TP5 - Planification d'actions

Auteurs du CR

Zhi Zhou, zhi.zhou@ensta-paris.fr Simon Queyrut, simon.queyrut@ensta-paris.fr

@zroykhi (repo source), @queyrusi

Résumé de cours

Étant donnés des actions génériques possibles, un état initial et des buts, on souhaite trouver un plan (séquence d'actions instanciées) qui amène le système à un état final (contenant les buts). L'explicitation de tous les états d'un vrai système est cependant impossible : on se donne donc des hypothèses simplificatrices. PDDL est une manière de décrire un tel problème dans un domaine (format STRIPS, ADL). Un plannificateur d'actions peut alors le résoudre (il en existe de nombreux types).

Description du TP

Avec le language PDDL nous traitons plusieurs problématiques de plannification simples en définissant des domaines (STRIPS), les problèmes et en les résolvants par un solveur basé SAT `cpt.exe` . Nous critiquons les résultats obtenus.

Question 1

Les opérateurs `pick-up` , `put-down` , `stack` et `unstack` sont les actions génériques qui composent la séquence du plan menant à l'état contenant les buts (boîtes empilées).

`put-down` met un objet sur la table et `stack` empile une boîte sur une autre. La distinction est importante car il faut notifier au solveur qu'une boîte qui "subit" un empilement (reçoit sur elle une boîte) doit être libre (précondition `(clear ?y)`) et qu'elle n'est plus en mesure d'en recevoir une autre après l'action (post condition `(not (clear ?y))`). Similairement, la pince exécute une action spécifique si l'objet est sur la table ou non (`(ontable ?x)`) donc si on pose un objet sur la table, il faut dire au solveur que cet objet peut subir une opération `pick-up` .

Le fluent `(holding ?x)` informe le solveur que la machine pince un objet `?x` en particulier. Si ce fluent n'était pas là, on devrait détailler autant d'actions que de combinaisons objet-table-pince.

Question 2

```
.\cpt.exe -o .\domain-blocksaips.pddl -f .\blocksaips01.pddl

0: (pick-up b) [1]
1: (stack b a) [1]
2: (pick-up c) [1]
3: (stack c b) [1]
4: (pick-up d) [1]
5: (stack d c) [1]
```

La longueur du plan solution (ce qui revient au nombre théorique d'itérations du solveur puisqu'à chaque itération il incrémente la taille maximale n de son domaine de recherche. Il vaut cependant 1 ici) est de 6. Temps de recherche proche de 0.

Ces autres solutions ne sont pas fournies par le solveur car il ne retourne que la solution optimale (plus faible nombre de coups). En principe, n'importe quelle combinaison aboutissant au but aurait pu marcher : insérer une boucle de `(pick-up b), (put-down b), (pick-up b), (put-down b)` indéfiniment avant de passer à la vraie étape significative `(stack b a)` en est un exemple.

Question 3

Commande utilisée pour résoudre le problème:

```
.\cpt.exe -o .\domain-blocksaips.pddl -f .\exe3.pddl
```

donne

```
0: (unstack b c) [1]
1: (put-down b) [1]
2: (unstack c a) [1]
3: (put-down c) [1]
4: (unstack a d) [1]
5: (stack a b) [1]
6: (pick-up c) [1]
7: (stack c a) [1]
8: (pick-up d) [1]
9: (stack d c) [1]
```

Longueur du plan solution 10 (itérations n vaut 1). Temps de recherche proche de 0.

Question 4

Commande utilisée pour résoudre le problème:

```
.\cpt.exe -o .\domain-blocksaips.pddl -f .\exe4.pddl -t 60
```

donne

```
--  Backtracks : 0  ---  Iteration time : 0.00
--  Backtracks : 0  ---  Iteration time : 0.00
---  Backtracks : 29  ---  Iteration time : 0.01
---  Backtracks : 36  ---  Iteration time : 0.01
 ---  Backtracks : 782  ---  Iteration time : 0.45
 ---  Backtracks : 787  ---  Iteration time : 0.46
 ---  Backtracks : 129  ---  Iteration time : 0.10

0: (unstack c g) [1]
1: (put-down c) [1]
2: (unstack g e) [1]
...
29: (stack b d) [1]
30: (pick-up c) [1]
31: (stack c b) [1]
```

Cette fois le solveur a mis 7 itérations grâce à une heuristique interne de recherche. La longueur du plan solution est de 32. Le temps de recherche est la somme de celui de chaque itération : 1,03 s.

Question 5

Commande utilisée pour résoudre le problème:

```
.\cpt.exe -o .\exe5-domain.pddl -f .\exe5.pddl
```

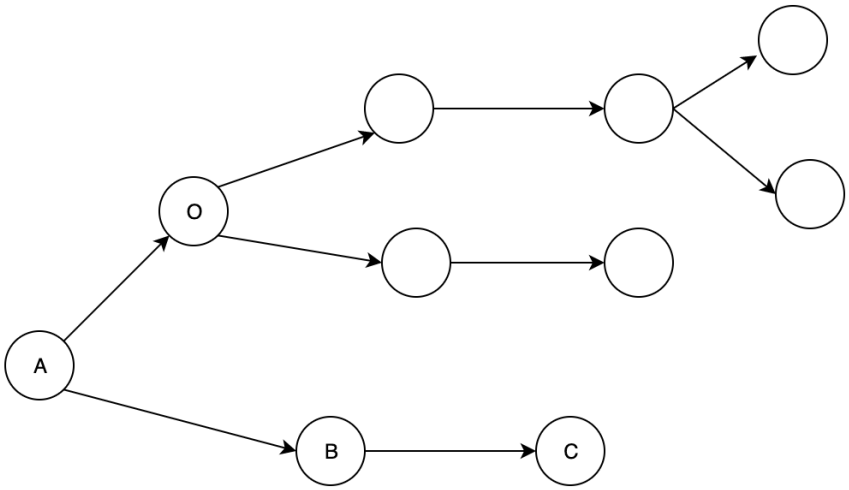
donne

```
0: (aller nodea nodeb agenta) [1]
1: (aller nodeb nodec agenta) [1]
2: (aller nodec noded agenta) [1]
```

Pour cet exemple nous ne rencontrons pas de problème (le voyageur n'a pas beaucoup de chances de se tromper car le nombre de nœuds est faible)

Néanmoins sa recherche n'est pas efficace. Le solveur va tester toutes les combinaisons de sauts possible entre les nœuds et s'attendre à un raccordement entre le départ et l'arrivée : on préfèrerait bénéficier au moins d'une heuristique de recherche (ci-

dessous, aller sur le nœud B plutôt que O), voire commencer du but pour aller vers l'état initial (recherche arrière).



Question 6

Commande utilisée pour résoudre le problème:

```
.\cpt.exe -o .\exe6-domain.pddl -f .\exe6.pddl
```

donne

```
0: (aller positiona positionb monkeym) [1]
1: (pousser positionb positionc boxb monkeym) [1]
2: (monter positionc boxb monkeym) [1]
3: (attraper positionc monkeym bananab) [1]
```

Longueur du plan solution 4. Une itération.

L'écriture de `pousser` est incomplète : il s'agit d'un exemple du problème de qualification. On devrait lister des caractéristiques de l'objet à pousser en tant que préconditions (poids, coefficient de frottement statique caisse/sol, ergonomie de la caisse etc.).

Question 7

Exemple de commande utilisée pour résoudre le problème:

```
.\cpt.exe -o .\exe7-domain.pddl -f .\exe7-3.pddl -t 1000
```

	1 disc	2 disc	3 disc	4 disc	5 disc
temps	0.01	0.14	0.22	10.96	too long
itérations	2	6	14	30	not found

pour 5 disques, le temps d'exécution est trop long et ne trouve pas la solution à cause de cela.

Lorsque le nombre de disques est n, `T(n)` étapes sont nécessaires. Pour déplacer n-1 disques de **PicDepart** à **PicIntermediaire**, `T(n-1)` étapes sont nécessaires. Une étape pour Le dernier disque de **PicDepart** est déplacé vers **PicArrivee**. Plus `T(n-1)` étapes pour déplacer n-1 disques de **PicIntermediaire** vers **PicArrivee**. donc la formule récursive

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

nous pouvons résoudre la récurrence et obtenir un formulaire fermé. La complexité temporelle du problème de la tour de Hanoi est donc **O(2^n)**.

Le nombre de mouvements n'est pas exactement le même dans ces deux questions parce que nous séparons l'action de déplacer le disque de A à B en deux mouvements différents, c'est-à-dire enlever avec la pince le disque de A et mettre le disque avec la pince sur B. Donc, le nombre de mouvements dans la question 3 est **double** que dans cette question. Mais ils ont la même complexité.

CPT utilise la méthode de retour arrière(Backtrack) pour trouver la solution finale tandis que l'algorithme ci-dessus utilise la méthode récursive. Backtrack perd beaucoup de temps à tester des solutions incorrectes, ce qui rend le calcul coûteux. C'est pourquoi CPT ne peut pas résoudre les tours de Hanoi pour 5 disques. En conclusion, la méthode récursive est meilleure que la méthode backtrack.