Un propagateur basé sur les positions pour le problème d'Open-Shop

Jean-Noël Monette, Yves Deville, Pierre Dupont

Un propagateur basé sur les positions pour le problème d'Open-Shop.*

Jean-Noël Monette, Yves Deville et Pierre Dupont

Département d'Ingénierie Informatique
Université catholique de Louvain
{jmonette,yde,pdupont}@info.ucl.ac.be

Résumé

L'Open-Shop est un problème difficile qui peut être résolu par des méthodes de Programmation par Contraintes ou de Recherche Opérationnelle. Les techniques existantes réduisent efficacement l'arbre de recherche mais elles prennent rarement en compte l'ordre d'exécution des tâches. Dans ce travail, nous développons un nouveau propagateur pour le problème d'ordonnancement sans interruption sur une machine, la contrainte de base de l'Open-Shop. Ce propagateur prend l'ordre des tâches en compte ce qui permet dans de nombreux cas de réduire la taille de l'arbre de recherche. Sa complexité temporelle pour une machine est de $\mathcal{O}(N^2 \log N)$, où N est le nombre de tâches sur la machine. Les expériences menées sur le problème d'Open-Shop montrent que le nouveau propagateur permet de détecter de nouvelles valeurs inconsistantes lorsqu'il est ajouté aux techniques de l'état de l'art.

Abstract

The Open-Shop Problem is a hard problem that can be solved using Constraint Programming or Operation Research methods. Existing techniques are efficient at reducing the search tree but they usually do not consider the absolute ordering of the tasks. In this work, we develop a new propagator for the One-Machine Non-Preemptive Problem, the basic constraint for the Open-Shop Problem. This propagator takes this additional information into account allowing, in most cases, a reduction of the search tree. The underlying principle is to use shaving on the positions. Our propagator applies on one machine or one job and its time complexity is in $\mathcal{O}(N^2 \log N)$, where N is either the number of jobs or machines. Experiments on the Open-Shop Problem show that the propagator adds pruning to state-of-the-art constraint satisfaction techniques to solve this problem.

*Traduction française d'un article présenté à CPAIOR'07 [16].

1 Introduction

Le problème d'Open-Shop est un problème d'ordonnancement disjonctif de tâches connu pour être particulièrement difficile à résoudre. Il existe encore des problèmes comprenant moins de 50 tâches qui restent non résolus malgré le développement de techniques et d'algorithmes qui permettent de réduire efficacement la recherche.

Le problème d'Open-Shop vise à trouver l'ordre dans lequel un ensemble de tâches est exécuté pour réduire le plus possible le makespan, c'est-à-dire la durée totale d'exécution. Chaque tâche doit être exécutée sur une machine particulière pour une durée donnée et sans interruption. Une machine ne peut pas traiter deux tâches en même temps. De plus, les tâches font partie de travaux (ou jobs) et deux tâches du même job ne peuvent être traitées en même temps. Il n'y a aucun ordre prédéterminé parmi les tâches d'un même job ou d'une même machine (par opposition au Job-Shop).

La Programmation par Contraintes fonctionne très bien pour ce problème. Plusieurs propagateurs ont été développés pour supprimer le plus tôt possible les valeurs inconsistantes et réduire la taille de l'arbre de recherche. Les techniques les plus connues sont appelées Edge-Finding (EF) et Not-First-Not-Last (NFNL). L'Edge-Finding [1, 2, 3, 4] essaye de déterminer si une tâche doit venir avant ou après un ensemble d'autres tâches. Les meilleures implémentations d'EF ont une complexité temporelle en $\mathcal{O}(N \log N)$, où N est le nombre de tâches d'une machine ou d'un job. Not-First-Not-Last [5, 6, 7], dont il existe des implémentations en $\mathcal{O}(N \log N)$, teste si une tâche peut être la première ou la dernière d'une ensemble de tâches.

La singleton consistance (ou shaving) [5, 6, 8] est une technique orthogonale qui donne de bons résul-

tats pour l'Open-Shop. Elle consiste à assigner successivement à une variable les différentes valeurs de son domaine et tester si cela entraîne une inconsistance. Si tel est le cas, la valeur est retirée du domaine de la variable. Pour détecter l'inconsistance, toutes les contraintes peuvent être propagées jusqu'à atteindre le point fixe. Comme cette méthode est coûteuse, une propagation plus simple est souvent utilisée. Par exemple dans [6], seul l'Edge-Finding est utilisé pour détecter les inconsistances. Le shaving reste malgré tout coûteux à cause de la taille du domaine des variables. C'est pour cela que le shaving ne considère souvent que les bornes des domaines.

Cet article propose un nouveau propagateur pour le problème à une machine et sans interruption. Celui-ci exploite l'information contenue dans la position absolue des tâches. Cette idée a déjà été utilisée avec succès dans plusieurs travaux. Tout d'abord, [9] utilise des variables de position comme variables de permutation dans une contrainte de tri et propose une extension de l'Edge-Finding. Ensuite, [10] propose une façon de décider si une tâche peut être exécutée à une certaine position en fonction du nombre de tâches qui peuvent venir avant et après la tâche. En troisième lieu, [11] étend cette idée avec des bornes plus fines et présente un algorithme avec une complexité temporelle en $\mathcal{O}(N^3)$.

Cet article présente une autre façon d'utiliser la position des tâches en se basant sur le principe du shaving. Pour chaque position possible d'une tâche, des bornes inférieure et supérieure sur le moment de départ de la tâche sont calculées à partir de la durée et des moments de départ possibles des autres tâches du même job ou de la même machine. Le propagateur qui en résulte peut être appliqué sur toutes les tâches d'une même machine ou d'un même job avec une complexité temporelle en $\mathcal{O}(N^2 \log N)$ où N est le nombre de tâches considérées. Ce propagateur permet de supprimer des valeurs inconsistantes qui ne l'auraient pas été par EF et NFNL et d'améliorer d'environ 14% la détection d'états inconsistants sur un jeu de test standard [12].

La section suivante explique le problème de façon formelle ainsi que sa modélisation en Programmation par Contraintes. La section 3 présente le nouveau propagateur et la section 4 décrit les résultats expérimentaux qui montrent le potentiel de l'approche. La dernière section présente les conclusions et des directions d'approfondissement.

2 Le Problème à une machine sans interruption

En tant que problème d'optimisation, l'Open-Shop peut être résolu à l'aide du Branch-and-Bound. L'optimisation consiste à minimiser le makespan ou le moment où se termine la dernière tâche (en supposant le moment où commence la première tâche fixé à zéro). Le Branch-and-Bound consiste a résoudre successivement plusieurs versions de satisfaction du problème. Le problème de satisfaction consiste à déterminer s'il existe une solution dont le makespan est plus petit qu'une certaine valeur. Chaque fois qu'une solution est trouvée, une autre solution dont le makespan est plus petit est recherchée. Quand il n'existe plus de nouvelle solution, la dernière solution trouvée est optimale.

L'Open-Shop sous sa version de satisfaction peut être exprimé comme une conjonction de problèmes plus petits, appelés problèmes à une machine et sans interruption (1SI). Le but du 1SI est d'ordonner un ensemble de tâches sur une machine pour qu'une tâche à la fois soit exécutée et ce sans interruption. Pour chaque tâche sont définis sa durée et les limites inférieure et supérieure sur le moment où la tâche peut commencer. L'Open-Shop est défini avec un 1SI pour chaque machine et un autre pour chaque job. Les jobs et les machines ont en effet le même comportement : Deux tâches qui en font partie ne peuvent être traitées en même temps. Le 1SI est aussi à la base d'autres problèmes tels que le Job-Shop. Le propagateur que nous présentons ici peut aussi s'y appliquer.

Formellement, le 1SI est défini comme suit. T est l'ensemble de tâches à traiter et N est sa cardinalité. Pour chaque tâche $t \in T$, sa durée, sa limite inférieure et sa limite supérieure sur le moment de départ sont définis respectivement par $d(t)$, $est(t)$ (pour "earliest starting time") et $lst(t)$ (pour "latest starting time"). Le problème est de trouver une valeur $S(t)$ pour chaque tâche t entre $est(t)$ et $lst(t)$ et satisfaisant la contrainte qu'une tâche soit traitée à la fois : $\forall t_1, t_2 \in T, S(t_1)+d(t_1) \leq S(t_2) \vee S(t_2)+d(t_2) \leq S(t_1)$.

2.1 Modèle en Programmation par Contraintes

Notre modélisation du 1SI utilise plusieurs variables pour chaque tâche $t \in T$. La variable entière $S(t)$ représente le moment auquel la tâche t commence. Son domaine initial est défini par $[est(t), lst(t)]$. La variable ensemble $B(t)$, dont le domaine est initialement $[\emptyset, \{u | u \in T, u \neq t\}]$, correspond à l'ensemble des tâches qui viennent avant t. Les symboles $\overline{B}(t)$ et $\underline{B}(t)$ représentent les bornes supérieure et inférieure de la variable $B(t)$. Une troisième variable $P(t)$ représente la position absolue de la tâche t dans l'ordre d'exécution. Son domaine va de 0 à $N-1$. Les positions

absolue et relative d'une tâche sont liées par le fait que $P(t)$ est égal à la cardinalité de $B(t)$.

Les variables $S(t)$ et $B(t)$ sont communes dans la modélisation des problèmes disjonctifs. L'utilisation d'une variable qui représente la position absolue d'une tâche a été introduite dans [9] où l'auteur y résout le problème du Job-Shop en fixant des permutations sur l'ordre des tâches. Dans sa formulation, des variables sont définies pour le temps de départ de chaque tâche, pour le temps de départ de la tâche à chaque position et pour la position de chaque tâche. Les trois ensembles de variables sont liés par une contrainte de tri. Dans notre approche, nous avons choisi de ne pas utiliser les variables qui représentent le moment de départ des tâches par position.

2.2 Contraintes

L'ensemble de contraintes du 1SI peut être exprimée de façon équivalente sur les trois ensembles de variables. On a ainsi trois ensembles équivalents de contraintes qui établissent que deux tâches ne peuvent être exécutées au même moment.

1. $\forall t_1, t_2 \in T,$
   $(S(t_1) + d(t_1) \leq S(t_2)) \vee (S(t_2) + d(t_2) \leq S(t_1))$
2. $\forall t_1, t_2 \in T, \quad (t_1 \in B(t_2)) \quad \vee \quad (t_2 \in B(t_1))$
3. $\forall t_1, t_2 \in T, (P(t_1) < P(t_2)) \quad \vee \quad (P(t_2) < P(t_1))$

Notre modèle va utiliser les trois ensembles de façon à accélérer la propagation. D'autres contraintes font le pont entre les trois ensembles de variables. Tout d'abord, la position d'une tâche t est égale au nombre de tâches qui viennent avant t ($|B(t)| = P(t)$). De plus, une tâche t_1 finit avant qu'une autre tâche t_2 ne commence si et seulement si sa position est plus petite que celle de t_2 ($S(t_1) + d(t_1) \leq S(t_2) \Leftrightarrow t_1 \in B(t_2) \Leftrightarrow P(t_1) < P(t_2)$).

En plus des contraintes de base, il est possible de définir des contraintes redondantes. Ainsi, si t_1 est avant t_2, toutes les tâches qui sont avant t_1 sont aussi avant t_2 ($t_1 \in B(t_2) \Leftrightarrow B(t_1) \subset B(t_2)$). Une contrainte de différence est aussi définie sur les variables de position vu qu'il n'est pas possible d'avoir deux tâches à la même position ($alldiff(\{P(t) : t \in T\})$).

Cette dernière contrainte est un premier exemple de contrainte globale. Les contraintes globales considèrent plus que deux tâches à la fois pour permettre plus de réduction des domaines. C'est le cas de EF et NFNL. Cependant ces dernières contraintes ne prennent pas en compte la position des tâches. Ce travail montre comment utiliser cette information supplémentaire.

3 Le Propagateur

L'idée générale du nouveau propagateur est de faire du shaving sur les variables de position. D'ordinaire, le shaving est appliqué sur les variables de départ et seulement sur leurs extrémités à cause de la taille de leurs domaines. Le domaine des variables de position contient au contraire peu de valeurs et peut être réduit en un temps raisonnable. Pour tester si une tâche peut être exécutée à une position particulière, le propagateur calcule des bornes sur les limites inférieure et supérieure du moment où la tâche peut commencer en cette position. Si l'intervalle obtenu est disjoint du domaine de $S(t)$, la tâche ne peut pas être placée à cette position. De plus, cette technique permet de réduire aussi le domaine de $S(t)$ à l'union des intervalles pour toutes les positions possibles. La section 3.1 présente la façon dont les domaines sont réduits à partir des bornes calculées sur le moment de départ. Ensuite la section 3.2 explique quelles sont les bornes utilisées et comment les calculer. Notre approche du shaving est entièrement locale à un propagateur.

Commençons par introduire quelques notations supplémentaires. Tout comme $est(t)$ est la limite inférieure sur le moment auquel la tâche peut commencer, nous allons écrire $ect(t)$ ("earliest completion time") pour la limite inférieure du moment auquel la tâche peut finir. Ces deux valeurs sont reliées par l'équation $ect(t) = est(t) + d(t)$. Ces valeurs peuvent aussi être définies pour des ensemble de tâches. Si U est un sous-ensemble non vide de T, $d(U)$ est la somme des durées des tâches qui font partie de U et $est(U)$ est le moment le plus tôt auquel une des tâches de U peut commencer ($est(U) = \min_{t \in U} est(t)$). La quantité duale, $ect(U)$, est le moment le plus tôt auquel toutes les tâches de U sont terminées. Cette dernière valeur ne peut être calculée facilement mais il existe différentes bornes inférieures. En particulier, ce travail utilise $b\_ect(U)$, le maximum parmi tous les sous-ensembles U' de U de la somme de la limite inférieure du moment de départ de U' et de la durée de U'. Il ne s'agit que d'une borne car la limite supérieure pour le moment auquel les tâches peuvent démarrer n'est pas prise en compte.

$$b\_ect(U) = \max_{\emptyset \neq U' \subseteq U} (est(U') + d(U')) \qquad (1)$$

3.1 Le shaving sur les variables de position

Le shaving énumère toutes les valeurs possibles pour $P(t)$. En supposant que $P(t)$ prenne une valeur p, le moment de départ de t appartient à l'intervalle $[est(t,p), lst(t,p)]$ où $est(t,p)$ et $lst(t,p)$ sont respectivement la limite inférieure et la limite supérieure du moment de départ de la tâche t lorsqu'elle se trouve

en position p.

La valeur $est(t,p)$ est liée à $ect(B(t),p)$ qui est le premier moment où au moins p tâches parmi celles dans $\overline{B}(t)$ sont terminées et parmi elles toutes les tâches dans $\underline{B}(t)$. En effet, en position p, la tâche t ne peut pas démarrer avant que p tâches parmi celles qui peuvent venir avant t n'aient été exécutées. De plus, t ne peut démarrer avant que toutes les tâches qui doivent venir avant soient finies. Cela donne la relation suivante :

$$est(t,p) = \max\left(ect(B(t),p), est(t)\right).$$

Dans cette formule, $ect(B(t),p)$ ne peut être calculé exactement avec une complexité raisonnable. Nous proposons donc de calculer une borne inférieure aussi proche que possible. La section suivante (Section 3.2) détaille le calcul de cette borne. Un raisonnement similaire, non détaillé ici, peut être fait pour trouver une borne supérieure pour $lst(t,p)$.

Lorsque les intervalles $[est(t,p), lst(t,p)]$ sont calculés pour tout $p \in P(t)$, les domaines de $P(t)$ et $S(t)$ peuvent être réduits avec deux règles simples :

$$\begin{aligned} &\forall p \in dom(P(t)): \qquad (2)\\ &([est(t,p), lst(t,p)] \cap dom(S(t)) = \emptyset) \Rightarrow P(t) \neq p \end{aligned}$$

$$\begin{aligned} &dom(S(t)) := \qquad (3)\\ &dom(S(t)) \cap (\cup_{p \in dom(P(t))}[est(t,p), lst(t,p)]) \end{aligned}$$

La première règle retire du domaine de $P(t)$ les valeurs p pour lesquelles il n'y a pas de moment de départ valide pour t, c'est-à-dire lorsque l'intervalle calculé est vide ou a une intersection vide avec le domaine de $S(t)$. La deuxième règle inclut le domaine de $S(t)$ dans l'union des intervalles qui ont été calculés. Alternativement, la règle (3) peut être remplacée par la règle suivante qui ne réduit que les bornes du domaine de $S(t)$ afin de le garder comme un seul intervalle. Cette pratique est standard en ordonnancement.

$$\begin{aligned} &dom(S(t)) := \qquad (4)\\ &[\min_{p \in dom(P(t))}(est(t,p)), \max_{p \in dom(P(t))}(lst(t,p))] \end{aligned}$$

Les expériences menées à la section 4 vont considérer les deux versions de la réduction de $S(t)$. Les réductions de $S(t)$ (avec la règle (4)) et de $P(t)$ peuvent être effectués avec une complexité temporelle en $\mathcal{O}(N)$ où N est le nombre de tâches à exécuter, une borne supérieure sur la taille du domaine de $P(t)$.

3.2 Une borne inférieure pour la fin d'un sous-ensemble de tâches

Cette section présente une approximation de $ect(B(t),p)$ utilisée pour calculer $est(t,p)$. L'algorithme pour $lst(t,p)$ est similaire mais non présentée. Pour calculer une borne inférieure de $ect(B(t),p)$, nous calculons le minimum du plus tôt moment de terminaison de tous les ensembles U de cardinalité p qui sont sous-ensembles de $\overline{B}(t)$ et dont $\underline{B}(t)$ est un sous-ensemble. Par la suite, $b\_ect(B(t),p)$ représente cette borne inférieure de $ect(B(t),p)$. Il s'agit bien d'une borne inférieure car elle dépend de $b\_ect(U)$ qui est elle-même une borne inférieure.

$$\begin{aligned} &b\_ect(B(t),p) = \min_{U}\left(b\_ect(U)\right) \qquad (5)\\ &\text{where } |U| \geq p \text{ and } \underline{B}(t) \subseteq U \subseteq \overline{B}(t) \end{aligned}$$

Cette borne peut être calculée en utilisant un ensemble de règles similaire au Jackson Preemptive Schedule [13] pour calculer le moment le plus tôt de fin d'exécution d'un ensemble de tâches où les interruptions sont acceptées. Notre algorithme autorise aussi l'interruption des tâches mais ne prend pas en compte la limite supérieure sur le moment de départ des tâches. En revanche, la durée des tâches est considérée pour planifier un sous-ensemble de tâches de taille donnée le plus tôt possible. Les règles à appliquer sont les suivantes :

- Quand une tâche t est disponible et que la machine est libre, commencer l'exécution de t.
- Quand une tâche t_1 devient disponible pendant qu'une autre tâche t_2 est effectuée et que la durée restante de t_1 est moindre que la durée restante de t_2, arrêter t_2 et démarrer t_1.
- Quand une tâche $t_1 \in \underline{B}(t)$ devient disponible pendant l'exécution d'une autre tâche $t_2 \notin \underline{B}(t)$, arrêter t_2 et démarrer t_1.

La valeur de $b\_ect(B(t),p)$ est obtenue lorsque toutes les tâches dans $\underline{B}(t)$ ont été exécutées et qu'au moins p tâches ont été exécutées.

Bien que l'algorithme suppose que les tâches puissent être interrompues, le résultat correspond exactement aux valeurs données par l'équation (5) où l'interruption n'est pas autorisée. En effet, il est possible de regrouper les différentes parties des tâches complétées en les réordonnant selon leur moment de départ. Il en résulte donc un ordonnancement non préemptif de l'ensemble de tâches. La préemption n'est pas utilisée comme une relaxation du problème mais comme un simple moyen de faciliter les calculs. La valeur calculée $b\_ect(B(t),p)$ est quand même bien une borne sur la valeur exacte vu que la limite supérieure sur le temps de départ des tâches ($est(t)$) n'est pas pris en compte.

Par ailleurs, un seul passage de l'algorithme donne la valeur $b\_ect(B(t), p)$ pour tous les p. En effet, il suffit de retenir les moments successifs auxquels une tâche se finit pour avoir les $b\_ect(B(t), p)$ pour les valeurs successives de p.

L'Algorithme 1 présente un pseudo-code de l'algorithme utilisé. Celui-ci utilise deux files de priorité. La première (Q1) trie les tâches par ordre croissant de limite inférieure sur le moment de départ. Cela permet d'insérer dans la seconde file de priorité (Q2) uniquement les tâches qui sont disponibles à un moment donné (lignes 9 à 15). Q2 trie les tâches en ordre croissant de durée d'exécution restante. Quand une tâche est sortie de Q2, deux situations sont possibles. Soit la tâche peut être exécutée entièrement avant qu'une autre tâche ne soit disponible et le moment auquel elle fini est enregistré (lignes 16 à 23). Soit la tâche doit être interrompue pour vérifier si une tâche nouvellement disponible ne peux pas se finir plus tôt (lignes 25 à 28).

Par soucis de simplicité, l'algorithme présenté est une version raccourcie qui ne prend pas en compte que certaines tâches appartiennent à $\underline{B}(t)$. Pour prendre cela en compte, il suffit d'utiliser une pénalité dans la seconde file de priorité pour obliger les tâches de $\underline{B}(t)$ à être choisies en premier lieu. Alternativement, deux files de priorité peuvent être utilisées en parallèle. Celle qui contient les tâches de $\underline{B}(t)$ est vidée en premier. De plus, un compteur doit être ajouté pour retenir quand toutes les tâches obligatoires sont finies.

La complexité temporelle de l'algorithme est en $\mathcal{O}(n \log n)$ avec $n = |\overline{B}(t)|$ qui dans le pire des cas est égal à $N - 1$ (N est le nombre de tâches à exécuter). En effet, les opérations put() et pop() des files de priorité peuvent être implémentées en $\mathcal{O}(\log n)$. Il y a exactement n tâches qui sont insérées dans Q1 (lignes 5 à 7) et au plus $2n$ tâches qui sont insérées dans Q2, vu qu'il y a exactement n tâches sorties de Q1 (lignes 9 à 15) et au plus n réinsertions de tâche pour cause d'interruption (lignes 24 à 28).

Exemple 1 *Pour illustrer le calcul de $b\_ect(B(t), p)$, supposons les tâches suivantes :*

- *t_0 est la tâche pour laquelle les valeurs sont calculées ; $dom(B(t_0)) = [\{t_4\}, \{t_1, t_2, t_3, t_4\}]$ et $dom(P(t_0)) = [1, 4]$*
- *t_1 avec $est(t_1) = 0$ et $d(t_1) = 5$.*
- *t_2 avec $est(t_2) = 1$ et $d(t_2) = 3$.*
- *t_3 avec $est(t_3) = 2$ et $d(t_3) = 1$.*
- *t_4 avec $est(t_4) = 3$ et $d(t_4) = 3$.*

Dans l'ordre chronologique, t_1 est exécutée en premier, en commençant à l'instant 0. Au temps 1, t_2 est disponible et comme elle a une durée ($d(t_2) = 2$) plus courte que la durée restante pour t_1 ($5 - 1 = 4$), t_1 est arrêtée et t_2 est démarrée. Au temps 2, t_2 est

Algorithme 1 : Algorithme Simplifié pour Calculer $b\_ect(B(t), p)$

Input : B : l'ensemble de tâches
D : vecteur de la durée des tâches
EST : vecteur du *est* des tâches
Output : ECT : vecteur de $b\_ect(B(t), p)$ pour chaque position p

```
1  Q1 := new PriorityQueue()
2  Q2 := new PriorityQueue()
3  time := 0
4  p := 0
5  forall t ∈ B do
6  |  RD(t) := D(t) //RD est le temps d'exécution
   |  restant
7  |  Q1.put(t,EST(t))
8  while not Q1.empty() do
9  |  t := Q1.pop()
10 |  time := EST(t)
11 |  Q2.put(t,RD(t))
12 |  while not Q1.empty() and
13 |  EST(Q1.top()) = time do
14 |  |  t := Q1.pop()
15 |  |  Q2.put(t,RD(t))
16 |  while not Q2.empty() and
17 |  (Q1.empty() or
18 |  time + RD(Q2.top()) < EST(Q1.top()) ) do
19 |  |  t := Q2.pop()
20 |  |  time := time +RD(t)
21 |  |  RD(t) := 0
22 |  |  p := p+1
23 |  |  ECT(p) := time
24 |  if not Q2.empty() then
25 |  |  t := Q2.pop()
26 |  |  RD(t) := RD(t) + time - EST(Q1.top())
27 |  |  Q2.push(t,RD(t))
28 |  |  time := EST(Q1.top())
29 |
30 return ECT
```

interrompue pour laisser exécuter t_3 dont la durée est plus courte que le temps d'exécution restant de t_2 ($3-1=2>1$). Au temps 3, t_3 est entièrement exécutée. Les tâches t_1, t_2 et t_4 sont disponibles mais t_4 est choisie car c'est la seule tâche obligatoire. En effet, par définition de $dom(B(t))$, t_4 est l'unique tâche qui doit être exécutée avant t_0. Cette tâche est exécutée pendant 3 unités de temps. Lorsqu'elle est finie, t_2 est exécutée avant t_1 car sa durée d'exécution restante est moindre. Après deux nouvelles unités de temps, t_2 est terminée et t_1 est exécuté jusqu'au temps 12. Le tableau ci-dessous donne les instants auxquels les tâches sont exécutés de façon préemptive.

Temps	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Tâches	t_1	t_2	t_3	t_4			t_2		t_1				

En enregistrant les moments auxquels les tâches se terminent, l'algorithme obtient les valeurs suivantes :

- $b\_ect(B(t_0),1) = b\_ect(B(t_0),2) = 6$. *En effet, la tâche obligatoire (t_4) ne s'est terminé qu'en second lieu.*
- $b\_ect(B(t_0),3) = 8$
- $b\_ect(B(t_0),4) = 12$

Bien que le calcul interrompe plusieurs tâches, les bornes obtenues correspondent bien à un ordonnancement non préemptif (comme attendu par l'équation (5)). Le tableau suivant montre comment les tâches peuvent être réordonnées pour chacune des positions.

Temps	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
$p=1$					t_4								
$p=2$			t_3		t_4								
$p=3$		t_2			t_3		t_4						
$p=4$	t_1						t_2		t_3		t_4		

Ainsi, lorsque les 4 tâches sont exécutées, il est possible d'exécuter t_1 de 0 à 5, moment auquel t_2 est exécutée jusqu'au temps 8. A ce moment, t_3 est démarrée pour une unité de temps et t_4 est ensuite exécutée jusqu'au temps 12 qui correspond à la valeur obtenue par l'algorithme.

Exemple 2 *La figure 1 présente un petit exemple où le nouveau propagateur permet de retirer des valeurs inconsistantes. Il y a 5 tâches à exécuter et leurs durées d'exécution et leurs domaines de départ sont les suivants.*

- $d(t_1) = 3$ *et* $dom(S(t_1)) = [8,17]$
- $d(t_2) = 5$ *et* $dom(S(t_2)) = [0,15]$
- $d(t_3) = 4$ *et* $dom(S(t_3)) = [5,16]$
- $d(t_4) = 4$ *et* $dom(S(t_4)) = [1,16]$
- $d(t_5) = 2$ *et* $dom(S(t_5)) = [7,18]$

L'application de NFNL et EF sur cet ensemble de tâches ne réduit aucun domaine. Par contre, notre propagateur permet de retirer la valeur 8 du domaine de $S(t_1)$. En utilisant l'algorithme pour calculer les limites inférieure et supérieure sur le moment de départ de t_1 dans chaque position, les valeurs obtenues sont :

- $est(t_1,0) = 8$ *et* $lst(t_1,0) = 2$

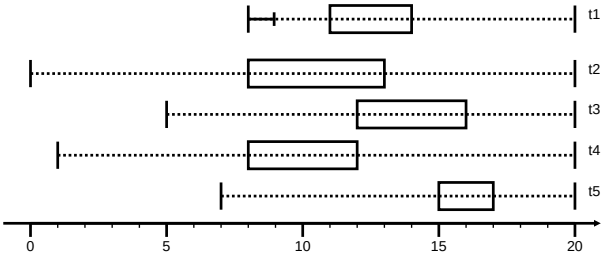

FIG. 1 – Exemple de réduction, voir l'exemple 2 pour les détails

- $est(t_1,1) = 8$ *et* $lst(t_1,1) = 7$
- $est(t_1,2) = 9$ *et* $lst(t_1,2) = 11$
- $est(t_1,3) = 11$ *et* $lst(t_1,3) = 15$
- $est(t_1,4) = 15$ *et* $lst(t_1,4) = 17$

De ces valeurs, il est possible de déduire que t_1 ne peut pas être exécuté en position 0 ou 1. Le domaine de son temps de départ peut donc être réduit à l'union des intervalles définis par les positions 2, 3 et 4, ce qui donne $dom(S(t_1)) = [9,17]$ où la valeur 8 a été retirée.

Le calcule de $est(t,p)$ et $lst(t,p)$ pour chaque $p \in dom(P(t))$ est effectué en $\mathcal{O}(N \log N)$ avec N le nombre de tâches et la réduction des domaines peut être faite en $\mathcal{O}(N)$. La complexité temporelle de l'ensemble de l'algorithme de filtrage pour une tâche t est donc en $\mathcal{O}(N \log N)$. Cela donne une complexité totale de $\mathcal{O}(N^2 \log N)$ pour un passage de l'algorithme de filtrage pour l'ensemble des tâches. En comparaison, les techniques NFNL et EF peuvent toutes deux être implémentées avec une complexité temporelle en $\mathcal{O}(N \log N)$.

4 Expériences

Nous avons implémenté le nouveau propagateur dans l'environnement de contraintes Gecode [14] dans deux versions. La première, appelée PS (pour "Position Shaving"), peut retirer des valeurs à l'intérieur des domaines des variables de départ, tandis que la seconde, PSB (pour "Position Shaving with Bound reduction"), est limitée à la réduction des bornes de ces variables. Nous avons aussi implémenté les techniques NFNL et EF selon les algorithmes décrits dans [15]. Il faut noter que ces implémentations ont une complexité temporelle en $\mathcal{O}(N^2)$ mais qu'elles utilisent des structures de données beaucoup plus simples que les algorithmes théoriquement plus efficaces décrits dans [3] et [7]. Finalement, nous avons modélisé le problème d'Open-Shop comme décrit dans la première section et en utilisant les propagateurs NFNL, EF, PS et PSB et la contrainte redondante AllDiff. PS et PSB ne sont jamais utilisés ensemble vu qu'il s'agit de deux versions

du même propagateur. En ce qui concerne le branchement, nous avons appliqué une heuristique simple qui utilise les variables de position. L'heuristique ordonne les tâches dans les machines avant de le faire dans les jobs. Parmi les tâches dont la position n'est pas fixée, la tâche pour laquelle il reste le moins de positions est choisie. En cas d'égalité, la tâche la plus courte est sélectionnée. L'heuristique de valeur choisit la plus petite valeur du domaine de la variable de position.

Nos tests ont été exécutés sur les instances de Guéret et Prins [12]. Il s'agit de 80 problèmes carrés (le nombre de jobs est égal au nombre de machines). Il y a 10 instances pour chaque taille entre 3x3 et 10x10 tâches. Les tests ont été exécutés sur un Intel Xeon 3 Ghz avec 512 KB de cache.

La première expérience consiste à observer le temps d'exécution total et la taille de l'arbre de recherche pour résoudre chaque instance en utilisant différentes combinaisons de propagateurs. Le temps d'exécution a été limité à une heure par instance. Les résultats sont présentés dans les Tableaux 1 et 2. Le Tableau 1 donne le nombre d'instances résolues et le nombre moyen de noeuds dans l'arbre de recherche. Cette moyenne est calculée sur les instances qui sont résolues en commun lorsque les instances résolues diffèrent (seulement pour la taille 7x7). Dans le Tableau 2, le même schéma est utilisé mais le temps moyen d'exécution (en secondes) remplace la taille de l'arbre de recherche.

Dans chacun des deux tableaux, les colonnes 2 et 3 présentent les résultats quand PS est utilisé seul (sans EF, ni NFNL). Les colonnes 4 et 5 donnent les résultats quand PSB est utilisé seul. Dans la troisième configuration (colonnes 6 et 7), NFNL et EF sont utilisés sans PS et PSB. Dans les colonnes 8-9 et 10-11, NFNL et EF sont utilisés respectivement en compagnie de PS ou de PSB.

Quand NFNL et EF sont utilisés, le même nombre d'instances est résolu avec ou sans le nouveau propagateur. Par contre, les instances résolues ne sont pas toujours les mêmes. Des deux premières configurations, il est possible de conclure que le nouveau propagateur n'est pas capable de résoudre des problèmes difficiles seul. En conjonction avec NFNL et EF, le Tableau 1 montre que PS et PSB sont capables de réduire l'arbre de recherche, parfois drastiquement, comme c'est le cas pour l'unique instance résolue de taille 8x8. Pour la taille 6x6, la taille moyenne augmente lorsque PS ou PSB sont utilisés. En détaillant les résultats pour chaque instance, il apparaît que seule l'instance GP06-01 a un plus grand arbre de recherche. Pour cette instance, l'arbre est dix fois plus grand avec PS ou PSB alors qu'il est en moyenne 30% plus petit pour les neufs autres instances de taille 6x6.

Si on considère le temps d'exécution, le Tableau 2 montre qu'il est en moyenne plus grand avec PS ou PSB que sans, à l'exception de l'instance de taille 8x8 pour laquelle le temps est entre 2 et 3 fois plus petit (pour un arbre de recherche 9 fois plus petit).

Il faut noter que les temps reportés sont bien plus longs que ceux présentés dans [15] parce que nous n'avons pas utilisé un environnement dédié au scheduling mais un moteur de contraintes à portée générale. Nous pensons qu'implémenter notre nouveau propagateur dans un environnement dédié serait bénéfique.

L'expérience suivante (Tableau 3) compare le temps moyen pour atteindre le point fixe lorsque NFNL, EF et PS(B) sont utilisés avec le temps moyen lorsque PS(B) n'est pas utilisé. Cette comparaison est effectuée le long de l'arbre de recherche obtenu lorsque tous les propagateurs sont actifs avec un nombre maximum de backtrack égal à 300000. Pour chaque instance, les temps d'exécution reportés pour atteindre les points fixes sont les moyennes sur tous les noeuds de l'arbre de recherche.

Au même moment, le filtrage est aussi comparé. Comme pour le temps, ce filtrage est calculé le long de l'arbre de recherche obtenu lorsque tous les propagateurs sont utilisés. Les nombres d'états reconnus inconsistants avec et sans PS(B) sont comptés. Le réduction des domaines ajoutée par l'utilisation de PS(B) est aussi comptée pour chaque type de variables ($S(t)$, $B(t)$ et $P(t)$) et ces valeurs sont sommées le long de tout l'arbre de recherche. La réduction est calculée comme la différence entre la taille des domaines dans l'état initial d'un noeud de l'arbre et leurs tailles après l'étape de propagation dans le même noeud. Si une inconsistance est détectée, le noeud n'est pas pris en compte pour la réduction des domaines.

Le Tableau 3 présente les résultats moyens pour chaque taille. Les trois premières paires de colonnes présentent le filtrage supplémentaire des variables $S(t)$, $B(t)$ et $P(t)$. Les deux colonnes suivantes montrent le nombre d'inconsistances détectées en plus et les deux dernières colonnes rapportent le temps supplémentaire passé pour atteindre ces améliorations. Deux cellules sont vides car le temps de calcul était trop court pour être compté avec assez de précision.

Les résultats de PS et PSB sont assez similaires, à l'exception des colonnes des variables de temps de départ, vu que PS permet de retirer des valeurs à l'intérieur des domaines des $S(t)$ alors que PSB ne peut pas. Cependant, cette différence n'influe pas sur les autres variables ou la détection d'inconsistances vu qu'il n'y a pas d'autre propagateur qui considère les valeurs retirées au milieu des domaines. Concernant l'augmentation du temps d'exécution pour atteindre le point fixe, il est plus petit pour PSB car moins de valeurs sont retirées par PSB. En prenant en compte le

Tab. 1 – Nombre d'instances résolues et taille moyenne de l'arbre de recherche

	PS		PSB		NFNL+EF		PS+NFNL+EF		PSB+NFNL+EF	
Taille	Résolus	Noeuds	Résolus	Noeuds	Résolus	Noeuds	Résolus	Noeuds	Résolus	Noeuds
3x3	10	39	10	38	10	38	10	39	10	38
4x4	10	128	10	127	10	134	10	127	10	126
5x5	10	451	10	456	10	371	10	369	10	373
6x6	10	3483	10	3896	10	2612	10	3402	10	3816
7x7	3	-	3	-	7	280914	8	208571	8	208582
8x8	0	-	0	-	1	120156	1	12953	1	12929
9x9	0	-	0	-	1	747146	0	-	0	-
10x10	0	-	0	-	0	-	0	-	0	-
Total	43		43		49		49		49	

Tab. 2 – Nombre d'instances résolues et temps d'exécution moyen en secondes

	PS		PSB		NFNL+EF		PS+NFNL+EF		PSB+NFNL+EF	
Taille	Résolus	Temps	Résolus	Temps	Résolus	Temps	Résolus	Temps	Résolus	Temps
3x3	10	0.008	10	0.008	10	0.006	10	0.01	10	0.008
4x4	10	0.075	10	0.047	10	0.054	10	0.069	10	0.068
5x5	10	0.38	10	0.32	10	0.19	10	0.36	10	0.32
6x6	10	3.9	10	3.5	10	1.9	10	4.3	10	3.9
7x7	3	-	3	-	7	338	8	496	8	432
8x8	0	-	0	-	1	106	1	43	1	37
9x9	0	-	0	-	1	1708	0	-	0	-
10x10	0	-	0	-	0	-	0	-	0	-
Total	43		43		49		49		49	

temps de calcul et le potentiel de filtrage, nous pouvons conclure que PSB est plus efficace que PS. Par ailleurs, il y a environ 14% d'état reconnus inconsistants en plus avec n'importe quelle version de notre propagateur. Et lorsque l'inconsistance n'est pas détectée, le domaine des variables est aussi substantiellement réduit.

En regardant l'évolution des résultats en fonction de la taille du problème, la réduction des domaines augmente jusqu'aux problèmes de taille 7x7 et décroît ensuite. Le temps utilisé suit le même schéma tandis que le nombre d'états inconsistants continue à augmenter. Étant donné qu'à partir de la taille 7x7 l'arbre de recherche n'est plus complet (car la recherche est coupée) et que la partie explorée diminue quand la taille augmente, nous pouvons supposer que le nouveau propagateur détecte plus d'inconsistances au début de la recherche mais réduit plus les domaines à la fin de la recherche. En observant les plus petites tailles, PS et PSB ne réduisent pas plus les arbres de recherche déjà fort petit de ces instances. C'est lorsque la taille (≥ 6) et la complexité augmentent que PS et PSB montrent leur utilité.

En conclusion, les expériences montrent que, bien que l'introduction de PS ou PSB n'augmente pas le nombre d'instances résolues, l'addition de ces propagateurs améliore le filtrage des noeuds de l'arbre de recherche et le nombre d'inconsistances découvertes.

5 Conclusion

Ce travail utilise la Programmation par Contraintes pour résoudre le problème d'Open-Shop. Il présente un nouveau propagateur, en deux versions, qui utilise la position absolue des tâches pour détecter des nouvelles inconsistances qui ne sont pas découvertes par les algorithmes standards que sont Not-First-Not-Last et Edge-Finding. Basé sur le principe du shaving, ce propagateur filtre les domaines des variables de temps de départ et les variables de position. Dans la première version, des trous peuvent être fait dans les domaines tandis que la seconde version se contente de réduire les bornes des domaines.

Des expériences sur un ensemble standard de problèmes montrent que le nouveau propagateur aide efficacement à réduire la taille des domaines et peut détecter jusqu'à 14% d'états inconsistents en plus mais

TAB. 3 – Filtrage additionnel et temps utilisé avec PS et PSB (en %)

	Red. S(t)		Red. B(t)		Red. P(t)		Inconsistance		Temps	
Taille	PS	PSB	PS	PSB	PS	PSB	PS	PSB	PS	PSB
3	7.6	1.5	0.3	0.3	5.7	3.6	0	0	-	-
4	13.6	5.6	7.0	7.4	14.2	12.7	2.1	2.1	184.0	165.7
5	14.1	5.6	4.8	4.9	11.2	9.8	0.7	0.8	192.1	182.2
6	27.3	14.9	9.0	9.2	15.6	14.1	8.0	8.2	241.3	181.0
7	108.7	58.3	21.3	21.8	42.5	42.7	13.9	14.2	333.2	325.3
8	116.9	34.9	17.4	16.7	30.1	26.1	13.3	13.9	281.1	254.0
9	78.9	25.4	13.9	13.2	20.5	18.0	37.7	36.3	291.5	272.0
10	64.6	17.9	9.9	10.3	20.9	19.5	37.4	37.3	155.5	196.5
Moyenne	54.0	20.5	10.4	10.5	20.1	18.3	14.1	14.1	239.8	225.3

à un coût plus élevé. La réduction des domaines n'est pas toujours refletée par une réduction de l'arbre de recherche. Si la recherche peut être jusqu'à 10 fois plus petite, dans la majorité des cas, la réduction est bien moins importante et dans quelques cas, l'arbre de recherche est même plus grand lorsque le nouveau propagateur est utilisé.

Une autre observation vient de la comparaison des deux versions du propagateur. Faire des trous dans les domaines des variables des temps de départ n'est pas récompensé par une réduction des domaines des autres variables ou de l'arbre de recherche. En effet, il n'y a pas d'autre propagateur ou contrainte qui fasse usage de cette information supplémentaire.

Pour conclure, nous affirmons que notre propagateur serait spécialement utile en conjonction avec d'autres contraintes qui prennent en compte la position des tâches ou les trous dans les domaines. Cela serait une bonne manière d'améliorer la résolution de problèmes difficiles d'ordonnancement de tâches. L'heuristique de branchement devrait aussi être adaptée pour éviter une augmentation de la taille de l'arbre de recherche lorsque le filtrage est renforcé.

Les possibilités de travail à venir comprennent la définition de bornes plus précises pour la limite inférieure sur le temps de terminaison d'un ensemble de tâches de taille fixée ainsi que la définition de meilleures heuristiques de recherche et d'autres propagateurs basés sur les positions absolues.

Remerciements

Cette recherche a été partiellement supportée par la Région Wallonne, projet TransMaze (516207).

Références

[1] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. Management Science, 35(2) :164-176, 1989.

[2] D. Applegate and B. Cook. A computational study of the job shop scheduling problem. ORSA J. Comput., 3(2) :149-156, 1991.

[3] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. European J. Oper. Res.,78 :146-161, 1994

[4] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. Proc. 11th Intl. Conf. on Logic Programming, 1994.

[5] J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. Ann. Oper. Res.,26 :269-287, 1990.

[6] U. Dorndorf, E. Pesch and T. Phan-Huy. Solving the open shop scheduling problem. J. Scheduling, 4 :157-174, 2001.

[7] P. Vilim. $\mathcal{O}(n \log n)$ filtering algorithms for unary resource constraint. Proc. CPAIOR 2004, LNCS 3011, 335-347, 2004.

[8] P. Martin and D. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. Proc. 5th Conf. Integer Programming and Combinatorial Optimization, 1996.

[9] Jianyang Zhou. A permutation-based approach for solving the job-shop problem. Constraints, 2 :185-213, 1997.

[10] W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with ILOG Scheduler. J. Heuristics, 3 :271-286, 1998.

[11] A. Wolf. Better propagation for non-preemptive single-ressource constraint problems. Proc. of CSCLP 2004, LNAI 3419, 201-215, 2005.

[12] C. Guéret and C. Prins. A new lower bound for the open-shop problem. Annals of Operation Research, 92 :165-183, 1999.

[13] J. R. Jackson. An extension of Johnson's results on job lot scheduling. Naval Research Logistics Quarterly, 3 :201-203, 1956.

[14] http ://www.gecode.org

[15] P. Baptiste, C. Le Pape and W. Nuijten. Constraint-based scheduling. Kluwer Academics Publisher, 2001.

[16] J.-N. Monette, Y. Deville and P.Dupont. A Position-Based Propagator for the Open-Shop Problem. Proc. CPAIOR 2007, LNCS 4510, 186-199, 2007.