ÉCOLE NATIONALE SUPÉRIEURE DE TECHNIQUES AVANCÉES

ROB316 Planification et contrôle
Enseignant : Philippe Morignot

TP - Planification d'actions

Luiz Henrique MARQUES GONÇALVES

Palaiseau, France 2025

1. Exercice 1

1.1. Que signifient les quatre opérateurs du fichier de domaine ?

Le fichier de domaine spécifie quatre opérateurs dans un système de planification pour manipuler des blocs.

- L'opérateur `pick-up (?x)` représente **l'action de saisir un bloc `x`**, à condition qu'il soit libre (`clear ?x`), qu'il soit sur la table (`ontable ?x`) et que la main du manipulateur soit vide (`handempty`). Cette action retire le bloc de la table (`not (ontable ?x)`), le rend non libre (`not (clear ?x)`), occupe la main du manipulateur (`not (handempty)`) et met le bloc en position "tenu" (`holding ?x`).
- L'opérateur `put-down (?x)` **représente l'action de poser un bloc `x` sur la table**, à condition qu'il soit tenu au moment de l'action (`holding ?x`). Cette action libère le bloc (`clear ?x`), rend la main du manipulateur vide (`handempty`), pose le bloc sur la table (`ontable ?x`) et enlève l'état "tenu" du bloc (`not (holding ?x)`).
- L'opérateur `stack (?x ?y)` **représente l'action d'empiler un bloc `x` sur un autre bloc `y`**, à condition que `?x` soit tenu et que `y` soit libre. Après cette action, `x` n'est plus tenu, devient libre, et se retrouve au-dessus de `y` (`on ?x ?y`). De plus, `y` n'est plus libre, et la main du manipulateur devient vide.
- Enfin, l'opérateur `unstack (?x ?y)` **représente l'action de retirer un bloc `x` empilé sur un bloc `y`**, à condition que `x` soit au-dessus de `y`, que `x` soit libre et que la main du manipulateur soit vide. Après cette action, la main n'est plus vide et tient `x`; `y` devient libre, `?x` n'est plus libre ni empilé sur `y`.

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

La différence principale entre les opérateurs `put-down` et `stack` réside dans la position finale du bloc manipulé. Avec `put-down`, le bloc manipulé (`?x`) est placé sur la table, tandis qu'avec `stack`, le bloc `x` est empilé sur un autre bloc `y`.

Il est nécessaire de dissocier ces deux cas car la table possède un comportement particulier, différent de celui des blocs, notamment concernant le paramètre `clear`. Il n'est pas nécessaire de vérifier si la table est libre pour y poser un bloc – on considère toujours qu'il y a suffisamment de place. Pour un bloc, en revanche, la situation est différente : un seul bloc peut être placé au-dessus. Par conséquent, il est essentiel de modifier l'état `clear` d'un bloc dès qu'un autre bloc est ajouté ou retiré de sa surface.

La même justification s'applique à la différence entre les opérateurs `unstack` et `pick-up` : il n'est pas pertinent de manipuler le paramètre `clear` pour la table, car cette dernière est toujours considérée comme accessible.

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

Le fluent `(holding ?x)` signifie que le bloc `x` est tenu par le manipulateur. Il sert à identifier le bloc actuellement dans la main pour vérifier les préconditions d'actions comme `put-down` ou `stack`, qui ne peuvent être réalisées que si le manipulateur tient déjà le bloc.

Si `(holding ?x)` n'existait pas, il faudrait remplacer cette condition par une vérification plus complexe:

- Vérifier que `x` n'est pas sur un autre bloc (`(not (on ?x ?y))` pour tous les blocs `y`).
- Vérifier que `x` n'est pas sur la table (`(not (ontable ?x))`).

Cela impliquerait que chaque opérateur gère tous les blocs individuellement dans ses paramètres d'entrée et dans ses préconditions. La précondition correspondante deviendrait :

`(and (not(on ?x ?a))(not(on ?x ?b))...(not(on ?x ?n))(not(ontable ?x)))`

En résumé, `(holding ?x)` simplifie le modèle en encapsulant toutes ces vérifications dans une seule condition, rendant la représentation du monde des blocs plus claire, efficace, et réutilisable car elle élimine la nécessité de connaître la quantité de blocs dans le système au préalable.

2. Exercice 2

2.1. Quelle est la longueur du plan-solution ? En combien de temps est-il trouvé? Combien y a-t-il eu d'itération ? Combien de temps dure chaque action du plan-solution ?

La longueur du plan-solution est de 6, trouvé en 0.02 seconde. Il n'y a eu qu'une seule itération.

En considérant que le "makespan" représente la durée totale de la tâche, le makespan est égal à la longueur, soit 6, et chaque action dure, donc, une unité de temps.

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 ?

Une infinité de plans-solutions plus longs peuvent être imaginées. Il suffit de réaliser des mouvements inutiles et de les annuler ensuite, ce qui ajoutera 2 à la longueur, comme par exemple : "empiler A sur B, puis le désempiler et finalement empiler correctement B sur A".

Ces plans ne sont pas fournis par ce planificateur car il cherche à trouver la planification optimale, c'est-à-dire qu'il affiche la meilleure solution trouvée.

3. Exercice 3

3.1. Écrire le problème de planification suivant (un fichier PDDL) pour ce domaine des cubes :

- **État initial : une tour de cubes B / C / A / D / table et la pince est vide.**
- **Buts : une tour de cubes D / C / A / B.**

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

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

La longueur du plan-solution est de 10, trouvé en 0.02 seconde. Il n'y a eu qu'une seule itération.

4. Exercice 4

4.1. Écrire le problème de planification suivant (un fichier PDDL) pour ce domaine des cubes :

- **État initial : une tour de cubes C / G / E / I / J / A / B / table, une deuxième tour de cubes F / D / H / table et la pince est vide.**
- **Buts : une tour de cubes C / B / D / F / I / A / E / H / G / J.**

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

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

La longueur du plan-solution est de 32, trouvé en 0.63 seconde. Il y a eu 7 itérations.

5. Exercice 5

5.1. Planification de chemin

```
(define (domain NOEUDS)
  (:requirements :strips)
  (:predicates (arc ?from ?to)
               (agentPos ?from))

  (:action avancer
       :parameters (?from ?to)
       :precondition (and (arc ?from ?to) (agentPos ?from))
       :effect (and (not (agentPos ?from)) (agentPos ?to))))
```

Un graphe simple est défini avec des nœuds A, B, C, D, E, et des arcs connectant ces nœuds de manière séquentielle. L'objectif est de trouver un chemin allant du nœud A au nœud E.

```
(define (problem GRAPH)
(:domain NOEUDS)
(:objects A B C D E)
(:INIT (ARC A B) (ARC B C) (ARC C D) (ARC D E) (ARC A B) (AGENTPOS A))
(:goal (AGENTPOS E))
)
```

Le planificateur trouve le chemin, d'une longueur de 4, en une seule itération.

Cependant, ce planificateur n'est pas particulièrement efficace pour la planification de chemins, car il est conçu pour résoudre des problèmes de planification de manière générale, et non pour se spécialiser dans le calcul des plus courts chemins dans des graphes.

Des méthodes spécialisées comme Dijkstra ou A* sont plus adaptées à ce type de problème, car elles intègrent des heuristiques bien définies et sont optimisées pour explorer efficacement les graphes. Par conséquent, elles sont probablement plus performantes pour traiter de graphes de grande taille.

6. Exercice 6

6.1. Écrire l'état initial et les buts de ce problème de planification «singe-bananes01».

```
(define (problem singe-bananes01)
  (:domain SINGES)
  (:objects A B C)
  (:init
    (singePos A)
    (caissePos B)
    (bananaPos C)
    (bas)
    (notHolding)
  )
  (:goal (holding))
)
```

6.2. Écrire les six opérateurs, formant le domaine de planification « singe » :

```
(define (domain SINGES)
  (:requirements :strips)
  (:predicates (caissePos ?cpos)
               (singePos ?spos)
               (bananaPos ?bpos)
               (haut)
               (bas)
               (holding)
               (notHolding))

  (:action aller
           :parameters (?from ?to)
           :precondition (and (singePos ?from) (bas))
           :effect (and (not (singePos ?from)) (singePos ?to)))

  (:action pousser
           :parameters (?from ?to)
           :precondition (and (caissePos ?from)(singePos ?from)(bas))
           :effect (and (not (caissePos ?from)) (caissePos ?to)
                        (not (singePos ?from)) (singePos ?to)))

  (:action monter
           :parameters (?pos)
           :precondition (and (caissePos ?pos)(singePos ?pos) (bas))
           :effect (and (haut) (not(bas))))

  (:action descendre
           :precondition (haut)
           :effect (and (not(haut)) (bas)))

  (:action attraper
           :parameters (?pos)
           :precondition (and (singePos ?pos) (bananaPos ?pos) (haut) (notHolding))
           :effect (and (not(notHolding)) (holding)))
```

```
35
36   (:action lacher
37            :precondition (holding)
38            :effect (and(not (holding)) (notHolding))))
```

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

Le plan-solution, correctement trouvé par le planificateur, est :

```
1 0: (aller a b) [1]
2 1: (pousser b c) [1]
3 2: (monter c) [1]
4 3: (attraper c) [1]
```

Il est trouvé en une seule itération en 0.02 seconde.

7. Exercice 7

7.1. Écrire en PDDL l'état initial pour une tour à 1, 2, 3, 4 et 5 disques, initialement sur le pic1 à déplacer sur le pic3 avec le pic2 comme intermédiaire (les pics sont numérotés de gauche à droite).

```
1  (define (problem hanoi-problem)
2    (:domain HANOI)
3    (:objects A PIC1 PIC2 PIC3)
4    (:init
5      (handEmpty)
6      (isPic PIC1) (isPic PIC2) (isPic PIC3)
7      (isEmpty PIC2) (isEmpty PIC3)
8      (isDisque A)
9      (on A PIC1)
10     (clear A)
11   )
12   (:goal (on A PIC3))
13 )
```

Listing 1: État initial pour une tour à 1 disque

```
1  (define (problem hanoi-problem)
2    (:domain HANOI)
3    (:objects A B PIC1 PIC2 PIC3)
4    (:init
5      (handEmpty)
6      (isPic PIC1) (isPic PIC2) (isPic PIC3)
7      (isEmpty PIC2) (isEmpty PIC3)
8      (isDisque A) (isDisque B)
9      (largerThan B A)
10     (on A B)(on B PIC1)
11     (clear A)
12   )
13   (:goal (and(on A B)(on B PIC3)) )
14 )
```

Listing 2: État initial pour une tour à 2 disques

```
(define (problem hanoi-problem)
  (:domain HANOI)
  (:objects A B C PIC1 PIC2 PIC3)
  (:init
    (handEmpty)
    (isPic PIC1) (isPic PIC2) (isPic PIC3)
    (isEmpty PIC2) (isEmpty PIC3)
    (isDisque A) (isDisque B) (isDisque C)
    (largerThan B A) (largerThan C B) (largerThan C A)
    (on A B)(on B C)(on C PIC1)
    (clear A)
  )
  (:goal (and(on A B)(on B C)(on C PIC3)) )
)
```

Listing 3: État initial pour une tour à 3 disques

```
(define (problem hanoi-problem)
  (:domain HANOI)
  (:objects A B C D PIC1 PIC2 PIC3)
  (:init
    (handEmpty)
    (isPic PIC1) (isPic PIC2) (isPic PIC3)
    (isEmpty PIC2) (isEmpty PIC3)
    (isDisque A) (isDisque B) (isDisque C) (isDisque D)
    (largerThan B A) (largerThan C B) (largerThan C A) (largerThan D C) (largerThan
    D B) (largerThan D A)
    (on A B)(on B C)(on C D)(on D PIC1)
    (clear A)
  )
  (:goal (and(on A B)(on B C)(on C D)(on D PIC3)) )
)
```

Listing 4: État initial pour une tour à 4 disques

```
(define (problem hanoi-problem)
  (:domain HANOI)
  (:objects A B C D E PIC1 PIC2 PIC3)
  (:init
    (handEmpty)
    (isPic PIC1) (isPic PIC2) (isPic PIC3)
    (isEmpty PIC2) (isEmpty PIC3)
    (isDisque A) (isDisque B) (isDisque C) (isDisque D) (isDisque E)
    (largerThan B A) (largerThan C B) (largerThan C A) (largerThan D C) (largerThan
    D B) (largerThan D A) (largerThan E D) (largerThan E C) (largerThan E B) (
   largerThan E A)
    (on A B)(on B C)(on C D)(on D E)(on E PIC1)
    (clear A)
  )
  (:goal (and(on A B)(on B C)(on C D)(on D E)(on E PIC3)) )
)
```

Listing 5: État initial pour une tour à 5 disques

7.2. Écrire en PDDL les 4 opérateurs pour le domaine des tours de Hanoi :

```
(define (domain HANOI)
  (:requirements :strips)
  (:predicates (largerThan ?x ?y)
               (handEmpty)
               (isPic ?x)
               (isDisque ?x)
               (holding ?x)
               (on ?x ?y)
               (isEmpty ?pic)
               (clear ?x)
)

  (:action enleverDuPic
           :parameters (?disque ?pic)
           :precondition (and (isPic ?pic) (on ?disque ?pic) (clear ?disque)
           (handEmpty))
           :effect (and (not (handEmpty)) (holding ?disque) (not(on ?disque ?pic))
           (isEmpty ?pic)))

  (:action mettreSurPicVide
           :parameters (?disque ?pic)
           :precondition (and (isEmpty ?pic)(isPic ?pic)(holding ?disque))
           :effect (and (not (isEmpty ?pic)) (on ?disque ?pic)
           (not (holding ?disque)) (handEmpty)))

  (:action mettreSurDisque
           :parameters (?disque1 ?disque2)
           :precondition (and (holding ?disque1)(largerThan ?disque2 ?disque1
           (clear ?disque2))
           :effect (and (on ?disque1 ?disque2) (not(holding ?disque1)) (handEmpty)
           (not(clear ?disque2)) ))

  (:action enleverDuDisque
     :parameters (?disque1 ?disque2)
           :precondition (and (isDisque ?disque2) (on ?disque1 ?disque2)
           (handEmpty) (clear ?disque1))
           :effect (and (not (handEmpty)) (holding ?disque1)
           (not(on ?disque1 ?disque2)) (clear ?disque2)  ))

)
```

Listing 6: État initial pour une tour à 5 disques

7.3. Générer un plan d'actions avec CPT pour résoudre les tours de Hanoi pour 1 à 4 disques (au besoin, ajuster le timer de CPT avec l'option « -t »). Quelle est la longueur de chaque plan-solution ? En combien de temps est-il trouvé ? Combien y a-t-il eu d'itérations ? Que se passe-t-il pour résoudre les tours de Hanoi pour 5 disques ?

N° disques	Longueur plan-solution	Temps	Nombre d'itérations
1	2	0.02	1
2	6	0.02	1
3	14	0.03	5
4	30	0.66	17

Quand on exécute la planification pour le cas avec 5 disques, cela prend énormément de temps à s'exécuter et il ne trouve pas la solution dans un temps raisonnable. Cela est dû au fait qu'il y a une explosion combinatoire du nombre d'actions possibles à prendre, et la méthode n'arrive pas à parcourir l'ensemble des états nécessaires pour trouver la solution optimale.

7.4. Que fait la fonction suivante, écrite en pseudo code ?

La fonction présentée réalise la description des étapes pour résoudre le problème de la Tour de Hanoï de façon récursive, en divisant la résolution du problème de taille n en deux sous-problèmes de taille $n-1$.

7.5. Quelle est la complexité algorithmique de cette fonction en nombre de mouvements ?

La récurrence pour le nombre de mouvements $T(n)$ est donnée par :

$$T(n) = 2T(n-1) + 1$$

Développons l'équation récursive :

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ T(n-1) &= 2T(n-2) + 1 \\ T(n-2) &= 2T(n-3) + 1 \\ &\vdots \\ T(1) &= 1 \end{aligned}$$

À partir de cette expansion, nous voyons que le nombre total de mouvements est une somme géométrique :

$$T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1}$$

La somme d'une série géométrique est donnée par :

$$T(n) = 2^n - 1$$

Cela implique que la complexité de l'algorithme récursif est :

$$O(2^n)$$

7.6. Cela correspond-il à ce qui est trouvé dans la question 3 ? Pourquoi ? Quelle est la différence entre la résolution du problème des tours de Hanoi avec cette fonction et avec un planificateur d'actions comme CPT ?

Ce résultat correspond à ce qui est trouvé dans la question 3 pour la longueur du plan-solution, mais pas pour le temps d'exécution.

Dans la fonction présentée en pseudo-code, le temps d'exécution pour trouver la solution est proportionnel à la quantité de mouvements nécessaires, en suivant une stratégie bien définie et optimisée. Cependant, avec le planificateur utilisé (CPT), il effectue une recherche non optimisée et beaucoup plus exhaustive, ce qui entraîne une explosion combinatoire. Cela signifie que la croissance du temps d'exécution dépasse largement une simple multiplication par deux, comme on peut l'observer lors des exécutions pour 3, 4 et 5 disques.

Ainsi, bien que les deux méthodes aboutissent à des solutions valides, le planificateur CPT est beaucoup moins efficace pour ce type de problème spécifique, parce qu'il s'appuie sur une recherche dans tous les états possibles plutôt que sur la recherche dirigée réalisée par l'algorithme spécialisé.