ?disk1)
           (not(clearDisk ?disk2))
        )
)
```

7.2.4 L'opérateur *Take_from_disk*

Cet opérateur permet de prendre un disque à partir d'un autre disque.

```
(  :action Take_from_disk
   :parameters (?disk1 ?disk2)
   :precondition  (and (onDisk ?disk1 ?disk2)
             (clearDisk ?disk1)
             (PinceFree)
           )
   :effect  (and  (not(onDisk ?disk1 ?disk2))
           (ReceiveDisk ?disk1)
           (clearDisk ?disk2)
           (not(PinceFree))
           (not(clearDisk ?disk1))
        )
)
```

7.3 Les résultats

Nous avons reporté ci-dessous les résultats obtenus dans les différents cas:

Nombre de disques	Longueur du plan-solution	Nombre des Itérations	Temps d'exécution
1	2	1	0.02
2	6	1	0.04
3	14	5	0.07
4	30	17	1.9
5	-	-	-

Nous pouvons observer que notre code arrive à trouver des solutions pour les différents cas avec 1,2,3 et 4 disques. Nous pouvons observer que la croissance de la complexité n'est pas linéaire et augmente fortement avec le nombre des disques.

Les solutions qui ont été trouvé représentent les meilleurs solutions possibles qui, étant n le nombre des disques, sont composé par un nombre de actions égal à $2 * (2^n - 1)$. Le facteur 2 est donné par le fait que nous allons diviser un déplacement en deux actions: l'action de *take* et l'action de *leave*.

Pour la solution 5 nous n'arrivons pas à trouver une solution. Cela est dû au fait que la complexité est augmentée et nous allons approfondir cette thématique dans la section suivante.

7.3.1 La complexité de l'algorithme

Pour analyser la complexité de l'algorithme nous avons analysé le comportement du code *recursif* proposé.

Nous avons étudié son comportement dans le cas d'un problème avec 2 disques et nous avons reporté ci dessous son comportement et les messages qui va afficher à pendant son exécution.

```
PROCEDURE Hanoi(2, P1, P2, P3)
   SI 2 different de 0 ALORS
      Hanoi(1, P1, P3, P2);

      NOEXEC--- Hanoi(0, P1, P2, P3);
      --- Afficher(Deplacer le disque numero   + 1 + du pic  + P1 + au  pic  + P2 +  \n  );
      NOEXEC--- Hanoi(0, P1, P2, P3);

      Afficher( Deplacer le disque numero   + 2 + du pic  + P1 + au  pic  + P3 +  \ n );

      Hanoi(1, P2, P1, P3);

      NOEXEC--- Hanoi(0, P2, P3, P1);
      --- Afficher(Deplacer le disque numero   + 1 + du  pic  + P2 + au  pic  + P3 +  \ n );
      NOEXEC--- Hanoi(0, P1, P2, P3);

   FINSI
FINPROCEDURE


---Messages affiches---
Afficher( Deplacer le disque numero   + 1 + du  pic  + P1 + au  pic  + P2 +  \ n );
Afficher( Deplacer le disque numero   + 2 + du  pic  + P1 + au  pic  + P3 +  \ n );
Afficher( Deplacer le disque numero   + 1 + du  pic + P2 + au  pic  + P3 +  \ n );
```

Nous pouvons voir que cet algorithme arrive directement à la solution meilleure et, pour un problème de *Hanoi* avec n disques, comporte donc un complexité de $2^n - 1$. Cela corresponde exactement à la longueur du plan-solution trouvé par notre algorithme et on peut l'écrire pour le fait qu'on connaît déjà comment trouver la solution du problème étant donné que le problème a une solution que peut être trouvé de façon récursive.

Notre algorithme, par contre, va essayer de trouver la solution la meilleure entre lesquels disponibles mais pour faire cela, il va essayer les différents solutions possibles qui dépendent du nombre de prédicats et d'actions que nous allons définir ce qui nous ramène à avoir une complexité beaucoup plus élevé.

Cette complexité nous empêche de trouver une solution dans le cas avec 5 disques: le chemin optimale est composé par 31 opérations que dans notre cas deviennent 62 pour le fait que on divise en deux opérations le déplacement d'un disque comme précédemment anticipé.

7.3.2 Quelle est la différence entre la résolution du problème des tours de *Hanoi* avec cette fonction et avec un planificateur d'actions comme CPT?

La différence, comme déjà remarqué, est dans le fait que le planificateur va essayer différents actions afin de trouver le plan-solution optimale et la code récursif va seulement exécuter son code dans lequel nous y avons déjà inséré la solution. Dans le code récursif c'est nous que on connaît comment arriver à la solution et on programme en sachant déjà qu'il va fonctionner. Dans le planificateur c'est le langage que va trouve la solution parmi les possibles configurations.