Le moindre engagement dans une approche de la planification basée sur la procédure de Davis et Putnam

Vincent Vidal
IRIT – Université Paul Sabatier
118 route de Narbonne
31062 Toulouse Cedex 04, France
email : vvidal@irit.fr

Résumé

Nous présentons une amélioration de la procédure d'extraction de plans DPPlan [2] qui exploite la structure de graphe de planification pour effectuer une recherche de type Davis et Putnam. Le graphe n'est pas codé sous forme d'une base de clauses comme dans Satplan, mais utilisé directement par diverses fonctions qui propagent des valeurs attachées aux nœuds d'action et de fluent, conduisant à une procédure de recherche bidirectionnelle. Nous reconstruisons cet algorithme sur la base d'une version « minimale » qui effectue le moins de propagations possibles, basée sur le codage du graphe de planification sous forme d'une base de clauses. Nous montrons alors comment utiliser la relation d'autorisation entre actions, qui est une relation moins contrainte que la relation classique d'indépendance qui garantit la possibilité pour deux actions d'être exécutées en parallèle. L'utilisation de la relation d'autorisation dans DPPlan, comme dans Graphplan, améliore considérablement les performances de cet algorithme.

1 Introduction

La planification (ou génération de plans, synthèse de plans) s'intéresse à la construction de systèmes capables de générer automatiquement des plans comme collections organisées d'actions. Un plan est élaboré à partir de la description de l'état initial de l'univers, de l'ensemble des actions applicables dans le monde et du but à atteindre. Il doit permettre, par son exécution réelle (par exemple par un système robotique) ou simulée, de faire évoluer l'univers modélisé de manière à satisfaire les objectifs préalablement fixés (par la description du but).

Nous nous intéressons dans cet article à la planification classique de type STRIPS, pour laquelle l'état initial est parfaitement connu, les effets des actions sont déterministes et les ensembles de données manipulées (actions, fluents) sont finis. Ce cadre très restrictif permet d'élaborer des méthodes de génération de plans relativement efficaces, comme les planificateurs Satplan [11], Blackbox [12], Graphplan [3], FF [8]... Les trois premières de ces méthodes, qui nous intéressent plus particulièrement, transforment le problème initial en une recherche de plan solution avec horizon borné.

Le planificateur Graphplan [3] développe une structure compacte appelée graphe de planification, qui entrelace des niveaux de nœuds de fluent et d'action construits en chaînage avant, plus des exclusions mutuelles qui représentent l'impossibilité pour deux actions d'être exécutées en parallèle, ou pour deux fluents d'être présents dans un même état après l'application d'un plan parallèle d'une longueur donnée. Cette structure est ensuite explorée par un algorithme de recherche en chaînage arrière qui peut être amélioré par exemple par l'utilisation de techniques de retour arrière intelligent et de mémoization [9], ou encore par l'utilisation de la relation d'autorisation [4, 5, 6, 15] pour le calcul des exclusions mutuelles. L'autorisation est une relation moins contrainte que l'indépendance, permettant de moins s'engager pendant la construction du graphe sur l'ordonnancement des actions, ce qui sera effectué pendant l'extraction de la solution : c'est en cela que la relation d'autorisation peut être considérée comme une stratégie de moindre engagement. Le graphe de planification est ensuite étendu niveau par niveau jusqu'à ce qu'une solution soit trouvée ou qu'une condition d'arrêt soit vérifiée. Le graphe de planification peut aussi être traduit en une formule booléenne dont les modèles correspondent aux plans solutions. Des prouveurs SAT efficaces peuvent alors être utilisés comme dans Satplan [11] et Blackbox [12]. Mais cette traduction est coûteuse en mémoire, et ne permet pas facilement l'utilisation des heuristiques indépendantes des domaines développées pour la planification.

Une autre utilisation du graphe de planification est faîte dans l'algorithme DPPlan [2], qui réduit le fossé entre Graphplan et Satplan. DPPlan exécute une procédure de type Davis et Putnam [7] directement sur le graphe de planification en associant une valeur à chaque nœud d'action et de fluent, et propage ces valeurs à travers le graphe par des fonctions spécifiques qui rappellent la propagation unitaire des prouveurs SAT. La recherche est dirigée par une liste de nœuds buts qui ont une valeur particulière (*requis* et *requis_faux*), ce qui permet l'utilisation d'heuristiques adaptées.

Après quelques définitions et un exemple en Section 2, nous montrons que la valeur *requis_faux* n'est pas nécessaire en exhibant une version « minimale » de DPPlan en Section 3 basée sur le codage SAT du graphe de planification. Nous reconstruisons DPPlan sur la base de cette version en Section 4, montrons comment utiliser l'autorisation en Section 5, et fournissons des résultats expérimentaux en Section 6 avant de conclure en Section 7.

2 Le graphe de planification

2.1 Définitions

Un fluent est une formule atomique de base qui représente un fait de l'univers. Une action de type STRIPS classique sans négation dans les préconditions, dénotée par a, est un triplet $\langle prec, add, del \rangle$, avec $prec$, add et del trois ensembles de fluents représentant respectivement les préconditions, ajouts et retraits de a. Un problème de planification est un triplet $\langle O, I, B \rangle$ où O est un ensemble d'actions, I est un ensemble de fluents représentant l'état initial et B est un ensemble de fluents représentant le but.

Soit PG un graphe de planification. Un nœud de fluent (resp. d'action) est dénoté par $n_f=f(i)$ (resp. $n_a=a(i)$) où f (resp. a) est un fluent (resp. une action) et i un entier dénotant le niveau du graphe de planification dans lequel se trouve le nœud. $N\_f(i)$ dénote le no-op[1] du fluent f au niveau i. $Valeur(n)$ dénote la valeur attachée au nœud n. $Nœuds(PG)$ dénote tous les nœuds du graphe de planification. $Fluents(PG)$ dénote tous les nœuds de fluent. $Init(PG)$ dénote les nœuds de fluent de l'état initial au premier niveau du graphe. $But(PG)$ dénote les nœuds de fluent du but au dernier niveau du graphe. $Precs(PG)$ dénote les arcs entre nœuds d'action et nœuds de fluent, dénotés par (n_f, n_a), où n_f est un nœud de fluent, n_a est un nœud d'action, et f est une précondition de a. $Adds(PG)$ (resp. $Dels(PG)$) dénote les arcs entre nœuds d'action et nœuds de fluent, dénotés par (n_a, n_f), où n_a est un nœud d'action, n_f est un nœud de fluent, et a produit (resp. retire) f. Deux actions a et b sont indépendantes ssi (1) $a \neq b$, (2) a ne retire ni une précondition ni un ajout de b et (3) b ne retire ni une précondition ni un ajout de a. Deux nœuds d'action n_a et n_b sont mutuellement exclusifs (mutex) ssi (1) $n_a \neq n_b$ et (2) a et b ne sont pas indépendantes ou une préconditions de n_a (ou n_b) est mutex avec une précondition de n_b (ou n_a). Deux nœuds de fluent n_f et n_g sont mutex ssi (1) $n_f \neq n_g$ et (2) tous les nœuds d'action qui produisent f sont mutex avec tous les nœuds d'action qui produisent g. $AMutex(PG)$ (resp. $FMutex(PG)$) dénote les paires de nœuds d'action (resp. nœuds de fluent) mutuellement exclusifs.

2.2 Exemple

Soit $\Pi = \langle O, I, B\rangle$ le problème de planification défini par :
- $O = \{A, B, C\}$ avec $A = \langle\{a\}, \{b\}, \{\}\rangle$, $B = \langle\{a\}, \{c\}, \{a\}\rangle$, $C = \langle\{b, c\}, \{d\}, \{\}\rangle$;
- $I = \{a\}$;
- $B = \{d\}$.

La Figure 1 représente le graphe de planification de taille minimale permettant de résoudre ce problème (quatre niveaux, car le fluent d qui représente le but du problème n'apparaît qu'au niveau 3). Pour plus de clarté, les nœuds sont représentés de manière abrégée : par exemple, le noeud A au niveau 2 devrait être noté $A(2)$. Le but n'apparaît qu'au niveau 3 car le mutex entre les fluents b et c au niveau 1 empêche d'utiliser l'action C au niveau 2. Ce mutex disparaît au niveau 2, puisqu'il existe deux actions non mutex qui produisent b et c au niveau 2 : le no-op de b (N_b) et l'action B.

Le seul plan solution que l'on peut extraire de ce graphe est la séquence d'actions $\langle A, B, C\rangle$.

3 Une version « minimale » de DPPlan : DPPMin

3.1 L'algorithme

Un de nos objectifs est de démontrer que la valeur *requis_faux* utilisée dans DPPlan n'est pas utile. Ceci est mieux compris en considérant le codage SAT du graphe de planification [11, 12] :

[1] Le no-op d'un fluent f est une action qui a pour unique précondition f et pour unique ajout f. Son utilisation dans Graphplan et Satplan permet de résoudre le problème du décor en garantissant que si f est présent à un niveau i et n'est retiré par aucune action, alors f reste présent au niveau $i + 1$.

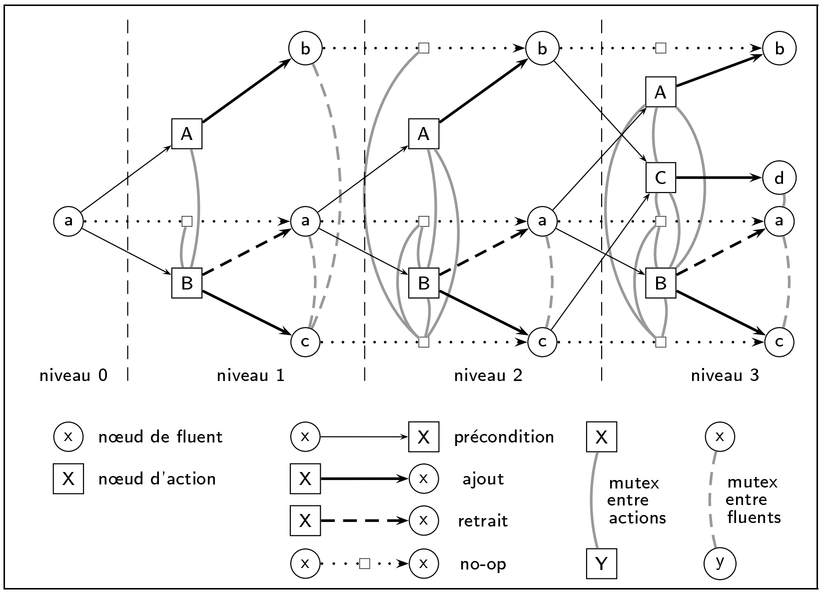

FIG. 1 – Le graphe de planification du problème Π

Règle 1. Les fluents de l'état initial et du but sont vrais :

$\left(\bigwedge_{n_f \in Init(PG)} n_f\right) \wedge \left(\bigwedge_{n_f \in But(PG)} n_f\right)$

Règle 2. Une action implique ses préconditions :

$\bigwedge_{(n_f, n_a) \in Precs(PG)} (n_a \Rightarrow n_f)$

Règle 3. Un fluent implique la disjonction des actions (en incluant son no-op) qui le produisent :

$\bigwedge_{n_f \in (Fluents(PG) \setminus Init(PG))} \left(n_f \Rightarrow \bigvee_{(n_a, n_f) \in Adds(PG)} n_a\right)$

Règle 4. Les actions mutuellement exclusives ne peuvent pas être utilisées simultanément :

$\bigwedge_{\{n_a, n_b\} \in AMutex(PG)} \neg (n_a \wedge n_b)$

Les valeurs attachées aux nœuds du graphe de planification dans DPPlan sont relatives à la valeur de vérité (*vrai*, *faux*) des variables SAT du codage. La valeur *indéfini* signifie que le nœud est dans un état indéterminé.

Les autres valeurs possibles pour un nœud d'action $n_a = a(i)$ sont :

- *utilisé* : a appartient au plan courant au niveau i (valeur *vrai* du codage SAT pour la variable n_a). Les clauses de la règle 2 (cf. [2]) doivent être satisfaites, donc les préconditions de n_a prendront la valeur *requis*. Les clauses de la règle 3 sont satisfaites, donc les fluents produits par n_a (qui sont en partie antécédent des clauses de la règle 3 dans lesquelles n_a est présent) prendront la valeur *asserté* et seront retirés de la liste des buts s'ils s'y trouvent. La seconde action des clauses de la règle 4 ne doit pas être utilisée.

[2]...qui contiennent n_a. Ceci sera implicite dans le reste de la description des valeurs.

- *exclu* : a n'appartient pas au plan courant au niveau i (valeur *faux* du codage SAT pour la variable n_a). Les clauses de la règle 2 sont satisfaites. Si toutes les actions d'une clause de la règle 3 sont exclues et le fluent est nié, cette clause ne peut pas être satisfaite : aucun plan ne peut être trouvé avec l'assignation courante. Les clauses de la règle 4 sont satisfaites.

Les valeur possibles pour un nœud de fluent $n_f = f(i)$ sont :

- *asserté* : f est présent dans un état au niveau i (valeur *vrai* du codage SAT pour la variable n_f) : f appartient soit à l'état initial, soit aux ajouts d'une action utilisée. Les clauses des règles 1 et 2 (cf. [3]) sont satisfaites. Les clauses de la règle 3 sont satisfaites, car une action qui produit n_f est utilisée. n_f ne reste pas requis, s'il l'était.
- *requis* : f doit être présent dans un état au niveau i (valeur *vrai* du codage SAT pour la variable n_f). n_f est soit un but (les clauses de la règle 1 sont satisfaites), soit une précondition d'une action utilisée (les clauses de la règle 2 sont satisfaites). Les clauses de la règle 3 doivent être satisfaites, donc une action qui produit n_f doit être trouvée. n_f est placé dans la liste des buts.
- *nié* : f n'est pas présent dans un état au niveau i (valeur *faux* du codage SAT pour la variable n_f), car toutes les actions qui le produisent sont exclues. Les clauses de la règle 2 sont satisfaites si l'action n'est pas utilisée. Les clauses de la règle 3 sont satisfaites.

Au départ, tous les nœuds du graphe de planification reçoivent la valeur *indéfini*. Les fluents de l'état initial sont assertés, et les fluents du but sont requis (règle 1 du codage SAT). La fonction **rechercher** est alors appelée et retourne le booléen *vrai* ou *faux* selon l'existence d'une solution. Si un plan solution existe, il est extrait du graphe de planification par la fonction **extraire-plan** (nœuds d'action qui ont la valeur *utilisé*).

fonction DPPMin()
  init();
  si rechercher() **alors** retourner(extraire-plan(PG)) **sinon** retourner(*échec*)
fin

fonction init()
  pour tout $n \in Nœuds(PG)$ **faire** $Valeur(n) \leftarrow$ *indéfini*;
  pour tout $n_f \in Init(PG)$ **faire** asserter(n_f);
  pour tout $n_f \in But(PG)$ **faire** requérir(n_f)
fin

La recherche est effectuée par une fonction de type Davis et Putnam [7] guidée par une liste de buts. S'il n'y a plus de buts, un plan est trouvé. S'il reste au moins un but, une action qui en produit un est choisie par une heuristique et est utilisée puis exclue en cas d'échec.

fonction rechercher()
  si $liste\_buts = \emptyset$ **alors** retourner(*vrai*);
  $n_a \leftarrow$ choisir($liste\_buts$);
  sauver-valeurs-et-buts();
  si utiliser(n_a) et rechercher() **alors** retourner(*vrai*);
  restaurer-valeurs-et-buts();
  si exclure(n_a) et rechercher() **alors** retourner(*vrai*);

[3]...qui contiennent n_f. Ceci sera implicite dans le reste de la description des valeurs.

retourner(*faux*)
fin

Un fluent est asserté quand il appartient à l'état initial ou est un ajout d'une action utilisée. Il est retiré de la liste des buts s'il était requis. Il ne peut pas être nié avant d'être asserté, comme dans DPPlan, car un fluent est nié seulement quand toutes les actions qui le produisent sont exclues : il n'y a donc pas lieu de faire la vérification.

fonction asserter(n_f)
cas-de
$\mathit{Valeur}(n_f) = \mathit{requis}$: $\mathit{Valeur}(n_f) \leftarrow \mathit{asserté}$;
$\mathit{liste\_buts} \leftarrow \mathit{liste\_buts} \setminus \{n_f\}$
$\mathit{Valeur}(n_f) = \mathit{indéfini}$: $\mathit{Valeur}(n_f) \leftarrow \mathit{asserté}$
fin

Un fluent est requis quand c'est un but ou une précondition d'une action utilisée. Il est alors ajouté à la liste des buts. S'il était nié avant d'être requis (toutes les actions qui le produisent sont exclues), une contradiction est trouvée : **requérir** retourne *faux*.

fonction requérir(n_f)
cas-de
$\mathit{Valeur}(n_f) = \mathit{nié}$: retourner(*faux*)
$\mathit{Valeur}(n_f) = \mathit{indéfini}$: $\mathit{Valeur}(n_f) \leftarrow \mathit{requis}$;
$\mathit{liste\_buts} \leftarrow \mathit{liste\_buts} \cup \{n_f\}$
fin

Les préconditions d'une action utilisée sont requises (règle 2 du codage SAT) et ses ajouts sont assertés (des clauses de la règle 3 du codage SAT sont satisfaites). On ne se soucie pas des retraits car ils sont pris en compte par la vérification des mutex entre actions (règle 4 du codage SAT) : si un fluent f est retiré par une action A, alors toutes les action qui produisent f sont exclues puisqu'elles sont mutex avec A. Une fois qu'elles sont toutes exclues, le fluent f est nié (cf. **exclure**).

fonction utiliser(n_a)
$\mathit{Valeur}(n_a) \leftarrow \mathit{utilisé}$;
pour tout $(n_f, n_a) \in \mathit{Precs}(PG)$ **faire**
si requérir(n_f) = *faux* **alors** retourner(*faux*);
pour tout $(n_a, n_f) \in \mathit{Adds}(PG)$ **faire** asserter(n_f);
pour tout $\{n_a, n_b\} \in \mathit{AMutex}(PG)$ **faire**
si $\mathit{Valeur}(n_b) = \mathit{utilisé}$ **alors** retourner(*faux*);
retourner(*vrai*)
fin

Quand une action est exclue, on doit vérifier que chacun de ses ajouts peut toujours être produit par une autre action (**nb-producteurs-possibles** retourne le nombre d'actions indéfinies ou utilisées qui produisent cet effet). Si ce n'est pas le cas, cet ajout doit être nié. S'il était requis, aucune solution ne peut être trouvée avec l'assignation courante (i.e. une clause de la règle 3 du codage SAT ne peut pas être satisfaite).

fonction exclure(n_a)
$\mathit{Valeur}(n_a) \leftarrow \mathit{exclu}$;
pour tout $(n_a, n_f) \in \mathit{Adds}(PG)$ **faire**
si nb-producteurs-possibles(n_f) = 0 **alors**
cas-de

$Valeur(n_f) = requis$: retourner($faux$)
$Valeur(n_f) = indéfini$: $Valeur(n_f) \leftarrow nié$
fin-cas;
retourner($vrai$)
fin

L'algorithme DPPMin que nous venons de décrire n'utilise pas la valeur *requis_faux* : un nœud de fluent ne devant pas être présent dans un état est simplement nié lorsque toutes les actions qui le produisent sont exclues.

3.2 Exemple

Reprenons l'exemple décrit dans la Section 2.2 et résolvons-le avec DPPMin. Lors de l'étape d'initialisation, le noeud $a(0)$ est asserté car il appartient à l'état initial et le fluent $d(3)$ est requis car c'est le but du problème. La liste des buts contient donc uniquement $d(3)$. Ensuite, $C(3)$ est utilisé. Ses préconditions $b(2)$ et $c(2)$ sont requises et placées dans la liste des buts. $d(3)$ est asserté et retiré de la liste des buts, et on vérifie que les noeuds d'action mutex avec $C(3)$ ne sont pas utilisés. La liste des buts contient maintenant $b(2)$ et $c(2)$. Une action qui produit un des buts est alors choisie par une heuristique (cf. Section 6 pour le détail de cette heuristique). Supposons que le noeud $A(2)$ soit choisi pour produire $b(2)$. Sa précondition $a(1)$ est requise et $b(2)$ est asserté et retiré de la liste des buts qui contient maintenant $a(1)$ et $c(2)$. On choisit ensuite un noeud d'action pour produire $c(2)$. Or, les deux actions qui le produisent ($B(2)$ et $N\_c(2)$) sont mutex avec $A(2)$, la recherche échoue. On effectue donc un retour arrière sur le choix d'une action qui produit $b(2)$, en excluant $A(2)$. On choisit alors $N\_b(2)$ pour produire $b(2)$: $b(1)$ est requis et placé dans la liste des buts, et $b(2)$ est asserté et retiré de la liste des buts. La liste des buts contient $b(1)$ et $c(2)$. On choisit maintenant par exemple $A(1)$ pour produire $b(1)$: $a(0)$ étant déjà asserté, il n'est pas requis, et $b(1)$ est asserté. On choisit ensuite $B(2)$ pour produire $c(2)$, $a(1)$ est requis et $c(2)$ est asserté. On vérifie que $B(2)$ n'est pas mutex avec $N\_b(2)$. On choisit enfin $N\_a(1)$ pour produire $a(1)$: $a(0)$ n'est pas requis car il est déjà asserté, et $a(1)$ est asserté. On vérifie que $N\_a(1)$ n'est pas mutex avec $A(1)$. La liste des buts est maintenant vide, le plan solution est donc la séquence d'ensembles d'actions $\langle\{A, N\_a\}, \{N\_b, B\}, \{C\}\rangle$ qui peut être simplifiée en enlevant les no-ops et donne $\langle A, B, C\rangle$.

4 Une version étendue de DPPMin : DPP

L'algorithme DPP décrit dans cette section est construit sur la base de DPPMin, auquel nous avons ajouté le plus possible de propagations. Certaines d'entre elles sont déjà utilisées dans DPPlan, d'autres sont nouvelles. A cause de toutes ces propagations, nous utiliserons un moyen plus approprié pour notifier les inconsistance plutôt que retourner *vrai* ou *faux*, inspiré des échappements dynamiques non locaux de Common Lisp [14]. Nous définissons les deux fonctions suivantes :

catch($\langle$*étiquette*$\rangle$, $\langle$*séquence d'instructions*$\rangle$) **:** les instructions sont évaluées. Si **throw** n'est jamais rencontré, **catch** retourne le résultat de la dernière instruction. Sinon, si **throw** est rencontré avec la même étiquette, un échappement est produit : l'exécution des instructions est stoppée et **catch** retourne l'étiquette.

throw($\langle$*étiquette*$\rangle$) **:** un échappement est produit vers le plus récent appel à **catch** avec la même étiquette.

Nous utiliserons deux étiquettes pour les échappements : *succès* quand une solution est trouvée, et *inconsistance* quand on essaie d'assigner deux valeurs incompatibles à un même nœud ou qu'aucune solution ne peut être trouvée (l'utilisation puis l'exclusion d'une action ont conduit à un échec).

L'initialisation des valeurs peut maintenant produire un échappement, donc l'appel à **init** et **rechercher** est protégé par **catch**. La fonction **init** est identique à celle de DPPMin.

fonction DPP()
  résultat $\leftarrow$ catch(*succès*, catch(*inconsistance*, {init(); rechercher()}));
  si *résultat* = *succès* **alors** retourner(extraire-plan(PG)) **sinon** retourner(*échec*)
fin

La fonction **rechercher** est modifiée pour prendre en compte les échappement : *succès* est produit si la liste des buts est vide, *inconsistance* est rattrapé quand l'utilisation d'une action ne peut conduire à une solution.

fonction rechercher()
  si *liste_buts* = $\emptyset$ **alors** throw(*succès*);
  $n_a \leftarrow$ choisir(*liste_buts*);
  sauver-valeurs-et-buts();
  catch(*inconsistance*, {utiliser(n_a); rechercher()});
  restaurer-valeurs-et-buts();
  exclure(n_a);
  rechercher()
fin

Asserter un fluent peut maintenant produire un échappement, lorsque le fluent était déjà nié. **asserter** est appelé seulement pour les ajouts d'une action par **utiliser**, qui exclut toutes les actions mutex avec l'action utilisée ; ainsi, il serait redondant de propager sur les fluents mutex avec le fluent asserté comme cela est fait dans DPPlan.

fonction asserter(n_f)
  cas-de
    Valeur(n_f) = *nié* : throw(*inconsistance*)
    Valeur(n_f) = *requis* : *Valeur*(n_f) $\leftarrow$ *asserté* ;
      liste_buts $\leftarrow$ *liste_buts* $\setminus \{n_f\}$
    Valeur(n_f) = *indéfini* : *Valeur*(n_f) $\leftarrow$ *asserté*
fin

Dans **requérir** nous avons ajouté beaucoup de propagations par rapport à DPPlan. Si un fluent n_f est asserté ou requis au niveau précédent, on peut utiliser directement son no-op. Si aucune action ne peut produire n_f (elles sont toutes exclues), un échappement est produit. Si une seule peut être utilisée, c'est fait immédiatement (**producteur-possible** retourne la première action ayant la valeur *indéfini* ou *utilisé*). Sinon, n_f prend la valeur *requis* et est ajouté à la liste des buts. Les fluents mutex sont niés et les actions qui retirent n_f sont exclues. **nier** est paramétrée par un booléen, dont la signification sera donnée plus loin.

fonction requérir(n_f)
  cas-de
    Valeur(n_f) = *nié* : throw(*inconsistance*)
    Valeur(n_f) = *indéfini* :
      /* avec $n_f = f(i)$ */
      si *Valeur*($f(i-1)$) $\in$ {*asserté*, *requis*} **alors** utiliser($N\_f(i)$)
      sinon
        cas-de
          nb-producteurs-possibles(n_f) = 0 : throw(*inconsistance*)
          nb-producteurs-possibles(n_f) = 1 : utiliser(producteur-possible(n_f))
          sinon
            Valeur(n_f) $\leftarrow$ *requis* ;
            liste_buts $\leftarrow$ *liste_buts* $\cup$ {n_f};
            pour tout $\{n_f, n_h\} \in$ *FMutex*(*PG*) **faire** nier(n_h, *vrai*);
            pour tout $(n_a, n_f) \in$ *Dels*(*PG*) **faire** exclure(n_a)
fin

Dans **nier** nous avons aussi ajouté plusieurs propagations. Si un fluent n_f est asserté ou requis au niveau précédent et qu'une seule action peut le retirer, cette dernière est utilisée immédiatement. Ce cas n'était pas détecté par **exclure** si $f(i-1)$ n'avait pas encore une valeur positive. On exclut ensuite les actions qui produisent n_f si le booléen *toutes_propagations* est vrai : ceci est toujours fait, à l'exception d'un cas dans **exclure** où c'est redondant. Les actions qui ont n_f comme précondition sont exclues. Finalement si aucune action ne peut retirer n_f, on le nie au niveau précédent.

fonction nier(n_f, *toutes_propagations*)
  cas-de
    Valeur(n_f) $\in$ {*asserté*, *requis*} : throw(*inconsistance*)
    Valeur(n_f) = *indéfini* :
      /* avec $n_f = f(i)$ */
      Valeur(n_f) $\leftarrow$ *nié* ;
      si nb-destructeurs-possibles(n_f) = 1 et *Valeur*($f(i-1)$) $\in$ {*asserté*, *requis*} **alors**
        utiliser(destructeur-possible(n_f))
      sinon
        si *toutes_propagations* **alors**
          pour tout $(n_a, n_f) \in$ *Adds*(*PG*) **faire** exclure(n_a);
        pour tout $(n_f, n_a) \in$ *Precs*(*PG*) **faire** exclure(n_a);
        si nb-destructeurs-possibles(n_f) = 0 **alors** nier($f(i-1)$, *vrai*)
fin

La fonction **utiliser** est quasiment la même que dans DPPMin, excepté le fait qu'une action peut être déjà exclue. Les propagations sur les actions mutex sont faites par **exclure** au lieu d'être simplement vérifiées.

fonction utiliser(n_a)
  cas-de
    Valeur(n_a) = *exclu* : throw(*inconsistance*)
    Valeur(n_a) = *indéfini* :
      Valeur(n_a) $\leftarrow$ *utilisé* ;
      pour tout $(n_f, n_a) \in$ *Precs*(*PG*) **faire** requérir(n_f);
      pour tout $(n_a, n_f) \in$ *Adds*(*PG*) **faire** asserter(n_f);
      pour tout $\{n_a, n_b\} \in$ *AMutex*(*PG*) **faire** exclure(n_b)
fin

Comparé à DPPlan, **exclure** est modifiée surtout pour prendre en compte la suppression de la valeur *requis_faux*. Quand toutes les actions qui produisent un fluent n_f sont exclues, n_f est nié. Dans DPPlan, il prenait la valeur *requis_faux*. **nier** est paramétrée par *faux*, car toutes les actions qui produisent n_f sont déjà exclues donc la propagation dans **nier** est redondante. Quand une seule action peut encore être utilisée pour produire un fluent requis, elle est utilisée immédiatement. Quand un fluent n_f a une valeur positive au niveau précédent et qu'aucune action ne peut le retirer, son no-op est utilisé directement. Dans DPPlan, n_f était requis. Quand un fluent n_f est nié et qu'aucune action ne peut le retirer, n_f est nié au niveau précédent. Dans DPPlan, n_f prenait la valeur *requis_faux* au niveau précédent. Finalement quand un fluent est nié, a une valeur positive au niveau précédent, et qu'il peut être retiré par seulement une action, cette dernière est utilisée immédiatement.

fonction exclure(n_a)
  cas-de
    Valeur(n_a) = *utilisé* : throw(*inconsistance*)
    Valeur(n_a) = *indéfini* :
      Valeur(n_a) $\leftarrow$ *exclu*;
      pour tout $(n_a, n_f) \in Adds(PG)$ **faire**
        cas-de
          nb-producteurs-possibles(n_f) = 0 : nier(n_f, *faux*)
          nb-producteurs-possibles(n_f) = 1 et *Valeur*(n_f) = *requis* :
            utiliser(producteur-possible(n_f))
        fin-cas;
      pour tout $(n_a, n_f) \in Dels(PG)$ **faire**
        / * avec $n_f = f(i)$ * /
        cas-de
          nb-destructeurs-possibles(n_f) = 0 :
            cas-de
              Valeur($f(i-1)$) $\in$ {*asserté*, *requis*} : utiliser($N\_f(i)$)
              Valeur(n_f) = *nié* : nier($f(i-1)$, *vrai*)
            fin-cas
          nb-destructeurs-possibles(n_f) = 1 et *Valeur*(n_f) = *nié* et
          Valeur($f(i-1)$) $\in$ {*asserté*, *requis*} :
            utiliser(destructeur-possible(n_f))
fin

5 L'utilisation de la relation d'autorisation dans DPP : LCDPP

5.1 Modifications de DPP

La relation d'autorisation [4, 5, 6, 15] est une relation moins contrainte que l'indépendance qui est traditionnellement utilisée pour calculer les exclusions mutuelles entre actions. Une action a autorise une action b (noté $a \angle b$) ssi (1) $a \neq b$, (2) a ne retire pas de précondition de b, et (3) b ne retire pas d'ajout de a. Une séquence d'actions $\langle a_1, \ldots, a_n \rangle$ est autorisée ssi $\forall i \in [1, n-1], \forall j \in [i+1, n]$, $a_i \angle a_j$. Un ensemble d'actions Q est autorisé ssi il existe au moins une linéarisation autorisée de Q. L'utilisation de l'autorisation dans DPP nécessite la modification des trois étapes suivantes de l'algorithme :

1. *Construction du graphe de planification :* l'autorisation remplace l'indépendance pour le calcul des mutex entre actions. Deux nœuds d'action n_a et n_b sont maintenant mutex ssi (1) $n_a \neq n_b$ et (2) $(\neg(a \angle b)$ et $\neg(b \angle a))$ ou une précondition de n_a (ou n_b) est mutex avec une précondition de n_b (ou n_a).
2. *Recherche d'un plan :* l'autorisation des actions utilisées à chaque niveau est vérifiée à chaque fois qu'une nouvelle action est utilisée. On emploie l'algorithme polynomial **SearchSeq** proposé dans [6, 15], qui calcule un tri topologique des actions utilisées par rapport à l'autorisation (**SearchSeq**(Q) retourne *échec* lorsqu'il n'existe aucun ordre partiel sur les actions de Q) :

   fonction utiliser(n_a)
     cas-de
       $\mathit{Valeur}(n_a) = \mathit{exclu}$: throw(*inconsistance*)
       $\mathit{Valeur}(n_a) = \mathit{indéfini}$:
         $\mathit{Valeur}(n_a) \leftarrow \mathit{utilisé}$;
         / * avec $n_a = a(i)$ et $Q = \{a \mid \mathit{Valeur}(a(i)) = \mathit{utilisé}\}$ * /
         si SearchSeq(Q) = *échec* **alors** throw(*inconsistance*);
         pour tout $(n_f, n_a) \in \mathit{Precs}(PG)$ **faire** requérir(n_f);
         pour tout $(n_a, n_f) \in \mathit{Adds}(PG)$ **faire** asserter(n_f);
         pour tout $\{n_a, n_b\} \in \mathit{AMutex}(PG)$ **faire** exclure(n_b)
   fin

   La fonction **requérir** doit aussi être modifiée, car un fluent peut maintenant être retiré par une action a et ajouté par une action b au même niveau si $a \angle b$. Ainsi, si n_a et n_b sont deux nœuds d'action tels que $a \angle b$:
   - Si un nœud de fluent n_f est requis, et est asserté ou requis au niveau précédent, on ne peut plus utiliser son no-op car n_f peut être retiré par n_a et ajouté par n_b ;
   - Si n_a retire un fluent requis n_f, on ne peut pas exclure n_a car n_f peut être ajouté par une autre action comme n_b.

   Ceci nous conduit à la fonction suivante :

   fonction requérir(n_f)
     cas-de
       $\mathit{Valeur}(n_f) = \mathit{nié}$: throw(*inconsistance*)
       $\mathit{Valeur}(n_f) = \mathit{indéfini}$:
         cas-de
           nb-producteurs-possibles$(n_f) = 0$: throw(*inconsistance*)
           nb-producteurs-possibles$(n_f) = 1$: utiliser(producteur-possible(n_f))
           sinon
             $\mathit{Valeur}(n_f) \leftarrow \mathit{requis}$;
             $\mathit{liste\_buts} \leftarrow \mathit{liste\_buts} \cup \{n_f\}$;
             pour tout $\{n_f, n_h\} \in \mathit{FMutex}(PG)$ **faire** nier(n_h, vrai)
   fin

3. *Retour du plan solution :* le plan solution extrait, qui consiste en une séquence d'ensembles d'actions autorisés, est réordonnancé de manière optimale en une séquence d'ensembles d'actions indépendants par l'algorithme polynomial **SearchReordering** défini dans [6, 15], qui est une version modifié de l'algorithme PRF [1, 13] :

fonction LCDPP()
  résultat $\leftarrow$ catch(*succès*, catch(*inconsistance*, {init(); rechercher()}));
  si *résultat* = *succès* **alors** retourner(**SearchReordering**(extraire-plan(PG)))
  sinon retourner(*échec*)
fin

5.2 Exemple

Reprenons l'exemple décrit dans la section 2.2. Le graphe de planification ne contient plus que trois niveaux (cf. Figure 2), car A autorise B (représenté par une flèche sur le graphe). Les fluents $b(1)$ et $c(1)$ ne sont pas mutex, donc C peut être utilisée au niveau 2 et le but apparaît à ce niveau.

La différence essentielle de la résolution du problème avec LCDPP (que nous n'allons pas détailler) par rapport à DPPMin vient du fait qu'il n'y a qu'une seule possibilité pour produire $b(1)$ et $c(1)$ qui sont les préconditions de $C(2)$ que l'on doit utiliser pour produire le but. De ce fait, la recherche va être faite entièrement dans l'étape d'initialisation et on ne fera donc jamais appel à la fonction de choix d'une action, grâce aux propagations de **requérir** qui vont forcer l'utilisation de $A(1)$ et $B(1)$. Il faut de plus vérifier qu'une séquence autorisée de $\{A, B\}$ existe, ce qui est immédiat. La plan autorisé produit est donc $\langle\{A, B\}, \{C\}\rangle$ que l'on peut réordonnancer en la séquence $\langle A, B, C\rangle$.

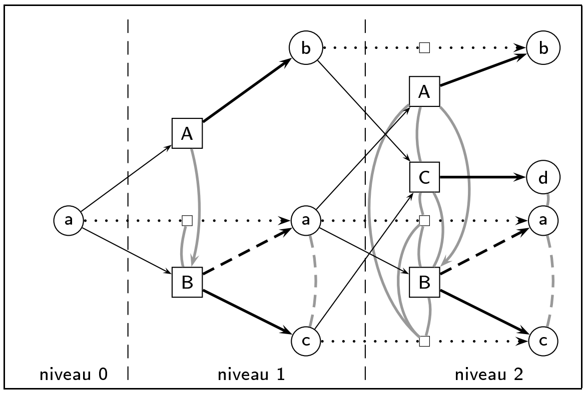

FIG. 2 – Le graphe de planification de Π avec autorisation entre actions

6 Évaluation expérimentale

Tous les planificateurs comparés ici sont implémentés en Allegro Common Lisp 5.0 et partagent la majorité de leur code, à l'exception du planificateur FF v2.2 [8], fonctionnant par recherche heuristique en chaînage avant dans les espaces d'états, implémenté en C. Tous les tests ont été exécutés sur un Pentium-II 450MHz avec 256Mo de RAM, sous Debian GNU/Linux 2.2. Nous ne reportons ici que le temps de recherche (en secondes) avec une limite maximale d'une heure ; la qualité des solutions en nombre d'actions est en effet comparable pour tous les planificateurs.

L'heuristique utilisée dans la phase de recherche de tous les planificateurs basés sur Graphplan et DPPlan est « le niveau d'apparition le plus élevé d'abord » pour le choix des fluents et « no-op d'abord, puis niveau d'apparition le plus faible d'abord » pour le choix des actions, ce qui paraît être généralement un bon choix [5, 6, 9, 10, 15]. En effet, plus un fluent apparaît tard dans le graphe, plus son obtention peut être considérée comme difficile : si la recherche doit échouer, autant que ce soit le plus rapidement possible. A l'inverse, pour établir un fluent difficile, une action apparaissant tôt dans le graphe aura des préconditions a priori plus faciles à obtenir ; ainsi, on essaie de maximiser les chances d'établir en premier les fluents les plus contraints.

Nous comparons d'abord notre propre version de DPPlan avec DPP (cf. Table 1) sur plusieurs problèmes issus de benchmarks classiques. DPP est toujours plus rapide que DPPlan, de 1,33 à 3,44 fois.

Problèmes	Temps CPU (s)		ratio DPPlan /
	DPPlan	DPP	DPP
ferry4	7,70	3,18	2,42
ferry5	475,33	142,34	3,34
gripper6	779,64	285,57	2,73
bw-large-a	2,95	2,22	1,33
bw-large-b	43,83	12,75	3,44
log018	306,09	148,93	2,06
log021	148,95	106,18	1,40
log030	225,02	121,35	1,85

TAB. 1 – DPPlan vs. DPP

Nous comparons ensuite DPP, GP (Graphplan avec la procédure de recherche classique) et GPBJ (Graphplan avec les améliorations EBL/DDB de [9]) sur les 30 problèmes Logistics de la distribution Blackbox [12] (cf. Figure 3 : en abscisse se trouvent les numéros référençant les problèmes, en ordonnée le temps de calcul en secondes). DPP, GP et GPBJ résolvent respectivement 17, 4 et15 problèmes, DPP étant généralement le plus efficace. Sur les mêmes problèmes, nous comparons DPP, LCDPP, LCGP (GP avec autorisation) et LCGPBJ (GPBJ avec autorisation). LCDPP résout 29 problèmes. 17 problèmes sont résolus par DPP avec un temps moyen de 298 sec., tandis qu'ils sont résolus par LCDPP avec un temps moyens de 2,51 sec. : l'utilisation de la relation d'autorisation améliore beaucoup les performances de l'algorithme DPPlan. On peut remarquer que si LCGPBJ

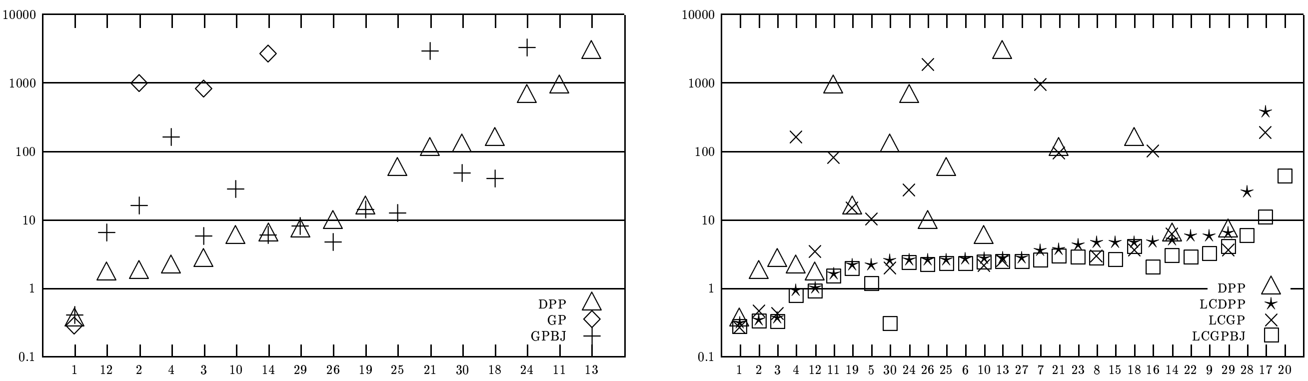

FIG. 3 – Domaine Logistics

est légèrement plus efficace que LCDPP, LCGP sans EBL/DDB est beaucoup moins efficace que LCDPP, qui n'utilise qu'un simple retour arrière chronologique.

Nous comparons ensuite les même planificateurs avec FF [8] au lieu de LCGP, dans une série de problèmes du domaine des cubes avec trois opérateurs provenant de la deuxième compétition de planificateurs lors de la conférence AIPS'2000 (cf. Figure 4). DPP est maintenant moins efficace que GP et GPBJ, qui sont à peu près équivalents. Mais avec l'autorisation, LCDPP qui résout les 50 problèmes de la série devient meilleur que LCGPBJ, qui n'en résout que 39. De plus, seulement 38 problèmes sont résolus par FF, qui était le planificateur le plus rapide de la compétition d'AIPS'2000.

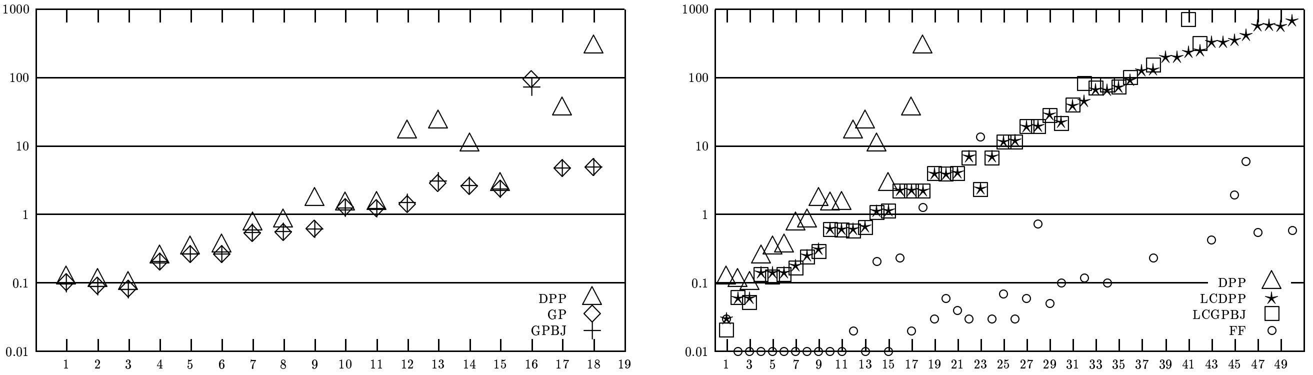

FIG. 4 – Domaine des cubes (3 ops.)

Nous comparons finalement DPP, LCDPP et FF dans les domaines Mprime et Mystery provenant de la première compétition de planificateurs lors de la conférence AIPS'98 (cf. Figure 5). DPP et LCDPP sont à peu près équivalents, et résolvent 32 problèmes sur 35 dans le domaine Mprime et 29 problèmes sur 30 dans le domaine Mystery. FF est plus rapide, mais ne résout que 31 problèmes de Mprime et 18 problèmes de Mystery.

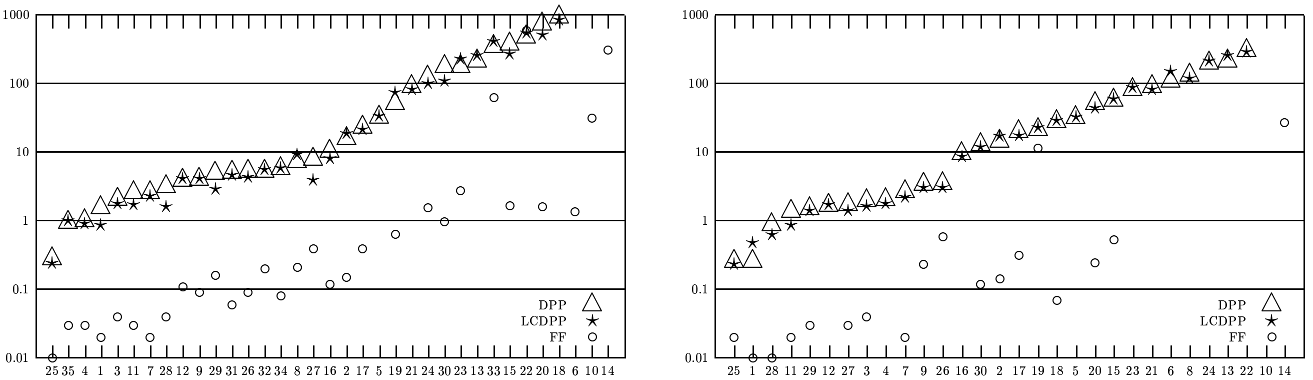

FIG. 5 – Domaines Mprime et Mystery

7 Conclusions et travaux futurs

Nous avons présenté dans cet article une amélioration de la procédure d'extraction de plans DPPlan, motivée par le fait que la valeur *requis_faux* n'est pas utile pour représenter le fait qu'un fluent doit être nié, comme cela est montré grâce à une version « minimale » de DPPlan basée sur le codage SAT du graphe de planification. Nous avons ajouté plusieurs propagations, spécialisé certaines d'entre elles en particulier par l'utilisation des no-ops, et supprimé d'autres pour des raisons de redondance. L'algorithme DPP que nous obtenons est significativement plus rapide que la version originelle de DPPlan. Nous avons alors utilisé la relation d'autorisation dans DPP, ce qui conduit comme dans Graphplan à des performances accrues. Le planificateur LCDPP qui en résulte peut ainsi trouver une solution à des problèmes difficiles pour lesquels un planificateur plus rapide comme FF échoue. LCGPBJ est parfois plus efficace que LCDPP, mais il utilise des techniques de retour arrière intelligent et de mémoization ; nous projetons maintenant d'améliorer LCDPP à l'aide de ces techniques.

Remerciements

Merci aux relecteurs pour leurs nombreuses et intéressantes remarques. Merci également à Michel Cayrol et Pierre Régnier pour leur aide.

Références

[1] C. Bäckström. Computational aspects of reordering plans. *JAIR*, 9:99–137, 1998.

[2] M. Baioletti, S. Marcugini, et A. Milani. DPPlan: An algorithm for fast solutions extraction from a planning graph. Dans *Proc. AIPS'2000*, pages 13–21, 2000.

[3] A. Blum et M. Furst. Fast planning through planning-graphs analysis. *Artificial Intelligence*, 90:281–300, 1997.

[4] M. Cayrol, P. Régnier, et V. Vidal. LCGP : une amélioration de graphplan par relâchement de contraintes entre actions simultanées. Dans *Proc. RFIA'2000*, pages 79–88, 2000.

[5] M. Cayrol, P. Régnier, et V. Vidal. New results about LCGP, a least committed Graphplan. Dans *Proc. AIPS'2000*, pages 273–282, 2000.

[6] M. Cayrol, P. Régnier, et V. Vidal. Least commitment in Graphplan. *Artificial Intelligence*, 130:85–118, 2001.

[7] M. Davis, G. Logemann, et D. Loveland. A computing procedure for quantification theory. *Communications of the ACM*, 5:394–397, 1962.

[8] J. Hoffmann et B. Nebel. The FF planning system: fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[9] S. Kambhampati. Planning graph as a (dynamic) CSP: Exploiting EBL, DDB and other CSP techniques in Graphplan. *JAIR*, 12:1–34, 2000.

[10] S. Kambhampati et R.S. Nigenda. Distance-based goal-ordering heuristics for Graphplan. Dans *Proc. AIPS'2000*, pages 315–322, 2000.

[11] H. Kautz, D. McAllester, et B. Selman. Encoding plans in propositional logic. Dans *Proc. KR'96*, pages 374–384, 1996.

[12] H. Kautz et B. Selman. Unifying SAT-based and Graph-based planning. Dans *Proc. IJCAI'99*, pages 318–325, 1999.

[13] P. Régnier et B. Fade. Complete determination of parallel actions and temporal optimization in linear plans of actions. Dans *Proc. EWSP-91*, pages 100–111, 1991.

[14] G.L. Steele. *Common Lisp the Language.* Digital Press, Woburn, MA, 2$^{\text{nd}}$ edition, 1989.

[15] V. Vidal. *Recherche dans les graphes de planification, satisfiabilité et stratégies de moindre engagement.* Thèse de doctorat, Université Paul Sabatier, Juillet 2001.