

Utilisation du backtrack intelligent dans un branch-and-bound Application au problème d'Open-Shop

Narendra Jussien et Christelle Guéret

École des Mines de Nantes

4 rue Alfred Kastler – BP 20722

F-44300 Nantes Cedex 03 – France

E-mail: {Narendra.Jussien, Christelle.Gueret}@emn.fr

Résumé :

La plupart des recherches arborescentes utilisées en recherche opérationnelle pour résoudre les problèmes d'atelier sont des procédures de séparation-évaluation en profondeur d'abord qui utilisent un backtrack chronologique qui présente quelques inconvénients comme le thrashing. Ces recherches arborescentes énumèrent non pas sur des variables, comme il est courant dans la communauté Programmation par contraintes, mais sur des contraintes et dans le cadre d'une minimisation. Nous proposons dans cet article une technique de backtrack intelligent adapté à ces recherches arborescentes. Les résultats obtenus sur des problèmes d'Open-Shop montrent que notre technique est efficace puisqu'elle a permis de résoudre un Open-Shop ouvert à 10 jobs et 10 machines.

Mots-clés : Programmation par Contraintes, Recherche Opérationnelle, Backtrack intelligent.

1 Introduction

Les problèmes d'ordonnancement d'atelier sont très étudiés en Recherche Opérationnelle. Ces problèmes consistent à ordonnancer n travaux (ou jobs) décomposés en m opérations (tâches) devant être exécutées sur m machines distinctes. Les tâches sont liées entre elles par des contraintes conjonctives (précédences) et des contraintes disjonctives (qui expriment le fait que deux tâches utilisant la même ressource ne peuvent se chevaucher). Le but du problème est généralement de déterminer un ordonnancement de durée totale (*makespan*) minimale.

Il existe trois types de problèmes d'atelier, selon la nature des contraintes liant les tâches d'un même job : lorsque l'ordre de passage de chaque job est fixé et est le

même pour tous les jobs, on parle de *Flow-Shop*. Si cet ordre varie d'un job à l'autre, il s'agit alors d'un *Job-Shop*. Enfin si le séquençement des tâches des jobs n'est pas imposé, le problème est appelé *Open-Shop*.

Pour résoudre optimalement ces problèmes, on utilise en général des procédures de séparation et évaluation (branch-and-bound) basées sur une technique d'exploration systématique de l'espace de recherche en profondeur d'abord et utilisant le retour-arrière. Les schémas de séparation généralement utilisés dans ces Branch-and-Bound se basent sur l'*arbitrage* de disjonctions¹. De nombreux travaux ont été publiés sur le calcul de bonnes bornes inférieure et supérieure pour ces problèmes. De plus, dans une telle recherche, après chaque arbitrage, une *propagation* peut être réalisée par l'intermédiaire des *arbitrages triviaux* [CP 89] (initialement développés pour le Job-Shop mais généralisables à tous les problèmes d'atelier).

Cependant, dans ce cadre, peu d'alternatives à une exploration systématique par retour-arrière ont été envisagées. Par contre, dans la communauté *programmation par contraintes*, de nombreux travaux ont été proposés pour améliorer les recherches arborescentes (par des techniques dites de *backtrack intelligent* [PRO 93, VS 95, DEC 90]) voire pour remplacer le retour-arrière par un processus moins destructif (l'algorithme *Dynamic Backtracking* [GIN 93], le système DECORUM [JB 97a, JB 97b]).

Nous proposons dans cet article une amélioration des techniques de recherche de solution optimale pour l'Open-Shop, basée sur la procédure d'exploration. Nous définissons un mécanisme de backtrack intelligent adapté à une recherche énumérant des contraintes (les disjonctions) dans le cadre d'une minimisation (du *makespan*). Nous présentons nos résultats pour l'Open-Shop, mais cette technique est directement réutilisable pour les autres problèmes d'atelier.

Notre article est organisé de la façon suivante : dans un premier temps, nous montrons les inconvénients des Branch-and-Bound utilisant un backtrack chronologique et proposons notre technique de backtrack intelligent. Puis, nous présentons une procédure de séparation-évaluation pour l'Open-Shop et l'intégration de notre technique de backtrack intelligent dans cette procédure. Nous terminons sur les résultats obtenus sur des problèmes de la littérature.

2 Amélioration de la recherche

En recherche opérationnelle, les recherches arborescentes en profondeur d'abord utilisent toutes² un backtrack chronologique pour assurer la complétude et l'exactitude

1. *i.e.* le remplacement de chaque disjonction (i, j) entre deux tâches par un point de choix entre les deux contraintes de précédence $i \rightarrow j$ ou $j \rightarrow i$.

2. Sadeh *et al.* [SSX 95] ont proposé une amélioration du backtrack standard dans le cas des problèmes de Job-Shop. Nous reviendrons dessus section 2.2.

de la recherche. On connaît bien ses inconvénients : une tendance au *thrashing* (réexploration de sous-parties de l'espace de recherche déjà rejetées) et une démarche de résolution aveugle (on ne profite pas de l'information que peut fournir un échec pour améliorer le backtrack). Nous allons illustrer ces inconvénients à l'aide d'un petit exemple.

2.1 Illustration

Soit un problème de Job-Shop à 3 machines et 2 jobs J_1 et J_2 pour lequel nous cherchons un ordonnancement de durée minimale³. Nous rappelons que dans ce type de problèmes, le séquençement des tâches de chaque job est fixé. Résoudre ce problème revient à déterminer l'ordre de passage des jobs sur les machines, *i.e.* à déterminer sur chaque machine quel job est avant l'autre. Nous pouvons résoudre ce problème par une recherche arborescente binaire dont l'arbre est présenté figure 1. La solution optimale de ce problème est de durée 20 (feuille dont la valeur est entourée sur la figure).

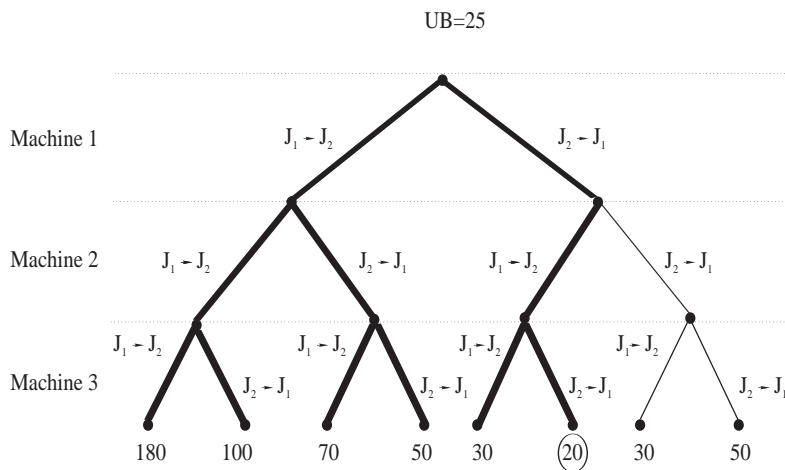


Figure 1 : Arbre binaire de recherche pour un job-shop 3×2

³. Nous considérons ici l'exemple d'un Job-Shop car l'arbre de recherche est plus petit que dans le cas d'un Open-Shop.

Supposons que nous disposions d'une solution heuristique à la racine de valeur 25. Il s'agit alors d'une borne supérieure pour un ordonnancement optimal. Nous supposerons pour simplifier que nous ne possédons pas de borne inférieure.

Explorons cet arbre de recherche : la première solution obtenue est celle du nœud de valeur 180. La solution de ce nœud dépasse la valeur de la borne supérieure (25) : ce nœud est donc un *échec*. On remonte alors au nœud père pour créer un second fils ; celui-ci est de valeur 100. La borne supérieure est une fois encore dépassée. On remonte encore au nœud père. Puisque l'on a développé tous ses fils, une approche classique va remettre en cause ce nœud père.

Mais, supposons que si J_1 passe avant J_2 sur la machine 1 (première décision prise lors de la recherche), alors considérer J_1 avant J_2 ou le contraire sur la machine 3 (les deux seules solutions envisageables), aboutit à un *makespan* supérieur à la borne supérieure. Il est alors inutile de tester la branche J_2 avant J_1 sur la machine 2 contrairement à ce que ferait un backtrack chronologique standard : la recherche peut directement repartir de la branche J_2 avant J_1 sur la machine 1 (tout en haut de l'arbre) économisant ainsi l'exploration de trois nœuds.

On peut penser qu'avec un bon schéma de séparation cette situation n'existe pas. En fait, malgré leur sophistication, il arrive souvent aux schémas de séparation de prendre des décisions complètement indépendantes le long d'une branche, créant ainsi des sous-problèmes déconnectés.

Le paragraphe suivant présente quelques techniques développées dans la communauté programmation par contraintes permettant de détecter de telles situations sans analyse *a priori*.

2.2 Éviter le thrashing : quelques approches

Le phénomène mis en évidence dans notre précédent exemple est bien connu de la communauté Programmation (Logique) par Contraintes : il s'agit du *thrashing*. Pour éviter de tels débordements, la communauté Programmation Logique a introduit la notion de *backtrack intelligent* [BP 84, COX 84] dont le but est de *remonter* directement à une mauvaise décision (passée) ayant provoqué l'échec courant.

Dans le cadre d'une énumération sur les valeurs des variables, [DBB 91] propose des mécanismes de backtrack intelligent destinés à être utilisés dans un simplexe incrémental pour résoudre des ensembles de contraintes linéaires, et [DEC 90] utilise un backtrack intelligent basé sur l'analyse du graphe des contraintes du problème dans le cadre des CSP (problème de satisfaction de contraintes). Mais, les travaux les plus intéressants concernent les améliorations du backtrack dirigées par les conflits. Il s'agit de déterminer dynamiquement à partir du conflit courant, la plus récente décision responsable de ce conflit (en évitant ainsi de revenir sur des choix qui n'y sont pas

liés). C'est la démarche utilisée par l'algorithme CBJ (*Conflict-directed BackJumping*) [PRO 93]. Cet algorithme a été développé pour la recherche d'une solution réalisable par le biais d'une énumération sur les variables du problème (recherche d'une instantiation). Cet algorithme a été amélioré [SV 93] par la conservation de certaines informations⁴ pour éviter de retrouver des situations d'échec déjà rencontrées. Il s'agit de l'algorithme *Nogood Recording*. L'obtention d'un algorithme raisonnable en terme d'occupation mémoire passe par la limitation des informations enregistrées (ici, en taille).

La remise en cause du backtrack chronologique peut aller encore plus loin : l'algorithme *Dynamic Backtracking* [GIN 93] remplace complètement le retour-arrière par un mécanisme de *saut* de solution prometteuse en solution prometteuse tout en garantissant la complétude de la recherche et de bons résultats en terme d'occupation mémoire.

Ces algorithmes ont été étendus dans le cadre d'une utilisation avec un solveur de contraintes que ce soit par la prise en compte d'un mécanisme de propagation de type *forward checking* [VS 95] ou pour tout type de solveur et de propagation dans le cas de *Dynamic Backtracking* [JB 97a, JB 97b]⁵.

Les techniques de backtrack intelligent citées ici utilisent un mécanisme d'*explanations* liant le rejet d'un membre d'une alternative (affectation d'une variable à une valeur) aux décisions prises précédemment (variables déjà instanciées).

Dans le cadre de la résolution de problèmes d'atelier, [SSX 95] proposent différentes techniques pour améliorer la résolution de problèmes de Job-Shop. Ces travaux utilisent une technique de propagation rudimentaire (propagation classique des contraintes de précédence et *forward checking* sur les contraintes disjonctives). L'énumération est réalisée sur les dates de démarrage des opérations (comme il est habituel dans la communauté contraintes).

Dans ce cadre, les auteurs présentent trois techniques :

1. une technique identifiant heuristiquement des sous-problèmes difficiles et les résolvant en priorité en backtrackant au *bon* endroit directement ;
2. un ordre dynamique de sélection des variables basé sur les échecs rencontrés ;
3. une approche heuristique utilisant un *backtrack intelligent*.

On pourra constater que notre proposition est plus générale que celle proposée par [SSX 95] et utilise des techniques de propagation plus performantes. De plus elle

4. Il s'agit de la notion de *nogood* une affectation partielle contradictoire.

5. Par *dégradation*, on obtient une telle extension pour les algorithmes *Conflict-directed BackJumping* et *Nogood Recording* puisque *Dynamic Backtracking* les généralise.

améliore simultanément chacune des trois techniques. En effet, les sous-problèmes difficiles sont identifiés lors des échecs et un *meilleur* backtrack est réalisé systématiquement ; le schéma de séparation utilisé permet de faire évoluer la sélection de disjonctions à arbitrer en fonction des informations déterminées au cours de la recherche ; notre *backtrack intelligent* offre une recherche systématique.

2.3 Notre approche

Nous rappelons que nous nous situons dans le cadre d'une recherche arborescente visant à minimiser un certain critère. Ici, il s'agit de la durée totale de l'ordonnancement. Pour cela, on utilise une borne supérieure de cette durée qui permet par simple propagation de contraintes de rejeter certaines branches explorées.

Une recherche arborescente en profondeur d'abord telles que celles utilisées en recherche opérationnelle pour résoudre les problèmes d'atelier présente un certain nombre de caractéristiques qui la distinguent d'une approche classique de programmation par contraintes :

- L'énumération n'est pas réalisée sur les valeurs possibles de certaines variables mais généralement sur les différentes possibilités de séquençement de deux tâches de l'ordonnancement. Il est donc nécessaire, pour utiliser une technique de backtrack intelligent, de fournir un mécanisme d'explication capable de lier les activités d'un solveur de contraintes (propagation) aux décisions prises lors de la recherche (*ajout* de contraintes de précédence).
- Le schéma de séparation peut produire des décisions multiples à chaque étape et non pas une unique décision.

Nous avons donc choisi de définir un système d'explication adapté à ces spécificités. Nous considérons ici la date de démarrage de chaque tâche de l'ordonnancement comme une variable dont les bornes du domaine courant définissent respectivement la date de démarrage au plus tôt de la tâche associée et sa date de démarrage au plus tard. Nous ne gérons donc pas les valeurs intermédiaires.

2.3.1 Un système d'explications

L'idée principale de notre système est de maintenir des explications liant l'activité du solveur (*i.e.* pour le retrait de valeurs des domaines des variables – modification des bornes) aux décisions induites par le schéma de séparation. Dans cette approche, une **explication** peut être définie comme un ensemble de nœuds tels que la conjonction des

décisions prises dans ces nœuds aboutit à la conclusion associée (modification d'une des bornes d'une variable) *i.e.* cette conclusion restera valide tant que les décisions considérées resteront vraies. Non seulement cette démarche est utilisable dans le cas où les décisions prises sont des contraintes, comme ici, mais en plus lorsqu'elles sont utilisées pour une recherche classique (énumération sur les valeurs des variables), elles produisent des informations plus précises que ce qui est utilisé dans les algorithmes *Conflict Directed Backjumping* ou *Nogood Recording*. Nous renvoyons le lecteur intéressé à [JUS 97] pour plus de détails.

2.3.2 Utilisation du système d'explications

Lors de la résolution, les échecs apparaissent lors de la phase de propagation des contraintes. En effet, lorsque le système courant de contraintes est contradictoire, au moins une variable présente un domaine vide à l'issue de la propagation. À partir des explications enregistrées pour les retraits successifs dans le domaine de cette variable, il est alors possible d'obtenir une justification de la contradiction utilisable pour le mécanisme de retour-arrière intelligent. L'union des explications des modifications de chacune des bornes est clairement une telle justification.

Il existe un autre cas de contradiction : lorsque tous les fils d'un nœud ont été testés, alors tout séquençement des contraintes de ce nœud est contradictoire. Une explication pour cet échec peut être obtenue de la façon suivante : quand un fils est testé, il est rejeté si et seulement si l'ajout de ses contraintes aboutit à une contradiction. Cette contradiction a une explication. L'union des explications de tous les fils qui ont échoué est donc une explication de l'échec du nœud père⁶.

D'une manière générale, les explications générées sont utilisées de la façon suivante.

Proposition 1 *Soit C_e une explication de contradiction. Soit r le nœud le plus récent élément de C_e . Tout retour-arrière sur n'importe lequel des nœuds compris entre r et le nœud courant est inutile car toutes les décisions prises dans les nœuds de C_e resteront valides et conduiront à une contradiction. On peut donc revenir directement au nœud r , économisant l'exploration de certains nœuds. L'ensemble $C_e \setminus \{r\}$ est alors une explication du rejet du fils du nœud r .*

Cependant, comme nous l'avons vu dans l'exemple paragraphe 2.1, aucune amélioration n'apparaîtra tant que tous les fils d'un nœud n'auront pas été testés. En effet, si un échec survient dans un nœud, la responsabilité incombe nécessairement aux dernières contraintes ajoutées. On reviendra donc systématiquement au nœud père.

6. En fait, on considère l'union de ces explications moins le nœud considéré.

Mais dès que tous les fils d'un nœud ont été testés, et s'ils ont tous échoué, alors on peut obtenir une amélioration en analysant l'union des explications de leurs échecs comme on vient de le décrire. On voit alors ici l'importance de la détermination d'une explication de contradiction même dans le premier cas où l'on connaît d'avance le nœud responsable le plus récent.

2.3.3 *Algorithme*

Voici un algorithme simple illustrant notre proposition (*cf.* figure 2). Notons que tout le mécanisme d'explication est délégué au solveur de contrainte utilisé.

```
(1) début
(2)   tant que il existe des disjonctions non arbitrées faire
(3)     créer un nouveau nœud
(4)     choixValide → faux
(5)     tant que il existe des fils et non choixValide
(6)       sélectionner le premier fils disponible
(7)       mettre en place les contraintes et propager
(8)       si une contradiction est levée alors
(9)         déterminer un explication pour la contradiction
(10)        rejeter le fils courant
(11)       sinon
(12)         choixValide → vrai
(13)       finsi
(14)   fintantque
(15)   si non choixValide
(16)     backtrack au nœud le plus récent expliquant les rejets des fils
(17)   finsi
(18) fintantque
(19) fin
```

Figure 2 : *Algorithme général de la recherche*

3 **Adaptation à la résolution du problème d'Open-Shop**

Nous avons intégré notre approche dans une procédure de séparation évaluation pour l'Open-Shop. Celle que nous avons choisie, proposée par Brucker *et al.* [BHJ 94] est la meilleure à notre connaissance. Dans cette procédure, nous utilisons comme

solution initiale la solution construite par une heuristique de liste à double priorité efficace présentée dans [GP 97]. Nous allons tout d'abord présenter brièvement la technique implémentée. Nous expliquerons ensuite le principe des *Arbitrages Triviaux*, règles d'élimination utilisées dans cette recherche arborescente, proposés par Carlier et Pinson [CP 89]. Dans un troisième paragraphe, nous proposerons notre système d'explications. Puis nous terminerons sur des résultats obtenus pour des problèmes de la littérature fournis par Taillard [TAI 93] après avoir présenté quelques résultats de complexités.

3.1 *Rappels et notations*

Supposons que nous disposions d'une borne supérieure de la date de fin d'ordonnement de valeur UB . Nous appellerons :

- p_i la durée de la tâche i
 - r_i la date de disponibilité (ou date de démarrage au plus tôt) de la tâche i .
 - f_i la date de démarrage au plus tard de la tâche i .
 - q_i la durée de latence de la tâche i .
- q_i, f_i et p_i sont liés par la relation suivante : $q_i = UB - f_i - p_i$.

Pour calculer les dates de disponibilité, nous pouvons utiliser la relation suivante où $\Gamma_J^{-1}(i)$ est l'ensemble des prédécesseurs de i appartenant au même job que i et $\Gamma_M^{-1}(i)$ est l'ensemble des prédécesseurs de i exécutés sur la même machine que i .

$$r_i := \max\left\{r_i, \min_{j \in \Gamma_M^{-1}(i)} r_j + \sum_{j \in \Gamma_M^{-1}(i)} p_j, \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j), \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j, \max_{j \in \Gamma_J^{-1}(i)} (r_j + p_j)\right\}$$

Cette relation est basée sur le fait que des tâches appartenant au même job ou exécutées sur la même machine ne peuvent se chevaucher. Une relation symétrique de celle-ci est utilisée pour calculer les durées de latence.

3.2 *La procédure de séparation-évaluation de Brucker*

La recherche arborescente pour l'Open-Shop présentée dans [BHJ 94] est basée sur deux concepts :

- la généralisation du schéma de séparation de Grabowski [GNZ 86] initialement développé pour le problème à une machine ;
- les arbitrages triviaux, proposés par Carlier et Pinson [CP 89].

Cette procédure de séparation-évaluation consiste en chaque nœud à construire une solution heuristique, à calculer un chemin critique⁷ de cette solution heuristique, puis à arbitrer des disjonctions sur ce chemin critique. Nous ne nous étendrons pas plus sur le schéma de séparation. Ce qu'il est important de noter ici est que ce schéma de séparation est basé sur l'arbitrage de disjonctions *i.e.* en chaque nouveau nœud, des contraintes de précédence sont ajoutées. Une fois ces contraintes ajoutées, on essaie de déduire des contraintes de précédence supplémentaires grâce aux Arbitrages Triviaux, et les dates de disponibilités et les durées de latence des tâches sont mises à jour grâce à la relation vue au paragraphe 3.1.

3.3 Arbitrages Triviaux

Les *Arbitrages Triviaux* [CP 89] sont des règles d'élimination qui fixent des disjonctions supplémentaires entre des tâches appartenant au même job ou exécutées sur la même machine.

Soient une tâche c et un ensemble de tâches I toutes exécutées sur la même machine que c (ou appartenant toutes au même job que c). Soit J un sous-ensemble de I et UB la valeur de la solution courante. Les *Arbitrages Triviaux* sont basés sur la proposition suivante :

Proposition 2 ([CP 89]) Si

$$r_c + p_c + \sum_{j \in J} p_j + \min_{j \in J} q_j > UB$$

et

$$\min_{j \in J} r_j + \sum_{j \in J} p_j + p_c + \min_{j \in J} q_j > UB$$

alors dans toutes solutions meilleures que UB , c doit être exécutée après toutes les opérations de J . J est alors appelé un ensemble ascendant de c .

En effet, la première relation est une borne inférieure d'un ordonnancement optimal des tâches de $J \cup \{c\}$ dans lequel c est exécutée avant toutes les tâches de J . La

⁷. Ensemble de tâches successives telles que tout retard sur ces tâches entraîne une augmentation de la date de fin de l'ordonnancement

deuxième relation est une borne inférieure d'un ordonnancement optimal de ce même ensemble de tâches mais dans lequel c est exécutée *au milieu* des tâches de J . Si ces deux bornes inférieures dépassent UB , alors c sera obligatoirement exécutée après toutes les tâches de J dans toute solution meilleure que UB .

Dans ce cas, la date de disponibilité de c peut être ajustée par :

$r_c \leftarrow \max(r_c, C^*)$ où

$$C^* = \max_{J' \subset J} \left\{ \min_{j \in J'} r_j + \sum_{j \in J'} p_j \right\}$$

Dans le cas où $|J| = 1$, la proposition 2 devient :

Proposition 3 ([CP 89]) *Si $r_c + p_c + p_i + q_i > UB$ alors c doit être exécutée après i .*

Nous pouvons alors fixer la disjonction $i \rightarrow c$, et l'opération c ne peut commencer avant : $r_c \leftarrow \max(r_c, r_i + p_i)$.

Ces Arbitrages Triviaux sont appelés *AT sur disjonctions*. Les premiers sont appelés *AT sur ensembles*.

Remarque : Dans le cas des AT ensembles, l'algorithme utilisé pour calculer la valeur de C^* ne détermine pas explicitement l'ensemble ascendant J (pour des raisons d'efficacité). Dans ce cas, il n'est pas possible de fixer de disjonctions entre c et les tâches de J . L'algorithme se contente donc uniquement de mettre à jour la date de démarrage au plus tôt de c . Cependant, les disjonctions pourront ensuite être fixées dans les AT sur disjonctions.

Dans la version des AT que nous venons de donner, nous ajustons les dates de disponibilité des tâches. Du fait de la symétrie existant entre les dates de disponibilité et les durées de latence, une version symétrique de ces relations permet d'ajuster les durées de latence.

3.4 Enregistrer les explications

Comme nous l'avons expliqué au paragraphe 2.3.1, l'idée de base de notre technique de backtrack intelligent est d'enregistrer des informations pour chaque retrait de valeur des domaines des variables. Dans le problème d'Open-Shop, les variables t_i sont les dates de démarrage des tâches, et leurs domaines sont les intervalles $[r_i, f_i]$ où r_i est la date de démarrage au plus tôt de la tâche i et f_i sa date de démarrage

au plus tard. Comme les r_i (*resp.* f_i) ne peuvent qu'augmenter (*resp.* diminuer) au cours de l'exploration, à chaque fois qu'un r_i ou un f_i est modifié, des valeurs sont retirées du domaine de la variable t_i . Donc, à chaque fois qu'un r_i (*resp.* un f_i) est modifié, nous proposons d'enregistrer les nœuds dont la conjonction des décisions est responsable de cette modification.

Enregistrer les modifications des r_i et f_i revient à enregistrer les modifications des r_i et q_i . En effet, nous savons que $q_i = UB - f_i - p_i$. Nous préférons travailler sur les q_i plutôt que sur les f_i afin de rester homogène avec les Arbitrages Triviaux qui sont basés sur les r_i et q_i .

En plus d'enregistrer les explications des modifications des r_i et q_i , il est également nécessaire d'enregistrer les explications de l'ajout de contraintes de précédence. En effet, une telle contrainte peut être déduite (par les AT) des décisions prises dans un nœud, et modifier des r_i et q_i . Les explications des ajouts des contraintes de précédence sont donc nécessaires pour maintenir les explications des r_i et q_i .

Nous allons voir maintenant quelles informations enregistrer et comment les enregistrer au cours de la recherche arborescente.

Au nœud racine, toutes les explications sont initialisées à $\{root\}$ (nœud racine). Ensuite, chaque fois qu'un événement a lieu (modification d'un r_i ou d'un q_i , arbitrage d'une disjonction), l'explication associée est enregistrée. Nous allons considérer les différentes situations dans lesquelles de tels événements apparaissent et expliquer comment les explications sont mises à jour.

3.4.1 Explications des ajouts de contraintes de précédence

De nouvelles contraintes de précédence peuvent être ajoutées soit lors de la séparation d'un nœud, soit lors du calcul des AT sur disjonctions.

– Ajout de contraintes de précédence lors de la séparation d'un nœud

Supposons qu'une disjonction $i \rightarrow j$ soit arbitrée par le schéma de séparation dans un nouveau nœud r dans l'exploration. Si cette contrainte de précédence n'existe pas, son explication est le nœud courant (r). Donc l'explication de cette nouvelle contrainte est : $Expl(i, j) := \{r\}$

Si $i \rightarrow j$ existe déjà, son explication ne change pas.

– Ajout de contraintes de précédence lors du calcul des AT sur disjonctions

Soient deux tâches i et j exécutées sur la même machine ou appartenant au même job.

Si $r_j + p_j + p_i + q_i > UB$ alors les AT sur disjonctions fixent la disjonction dans le sens $i \rightarrow j$. Si cette contrainte n'existe pas, nous l'ajoutons donc à cause des valeurs de r_j et de q_i . Son explication devient : $Expl(i, j) := Expl(r_j) \cup Expl(q_i)$

Si cette contrainte existe déjà, son explication ne change pas.

3.4.2 Explications des modifications des valeurs r_i et q_i

Les r_i et q_i sont modifiés dans trois situations : lors du calcul des AT sur disjonctions, lors du calcul des AT sur ensembles, et lors de leur mise à jour après la séparation du nœud et le calcul des AT.

– Modification des r_i et q_i lors du calcul des AT sur disjonctions

Si deux tâches i et j exécutées sur la même machine ou appartenant au même job vérifient $r_j + p_j + p_i + q_i > UB$ alors les AT sur disjonctions fixent la disjonction $i \rightarrow j$, puis ajustent la date de disponibilité de la tâche j qui ne peut démarrer avant : $\max(r_j, r_i + p_i)$.

Si r_j est modifié, c'est à cause de l'ajout de la contrainte $i \rightarrow j$, et sa nouvelle valeur est fonction de r_i . Donc l'explication de la nouvelle valeur de r_j devient : $Expl(r_j) := Expl(r_j) \cup Expl(i, j) \cup Expl(r_i)$

Si r_j n'est pas modifié, son explication ne change pas.

– Modification des r_i et q_i lors du calcul des AT sur ensembles

Soit une machine M_k sur laquelle on calcule les AT sur ensembles. Nous rappelons que les AT sur ensembles ne fixent pas de disjonctions. Elles se contentent d'ajuster les r_i et q_i . Les contraintes de précédence déduites sont ensuite ajoutées dans les AT sur disjonctions.

Soit une tâche c exécutée sur M_k et soit J un ensemble ascendant de c contenant des tâches exécutées sur M_k (le cas où l'ensemble ascendant de c contient des tâches appartenant au même job que c se déduit facilement). Si la date de disponibilité de c est modifiée, elle prend pour valeur (cf. paragraphe 3.3):

$r_c := \max(r_c, C^*)$ où

$$C^* = \max_{J' \subseteq J} \left\{ \min_{j \in J'} r_j + \sum_{j \in J'} p_j \right\}$$

Cette modification est donc due aux valeurs des r_j et q_j de toutes les tâches j appartenant à J' . Nous avons vu que cet ensemble J' n'est pas déterminé ex-

plicitement dans l'algorithme. Nous nous contenterons⁸ donc de supposer que la modification de r_c est due aux valeurs des r_i et q_i de toutes les tâches exécutées sur M_k (ce qui est clairement une explication valide). Si r_c est modifié, l'explication de sa nouvelle valeur est donc :

$$Expl(r_c) := \{Expl(r_i) \mid i \text{ exécuté sur } M_k\} \cup \{Expl(q_i) \mid i \text{ exécuté sur } M_k\}$$

– **Modification des r_i et q_i lors de leur recalcul**

Pour recalculer les r_i , nous utilisons la relation :

$$r_i := \max\{r_i, \min_{j \in \Gamma_M^{-1}(i)} r_j + \sum_{j \in \Gamma_M^{-1}(i)} p_j, \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j), \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j, \max_{j \in \Gamma_J^{-1}(i)} (r_j + p_j)\}$$

où $\Gamma_J^{-1}(i)$ est l'ensemble des prédécesseurs de i appartenant au même job que i et $\Gamma_M^{-1}(i)$ est l'ensemble des prédécesseurs de i exécutés sur la même machine que i .

Nous considérons alors cinq cas :

1. r_i n'est pas modifié : son explication ne change pas.

2. $r_i = \min_{j \in \Gamma_M^{-1}(i)} r_j + \sum_{j \in \Gamma_M^{-1}(i)} p_j$:

La modification de r_i est due à toutes les valeurs r_j , $j \in \Gamma_M^{-1}(i)$, et à toutes les contraintes $j \rightarrow i$, $j \in \Gamma_M^{-1}(i)$. Donc nous avons :

$$Expl(r_i) = Expl(r_i) \cup \{Expl(r_j), j \in \Gamma_M^{-1}(i)\} \cup \{Expl(j, i), j \in \Gamma_M^{-1}(i)\}$$

3. $r_i = \max_{j \in \Gamma_M^{-1}(i)} (r_j + p_j) = r_k + p_k$:

L'explication de la nouvelle valeur de r_i devient :

$$Expl(r_i) = Expl(r_i) \cup Expl(r_k) \cup Expl(k, i)$$

4. $r_i = \min_{j \in \Gamma_J^{-1}(i)} r_j + \sum_{j \in \Gamma_J^{-1}(i)} p_j$:

La modification de r_i est due à toutes les valeurs r_j , $j \in \Gamma_J^{-1}(i)$, et à toutes les contraintes $j \rightarrow i$, $j \in \Gamma_J^{-1}(i)$. Donc nous avons :

$$Expl(r_i) = Expl(r_i) \cup \{Expl(r_j), j \in \Gamma_J^{-1}(i)\} \cup \{Expl(j, i), j \in \Gamma_J^{-1}(i)\}$$

5. $r_i = \max_{j \in \Gamma_J^{-1}(i)} (r_j + p_j) = r_k + p_k$:

L'explication de la nouvelle valeur de r_i devient :

$$Expl(r_i) = Expl(r_i) \cup Expl(r_k) \cup Expl(k, i)$$

8. Il serait bien sûr plus précis d'enregistrer comme explication d'ajustement l'union des explications des r_i pour l'ensemble de tâche J' et des explications des q_i pour l'ensemble des tâches de J mais une implémentation efficace des arbitrages triviaux [GUE 97] implique que l'on ne connaisse pas explicitement ni J ni J' , il est moins coûteux – et quand même efficace – d'utiliser cette explication.

Proposition 4 En un nœud donné, pour toute tâche i , $Expl(r_i)$ (resp. $Expl(q_i)$) contient tous les numéros des nœuds responsables d'une modification de r_i (resp. q_i).

Proposition 5 En un nœud donné, pour chaque disjonction arbitraire $i \rightarrow j$, $Expl(i, j)$ contient tous les numéros des nœuds responsables de l'ajout de la contrainte $i \rightarrow j$.

Preuve Ces propositions se démontrent facilement par récurrence.

Ces propositions sont vraies au nœud racine. Par construction, on peut vérifier que si elles sont vraies au niveau r de la recherche arborescente, elles restent vraies au niveau $r + 1$. \square

3.5 Utiliser les explications

Toutes ces informations sont ensuite utilisées pour déterminer des points de backtrack pertinents en cas d'échec. Dans la recherche arborescente, nous distinguons 2 possibilités d'échecs.

1. Il existe une tâche i pour laquelle $r_i + p_i + q_i \geq UB$ (le domaine de la variable t_i devient vide)

Dans ce cas nous remontons dans l'arbre de recherche au nœud le plus récent dans lequel r_i ou q_i a été modifié. Le point de backtrack est le nœud le plus récent parmi les nœuds de $Expl(r_i) \cup Expl(q_i)$.

2. On a exploré tous les fils d'un nœud sans succès.

Dans ce cas, le nœud père est lui aussi un échec. Le point de backtrack est déterminé en choisissant le nœud le plus récent de l'union des explications des échecs de tous les nœuds fils, moins le nœud père.

3.6 Complexités

Les complexités spatiale et temporelle de notre approche sont liées au nombre de disjonctions du problème. Étant donné que toutes les tâches d'un même job ou d'une même machine sont en disjonction, ce nombre est en $O(m \times n(m + n))$.

Le nombre de disjonctions correspond à la profondeur maximale dans le pire cas de l'arbre de recherche et par conséquent définit aussi la taille maximale d'une explication dans notre approche.

L'enregistrement de toutes les explications (pour les modifications des bornes des variables et pour les disjonctions elles-mêmes) requiert donc un espace en $O(m^2 \times n^2(m+n))$ dans le pire cas.

Le surcoût engendré par le calcul d'explications lors de la phase de propagation se mesure selon le nombre d'unions réalisées entre explications : pour les arbitrages triviaux sur ensembles, il y en a $n-1$ (resp. $m-1$) si on les calcule pour une machine (resp. job) donnée et une seule pour toutes les autres propagations.

En pratique, le surcoût en temps engendré par le mécanisme d'enregistrement d'explications est négligeable.

3.7 Résultats

La tableau 1 présente les résultats obtenus en incluant notre technique dans la recherche arborescente de Brucker *et al.* sur les problèmes proposés par Taillard [TAI 93], en comparaison avec la même version sans backtrack intelligent. Ces problèmes se composent de dix Open-Shop carrés de taille 4, 5, 7 et 10. Pour chaque problème nous donnons une borne inférieure LB , une borne supérieure UB , ainsi que sa valeur optimale OPT (inconnue pour quatre des problèmes de taille 10).

Notre tableau indique, pour chaque version, le nombre de backtracks réalisés pour atteindre l'optimum. Nous avons stoppé notre recherche arborescente à 250000 backtracks (ce qui correspond à peu près à 3 heures de calcul sur un Pentium 133MHz). Si l'optimum n'est pas atteint à cette limite, nous indiquons entre parenthèses la meilleure solution courante.

Bien qu'il soit difficile de généraliser à partir des résultats obtenus sur seulement 10 problèmes de chaque taille, nous observons que la fréquence des diminutions augmente avec la taille des problèmes : alors qu'il n'y a aucune amélioration sur les problèmes de taille 4 de Taillard, cette technique permet de diminuer le nombre de backtracks pour 9 problèmes sur les 10 problèmes de taille 5, et pour tous les problèmes de taille 7.

Sur certains problèmes, l'amélioration est très significative, notamment sur le dixième problème de taille 5 où l'amélioration est de plus de 90%, mais aussi et surtout sur les problèmes de taille 10 : notre technique de backtrack permet d'en résoudre 3, dont un ouvert (celui dont la valeur optimale est encadrée) en seulement 4843 backtracks, alors que la version sans backtrack intelligent ne résout aucun de ces problèmes en moins de 250000 backtracks. On peut ainsi noter une diminution du nombre de backtracks d'au moins 90% pour le quatrième 10x10 et de plus de 98% pour le septième.

Taille	<i>OPT</i>	<i>LB</i>	<i>UB</i>	sans backtrack intelligent nombre de backtracks	avec backtrack intelligent nombre de backtracks
4	193	186	197	18	18
4	236	229	253	37	37
4	271	262	272	29	29
4	250	245	260	26	26
4	295	287	305	55	55
4	189	185	193	19	19
4	201	197	203	22	22
4	217	212	217	14	14
4	261	258	268	32	32
4	217	213	224	31	31
5	300	295	303	273	270
5	262	255	266	252	238
5	323	321	347	943	940
5	310	306	319	678	678
5	326	321	330	840	836
5	312	307	325	756	737
5	303	298	308	420	411
5	300	292	304	853	851
5	353	349	368	1000	987
5	326	321	337	2377	237
7	435	435	452	2840	1860
7	443	443	465	18196	16862
7	468	468	495	98903	96502
7	463	463	482	1494	1410
7	416	416	425	317	256
7	451	451	471	21492	20364
7	422	422	449	56944	53773
7	424	424	433	1939	1552
7	458	458	476	560	535
7	398	398	411	731	710
10		637	654	> 250000 (644)	> 250000 (644)
10	588	588	600	> 250000 (594)	> 250000 (591)
10		598	611	> 250000 (610)	> 250000 (604)
10	577	577	588	> 250000 (584)	26777
10	640	640	661	> 250000 (658)	> 250000 (658)
10	538	538	544	> 250000 (544)	> 250000 (544)
10	616	616	633	> 250000 (633)	4843
10	595	595	604	> 250000 (604)	> 250000 (604)
10	595	595	612	> 250000 (597)	245100
10		596	612	> 250000 (611)	> 250000 (606)

Table 1 : Comparaison des résultats obtenus avec la recherche arborescente de Brucker et al. avec et sans backtrack intelligent

En ce qui concerne les temps d'exécution, l'utilisation de notre technique de backtrack intelligent n'est pas plus coûteuse en chaque nœud sur les problèmes de petite taille (4 et 5). Elle devient légèrement plus lente sur des problèmes de taille 7 (0,204 s/nœud au lieu de 0.199s/nœud). Mais cette perte de temps est largement compensée par la diminution du nombre de backtracks qui réduit le temps moyen de résolution des problèmes de taille 7 à 643,58s, contre 725,80s sur un Pentium 133 MHz.

4 Conclusion

Dans cet article, nous avons présenté une nouvelle technique de backtrack intelligent que nous avons appliquée à une recherche arborescente pour l'Open-Shop. Les résultats obtenus sont très satisfaisants puisque cette technique permet de diminuer nettement le nombre de backtracks nécessaires pour atteindre l'optimum. Elle nous a permis entre autre de résoudre un problème ouvert de la littérature à 10 jobs et 10 machines.

Notre technique peut facilement s'adapter à n'importe quelle recherche arborescente pour les problèmes d'atelier de type Open-Shop, Job-Shop ou Flow-Shop dont le schéma de séparation est basé sur l'arbitrage de disjonctions.

Cette technique correspond à l'adaptation aux Branch-and-Bound pour l'Open-Shop d'une partie de nos précédents travaux sur le système DECORUM dans lequel le backtrack est complètement remplacé par une approche *par sauts* [JUS 97, JB 97b]. Nous travaillons actuellement sur l'adaptation complète de ce système à la résolution de l'Open-Shop.

Bibliographie

- [BJH 94] BRUCKER P., HURINK J., JURISCH B., et WÖSTMANN B., « A Branch and Bound Algorithm for the Open-Shop Problem », *Discrete Applied Mathematics*, vol. 76, :43–59, 1997.
- [BP 84] BRUYNNOOGHE M. et PEREIRA L., « *Implementations of Prolog* », chapitre Deduction revision by intelligent backtracking, pp. 194–215, Ellis Horwood, 1984.
- [COX 84] COX P., « *Implementation of Prolog* », chapitre Finding backtrack points for intelligent backtracking, pp. 216–233, Ellis Horwood, 1984.
- [CP 89] CARLIER J. et PINSON É., « An algorithm for solving the job shop problem », *Management Science*, vol. 35, :164–176, 1989.
- [DBB 91] BACKER B. D. et BERINGER H., « Intelligent Backtracking for CLP Languages: An Application to CLP(\mathcal{R}) », In SARASWAT V. et UEDA K., éditeurs, *ILPS'91: Proceedings International Logic Programming Symposium*, pp. 405–419, San Diego, CA, October 1991, MIT Press.

- [DEC 90] DECHTER R., « Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition », *Artificial Intelligence*, vol. 41, n° 3:273–312, January 1990.
- [GIN 93] GINSBERG M., « Dynamic Backtracking », *Journal of Artificial Intelligence Research*, vol. 1, :25–46, 1993.
- [GNZ 86] GRABOWSKI J., NOWICKI E., et ZDRZALKA S., « A block approach for single-machine scheduling with release dates and due dates », *European Journal of Operational Research*, vol. 26, :278–285, 1986.
- [GP 97] GUÉRET C. et PRINS C., « Classical and New Heuristics for the Open-Shop Problem », *European Journal of Operations Research*, vol. à paraître, , 1997.
- [GUE 97] GUÉRET C., Problèmes d’ordonnancement sans contraintes de précédence, thèse de doctorat, Université de Technologie de Compiègne, 29 October 1997.
- [JB 97a] JUSSIEN N. et BOIZUMAULT P., « Stratégies en Meilleur d’abord pour la relaxation de contraintes », In *Journées Francophones de Programmation en Logique et avec Contraintes*, Orléans, May 1997.
- [JB 97b] JUSSIEN N. et BOIZUMAULT P., « Dynamic Backtracking with Constraint Propagation – Application to static and dynamic CSPs », In *CP97 Workshop on The Theory and Practice of Dynamic Constraint Satisfaction*, Schloss Hagenberg, Austria, 1 November 1997.
- [JUS 97] JUSSIEN N., Relaxation de Contraintes pour les problèmes dynamiques, thèse de doctorat, Université de Rennes I, 24 October 1997.
- [PRO 93] PROSSER P., « Hybrid algorithms for the Constraint Satisfaction Problem », *Computational Intelligence*, vol. 9, n° 3:268–299, 1993.
- [SSX 95] SADEH N., SYCARA K., et XIONG Y., « Backtracking techniques for the job shop scheduling constraint satisfaction problem », *Artificial Intelligence*, vol. 76, :455–480, 1995.
- [SV 93] SCHIEX T. et VERFAILLIE G., « Nogood recording for static and dynamic CSP », In 5th *IEEE International Conference on Tools with Artificial Intelligence*, pp. 48–55, Boston, MA., 1993.
- [TAI 93] TAILLARD E., « Benchmarks for Basic Scheduling Problems », *European Journal of Operations Research*, vol. 64, , 1993.
- [VS 95] VERFAILLIE G. et SCHIEX T., « Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes : bilan de quelques approches », *Revue d’Intelligence Artificielle*, vol. 9, n° 3, 1995.