Université de Monastir

Cours: Programmation déclarative

Chapitre 5: Introduction au langage PDDL

Réalisé par:
Dr. Taoufik Sakka Rouis
https://github.com/srtaoufik/Cours-Prog-Declarative/

Introduction

Le standard PDDL (Planning Domain Definition Language) est le langage **de facto** utilisé pour la modélisation formelle des problèmes de planification.

Étant donné qu'un problème de planification est défini par des états et des opérateurs de changement d'état. PDDL est basé sur le langage des prédicats du premier ordre pour modéliser les états et sur une spécification de pré/post condition pour modéliser les opérateurs de changement d'état.

Introduction

En IA, la planification désigne un domaine de recherche visant à générer automatiquement, via une procédure formalisée, un résultat articulé appelé plan solution. Celui-ci est destiné à orienter l'action d'un ou plusieurs exécutants (robots ou humains).

Introduction

La planification automatique fait une distinction claire entre deux activités: **la modélisation et la résolution**.

- L'activité de modélisation vise à décrire un problème de planification à l'aide d'un formalisme approprié tel que PDDL.
- L'activité de résolution, elle concerne la génération automatique des plans-solution à l'aide des algorithmes de planification.

➔ In plan-solution est une séquence d'actions élémentaires menant d'un état initial à un état but. Les outils logiciels utilisés pour générer automatiquement des plans-solution sont appelés **planificateurs.**

Introduction

Un problème de planification comprend :

- Un état initial,
- Des opérateurs de changement d'état,
- Une condition logique permettant de spécifier les états buts (ou cibles).

Exemple: Problème de deux jarres

Données :

Un récipient non gradué contenant du liquide.
Une jarre J3 non graduée pouvant contenir au maximum 3 litres.
Une jarre J4 non graduée pouvant contenir au maximum 4 litres.
Initialement, les deux jarres sont vides.

Résultats souhaités :

On souhaite obtenir 2 litres soit dans J3, soit dans J4.

➔Pour ce problème, on peut définir un état par deux composantes x et y : (x, y), où x représente la quantité dans J3, et y représente la quantité dans J4.

5

Introduction

- **état initial:** (0,0)
- **états buts:** (2, q) ou (q, 2) avec q dénote une quantité quelconque respectivement dans J4 u dans J3.

Quel est l'ensemble des états possibles ou potentiels (cardinalité)?

Card(0..3) * Card(0..4)= 4*5= 20 états différents
➔ C'est le produit cartésien : (0,0) (0,1) (0,2), (0,3), …

- **Un opérateur de changement d'état** est définit comme suit:

- État origine
- État destination
- Une étiquette comportant le nom de l'action l'égale a exécuter afin de passer de l'état origine vers l'état destination

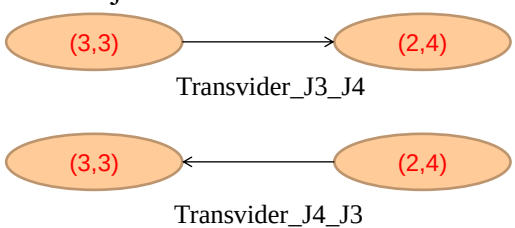
6

Introduction

Pour ce problème

- Remplir_J3: permet de remplir J3 a partir du réception
- Vider_J3 : permet de vider J3 dans le réception
- Remplir_J4: permet de remplir J4 a partir du réception
- Vider_J4: permet de vider J4 dans le réception
- Transvider_J3_J4: prendre le maximum possible venant de la jarre J3 et l'ajouter à la jarre J4
- Transvider_J4_J3: prendre le maximum possible venant de la jarre J4 et l'ajouter à la jarre J3

Illustration



Introduction

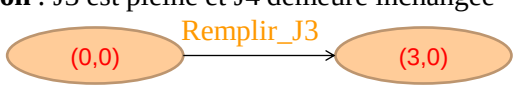
Les actions autorisées liées au problème de planification traité (ici le problème des deux jarres) peuvent être spécifiées (ou décrites) selon la sémantique des pré-conditions et des post-conditions.

- Pré-condition: elle décrit la condition d'applicabilité de l'action concernée
- Post-condition: elle décrit l'effet de l'exécution de la dite action

Exemples:

- Remplir_J3 a la spécification informelle suivante :

Pré-condition : J3 n'est pas pleine et J4 est quelconque
Post-condition : J3 est pleine et J4 demeure inchangée



- Transvider_J3_J4 a la spécification informelle suivante :

Pré-condition : J3 n'est pas vide et J4 n'est pas pleine
Post-condition : retirer le maximum possible de J3 et l'ajouter à J4

Introduction

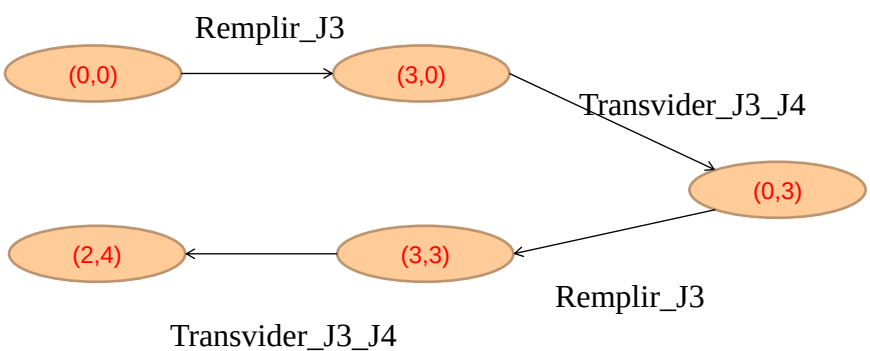
Les états et les opérateurs de changement d'états forment un espace dit **espace d'états**.

Un problème modélisé par le concept espace d'état possède 0 ou plusieurs solutions dites: **plan-solutions.**

➔ **Un plan-solution** est une séquence d'actions autorisées qui mène de l'état initial vers l'état but.

Introduction

Un second plan-solution associé au problème de deux jarres est :

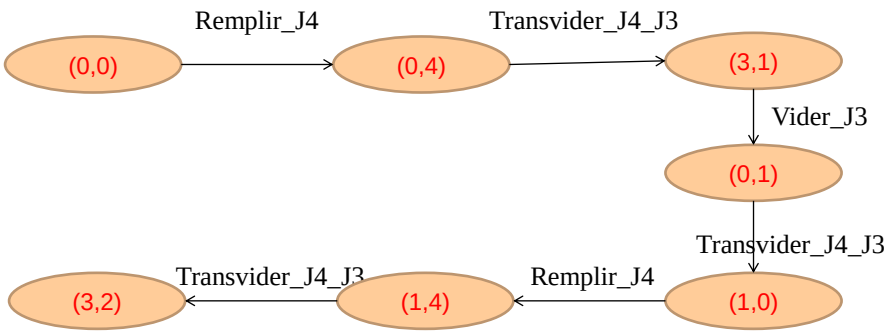


➔Ce plan-solution est noté:
<remplir_j3, transvider_j3_j4, remplir_j3, transvider_j3_j4>.

Chapitre 5: Introduction au langage PDDL

Introduction

➔ Un plan-solution associé au problème de deux jarres est :



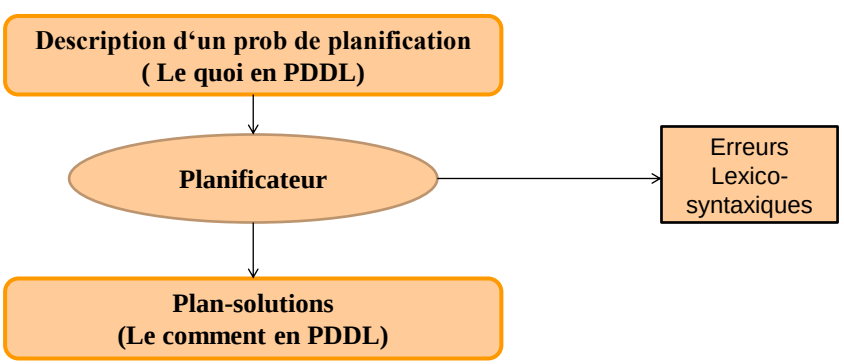
➔La longueur de la solution 2 est égale à 6.
➔La solution 1 est efficace que la solution 1.

11

Chapitre 5: Introduction au langage PDDL

Introduction

Les plan-solutions sont générés automatiquement grâce à un outil logiciel appelé **planificateur**.



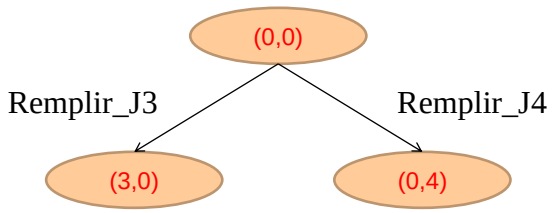
12

Introduction

Est-ce que le planificateur (planner cas du PDDL) est doté d'une certaine intelligence?

➔ La réponse est oui.

Illustration:



Choix a faire?

➔ Planificateur fait appel a des heuristiques afin de faire le « bon » choix.

13

Les constructions de PDDL

Le langage PDDL offre deux constructions obligatoires **domain** et **problem** permettant de modéliser un problème de planification.

- La construction **domain** offerte par PDDL permet de décrire tous les aspects communs à une classe de problèmes dite domaine générique.
- La construction **problem** fournie par PDDL permet de formaliser un problème appartenant au domaine décrit par la construction **domain**. Elle englobe la définition d'un état initial et la condition logique des états buts.

14

Chapitre 5: Introduction au langage PDDL

Les constructions de PDDL

➔ Syntaxe générale de la construction domaine en PDDL:

```
(define   (domain   DOMAIN_NAME)
    (:requirements :strips : equality :typin g :adl )
    ( :types type1  type2  … _typeN)
    ( :constants const1 const2 … constN)
    (:predicates (PREDICATE_1_NAME  ?A1 ?A2  …  ?AN)(PREDICATE_2_NAME ?A1  ?A2  … ?AN)
    )
    (:functions (name_function1 [?A1 ?A2 … ?AN](name_function2 (?A1 ?A2 … ?AN)
    )
    (:action ACTION_1_NAME
    :parameters (?P1 ?P2 … ?PN)
    :precondition PRECOND_FORMULA
    :effect EFFECT_FORMULA)
    (:action ACTION_2_NAME
    )
) ;  fin de la construction du domaine
```

Chapitre 5: Introduction au langage PDDL

Les constructions de PDDL

➔ Syntaxe générale de la construction domaine en PDDL

Explications:

Une construction du domaine comporte plusieurs sections:

DOMAIN_NAME: est le nom du domaine (un identificateur valide en PDDL).

:requirements: ce sont des exigences qui sont optionnelles pour certains planificateurs.

:strips: est un langage de planification, un sous-ensemble de base du langage PDDL. Sa déclaration signifie que PDDL va utiliser le langage STRIPS.

:equality: signifie que le domaine utilise le prédicat = interprété comme « égalité».

:typing: spécifie le type des objets utilisés dans le domaine.

:adl: signifie que le domaine utilise tout ou une partie de l'ADL qui est aussi un langage de planification (c'est à dire disjonctions et quantificateurs dans les conditions préalables et les buts (états finaux) et les effets conditionnels).

16

Les constructions de PDDL

- **Les types en PDDL**

Un type PDDL (déclaré au sein de la section:**types**) est désigné par un identificateur PDDL. Il introduit un ensemble potentiel contenants des objets à fixer dans la construction problème. PDDL fournit un fondateur de type appelé « **objet** ». Tout type introduit dérive directement ou indirectement de type objet, ceci permet de créer une hiérarchie de type.

Illustrations:

(:types type_name1 -objet)
(:types type_name2 -type_nameN)

Exemples:

(:types ELEM -objet))
(:types VILLE -objet)

17

Les constructions de PDDL

- **Les prédicats**

La section prédicates sert à décrire en PDDL **l'état** d'un problème de planification.

Un prédicat possède un nom (identificateur) zéro ou plusieurs paramètres. **Un paramètre** PDDL commence par « **?** » suivi d'un identificateur. Un tel paramètre est de typé par défaut, il est de type objet.

Syntaxe

```
(:predicates
        (name_1 ?A1 ?A2 … ?An)
        (name_2 ?A1 ?A2 … ?An)
        ...
        (name_n ?A1 ?A2 … ?An)
)
```

Exemple 1

```
(:predicates
        (x)
        (y)
)
; Prédicates non parametrés
```

Exemple 2

```
(:predicates
        (Tab ?e -ELEM)
        ( connexion ?V1 -VILLE
                    ?V2 -VILLE )
)
; Prédicates parametrés
```

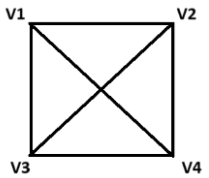
18

Les constructions de PDDL

- **Les prédicats**

Exemple 2

```
(:predicates
        (Tab ?e -ELEM)
        ( Connexion ?V1 -VILLE
                    ?V2 -VILLE )
)
; Prédicates parametrés
```



ELEM={a, b, c, d }

(Tab a) vrai?
(Tab b)
(Tab c) faux?
(Tab d)

Question: Combien de propositions issues du prédicat Tab?

➔ Card(ELEM)

VILLE={Ville1, Ville2, Ville3, Ville4 }

Question: Combien de propositions P issues du prédicat Connexion?

➔Card(VILLE)* Card(VILLE)=4*4=16

(connexion Ville1 Ville1) → faux
(connexion Ville1 Ville2) → vrai
…

Les constructions de PDDL

- **Les fonctions**

En PDDL les fonctions peuvent être assimilées à des fonctions mathématiques, ayant un ensemble de départ composé d'objets et un ensemble d'arrivée composé de nombres réels. Une fonction PDDL peut avoir zéro ou plusieurs arguments. La syntaxe générale d'une fonction en PDDL est la suivante:

```
(:functions
        (name_1 ?A1 ?A2 … ?An)
        (name_2 ?A1 ?A2 … ?An)
        ...
        (name_n ?A1 ?A2 … ?An)
)
```

Les constructions de PDDL

- **Les actions**

Une action PDDL permet de passer d'un état à un autre sous certaines conditions de l'état sur lequel on va appliquer l'action (l'état courant).

➔ **Elle se compose de 3 parties:**

Paramètres (:parameters) : Il s'agit d'une liste des paramètres typés. Cette partie peut être vide. Aussi, l'action n'est pas paramétrée.

Pre-condition (:precondition): Il s'agit d'un prédicat basé sur les atomes introduits dans la clause (:predicates) et comportant des connecteurs logiques tels que:

not, and, or, forall, exists, when, imply, ...

Post-condition (:effect) : Il s'agit d'un prédicat exprimant l'effet de l'action sur l'état du domaine. Un tel état est décrits par les prédicats atomiques introduits dans la clause (:predicates)

Les constructions de PDDL

- **Les actions**

Exemple Illustratif sur l'action d'ajout d'un élément dans un table

```
(: action ajouter
   :parameters (?e -ELEM)

   :precondition (not (Tab ?e))

   :effect (Tab ?e)
)
```

Exemple Illustratif sur l'action d'ajout d'un route reliant deux villes

```
(: action ajouterRoute
   :parameters (?V1 –VILLE  ?V2 -VILLE)

   :precondition
      and(
         (not (connexion ?V1 ?V2))
         (not (connexion ?V2 ?V1))
         (not (= ?V1 ?V2))

       )

   :effect (
      and(
         (connexion ?V1 ?V2)
         (connexion ?V2 ?V1)
       )
)
```

Les constructions de PDDL

Construction du problème : une description d'un scénario où le problème de construction dépend du domaine de construction. Il réutilise les éléments de modélisation issus du domaine afin de définir ses constituants.

➔ Syntaxe générale de la construction du problème en PDDL:

```
(Define (problem PROBLEM_NAME)
   (:domain DOMAIN_NAME)
   (:objects OBJ1 OBJ2 ... OBJ_N)
   (:init  ATOM1 ATOM2 ... ATOM_N)
   (:goal  CONDITION_FORMULA)
)
```

23

Les constructions de PDDL

- Le typage des objets utilisés par le problème de planification concerné est issu du fichier domaine explicité dans la clause domaine.
- La description de l'état initial (clause :**init**) est simplement une liste de tous les atomes de base qui sont **vrais** dans l'état initial. Tous les autres atomes sont par définition **faux**.
- La clause (**:goal**) décrit l'état but sous forme des littéraux positifs. Tous les prédicats utilisés dans l'état initial et le but devraient naturellement être déclarés dans le domaine correspondant.

24

Les constructions de PDDL Exemple 1

```
(define (problem Deux_jarres_probleme)
        (:domain Deux_jarres_domaine)
        (:init
                (= (j3) 0)
                (= (j4) 0)
        )

        (:goal
            (or
                (= (j3) 2)
                (= (j4) 2)
             )
         )
)
```


Les constructions de PDDL Exemple 2

```
(define (domain ensemble)
 (:types ELEM -object)
 (:predicates
  (Tab ?e - ELEM)
 )
 ; oper de changement d'état
 (:action ajouter
  :parameters (?e - ELEM)
  :precondition (not (Tab ?e))
  :effect (Tab ?e)
 )
 ; oper de changement d'état
 (:action retirer
  :parameters (?e - ELEM)
  :precondition (Tab ?e)
  :effect (not (Tab ?e))
 )
)
```

```
(define (problem plein) ; Version 1
 (:domain ensemble)
  (:objects e1 e2 e3 e4 - ELEM)
 (:init
  (Tab e1)
 (:goal
  (and
   (Tab e1)
   (Tab e2)
   (Tab e3)
   (Tab e4)
  )
 )
)
```

➔ **Plan:**
(ajouter e2)
(ajouter e3)
(ajouter e4)

Les constructions de PDDL Exemple 2

```
(define (domain ensemble)
  (:types ELEM -object)
  (:predicates
    (Tab ?e - ELEM)
  )
   ; oper de changement d'état
  (:action ajouter
    :parameters (?e - ELEM)
    :precondition (not (Tab ?e))
    :effect (Tab ?e)
   )
   ; oper de changement d'état
  (:action retirer
    :parameters (?e - ELEM)
    :precondition (Tab ?e)
    :effect (not (Tab ?e))
  )
)
```

```
(define            ; Version 2
   (:problem plein)
      (:domain ensemble)
      (:objects e1, e2, e3, e4 –ELEM)
           ; dom de variation
       (: init
            (Tab e1)
       )
      (:goal
            (forall (?e –ELEM)
                  (Tab ?e)
            )
       )
)
```

forall existes

➔ **Plan:**
(ajouter e2)
(ajouter e3)
(ajouter e4)

Exercice d'application

Trouver les nombres premiers compris entre 2 et un nombre donnés jouant le rôle d'une borne supérieure.

- Présentation informelle

Etape A: Enumérer tous les nombres concernés
Etape B: Ecarter les multiplies
Etape C: les nombres non écartés sont forcément des nombre premiers, (c-à-d divisible uniquement par 1 et lui-même)

- Animation sur les nombres compris entre 2 et 13 par exemple

Etape A: 2 3 4 5 6 7 8 9 10 11 12 13
Etape B: 2 3 4 5 6 7 8 9 10 11 12 13 (écarter = en rouge)
Etape C: 2 3 5 7 11 13

Exercice d'application

PDDL utilisé dans cet exercice ne supporte pas les nombres (N)

```
(define (domain premiers)
  (:requirements :strips :equality :typing :adl )
  (:types nombre - object)
  (:predicates
    (nb ?n - nombre) (multiple ?n1 ?n2 - nombre)  )
  (:action ecarter
    :parameters (?n1  ?n2 - nombre)
    :precondition (and (not (= ?n1 ?n2))
                       (nb ?n1)
                       (nb ?n2)
                       (multiple ?n1 ?n2))

    :effect (not (nb ?n1 )) ))
```


Exercice d'application

```
(define (problem p1)
(:domain premiers)
(:objects n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 - nombre)
; L'etat initial du probleme ?
; PDDL ne supporte pas les opérateurs arithmitique tq "mod"
(:init (nb n2) (nb n3) (nb n4) (nb n5) (nb n6) (nb n7) (nb n8) (nb n9) (nb n10)
(nb n11) (nb n12) (nb n13)

(multiple n4 n2) (multiple n6 n2) (multiple n6 n3) (multiple n8 n2) (multiple n8
n4) (multiple n9 n3)
(multiple n10 n2) (multiple n10 n5) (multiple n12 n2)  (multiple n12 n3)
(multiple n12 n4) (multiple n12 n6)
)
```

Exercice d'application

```
(:goal
(and
 (nb n2) (nb n3) (nb n5) (nb n7) (nb n11) (nb n13) (not (nb n4)) (not (nb n6))
(not (nb n8)) (not (nb n9)) (not (nb n10)) (not (nb n12))
)
```

Solution 1

```
(:goal
    (not (exists(?n1 ?n2 - nombre )
   (and (nb ?n1) (nb ?n2) (multiple ?n1 ?n2 ))
 )))
```

Solution 2

31