Planification

Ce TME vise à programmer un planificateur simple basé sur un encodage en ASP.

Les problèmes considérés sont représentés dans le langage standard de planification PDDL (Planning Domain Definition Language), dont les éléments principaux sont décrits en annexe.

1 Représentation de problème de planification en PDDL

Pour vérifier et tester des domaines et problèmes en PDDL, nous utilisons ici la page `https://editor.planning.domains/`, qui permet de charger et éditer des fichiers PDDL et contient un certain nombre de solveurs pour générer les plans correspondant.

Exercice 1 Prise en main de PDDL

1. Lire la partie de l'annexe consacrée à la syntaxe de PDDL, c'est-à-dire la section 4.1.
2. Créer un fichier `blockWorld-domain.pddl` complétant l'extrait donné en annexe pour définir toutes les actions définies dans l'exercice 1 de la feuille de TD5.
3. Créer un fichier `blockWorld-problem.pddl` inspiré de l'exemple donné en annexe pour décrire le problème de l'exercice 1 de la feuille de TD5.
4. utiliser l'interface Solveur pour générer un plan.
5. Créer des problèmes plus complexes, avec plus de blocs et plus de déplacements nécessaires pour observer des plans plus longs.

Exercice 2 Variante du monde des blocs

1. Proposer une version simplifiée du monde des blocs qui n'utilise que deux actions : `moveTo(X,Y,Z)` qui prend un bloc `X` qui est sur `Y` et le met sur `Z`. La seconde action est `moveToTable(X,Y)` qui prend un bloc `X` qui est sur `Y` pour le mettre sur la table, `X` et `Y` étant tous deux des blocs.

   On peut alors se passer des prédicats `handempty`, `holding` et `ontable` mais il faut généraliser les prédicats `clear(Y)` et `on(X,Y)` pour qu'ils s'appliquent aussi quand `Y` est la constante `table`.

   De plus, quelle(s) précondition(s) faut-il ajouter aux actions et pourquoi ?
2. A partir de `blocks-domain.pddl` et `blocks-problemTD.pddl`, écrire les fichiers `blocksSimp-domain.pddl` et `blocksSimp-problemTD.pddl` correspondant à cette version simplifiée. Utiliser und es solveurs pour obtenir un plan.

2 Planification par ASP

Exercice 3 Planificateur STRIPS

Le but de cet exercice est d'écrire le moteur d'un planificateur STRIPS en ASP, en s'appuyant sur la représentation ASP d'un problème de planification décrite dans l'annexe 4.2.

1. **Domaine et problème.** Ecrire le programme ASP correspondant au domaine du monde des blocs et au problème utilisé dans l'exercice 1.

On ajoute une horloge discrète avec un prédicat `time(0..n)`. Le temps 0 correspond à la situation initiale et le but doit être atteint à l'*horizon* n : on teste avec des n croissants jusqu'à trouver un plan de taille minimale. Les différents T représentent des étapes, et non forcément un écoulement régulier du temps (on change d'étape chaque fois qu'il se passe quelque chose).

Pour relier les états et actions à cette horloge, on introduit aussi le prédicat `holds(P,T)`, qui signifie que le prédicat `P` est vrai au temps `T`, et le prédicat `perform(A,T)` qui indique que l'on fait l'action `A` au temps `T`. Cela n'est possible que si les préconditions de `A` sont vérifiées au temps `T`, et les effets de l'action détermineront l'état au temps `T+1`.

2. **Etat initial.** Traduire en ASP l'état initial (*si quelque chose est initialement vrai, cela veut dire que c'est vrai au temps 0*).
3. **Préconditions.** Traduire en ASP le fait qu'une action ne peut se produire que si toutes ses préconditions sont vérifiées. On utilisera pour cela une contrainte d'intégrité, en exprimant que *si on fait une action alors qu'une de ses préconditions n'est pas vérifiée, on a une contradiction.*
4. **Effets positifs.** Traduire en ASP les effets positifs d'une action : si une action a lieu au temps T, au temps $T + 1$ tous ses effets positifs sont vrais.
5. **Inertie et effets négatifs.** Traduire en ASP le fait que ce qui est vrai au temps T, reste vrai au temps suivant, à moins qu'une action n'y ait mis fin (c'est à dire, à moins qu'une action réalisée au temps T ne l'ait eu comme effet négatif).
6. **Choix d'action.** Traduire en ASP le choix d'une et une seule action effectuée à chaque pas de temps (sauf le dernier).
7. **Spécification du but.** Traduire par une contrainte d'intégrité l'exigence que le but soit atteint au temps n.
8. **Test.** Tester ce moteur de planification avec les problèmes de la feuille de TDs traduits dans les exercices précédents.

Exercice 4 Convertisseur PDDL vers ASP-STRIPS

On considère la représentation ASP d'un problème de planification décrite dans l'annexe 4.2.

Ecrire un programme qui parse un domaine PDDL et le traduit en un programme ASP définissant les types (déclaration des objets) ainsi que les prédicats `pred(`P`)`, `action(`A`)`, `pre(`A`,`P`)`, `del(`A`,`P`)`, `add(`A`,`P`)`, `init(`P`)` et `but(`P`)` (générés par des règles selon le typage).

On peut séparer cette traduction en deux modules (domaine et problème), comme pour les fichiers PDDL.

Il est recommandé d'utiliser des bibliothèques pour gérer le parsing des fichiers PDDL :

— En Python, on pourra par exemple installer et utiliser `https://pypi.org/project/pddl/`

— En Java : `http://pddl4j.imag.fr/`

Exercice 5 Combinaison

1. Ecrire un programme `ASPPLAN` qui, à partir de deux fichiers PDDL définissant le domaine et le problème, génère un plan minimal. Il faut en particulier tester des valeurs de n croissantes jusqu'à trouver un plan, et mettre en forme l'answer set pour afficher lisiblement le plan obtenu.
2. Comparer la vitesse de votre programme avec les solveurs en ligne.

Annexes

2.1 Formalisme PDDL

Le langage PDDL (Planning Domain Definition Language) est un standard de la planification, qui est accepté comme entrée de la plupart des planificateurs et est utilisé dans les compétitions de planification. Ses versions les plus récentes sont très expressives, ce TME se restreint à la partie du langage permettant de traduire le formalisme STRIPS, avec un léger enrichissement pour gérer un typage des objets.

Il décompose la représentation du problème de planification en deux fichiers texte : d'une part le domaine, qui indique principalement les prédicats considérés et les actions autorisées, d'autre part le problème à résoudre, essentiellement défini par l'état initial et les buts à atteindre.

La syntaxe de PDDL, résumée ici, est inspirée de celle de lisp : elle utilise une notation préfixe parenthésée (voir les exemples ci-dessous). Les commentaires sont des lignes précédées de points-virgules.

Représentation du domaine de planification

Comme illustré à la page suivante, un domaine est défini par la fonction `(define (domain` *nomDuDomaine*`)` *champs*`)`, où les champs sont

- `:requirements` qui indique l'expressivité souhaitée. Nous considérons ici le cas de STRIPS, enrichi par un typage des objets et éventuellement l'utilisation du prédicat d'égalité. Aussi, le champ peut prend la forme `(:requirements :strips :typing :equality)` si l'on considère les trois composants.
- `:types` qui liste les types considérés, par exemple, pour un unique type, `(:types block)`
- `:predicates` qui liste les prédicats considérés. Chacun est défini dans une paire de parenthèses par un nom suivi de ses variables associées à leur type. Une variable commence par un `?`, le type est signalé par un tiret (attention aux espaces).

  On peut avoir par exemple `(:predicates (on ?x - block ?y - block))` pour définir le prédicat *on* qui a 2 arguments de type bloc.
- `:constants` qui définit éventuellement des constantes, avec leurs types
- les actions considérées

Une action est définie, dans une paire de parenthèses, par le mot-clé `:action`, suivi de son nom puis des champs (sans parenthèses cette fois) :

- `:parameters` qui liste les paramètres, chacun étant défini comme un nom de variable et un type
- `:precondition` suivi d'une formule logique, en notation préfixe parenthésée, utilisant les prédicats définis auparavant dans le champ `:predicates` (éventuellement `=`) appliqués aux variables définies dans le champ `:parameters`, et combinés par les mots clés `and` et `not`
- `:effect` qui définit les effets à la fois *add* et *del*, par des formules logiques, précédées par `not` pour *del*.

Représentation du problème

Un problème de planification particulier est ensuite défini, dans un autre fichier, par les champs

- `:domain` qui indique le domaine auquel il se réfère
- `:objects` qui indique les objets particuliers considérés, avec leurs types (par exemple quels blocs)
- `:init` qui décrit les prédicats vrais initialement
- `:goal` qui décrit le but, sous la forme d'une formule logique

Documentation détaillée

Une documentation complète de PDDL est téléchargeable à l'adresse suivante : `http://homepages.inf.ed.ac.uk/mfourman/tools/propplan/pddl.pdf`. Les sections 4 (Domaine), 5 (Actions), 6 (Goal Description pour but et precondition), 7 (Effets) et 13 (Problème) détaillent les concepts abordés dans ce TME. La section 2 qui donne un exemple introductif et la section 3 qui expose les notations peuvent aussi être utiles.

Exemple : cas du monde des blocs

Définition partielle du domaine

```
(define (domain blockWorld)
   (:requirements :strips :typing)
   (:types block)
   (:predicates
      (on ?x - block ?y - block)
      (ontable ?x - block)
      (clear ?x - block)
      (handempty)
      (holding ?x - block))
   (:action pick-up
      ;;; action qui ramasse un bloc pose sur la table
      :parameters (?x - block)
      :precondition (and (clear ?x) (ontable ?x) (handempty))
      :effect (and (not (ontable ?x))
               (not (clear ?x))
               (not (handempty))
               (holding ?x))))
```

Représentation d'un problème

```
(define (problem blockProblem)
   (:domain blockWorld)
   (:objects A B - block)
   (:init (clear B)
          (ontable A)
          (on B A)
          (handempty))
   (:goal (and (on A B) (handempty))))
```

2.2 Représentation ASP d'un problème STRIPS

Un problème exprimé en STRIPS contient un ensemble de *prédicats* (qui permettent de décrire des états), un ensemble de *constantes* ou d'*objets* sur lesquels portent les prédicats, un *état initial* et un *but*, tous deux exprimés par un ensemble de prédicats instanciés, et enfin, un ensemble d'*actions* définies sur ces prédicats. Les actions (éventuellement précisées par un ensemble de variables appelées *paramètres*) sont spécifiées par leur *préconditions* (PRE), et leurs *effets*, qui sont *positifs* (ADD) ou *négatifs* (DEL). PRE, ADD et DEL sont tous trois des ensembles de prédicats, dans lequels ne peuvent apparaître que des variables présentes dans les paramètres de l'action.

Cette absence de variables libres dans PRE, ADD et DEL, fait que l'on peut toujours examiner les élements des ces ensembles individuellement par rapport à l'action (ils ne sont pas liés entre eux autrement que par l'action elle-même). On peut donc traduire par des prédicats `pred(P)`, `action(A)`, `pre(A,P)`, `del(A,P)`, `add(A,P)`, `init(P)` et `but(P)` tel qu'illustré ici sur l'exemple du monde des blocs. On utilise aussi des faits de typage pour identifier les objets et constantes ainsi que les instanciations possibles des prédicats ci-dessus.

```
%%% MONDE DES BLOCS %%%
%Déclaration des prédicats (domain)
pred(on(X,Y)):-block(X;Y).
pred(ontable(X)):-block(X).
pred(clear(X)):-block(X).
pred(handempty).
pred(holding(X)):-block(X).
```

```
%Déclaration des actions (domain)
action(pickup(X)):-block(X).
%preconditions
pre(pickup(X),clear(X)):-action(pickup(X)).
pre(pickup(X),ontable(X)):-action(pickup(X)).
pre(pickup(X),handempty):-action(pickup(X)).
%effects
del(pickup(X),clear(X)):-action(pickup(X)).
del(pickup(X),ontable(X)):-action(pickup(X)).
del(pickup(X),handempty):-action(pickup(X)).
add(pickup(X),holding(X)):-action(pickup(X)).
action(putdown(X)):-block(X).
...
action(stack(X,Y)):-block(X;Y).
...
action(unstack(X,Y)):-block(X;Y).
% déclaration des objets (problem) ou constantes (domain)
block(a;b;c;d).
% état initial
init(clear(a)).
init(clear(b)).
...
init(ontable(a)).
...
% but
but(on(d,c)).
but(on(c,b)).
but(on(b,a)).
```