Application de Resolution Search au RCPSP

Sophie Demassey[1], Serigne Gueye[2],
Philippe Michelon[1], and Christian Artigues[1]

[1] Laboratoire Informatique d'Avignon, CNRS FRE 2487,
339, chemin des Meinajariés, Agroparc, BP 1228,
84911 Avignon Cedex 9
sophie.demassey@lia.univ-avignon.fr
[2] Laboratoire de Mathématiques Appliquées du Havre,
25 rue Philippe Lebon, BP 540, 76058 Le Havre cedex
serigne.gueye@univ-lehavre.fr

Résumé En 1997, Chvátal [4] publie une nouvelle méthode, appelée *resolution search*, pour résoudre les programmes linéaires en variables binaires. Elle se distingue des méthodes usuelles de recherche arborescente par séparation et évaluation, par une exploration particulière de l'espace de recherche et par la mémorisation des portions éliminées au fur et à mesure de cette exploration. En ce sens, elle est analogue aux méthodes de *nogood recording* en programmation par contraintes, et plus particulièrement au *dynamic backtracking*.

Dans le but de proposer une alternative originale aux nombreuses procédures de résolution exacte développées pour le RCPSP, nous cherchons à appliquer resolution search à ce problème d'ordonnancement. Nous testons ici une version basique de resolution search sur une modélisation linéaire en variables binaires courante du RCPSP. Afin d'établir une stricte comparaison des performances, nous appliquons à cette même formulation une procédure d'énumération implicite, d'implémentation équivalente, non moins basique.

Mots-Clefs. RCPSP ; Resolution search ; Backtracking intelligent.

1 Introduction

Resolution search est une procédure originale pour la résolution exacte de programmes linéaires en variables binaires, proposée par Chvàtal [4]. Cette alternative aux méthodes d'énumération implicite repose sur l'exploration particulière de l'arbre de recherche, qui, à l'inverse des procédures usuelles de séparation et évaluation, peut être vue comme allant des feuilles à la racine de l'arbre. Il s'agit en effet, de rechercher, le plus haut dans l'arbre, des noeuds à éliminer sachant qu'ils ne conduisent pas à de meilleures solutions. Ces noeuds éliminés sont alors stockés intelligemment, de sorte que la recherche s'effectue en dehors de leurs sous-arborescences.

À notre connaissance, l'emploi de cette procédure n'a donné lieu, à ce jour, à aucune autre publication. Pour tester son efficacité et proposer une alternative

aux méthodes couramment employées, nous appliquons resolution search au problème d'ordonnancement à contraintes de ressource (RCPSP), un des problèmes d'ordonnancement les plus généraux, et les plus étudiés, de la littérature.

Le RCPSP consiste à ordonnancer des activités liées par des contraintes de précédence, et qui entrent en concurrence dans l'utilisation de ressources disponibles en quantité limitée. Plus précisément, étant donnés :

- $A_1, \ldots, A_n$ activités, de durées respectives $p_1, \ldots, p_n$, et un graphe acyclique (A, E) des contraintes de précédence entre ces activités,
- $R_1, \ldots, R_m$ ressources de capacités limités $c_1, \ldots, c_m$, chaque activité A_i nécessitant durant son exécution une quantité constante r_{ik} de chaque ressource R_k,

Il s'agit d'attribuer des dates de début $(S_1, \ldots, S_n)$ aux activités, en respectant à la fois les contraintes de précédence (A_j ne peut démarrer avant que A_i soit terminée : $S_j - S_i \geq p_i$, si $(i,j) \in E$) et les contraintes de ressources (à tout temps t et pour chaque ressource R_k, la somme des quantités de R_k utilisées par les activités en cours au temps t ne peut excéder la disponibilité de la ressource R_k : $\sum_{i|S_i \leq t < S_i + p_i} r_{ik} \leq c_k$). Pour en faciliter l'écriture, il est d'usage d'ajouter deux activités fictives, de durées et de consommations nulles : A_0 précédant toutes les activités du projet et A_{n+1} leur succédant. L'objectif le plus naturel et auquel on s'attache ici, est la minimisation de la durée totale d'exécution du projet, autrement dit, en posant $S_0 = 0$, la date de fin (et de début) de la dernière tâche S_{n+1}.

La seule variante décisionnelle du RCPSP est NP-difficile, ainsi, le RCPSP a fait l'objet de nombreux travaux et l'essentiel des méthodes exactes développées pour ce problème sont de type PSE (procédure par séparation et évaluation) comme, par exemple, [3,5,1,8]. [6,2] fournissent un état de l'art sur le RCPSP.

Dans un premier temps, nous nous proposons de montrer l'efficacité de resolution search par rapport à une PSE basique pour résoudre le RCPSP en tant que programme linéaire en variables binaires. Nous chercherons par la suite à intégrer dans Resolution Search des schémas de branchement spécifiques au RCPSP.

Le papier est organisé comme suit : La section 2 décrit rapidement la formulation linéaire en variables binaires du RCPSP pour laquelle nous avons implémenté resolution search. La section 3 présente plus en détail l'algorithme de resolution search. Les résultats expérimentaux préliminaires sont donnés à la section 4. Enfin, des pistes pour l'intégration à resolution search de branchements spécifiques au RCPSP sont proposées en conclusion, section 5.

2 Modèle linéaire 0-1 pour le RCPSP

La formulation linéaire du RCPSP la plus usuelle est celle introduite par Pritsker et al. [9]. Étant donnée une évaluation par excès T de la valeur optimale du problème, cette formulation est basée sur des variables binaires x_{it}, $i \in \{0, \ldots, n+1\}$ et $t \in \{0, \ldots, T\}$, définies par : x_{it} est égale à 1 si l'activité A_i commence à l'instant t $(S_i = t)$, égale à 0 sinon. Il est possible de réduire

considérablement le nombre de variables en calculant au préalable, pour chaque activité A_i, la fenêtre de temps $[ES_i, LS_i]$, durant laquelle A_i doit nécessairement commencer, par simple calcul des plus longs chemins dans le graphe de précédence, de 0 à i et de i à $n+1$. Ainsi, les variables x_{it} ne sont plus définies que pour des dates t comprises dans cet intervalle, et le programme linéaire de Pritsker peut s'écrire comme suit :

$$\min \sum_{t=ES_{n+1},\ldots,LS_{n+1}} t y_{(n+1)t}$$

sujet à :

$$\sum_{t=ES_j}^{LS_j} y_{jt} = 1 \qquad \forall\, j \in \{0,\ldots,n+1\} \tag{1}$$

$$\sum_{t=ES_j}^{LS_j} t y_{jt} - \sum_{t=ES_i}^{LS_i} t y_{it} \geq p_i \qquad \forall\, (i,j) \in E \tag{2}$$

$$\sum_{j=1}^{n} r_{jk} \sum_{\tau=\min\{t-p_j+1, ES_j\}}^{\min\{t, LS_j\}} y_{j\tau} \leq R_k \qquad \forall\, k \in \mathcal{R}, \forall\, t \in \{0,\ldots,T\} \tag{3}$$

$$y_{jt} \in \{0,1\} \qquad \forall\, j \in \{0,\ldots,n+1\}, \forall\, t \in \{ES_j,\ldots,LS_j\} \tag{4}$$

3 Resolution search

Comme il est évoqué dans l'introduction, resolution search se distingue des méthodes d'énumération implicite par sa façon de parcourir l'arbre de recherche : partant de noeuds terminaux de l'arbre pour en atteindre la racine. Ce comportement est lié au fait que, comme les procédures de backtracking intelligent en programmation par contraintes, resolution search recherche la *cause* de l'échec d'un branchement, autrement dit, les décisions prises qui sont seules responsables de l'infaisabilité (violation des contraintes ou de la borne).

Tout d'abord, la procédure peut commencer de n'importe quel noeud de l'espace de recherche. Branchements et évaluations successives sont alors effectués, tels qu'habituellement, en descente, dans une énumération implicite. Dans le but de réduire la taille de l'espace de recherche, dès qu'un noeud terminal est atteint, c'est à dire, un noeud dont la sous-arborescence ne peut contenir de solution meilleure que la meilleure solution courante, on lui recherche un noeud ascendant qui soit aussi terminal. Pour cela, on teste une à une dans l'ordre inverse, les décisions qui ont été prises lors de la descente, puis, les décisions déjà présentes dans le noeud initial. On commence par supprimer l'avant-dernière décision. Si les évaluations, par défaut et par excès, indiquent que le noeud n'est pas terminal, la décision est rétablie. On recommence avec la décision précédente, et ainsi de suite pour toutes les décisions. On notera au passage, que la remontée ne se fait pas le long de la branche de descente : certaines décisions sont supprimées, d'autres (en particulier la plus récente) non. Ainsi, l'espace de

recherche de resolution search se présente davantage comme un graphe plutôt que comme un arbre. À la fin du processus de backtrack, le dernier noeud trouvé, est stocké dans une famille F de noeuds terminaux et la procédure reprend dans un autre secteur de l'espace de recherche. Dès que la racine de l'« arbre » devient un noeud terminal, le problème est résolu, étant prouvé que la meilleure solution courante est optimale. L'idée naïve de conserver tous les noeuds terminaux rencontrés peut être, bien sûr, techniquement impossible. De plus, il serait difficile alors de s'assurer qu'à tout moment, la recherche ne recoupe pas des noeuds préalablement éliminés.

Le coeur de resolution search réside véritablement dans la façon dont est gérée la famille F de noeuds éliminés et la manière dont est construit le nouveau noeud d'où va partir la recherche à la suite de l'ajout d'un nouveau noeud terminal à F. Schématiquement, si deux noeuds u_0 et u_1 appartiennent à F, et si u_0 et u_1 sont les (deux seuls) fils d'un même noeud u, alors u est nécessairement un noeud terminal et il est plus intéressant de ne conserver dans F que le seul noeud u, plutôt qu'à la fois, u_0 et u_1. En logique propositionnelle, u est appelée la *résolvante* de u_0 et u_1, et ce principe de déduction est appelé *principe de résolution*. Par la suite, si les deux fils de la racine de l'arbre sont dans F, alors la racine est elle-même un noeud terminal et donc, il est prouvé que la meilleure solution courante est optimale. Enfin, un noeud u_F est associé à F de sorte que, pour chaque noeud u^k de F, il existe un branchement (par exemple sur une variable binaire x^k, $x^k = 0$) dont est issu u, et tel que u^F est issu du branchement opposé ($x^k = 1$). De plus, la variable x^k est libre dans tous les autres noeuds de F. u_F appartient donc à l'espace de recherche non encore éliminé et c'est à partir de ce noeud là que la procédure est itérée. Ainsi, à l'itération suivante, on est assuré de ne jamais croiser les sous-arborescences des noeuds de F, que ce soit en descente ($x^k = 1$) ou lors de la remontée ($x^k = 1$ ou x^k non fixée).

En pratique, la gestion de la famille F est plus compliquée, de manière à lui conserver une structure particulière *path-like*. Un noeud est, soit simplement ajouté à F, soit utilisé pour réduire F au cas où la sous-arborescence du noeud couvre en partie celle d'un noeud de F. Il se peut alors que la portion de l'espace de recherche éliminée par la famille F ainsi mise à jour ne recouvre pas entièrement celle éliminée auparavant. Cependant, elle est plus étendue et la convergence de l'algorithme est assurée. On se référera à [4] pour les preuves de convergence et la maintenance de la structure de la famille F.

Pour illustrer ce principe, on déroule maintenant l'algorithme sur l'exemple formel simple d'un problème de minimisation à 4 variables binaires x_1, x_2, x_3, x_4, en branchant sur les variables dans cet ordre, d'abord sur la valeur 0, puis sur 1. On représente un noeud du graphe de recherche par un vecteur (x_1, x_2, x_3, x_4). Par exemple, $(*, 1, *, *)$ correspond à la décision $x_2 = 1$, les autres variables n'étant pas instanciées. On suppose connaître une borne supérieure égale à 1 et une borne inférieure égale à 1 si x_2 et x_4 sont toutes deux instanciées (à 0 ou 1), et 0 sinon. La valeur optimale est donc nécessairement 1.

On démarre la recherche à partir du noeud $(1, 0, *, *)$. La borne inférieure étant 0, on branche sur $(1, 0, 0, *)$ puis sur $(1, 0, 0, 0)$. La solution n'étant pas

meilleure, on remonte en défixant progressivement à partir de l'avant-dernière décision prise $(1, 0, *, 0)$. La borne étant encore à 1, cela signifie que x_3 n'intervient pas dans la cause de l'échec. Au contraire, en défixant x_2, le noeud $(1, *, *, 0)$ n'est plus terminal. On conserve donc la décision $x_2 = 0$, puis on teste x_1. À la fin du backtrack, on déduit que $(*, 0, *, 0)$ est un noeud terminal, on le stocke dans F et on repart dans une autre partie de l'espace de recherche en inversant une décision, par exemple $(*, 1, *, 0)$. Comme ce noeud est terminal, on l'utilise pour mettre à jour F. Ici, on peut remplacer les deux noeuds de F par leur résolvante, en posant $F = \{(*, *, *, 0)\}$. On repart de $(*, *, *, 1)$, on branche jusqu'au noeud terminal $(0, 0, *, 1)$, puis on remonte jusque $(*, 0, *, 1)$. F peut de nouveau être mis à jour par résolvante : $F = \{(*, *, *, 0), (*, 0, *, *)\}$. La recherche reprend alors sur le noeud $(*, 1, *, 1)$ qui est terminal. Par résolvante avec le second noeud de F, on déduit que $(*, *, *, 1)$ est terminal, puis, avec le premier noeud, que $(*, *, *, *)$ est terminal. La recherche s'arrête alors prouvant qu'il n'existe pas de solution de valeur strictement inférieure à 1. Ici, resolution search a exploré 14 noeuds, tandis qu'une PSE équivalente en aurait exploré 31 (la racine $(*, *, *, *)$, puis de $(0, *, *, *)$ à $(0, 1, 1, 1)$, puis de $(1, *, *, *)$ à $(1, 1, 1, 1)$). De plus, en choisissant un meilleur noeud de départ (par exemple $(*, 0, *, 0)$), la recherche n'aurait évalué que 11 noeuds. À noter enfin que, dans cet exemple, après chaque mise à jour, l'espace de recherche éliminé par F augmente strictement, ce qui n'est pas toujours le cas.

4 Résultats expérimentaux préliminaires

Nous avons implémenté la procédure de resolution search telle que décrite dans [4], pour le modèle linéaire en variables binaires, de Pritsker, présentés à la section 2. L'évaluation par défaut LB_u est faite à chaque noeud rencontré u par la résolution du programme linéaire PL_u : la relaxation continue du modèle, où les variables de décision sont fixées à leur même valeur 0 ou 1 que dans le noeud u. Le programme linéaire continu est résolu au moyen de Cplex. L'évaluation par excès n'est pas calculée en chaque noeud mais est égale à la valeur de la meilleure solution courante. Avant d'exécuter resolution search, un premier ordonnancement réalisable S est calculé par une heuristique. Nous avons utilisé ici la procédure de recherche tabou de Brucker et al. [1]. Les intervalles $[ES_i, LS_i]$ des dates possibles de début d'exécution des activités sont aussi calculés par les algorithmes de Bellman-Forward et Bellman-Backward, ne prenant en compte que les contraintes de précédence. Si $ES_{n+1} = S_{n+1}$, alors S est un ordonnancement optimal. Sinon, resolution search est exécuté en partant du noeud terminal S. Nos premiers résultats expérimentaux semblent en effet indiquer qu'il est plus efficace de partir d'une feuille plutôt que de la racine. Le choix de la variable de branchement se fait sur la variable la plus fractionnaire de la solution courante. Ainsi, on améliore la méthode de Chvàtal en détectant plus tôt les solutions entières du programme linéaire continu.

Partant de ce code, il est facile d'implémenter une procédure d'énumération implicite avec les mêmes évaluations et le même schéma de branchement. Le ta-

bleau 1 présente les résultats comparés de ces deux procédures, resolution search (RS, ligne 2) et énumération implicite (PSE, ligne 3) sur les 173 premières instances à 30 activités de Kolisch et al. [7], et dont la valeur optimale dépasse strictement la borne du chemin critique. La colonne 2 présente le nombre d'optimaux prouvés dans la limite de 30 minutes ; colonnes 3 et 4, les temps CPU moyen et max en secondes ; colonnes 5 et 6, les déviations moyenne et max par rapport à l'optimal ; colonne 7, le nombre moyen de noeuds parcourus.

Tab. 1. Résultats sur 173 instances non-triviales de KSD30

	#opt	CPU moy.	CPU max.	Δ moy.	Δ max	# noeuds
RS	**84**	964	1800	2,92%	14,06%	157 434
BB	52	1322	1800	3,31%	14,06%	184911

À implémentation égale, RS est nettement plus performant que PSE puisqu'il prouve l'optimalité pour 84 instances, contre 52 pour PSE, dans la limite des 1800 secondes. En particulier, resolution search tire avantage de pouvoir débuter la recherche d'une solution réalisable. L'existence de très bonnes heuristiques pour le RCPSP est d'ailleurs un atout à cette méthode. Bien sûr, ces résultats sont bien en-deçà des résultats de la littérature, mais ils sont prometteurs quant aux capacités de resolution search et nous encouragent à en développer une version plus spécifique, utilisant la structure du RCPSP.

5 Conclusion

Nous montrons l'efficacité de resolution search, une procédure originale de résolution exacte, vis-à-vis d'une PSE classique, sur une formulation linéaire en variables binaires du RCPSP. Au vu de la faiblesse des programmes linéaires, mais surtout de l'existence de méthodes arborescentes très perfectionnées pour le RCPSP, nous allons maintenant, dans un second temps, tenter de nous inspirer de ces procédures et d'intégrer leurs schémas de branchement dans des versions plus élaborées de resolution search. Nous nous intéressons en particulier à trois PSE originales pour le RCPSP : l'une suivant un schéma chronologique (Demeulemeester et Herroelen [5]), une autre basée sur les *schémas d'ordonnancement* (Brucker et al. [2]) et enfin la méthode de Carlier et Latapie de réduction des intervalles des activités [3]. L'intégration de ces méthodes à resolution search présente en fait trois difficultés. La première est la représentation des noeuds de l'arbre de recherche en vecteurs 0-1 (évident pour [5] et [2], moins pour [3]). Il s'agit aussi d'adapter la technique de branchement pour pouvoir brancher à partir de n'importe quel noeud de l'arbre (par exemple pour [5], le noeud u_F, construit à partir de la famille F dans resolution search, ne représente pas nécessairement un ordonnancement partiel (chronologique) mais plutôt un ordon-

nancement « à trous »). Enfin, il s'agira de gérer les remontées (backtracking) successives dans l'arbre.

Références

1. Brucker P., Knust S., Schoo A., Thiele O. : A branch and bound algorithm for the resource-constrained project scheduling problem, European Journal of Operational Research, 107, 272–288 (1998)
2. Brucker P., Drexl A., Möhring R., Neumann K., Pesch E. : Resource-constrained project scheduling : Notation, classification, models, and methods, European Journal of OperationalResearch, 112, 3–41 (1999)
3. Carlier J., Latapie B. : Une méthode arborescente pour résoudre les problèmes cumulatifs, RAIRO, 25, 311-340 (1991)
4. Chvàtal V. : Resolution Search, Discrete Applied Mathematics, 73, 81–99 (1997)
5. Demeulemeester E., Herroelen W. : New benchmark results for the resource-constrained project scheduling problem, Management Science, 43(11), 1485–1492 (1997)
6. Demeulemeester E., Herroelen W. : Project Scheduling, Kluwer Academic Publishers, Boston (2002)
7. Kolisch R., Sprecher A., Drexl A. : Characterization and generation of a general class of RCPSP, Management Science, 41, 1693–1703 (1995)
8. Mingozzi, A., Maniezzo V., Ricciardelli S., Bianco L. : An exact algorithm for the multiple resource-constrained project scheduling problem based on a new mathematical formulation, Management Science, 44, 714–729 (1998)
9. Pritsker A.A., Watters L.J., Wolfe P.M. : Multi-project scheduling with limited resources : a zero-one programming approach, Management Science, 16, 93–108 (1969)