Planification

Préliminaires

le langage PDDL. Le langage PDDL (Planning Domain Definition Language) est un standard de la planification, qui a été utilisé dans des compétitions et est accepté comme entrée de la plupart des planificateurs. Il est assez expressif, mais nous nous restreignons dans ce TME à la partie du langage permettant de traduire le formalisme STRIPS (avec un léger enrichissement pour gérer un typage des objets).

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) exposent tous 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.

En PDDL, on définit d'abord un *domaine de planification*, qui contient la définition du langage utilisé (prédicats via `:predicates` et constantes via `:constants`) ainsi que la description des actions existantes via `:action`, qui contient leurs arguments `:parameters`, leurs préconditions `:precondition` et leurs effets `:effect` (positifs et négatifs rassemblés par un `and` et distingués par la présence ou not d'un `not`). Un domaine s'écrit de la façon suivante (on met entre $ les éléments à compléter et entre < et > les expressions particulières)) :

```
(define (domain $nomDuDomain$)
    (:requirements :strips :typing)
    (:types $type_1 ... type_t$)
    (:predicates ($Pred_1 <listeTypeeVars>$)
       ($Pred_2 <listeTypeeVars>$)
       ...
       ($Pred_n <listeTypeeVars>$)
    )
    (:constants $<listeTypeeConst>$)
    (:action $action_1$
      :parameters $<listeTypeeVars>$
     :precondition $(and (Ppre_1 <Arg>) ... (Ppre_k (Arg>)$
     :effect $(and
      (not (Pdel_1 <Arg>) ... (not (Pdel_m <Arg>)
            (Padd_1 <Arg>) ... (Padd_p <Arg>)
            )
    )
    ...
    (:action $action_q$ ...)
)
```

Toute variables commence par un `?`, constantes et predicats sont non sensibles à la casse et sont des caines alphanumérique commençant par une lettre. `<listeTypeeVars>` représente des listes typées de variables de la forme $?x_1^1 \dots ?x_i^1$ - $type_1$ $?x_1^2 \dots$ - $type_2 \dots ?x_j^k$ - $type_k$. Les listes typées de constantes ont la même forme (mais sans les `?`). Un prédicat est toujours suivi de ses arguments (notés `<Arg>` ci dessous), qui peuvent être des constantes ou ds variables. Ils sont donnés dans l'ordre et juste séparés par des espaces.

A cela s'ajoute la description d'un *problème*, donnant une instance spécifique de problème de planification dans un domaine. Il s'agit en particulir d'enrichir le langage avec les constantes correspondant

aux objets particuliers (par exemple quels blocs) via `:objects`, et de spécifier la situation initiale avec `:init` et le but avec `:goal`.

```
(define (problem $pb_name$)
(:domain $domainName$)
(:objects $<ListeTypeeConst>$)
(:init
(Pred_1 <Arg>)
...
(Pred_m <Arg>)
)
(:goal (and (but_1 <Arg>) ...(but_k <Arg>) )
)
```

SATPLAN. Dans un premier temps, on utlisera `SATPLAN` un planificateur basé sur un encodage en SAT du problème, qui convertit le problème en des formules booléennes d'horizons croissants au format DIMACS avant de les fournir à un solveur SAT et dès que le problème devient satisfiable, d'extraire du modèle renvoyé le plan minimal correspondant. Il est téléchargeable à l'adresse `http://www.cs.rochester.edu/users/faculty/kautz/satplan/index.htm` (recherche `SATPLAN planner`). Télécharger l'archives des binaires linux (lien Linux Binaries de la section Downloads) et la decompresser. Il n'y alors pas d'autre installation à faire ?

L'exécutable est `satplan`. Un exécution vous indique la syntaxe. En considérant qu'on a deux fichiers `ex-domain.pddl` et `ex-problem.pddl` définissant le problème de planification que l'on veut résoudre, il faut exécuter `./satplan -domain ex-domain.pddl -problem ex-problem.pddl`. On peut rajouter un certain nombre d'options

- `-solution ex-plan.txt` permet de récupérer la solution (le plan) dans un fichier texte dont on fournit le nom (ici `ex-plan.txt`). Ce fichier sera crée s'il n'existe pas.
- `-cnf ex-cnf.cnf` permet de récupérer l'encodage du problème dans un fichier au format DIMACS dont on fournit le nom (ici `ex-cnf.cnf`). Ce fichier sera crée s'il n'existe pas.
- `-cnfonly 1` demande à ce que seul l'encodage soit réalisé. La phase de résolution par solveur SAT et décodage n'est alors pas effectuée.
- `-verbose 0` désactive le mode verbose qui est actif par défaut.

Exercice 1 Prise en main de DPLL et SATPLAN

Télécharger le fichier `PDDL3.1-examples.zip` sur la page `http://ipc.informatik.uni-freiburg.de/PddlExtension` (rechercher `PDDL 3.1 examples`) et décompresser l'archive dans le répertoire où vous avez installe SATPLAN.

1. Dans le répertoire `pddl3.1-examples/blocks/`, ouvrir les fichiers `blocks-domain.pddl` et `blocks-problem.pddl`. Quel est le problème de planification représenté par ces fichiers ? Illustrer la situation intiale et le but.
2. Utiliser SATPLAN pour résoudre ce problème, enregistrer le plan obtenu dans un fichier et vérifier le.
3. Créer un fichier `blocks-problemTD.pddl` qui corresponde à l'exercice 1 de la feuille de TD 7, et obtenir un plan pour ce problème avec SATPLAN. Comparer le à celui obtenu en TD.
4. Proposer une version simplifiée du monde des blocs qui n'utilise que deux actions : moveToBlock(X,Y,Z) qui prend un bloc X qui est sur Y pour le mettre sur Z, X étant un bloc, et Y et Z étant des objets (blocs ou table) et 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 prédicat `handempty`, `holding` et `ontable` mais il faudra généraliser les prédicats `clear(`Y`)` et `on(X,`Y`)` pour qu'il s'applique aussi quand Y est la constante `table`.
5. A partir de `blocks-domain.pddl` et `blocks-problemTD.pddl`, écrire les fichiers `blocksSimp-domain.pddl` et `blocksSimp-problemTD.pddl` correspondant aux versions simplifiées de ces fichiers (question précédente). Utiliser SATPLAN pour obtenir un plan.

Exercice 2 Modélisation d'un problème de planification

Définir le problème du *singe et des bananes* (exercice 2 de la feuille de TD7) en PDDL et le résoudre avec SATPLAN.

Programmation d'un planificateur en ASP

Exercice 3 Parseur DPLL vers ASP-STRIPS

Rappel 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 instantiés, 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 être *positifs* (ADD) ou *négatifs* (DEL). PRE, ADD et DEL sont tous trois des ensembles de prédicats, dans lequelles ne peuvent apparaître que des variables présentes dans les paramètres de l'action.

Cette absence de variables libres dans PRE, ADD et DEL, fait qu'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 utilisera aussi des faits de typage pour identifier les objets et constantes ainsi que les instantiations 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)).
```

Ecrire un programme qui parse un domaine PDDL et le traduise 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 pourra séparer cette traduction en deux modules (domaine et problème), comme pour les fichier PDDL.

Exercice 4 Planificateur STRIPS

Le but de cet exercice est d'écrire le moteur d'un planificateur STRIPS en ASP, en s'appuyant sur les prédicats définis à l'exercice précédent. On rajoute 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 testera 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 introduti 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 $T + 1$.

1. **Etat initial.** Traduire en ASP l'état initial (*si quelque chose est initialement vrai, cela veut dire que c'est vrai au temps 0*).
2. **Préconditions.** Traduire en ASP le fait qu'une action ne peut se produire que si toutes ses préconditions sont vérifiés. 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.*
3. **Effets positifs.** Traduire en ASP les effets positifs d'une action : si une action a lieu au temps T, au temps $T + 1$ tout ses effets positifs seront vrais.
4. **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).
5. **Choix d'action.** Traduire en ASP le choix d'une et une seule action effectuée à chaque pas de temps (sauf le dernier).
6. **Spécification du but.** Traduire par une contrainte d'intégrité l'exigence que le but soit atteint au temps n.
7. **Test.** Tester ce moteur de planification avec les problèmes de la feuille de TDs traduits dans les exercices précédents.
8. Ecrire un programme `ASPPLAN` qui, à partir de deux fichiers PDDL définissant le domaine et le problème, génère un plan minimal. Il faudra en particulier tester des valeurs de n croissante jusqu'à trouver un plan, et mettre en forme l'answer set pour afficher lisiblement le plan obtenu/
9. Comparer la vitesse de votre programme avec `SATPLAN`. Note : `clasp` pouvant aussi être utilisé comme solveur SAT, il est possible d'avoir une comparaison plus juste en séparant l'encodage (`satplan -cnfonly 1` ou votre encodage SAT) et en résovlant avec `clasp` dans les deux cas.