L'algorithme *path-repair*

Narendra Jussien et Olivier Lhomme

École des Mines de Nantes - La Chantrerie
4, rue Alfred Kastler - B. P. 20722
44 307 Nantes Cedex 3 - France
{jussien, lhomme}@emn.fr

1 Introduction

De nombreux problèmes réels rencontrés aussi bien dans le domaine industriel qu'en ingénierie peuvent être appréhendés sous l'angle des problèmes de satisfaction de contraintes (CSP). Un CSP est défini par un ensemble de variables possédant chacune un domaine de valeurs possibles et un ensemble de contraintes sur ces variables.

La plupart des algorithmes de résolution de systèmes de contraintes utilisent une recherche basée sur le retour arrière. Ces algorithmes explorent donc habituellement l'espace de recherche d'une manière systématique garantissant ainsi la découverte d'une solution réalisable si celle-ci existe. Bien sûr, ces algorithmes sont généralement utilisés conjointement à des techniques de filtrage dont le but est de réduire l'espace de recherche afin de diminuer la durée totale de calcul de solution.

Une autre famille d'algorithmes de recherche est constituée par les méthodes de recherche locale. Dans ce cas, une exploration stochastique de l'espace de recherche est effectuée ne permettant alors en aucun cas de garantir la découverte d'une solution. L'intérêt de ces méthodes locales (par exemple : la recherche Taboue [10], l'algorithme GSAT [20]) réside dans le fait qu'utilisant des techniques de descente de gradient, elles peuvent être beaucoup plus efficace (en terme de temps de réponse) que des approches systématiques.

Plueiseurs travaux ont exploré différentes voies de coopération entre recherche locale et recherche systématique [23], [16],[18], [7], [17]. Ces approches coopératives ont montré de très bons résultats expérimentaux sur des problèmes de grande taille.

À notre connaissance cependant, aucune véritable intégration de techniques de filtrage dans les algorithmes de recherche locale n'a été réalisée jusqu'à présent.

Cet article aborde l'intégration d'un algorithme de filtrage dans un algorithme de recherche locale. Ceci nous mène vers une nouvelle technique de recherche sur les CSP que nous appelons *path-repair*. Nous présentons différentes variations autour de cette technique. Les résultats obtenus au cours des premières expérimentations semblent très prometteurs.

La section 2 présente la notation utilisée, la section 3 décrit l'algorithme *path-repair*, et la section 4 présente les premiers résultats expérimentaux.

2 Préliminaires

Un CSP est un couple $< V, C >$ où V est un ensemble de variables, et $C = \{c_1, \ldots, c_m\}$ est un ensemble de contraintes. Les domaines des variables sont représentés par des contraintes unaires.

Pour un ensemble donné de contraintes $S = \{c_1, \ldots, c_k\}$, nous noterons $\hat{S}$ la conjonction logique des contraintes de S. $\hat{S} = (c_1 \wedge \ldots \wedge c_k)$. On supposera $\hat{\varnothing} = true$.

La résolution classique d'un CSP passe par l'utilisation simultanée d'un algorithme de filtrage (pour réduire l'arbre de recherche *a priori*) et d'une mécanisme d'énumération (pour pallier l'incomplétude des algorithmes de filtrage). Par exemple, pour les CSP binaires sur les domaines finis, on peut utiliser la consistance d'arc comme technique de filtrage. Dans ce cas, après filtrage plusieurs situations peuvent se présenter :

1. le domaine d'une des variables du problème est vide : il n'y a pas de solution pour le problème considéré.

2. tous les domaines sont réduits à un singleton : les valeurs restantes, une fois affectées à leur variable respective donne une solution pour le problème.
3. il existe au moins un domaine qui n'est pas un singleton : la recherche n'est pas terminée, il faut énumérer. Dans le cas d'une approche classique, on utilise ici une approche basée sur le retour arrière pour énumérer.

D'une manière plus générale, on peut considérer que pour tout algorithme de filtrage[1] Φ appliqué à l'ensemble des contraintes C d'un CSP donné (on obtient alors $C' = \Phi(C)$, il existe une fonction $obviousInference()$ qui appliquée à C' répond :

- $noSolution$ si et seulement si il est immédiat de constater qu'aucune solution ne peut vérifier C (situation 1 ci-dessus) ;
- $allSolution$ si et seulement si le système de contraintes[2] permet de donner directement une solution vérifiant l'ensemble des contraintes de $C' = \Phi(C)$ (situation 2) ;
- $flounder$ dans tous les autres cas (situation 3).

Un pas d'énumération pour un CSP binaire discret revient à donner une valeur[3] a à une variable v_k *i.e.* à ajouter une contrainte $v_k = a$. Pour d'autres types de problèmes le pas d'énumération est quelque peu différent : par exemple, pour les CSP numériques, faire un pas d'énumération consiste à ajouter une contrainte de splitting (*eg.* $v_\ell < a$). Lorsqu'on considère des problèmes d'ordonnancement, faire un pas d'énumération revient souvent à ajouter une contrainte de précédence entre deux tâches du problèmes.

Le concept clé de ces différents types de contraintes d'*énumération* est qu'elles vérifient un critère simple : il existe un entier N_e tel que quel que soit l'ensemble E d'au moins N_e contraintes différentes d'énumération, la fonction $obviousInference(\Phi(C \cup E))$ ne répondra pas *flounder*.

3 L'algorithme *path-repair*

Les idées qui sous-tendent l'algorithme *path-repair* sont très simples. On peut tout d'abord remarquer que :

- Les algorithmes de recherche locale existants travaillent sur une instanciation complète des variables du problème ;
- Les algorithmes de recherche basés sur le retour arrière travaillent eux sur une instanciation partielle des variables du problème.

La capacité des algorithmes basés sur le retour arrière à être combinés à des techniques de filtrage provient uniquement du fait qu'ils travaillent sur une instanciation partielle des variables. Ainsi, un *algorithme de recherche locale travaillant sur une instanciation partielle aura aussi cette capacité.*

L'algorithme *path-repair* possède cette particularité. L'instantiation partielle considérée est définie par un ensemble de contraintes d'énumération (au sens où on vient de les définir) sur les variables du problème. Cet ensemble de contraintes d'énumération définit en fait un chemin dans l'arbre d'exploration du problème.

3.1 Principes de Path-Repair

Le principe de l'algorithme *path-repair* détaillé figure 1 est le suivant : soit P un chemin dans l'arbre de recherche. En chaque nœud de cet arbre, une contrainte d'énumération est ajoutée. Soit C_P l'ensemble des contraintes d'énumération ajoutées lors du parcours du chemin P.

1. Un algorithme de filtrage Φ appliqué à un ensemble de contraintes C donne un nouvel ensemble $C' = \Phi(C)$, tel que $C \subseteq C'$ (des contraintes redondantes ayant pu être ajoutées).
2. On considère les réductions de domaine comme des contraintes ajoutées au système de contraintes.
3. a est un élément du domaine de v_k.

```
procédure Path-repair(C)
P := chemin initial
boucle
        si conditions d'échec vérifiées alors
           return échec
        sinon
           C':= Φ(C ∪ C_P)
        si ObviousInference(C') = noSolution alors
           soit k un nogood expliquant l'échec
           P := neighbor(P, k, Γ)
        sinon si ObviousInference(C') = allSolution alors
           return C'
        sinon
           P:= extend(P,Γ)
fin boucle
```

Fig. 1 –. *L'algorithme* Path-repair

Un algorithme de filtrage est appliqué à $C \cup C_P$ donnant un nouvel ensemble de contraintes $C' = \Phi(C \cup C_P)$. La fonction $ObviousInference$ est alors appelée sur C'. Trois cas sont à envisager :

- $ObviousInference(C') = allSolution$: une solution a été trouvée. L'algorithme termine et renvoie C'.
- $ObviousInference(C') = flounder$: l'algorithme *Path-Repair* étend le chemin courant P en lui ajoutant une contrainte d'énumération. Ce comportement est tout à fait classique dans une approche basée sur le retour arrière. Nous supposons l'existence d'une fonction $extend(P, \Gamma)$ qui choisit une nouvelle contrainte et l'ajoute à P. L'utilisation et la signification du paramètre Γ seront discutées plus tard.
- $ObviousInference(C') = noSolution$: $C \cup C_P$ est inconsistant. Nous dirons donc que P est un cul-de-sac ou est inconsistant : P ne peut plus être étendu. Notre algorithme va donc maintenant *réparer* le *chemin* courant en choisissant un nouveau chemin à l'aide de la fonction $neighbor(P, k, \Gamma)$. Comme précédemment, k et Γ seront expliqués plus tard.

3.2 Propriétés du schéma de voisinage de Path-Repair

Dans un algorithme de recherche locale de type GSAT (sur les CSP booléens), une instanciation inconsistante est remplacée par une nouvelle instanciation construite à partir de la première en *niant* (au sens logique) la valeur d'une des variables. Cette variable est choisie à l'aide d'une heuristique (par exemple : celle dont la négation permet la plus grande augmentation du nombre de clauses satisfaites). Plus généralement, une recherche locale utilise des instanciations complètes (appelées *états*) et remplace un état inconsistant par un autre état choisi parmi ses *voisins*.

L'algorithme *path-repair* fonctionne un peu de la même façon, sauf qu'il utilise des instanciations partielles (*chemins*) : lorsqu'un chemin devient inconsistant il faut sélectionner un de ses voisins. Un chemin (une instanciation partielle) permet de synthétiser toutes les instanciations complètes en découlant. Changer de chemin revient donc à mettre de côté de nombreuses instanciations partielles en une seule fois.

Comme un algorithme classique de recherche locale, l'algorithme *path-repair* peut suivre une heuristique pour choisir un voisin intéressant. Cet algorithme peut aussi choisir un voisin de telle sorte que l'algorithme ainsi défini soit une méthode systématique d'exploration. La complétude est permise par le fait qu'un chemin regroupe en fait plusieurs instanciations complètes.

Les sections suivantes expliquent la notion de chemin voisin, montrent comment choisir heuristiquement un tel voisin et comme choisir les voisins pour obtenir une exploration systématique.

Nous avons introduit précédemment dans les fonctions de calcul de voisinage ($neighbor$) et d'extension du chemin courant ($extend$) une variable Γ. Γ peut être utilisée pour enregistrer un

contexte et peut varier suivant les différentes version de l'algorithme. Dans la version primitive de l'algorithme, ce contexte n'est pas utilisé.

3.3 La notion de chemin voisin

Il semble judicieux de choisir un voisin P' qui ne présente pas les mêmes inconvénients que le chemin courant P (rappelons que dans l'algorithme *path-repair*, lorsque l'on cherche un voisin de P, P est inconsistant). Il serait alors très intéressant d'être capable d'aller directement sur un voisin consistant *i.e.* tel que : $ObviousInference(\Phi(C \cup C'_P)) = allSolution$. Bien sûr, ceci n'est pas possible dans le cas général.

Aussi, on peut vouloir à juste titre être moins exigeant et se tourner vers un voisin au moins partiellement consistant *i.e.* tel que : $ObviousInference(\Phi(C \cup C'_P)) \neq noSolution$. Hélas, la seule manière d'y parvenir est de revenir sur un voisin déjà exploré mais faisant cela, on reviendrait alors à utiliser un mécanisme basé sur le retour arrière ce qui est contraire à notre objectif.

Néanmoins, il est quand même possible au moins d'éviter les voisins dont on peut savoir qu'ils sont inconsistants. Ce genre d'information peut être extraite d'un chemin inconsistant P. En effet, l'inconsistance signifie que $\hat{C} \wedge \hat{C_P} \Longrightarrow false$. Il est alors possible d'isoler un sous-ensemble de C_P qui est inconsistant avec C. Un tel sous-ensemble est alors appelé un *nogood* [6].

Définition 1 (Nogood) *Un nogood k, pour un ensemble de contraintes C et un chemin P, est un ensemble de contraintes tel que : $k \subset C_P$ et $\hat{C} \wedge \hat{k} \Longrightarrow false$.*

Tant que les contraintes d'un nogood calculé k seront toutes présentes dans un chemin quelconque P', celui-ci restera bien sûr inconsistant. Ainsi, pour obtenir un chemin convenable pour notre voisinage, il est nécessaire de retirer du chemin courant P au moins une contrainte appartenant au nogood k.

On peut noter que dans tous les cas, C_P est un nogood valide. Il est évident qu'un nogood strictement inclus sera plus intéressant et fournira des voisinages plus réduits. En particulier, un nogood minimal au sens de l'inclusion serait certainement très intéressant, mais très couteux à calculer [22]. En pratique on se contentera d'un nogood intermédiaire entre ces deux extrêmes.

Pour l'instant le voisinage ainsi défini reste très général. Dans la suite, nous définissons des voisinages plus pertinents. Le point que nous souhaitons ici mettre en avant est que le concept de nogood est crucial pour l'algorithme *path-repair* :

- un nogood permet de considérer un voisinage véritablement pertinent ;
- nous verrons comment dériver depuis les nogoods des heuristiques intéressantes permettant de donner une spécialisation non systématique de *path-repair* ;
- les nogoods peuvent aussi être utilisés pour donner une version complète de *path-repair*.

Les nogoods peuvent être fournis par l'algorithme de filtrage à chaque fois qu'il peut prouver qu'aucune solution n'existe dans les descendants du chemin courant. Dans les systèmes de contraintes basés sur le filtrage, une contradiction est identifiée dès que le domaine d'une variable v devient vide. Nous supposons que pour chaque valeur (ou ensemble de valeurs) a_i qui a été retirée du domaine de v, un ensemble de contraintes d'énumération $k_i \subset C_P$, appelé une explication du retrait de a_i, est fourni, avec k_i tel que $\hat{C} \wedge \hat{k_i} \Longrightarrow v \neq a_i$. Alors, $k = \bigcup_i k_i$ est un nogood. Ainsi, pour calculer des nogoods, il suffit d'être capable de déterminer une explication pour chaque retrait de valeur (ou d'ensemble de valeurs) du domaine de la variable en échec.

Les retraits de valeurs sont les conséquences de l'application des algorithmes de filtrage. Ainsi, les explications de retrait de valeur peuvent être calculées simplement en "traçant" l'algorithme de filtrage, et en mémorisant les raisons de chaque retrait réalisé [12].

Par exemple, si on considère deux variables v_1 et v_2 dont les domaines sont $[1, 2, 3]$. Soit la contrainte $c_1 : v_1 \geqslant v_2$. Supposons qu'une contrainte c_2 explique que $[1, 2]$ doit être retiré de v_2. Si on applique l'opérateur de consistance de c_1, il est alors nécessaire de retirer $[1, 2]$ du domaine de v_1. Une explication sera alors tout simplement : $c_1 \wedge c_2$ puisque c_1 ne provoque ce retrait que parce que certains retraits ont eu lieu dans v_2 à cause de c_2.

3.4 Instances de Path-Repair

Choix heuristique d'un voisin Dans un algorithme de recherche locale, le choix du voisin est très important. De nombreuses heuristiques peuvent être utilisées. De même pour *path-repair*, différentes heuristiques sont utilisables.

Jusqu'à présent, nous avons défini un voisin à partir d'un chemin P et d'un nogood k comme un chemin qui ne contient pas au moins une contrainte de k. En fait, on peut faire nettement mieux et définir un voisinage plus précis utilisable dans le cas d'une recherche non systématique. Soit c une contrainte destinée à être retirée de C_P. On sait déjà que c ne sera jamais satisfaisable tant que les contraintes $k \setminus c$ seront présentes. Ainsi, on peut ajouter la négation de k dans le nouveau chemin.

Ainsi, un voisinage possible pour un ensemble de contraintes inconsistant C_P, étant donné un nogood $k \subset C_P$, est constitué de tous les ensembles de contraintes C'_P différant de C_P par la négation d'une contrainte de k.

Maintenant, il reste un degré de de liberté dans le choix de la contrainte de k à nier. Notre approche initiale consistait à essayer d'adapter l'heuristique *min-conflict* [15] qui minimise le nombre de contraintes non vérifiées. Mais, cette notion n'a pas vraiment de sens lorsqu'un algorithme de filtrage est utilisé (en effet, la première contrainte non vérifiée stoppe l'algorithme).

L'approche que nous avons choisie d'utiliser associe à chaque contrainte un poids, qui comptabilise le nombre de fois qu'une contrainte est apparue dans un nogood. L'heuristique *poids-max* consiste alors à nier la contrainte de k ayant le plus fort poids. Une approche similaire comptabilisant le nombre de fois qu'une contrainte a été violée a été utilisée avec succès pour GSAT [19].

Path-repair(tabou) La version *taboue* de l'algorithme *path-repair* utilise une liste taboue d'une taille s fixée. Les s derniers nogoods sont alors conservés dans une liste Γ et un voisin est défini comme un chemin qui ne contient complètement aucun des nogoods de Γ. En d'autre termes, au moins une contrainte par nogood est absente du nouveau voisin. Un voisin peut être trouvé en résolvant un problème de recouvrement sur Γ [4]. La figure 2 décrit l'implémentation que nous avons utilisée de la fonction *neighbor* avec l'heuristique *poids-max* pour *Path-repair(tabou)*. Notons que de la même façon la fonction $extend(P, \Gamma)$ doit utiliser Γ pour étendre correctement le chemin partiellement consistant courant.

```
fonction neighbor(P, k, Γ)
ajouter k à la liste des nogoods Γ
si taille(Γ) > s alors
   retirer de Γ son élément le plus ancien
L:= liste triée par ordre de poids décroissants des contraintes de C_P
répéter
   retirer la 1ère contrainte c de L
   P' := P excepté que ¬c remplace c
   si C_P' recouvre tous les nogoods de Γ alors
      retourner P'
jusqu'à L vide
retourner stop (ou bien élargir le voisinage)
```

Fig. 2 –. *Fonction neighbor avec heuristique* poids-max*pour* Path-repair(tabou)

Instance systématique L'intérêt d'une approche basée sur le backtrack est qu'elle produit un algorithme de recherche systématique. Les techniques de filtrage sont utilisées dans ces approches

4. Plusieur algorithmes sub-optimaux efficaces existent pour ce genre de problèmes [4].

pour des raisons évidentes d'efficacité. Une telle utilisation (conjointement au traitement d'instanciations partielles) est encore plus intéressante dans un algorithme de recherche locale : elles peuvent non seulement apporter de l'efficacité mais aussi la complétude. Comment rendre *path-repair* systématique ?

Les nogoods permettent cette complétude. La manière la plus simple est de conserver tous les nogoods déjà calculés et de considérer comme voisin les chemins qui ne contiennent aucun nogood connu. Ainsi, si à un moment donné de la résolution aucun voisin n'existe, le problème considéré ne possède aucune solution réalisable. Bien sûr, l'inconvénient majeur de se comportement est qu'il requiert un espace de stockage potentiellement exponentiel pour conserver tous les nogoods.

Il est possible d'éviter ce problème et de conserver un espace de stockage polynomial. C'est ce qui est fait dans des algorithmes de type *dynamic backtracking* [9], *partial order dynamic backtracking* [8] et *general partial order backtracking* [2].

Ces algorithmes travaillent sur une affectation partielle des variables qui est réparée localement en utilisant les nogoods. Le mécanisme d'enregistrement utilisé ne nécessite qu'un espace polynomial. Par exemple, dans le cas de *dynamic backtracking* :

- Seuls sont conservés les nogoods pour lesquels toutes les contraintes sont dans le chemin courant exceptée au plus une.
- Le choix du voisin suivant à explorer est totalement déterministe (l'affectation choisie pour être défaite est la plus récente dans le nogood).

Dans *path-repair*, on peut par exemple utiliser une stratégie de gestion des nogoods similaire à celle utilisée dans *dynamic backtracking*, donnant un algorithme de recherche systématique *path-repair(dynamic backtracking)*.

En fait, ce type d'algorithmes *path-repair(dynamic backtracking)* est présenté sous une forme un peu différente pour les CSP dynamiques [12], et les CSP numériques [13].

4 Expérimentations sur des problèmes d'ordonnancement

Les problèmes classiques d'ordonnancement d'atelier pour lesquels un ensemble J de n *jobs*, constitués chacun de m opérations (ou tâches) doit être traité sur un ensemble M de l machines, peut être considéré comme un CSP sur les intervalles[5]. Un de ces problèmes classique s'appelle l'Open Shop. Pour ce problème, les opérations de chaque job peuvent être séquencées dans n'importe quel ordre, mais en tout état de cause une seule à la fois. Nous considérons ici la construction d'ordonnancement non préemptif de durée totale d'exécution minimal. Ce problème est NP difficile pour $min(n, m) \geqslant 3$.

Les contraintes sur les ressources (machines ET jobs) sont propagées à l'aide des *arbitrages triviaux* de [5]. Le degré de consistance atteint par cette technique ne permet pas toujours de fournir une solution réalisable. Une étape d'énumération est alors requise. Pour les problèmes d'ateliers, l'énumération est classiquement réalisée sur l'ordre relatif des tâches pour chacune des ressources. Quand toutes les précédences possibles sont installées, fixer la date de démarrage de tâche à leur date de démarrage au plus tôt fourni une solution réalisable. Une telle contrainte de précédence est donc bien une contrainte d'énumération au sens où nous l'avons définie précédemment.

Une des meilleures recherche systématique pour l'Open Shop est un algorithme de type branch and bound présenté dans [3]. Elle consiste en chaque nœud à ajouter des contraintes de précédence sur le chemin critique d'une solution heuristique. Bien que cette méthode soit une des meilleurs connues à ce jour, des problèmes de taille 7×7 restent non résolus.

Les techniques d'énumération utilisées sur l'Open Shop sont intéressantes pour *path-repair* car elles construisent dynamiquement des sous-problèmes indépendants (en ajoutant des contraintes de précédence). Nous pouvons donc supposer que *path-repair* saura tirer profit de cette situation.

Nous avons tout d'abord testé ds versions systématiques de *path-repair* sur l'Open Shop. Nous avons alors obtenus de très fortes améliorations en nombre de chemins explorés par rapport aux

5. Les variables sont les dates de démarrage des tâches. Les bornes représentent alors la date de démarrage au plus tôt et au plus tard.

résultats de [3]. De plus, un problème de taille 10×10 a été résolu pour la première fois parallèlement à une équipe allemande. Ces résultats ont été présentés dans [11].

Nous avons aussi réalisé des expérimentations avec une version *path-repair(tabou)*. Nous présentons ici les résultats obtenus sur une série de 30 problèmes due à Taillard [21]. Pour donner une idée de la qualité des résultats obtenus nous rappelons les résultats présentés dans [1] et [14]. Ces deux articles présentent des recherches taboues particulièrement évoluées et paramètrées avec attention qui ont été spécialement développées pour l'Open Shop. Les résultats présentés dans le tableau 1 montrent que notre approche très simple qui applique simplement les principes présentés donne des résultats particulièrement honorables.

Problème	Solution	PR [a]	Dist.	L	A
4x4-1	193	193	-	193	-
4x4-2	236	236	-	236	-
4x4-3	271	271	-	271	-
4x4-4	250	250	-	250	-
4x4-5	295	295	-	295	-
4x4-6	189	189	-	189	-
4x4-7	201	201	-	201	-
4x4-8	217	217	-	217	-
4x4-9	261	261	-	261	-
4x4-10	217	217	-	217	-
5x5-1	300	301	0.33 %	300	-
5x5-2	262	262	-	262	-
5x5-3	323	323	-	326	-
5x5-4	310	311	0.32 %	310	-
5x5-5	326	326	-	326	-
5x5-6	312	314	0.64 %	303	-
5x5-7	303	304	0.33 %	303	-
5x5-8	300	300	-	300	-
5x5-9	353	356	0.85 %	353	-
5x5-10	326	326	-	326	-
7x7-1	435	435	-	435	437
7x7-2	443	449	1.35 %	447	444
7x7-3	468	**473**	1.07 %	474	476
7x7-4	463	463	-	463	464
7x7-5	416	**416**	-	417	417
7x7-6	451	460	2.00 %	459	-
7x7-7	422	430	1.90 %	429	429
7x7-8	424	424	-	424	-
7x7-9	458	458	-	458	458
7x7-10	398	398	-	398	398

Tab. 1 –. *Résultats sur les problèmes de Taillard*

[a] **PR** : résultat obtenu avec Path Repair limité à 1500 mouvements sans amélioration, **Dist.** représente la distance à l'optimum pour cette solution. **L** : résultat obtenu par Liaw avec 50 000 mouvements sans amélioration et **A** : résultat obtenu par Alcaide *et al.* avec 100 000 mouvements sans amélioration. **-** : représente les valeurs non fournies ou non pertinentes (distance à l'optimum quand l'optimum est atteint).

5 Conclusion et perspectives

Dans cet article, nous avons présenté un nouvel algorithme de résolution pour les CSPs: l'algorithme *path-repair*. Il y a deux idées principales dans cet algorithme : la première est de se baser sur un algorithme de réparation (recherche locale) pour résoudre le problème, la deuxième est qu'il

convient de travailler à partir d'une instantiation partielle si on veut pouvoir utiliser efficacement des techniques de filtrage. Nous avons alors montré que l'outil privilégié pour la mise en œuvre de cet algorithme était l'utilisation des nogoods.

Des premières expérimentations avec des versions systématiques de *path-repair*, reposant sur la stratégie de gestion des nogoods de *dynamic backtracking*, comme avec des versions non systématiques utilisant une liste taboue, ont fourni des résultats prometteurs.

Références

1. David Alcaide, Joaquín Sicilia, and Daniele Vigo. A tabu search algorithm for the open shop problem. *TOP : Trabajos de Investigación Operativa*, 5(2):283–296, 1997.
2. C. Bliek. Generalizing partial order and dynamic backtracking. In *Proceedings of AAAI*, 1998.
3. P. Brucker, J. Hurink, B. Jurisch, and B. Westmann. A branch and bound algorithm for the open-shop problem. Technical report, Osnabrueck University, 12 1994.
4. Alberto Caprara, Matteo Fischetti, and Paolo Toth. A heuristic algorithm for the set covering problem. In *Integer Programming and Combinatorial Optimization*, Vancouver, June 1996.
5. Jacques Carlier and Éric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.
6. J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.
7. C. Gervet. Large combinatorial optimization problem methodology for hybrid models and solutions (invited talk). In *JFPLC*, 1998.
8. Matthew Ginsberg and David McAllester. GSAT and dynamic backtracking. In Alan Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*. Springer, May 1994. (PPCP'94: Second International Workshop, Orcas Island, Seattle, USA).
9. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
10. F. Glover and M. Laguna. *Modern heuristic Techniques for Combinatorial Problems, chapter Tabu Search, C. Reeves*. Blackwell Scientific Publishing, 1993.
11. Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: an application to open-shop problems. *European Journal of Operational Research*, page to appear, 1999.
12. Narendra Jussien. *Relaxation de Contraintes pour les problèmes dynamiques*. 1. thèse, Université de Rennes I, 24 October 1997.
13. Narendra Jussien and Olivier Lhomme. Dynamic domain splitting for numeric csp. In *European Conference on Artificial Intelligence*, pages 224–228, Brighton, United Kingdom, August 1998.
14. Ching-Fang Liaw. A tabu search algorithm for the open shop scheduling problem. *Computers and Operations Research*, 26, 1998.
15. S. Minton, M.D. Johnston, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–206, 1992.
16. G. Pesant and M. Gendreau. A view of local search in constraint programming. In *Proc. of the Principles and Practice of Constraint Programming*, pages 353–366. Springer-Verlag, 1996.
17. E. T. Richards and E. B. Richards. Non-systematic search and learning: An empirical study. In *Proc. of the the Conference on Principles and Practice of Constraint Programming*, Pisa, 1998.
18. Andrea Schaerf. Combining local search and look-ahead for scheduling and constraint satisfaction problems. In *Proc. of the 15th International Joint Conf. on Artificial Intelligence (IJCAI-96)*, pages 1254–1259, Nagoya, Japan, 1997. Morgan Kaufmann.
19. Bart Selman and Henry Kautz. Domain-independent extensions to gsat: Solving large structured satisfiability problems. In Ruzena Bajcsy, editor, *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 290–295, Chambery, France, August 1993. Morgan Kaufmann.
20. Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *AAAI 92, Tenth National Conference on Artificial Intelligence*, pages 440–446, 1992.
21. É. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operations Research*, 64:278–285, 1993.
22. G. Verfaillie and L. Lobjois. Problemes incohérents: expliquer l'incohérence, restaurer la cohérence. In *Actes des JNPC*, 1999.
23. M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of AAAI*, 1994.