Une contrainte globale pour l'ordonnancement des transferts de données dans les missions spatiales

J. Rouzot[1,2], C. Artigues[1], P. Garnier[2], E. Hebrard[1], P. Lopez[1]

[1] LAAS-CNRS, Université de Toulouse, CNRS, Toulouse
[2] IRAP, Université de Toulouse, CNRS, UT3, CNES, Toulouse

Résumé

Dans le cadre des missions spatiales où les ressources à bord sont limitées, il est essentiel de transférer les données acquises de manière efficace afin de limiter le risque de saturation des mémoires. Nous nous intéressons ici en particulier à l'ordonnancement des transferts de données de la mission Rosetta de l'ESA (l'agence spatiale européenne). Ce problème a été résolu par des méthodes heuristiques qui ont donné de bons résultats, voire des résultats optimaux dans certains cas. Dans la lignée d'un travail précédent, nous proposons dans cet article un modèle de programmation par contraintes pour résoudre exactement ce problème, nous proposons une nouvelle contrainte globale pour le partage de bande passante en fonction de priorités, ainsi qu'une contrainte globale pour les classements de type Dense Ranking.

Des premiers résultats montrent la pertinence et l'efficacité d'une telle approche exacte pour la résolution de ce problème. Un des buts dans la conception de ces contraintes est leur généricité, afin qu'elles puissent être appliquées à toute mission spatiale nécessitant des transferts de données, ainsi qu'à tout problème nécessitant un classement Dense Ranking.

Mots-clés

Ordonnancement, Programmation par contraintes, Contrainte globale, Mission d'exploration spatiale, Dense Ranking

1 Introduction

La mission Rosetta a été lancée par l'Agence spatiale européenne (ESA) en 2004 dans le but d'analyser la comète 67P/Tchourioumov-Gerassimenko, et a abouti en 2014. Cette mission est représentative de nombreuses missions soutenues par le "*Deep Space Network*", et l'étudier permet donc de concevoir des systèmes de planification efficaces pour les missions futures.

Nous nous intéressons ici au problème du vidage des données de l'orbiteur Rosetta, défini par Chien, Rabideau et al. [2, 5], similaire au problème de vidage des données de l'atterrisseur Philae [7]. Dans les deux cas, un certain nombre d'activités, qui produisent des données temporairement stockées dans la mémoire de leur instrument, doivent être ordonnancées. De plus, un ordre de priorité doit être établi pour chacune des fenêtres de visibilité entre la sonde et la Terre. Cet ordre de priorité dicte le partage de bande passante entre les différents instruments lors du transfert de données pendant cette fenêtre. L'objectif est de minimiser le pic d'utilisation maximal des mémoires (i.e. maximiser la marge de sécurité, où la marge de sécurité est le ratio entre la capacité de la mémoire et son remplissage maximal au cours de la mission).

Dans le cas de Philae, l'ordre de priorité est considéré fixe, et les variables de décision correspondent à l'ordonnancement des activités. Dans le cas de Rosetta, à l'inverse, l'ordonnancement des activités est une entrée, et les variables correspondent à l'ordre de priorité pour le transfert des données.

Il s'agit d'un ordre partiel, plus précisément un *classement* : les mémoires de même priorité se partagent la bande passante grâce à un algorithme de type Round-Robin (file d'attente circulaire). Ainsi, les mémoires avec la plus forte priorité peuvent utiliser toute la bande passante initiale, puis la bande passante restante est redistribuée jusqu'à ce qu'il n'en reste plus, ou que les mémoires soient toutes vides. On considère ici que le remplissage des mémoires est complètement déterminé par la planification des expériences scientifiques et donc connu à l'avance. Les priorités ne peuvent être changées qu'en début de visibilité, mais sont déterminées en amont, même si elles peuvent être amenées à être recalculées au cours de la mission.

Comme la relation entre l'utilisation des mémoires et les priorités est complexe, utiliser une modélisation de programmation par contraintes classique est très peu efficace. Afin d'accélérer la résolution, nous avons donc développé les contraintes globales suivantes :

- La contrainte `PRIORITY TRANSFER` permet de déterminer le pic maximal d'utilisation des mémoires au cours d'une fenêtre de vidage et l'utilisation des mémoires à la fin de cette fenêtre en fonction d'une affectation de priorités. Cette contrainte permet également d'inférer de nouvelles bornes sur les variables de priorité en fonction de la borne supérieure du pic d'utilisation maximal courant. Cette contrainte utilise les algorithmes décrits par Hebrard et al. [3].
- La contrainte `DENSE RANKING` permet de filtrer efficacement les domaines des variables de priorité pour obtenir un classement sans symétries.

Nous décrivons dans cet article les contraintes globales précédentes, puis un algorithme de calcul rapide de bornes inférieures sur les mémoires et les pics d'utilisation pour chaque fenêtre de vidage. Nous concluons sur l'intérêt de ces contraintes globales ainsi que sur les perspectives pour améliorer le modèle.

2 Modélisation du problème

Dans le problème du vidage des données de Rosetta, ou *overlapping Memory Dumping Problem (oMDP)* [6], on considère m fenêtres de vidage aux dates de début et fin $[s_j, e_j[_{\{j=1..m\}}$ pendant lesquelles la bande passante à partager entre les mémoires est une constante δ_j. On a n mémoires concurrentes, avec une capacité limitée. On note $r_{i,j}$ l'usage maximum de la mémoire i par rapport à sa capacité au cours de la fenêtre j. Pour ce problème, nous considérons que la fonction de remplissage de chaque mémoire au cours du temps est une fonction constante par morceaux. Cela implique que l'évolution de l'utilisation des mémoires au cours du temps suit une fonction linéaire par morceaux, puisque le transfert des données est aussi constant par morceaux.

Nous définissons les priorités de la façon suivante :

— Des groupes de priorité sont définis, pouvant prendre une valeur de 1 à n. Le groupe 1 est le plus prioritaire, tandis que le groupe n est le moins prioritaire.

— Les mémoires sont affectées à un groupe de priorité pour chaque fenêtre de vidage ; plusieurs mémoires peuvent être affectées au même groupe.

— Les mémoires au sein d'un même groupe de priorité partagent la bande passante disponible grâce à un algorithme de type Round-Robin.

Nous proposons un modèle de programmation par contraintes pour résoudre les instances du oMDP. Dans notre modèle, nous séparons les instances par fenêtre de visibilité j et nous introduisons les variables suivantes :

— $p_{i,j}$ représente le groupe de priorité de la mémoire i au cours de la fenêtre j.

— $mem^s_{i,j}$ et $mem^e_{i,j}$ représentent respectivement l'utilisation de la mémoire i au début et à la fin de la fenêtre j.

— $r_{i,j}$ représente le pic d'utilisation maximum de la mémoire i au cours de la fenêtre j.

— $rmax$ représente le pic d'utilisation maximum des mémoire sur l'instance globale.

L'objectif est de minimiser l'usage maximal des mémoires sur l'instance : $\min\ rmax$.

La figure 1 montre l'évolution de l'utilisation des mémoires au cours du temps pour une solution particulière (affectation de priorités pour chaque fenêtre de vidage). Les zones vertes représentent les fenêtres de visibilité, pendant lesquelles les données des mémoires peuvent être transférées sur Terre à un taux δ_j, partagé entre les mémoires en fonction de leur priorité. Pendant ces fenêtres de visibilité, les mémoires ont une priorité fixe, représentée sur la figure 1 en rouge. Pour chaque mémoire, on représente le remplissage

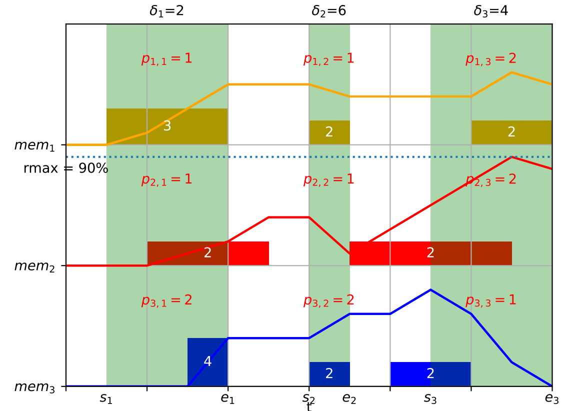

FIGURE 1 – Utilisation des mémoires au cours du temps et pic maximum pour une affectation de priorités.

au cours du temps par des rectangles de couleur où la hauteur représente le taux (constant) de remplissage. L'évolution de l'utilisation de chaque mémoire est représentée par une fonction linéaire par morceaux. Dans cet exemple, la capacité de chaque mémoire est de 10 unités ; le pic maximum est donc atteint pour la mémoire 2, pour une valeur de 9 unités, soit $rmax = r_{2,3} = \frac{9}{10} = 90\,\%$.

3 Contraintes globales

La principale difficulté pour oMDP est d'exprimer efficacement la relation entre une affectation de priorité et l'évolution de l'utilisation des mémoires résultant. Cette difficulté a motivé le développement de la contrainte globale `PRIORITY TRANSFER` qui, pour chaque fenêtre j, lie les variables $p_{i,j}$, $mem^e_{i,j}$, $mem^s_{i,j}$ et $r_{i,j}$.

De plus, la modélisation des variables de priorité a l'avantage d'être directe par rapport à des priorités relatives entre chaque mémoire, mais comporte de nombreuses symétries qui doivent être cassées pour accélérer la résolution. Nous présentons la contrainte globale `DENSE RANKING` qui permet de briser ces symétries efficacement.

3.1 Contrainte Priority Transfer

Une fois que les variables $p_{i,j}$ et $mem^s_{i,j}$ sont fixées, on peut calculer l'évolution des mémoires au cours du temps dans la fenêtre j. On utilise l'algorithme `SIMULATION` [3] pour déterminer les valeurs des variables $mem^e_{i,j}$ et $r_{i,j}$ au cours de la résolution. Cependant, cet algorithme ne permet pas de faire de propagation sur les priorités.

Comme l'objectif de notre modèle est de minimiser le pic d'utilisation maximum de la mémoire, il est possible d'éliminer certaines valeurs dans les domaines des variables de priorité, en fonction de la valeur de la borne supérieure de la variable $rmax$ au cours de la résolution. En effet, on peut montrer que certaines affectations de priorité mènent nécessairement à un dépassement du pic maximum courant, grâce à l'algorithme `SINGLE WINDOW` [3].

Plus précisément, cet algorithme construit une affectation

de priorités partielles pour un pic d'utilisation maximum donné. L'affectation est partielle car elle ne donne pas une valeur stricte pour les variables de priorité, mais un ordre de priorité relatif entre les mémoires.
Puisque cet ordre partiel doit être respecté pour satisfaire le pic maximal courant sur la fenêtre, on peut inférer de nouvelles bornes sur les variables de priorité sur la fenêtre. L'ordre partiel nous donne un ensemble de relations $p_{i,j} < p_{i',j}$ qui permettent de filtrer les bornes des variables de priorité.
La complexité de l'algorithme SINGLE WINDOW étant en $O(n^3 \log(n))$, la contrainte PRIORITY TRANSFER permet donc un filtrage des domaines des variables de priorité aussi en $O(n^3 \log(n))$ puisque la mise à jour des domaines, après avoir construit l'ordre partiel entre les mémoires, se fait en $O(n^2)$.

3.2 Contrainte Dense Ranking

Il existe plusieurs façon d'effectuer des classements (ex. classement standard "1224", classement modifié "1334", etc.) dont certains ont déjà été étudiés en programmation par contraintes, comme le classement standard [1].
Dans notre modèle, les priorités sont définies pour chaque mémoire par l'appartenance à un groupe de priorité et on autorise plusieurs mémoires à avoir une priorité égale. Afin d'éviter les symétries dans les solutions réalisables, nous avons décidé d'adopter un classement *Dense Ranking*.
Les règles suivantes doivent être respectées pour qu'une solution soit considérée valide à l'égard d'un *Dense Ranking* :

- Le groupe de priorité 1 doit être non vide.
- Si le groupe de priorité $i + 1$ est non vide, alors le groupe de priorité i doit être non vide.

Pour éliminer les solutions qui ne respectent pas ces règles, nous présentons la contrainte globale DENSE RANKING qui permet un filtrage efficace de type cohérence de borne des priorités. L'algorithme de filtrage des domaines est découpé en deux parties :

- L'algorithme PRIORITY SAT vérifie si les domaines des variables de priorité permettent de construire une solution qui satisfait la contrainte DENSE RANKING.
- L'algorithme SYMMETRY FILTERING permet de réaliser la cohérence de bornes en utilisant l'algorithme précédent comme oracle.

Dans oMDP, on a n variables de priorité avec des domaines $[1, ..., n]$. Les domaines des variables de priorité sont valides si et seulement si l'on peut construire une solution en respectant les règles de *Dense Ranking*. L'algorithme PRIORITY SAT (algorithme 1) permet de vérifier l'existence d'une solution à partir des domaines des variables de priorité en $O(n \ \log(n))$. Les variables de priorité p sont d'abord triées par borne inférieure croissante (ligne 2) puis, pour chaque rang de priorité, on ajoute dans un tas h, ordonnées par borne supérieure croissante, toutes les variables de priorité dont la borne inférieure est égale au rang courant (lignes 5-8). S'il est possible d'affecter au moins une variable au rang courant à chaque étape (lignes 9-10), jusqu'à ce que toutes les variables soient affectées, il existe une solution réalisable pour les domaines courants. Si l'on ne peut pas traiter un des rangs (lignes 12-13), et qu'il reste des variables qui n'ont pas été affectées, alors il n'existe pas de solution réalisable pour les domaines courants. Pour chaque rang, il faut retirer du tas les variables qui n'ont pas été affectées, mais dont la borne supérieure est égale au rang courant, ce qui revient à affecter la même priorité à plusieurs variables (ligne 13-14).
En effectuant une recherche dichotomique sur les domaines des variables de priorité, on peut donc faire le filtrage des bornes en $O(n^2 \ \log(n)^2)$.
La contrainte globale Dense Ranking peut s'exprimer de manière équivalente avec la contrainte AtLeastNValues(N, X), basée sur SoftAllDiff(X), qui assure que les variables de l'ensemble X prennent au moins N valeurs distinctes [4]. En effet, Dense Ranking(X) est équivalent à AtLeastNValues(Max(X), X). Cependant, l'algorithme pour trouver un support pour la contrainte AtLeastNValues se base sur le couplage de cardinalité maximum dans un graphe biparti. Avec $n = |X|$ le nombre de variables et $m = \sum |D(X)|$, l'algorithme est en $O(m \ n^{\frac{1}{2}})$, soit en $O(n^{\frac{5}{2}})$ pour nos domaines. Le filtrage complet des domaines se fait aussi en $O(n^{\frac{5}{2}})$, ce qui est moins efficace qu'avec notre méthode. De plus, le test de satisfaisabilité est bien plus efficace avec la contrainte globale Dense Ranking.

Algorithm 1 PRIORITY SAT

```
1: Input : p
2: Trier p par borne inférieure
3: i ← 0, current ← −1
4: while i < n do
5:    current ← current + 1
6:    while p_i.Lb() = current do
7:       h.Add(p_i)
8:       i ← i + 1
9:    if not h.Empty() then
10:      h.RemoveMin()
11:   else
12:      return False
13:   while h.Min().Ub() = current do
14:      h.RemoveMin()
15: return True
```

4 Borne inférieure

Une borne inférieure sur les variables $mem^s_{i,j}$, $mem^e_{i,j}$ et $r_{i,j}$ est facile à déterminer en utilisant l'algorithme SIMULATION [3]. Du point de vue d'une mémoire i, le meilleur scénario possible est d'être plus prioritaire que les autres mémoires à chaque fenêtre de vidage. Ainsi, pour chaque mémoire i, en affectant la priorité $p_{i,j} = lb(p_{i,j})$ et $p_{i',j} = ub(p_{i',j}) \ \forall \ i' \neq i$, et en supposant que $mem^s_{i,j}$ est égale à sa borne inférieure on peut déterminer un meilleur cas d'utilisation de mémoire i en simulant cette affectation particulière de priorités, et donc une borne inférieure sur $mem^e_{i,j}$ et $r_{i,j}$. L'algorithme SIMULATION est de complexité $O(n \log(n))$, ce qui est relativement peu coûteux.

Instances	Dense Ranking	Temps moyen	Temps médian	% solutions optimales
2 mémoires, 2 fenêtres (*2m/2f*)	Non	**351 μs**	**341 μs**	100%
4 mémoires, 4 fenêtres (*4m/4f*)	Non	**170 ms**	169 ms	40%
6 mémoires, 6 fenêtres (*6m/6f*)	Non	**6 s**	**448 ms**	30%
2 mémoires, 2 fenêtres (*2m/2f*)	Oui	650 μs	572 μs	100%
4 mémoires, 4 fenêtres (*4m/4f*)	Oui	2 s	**12 ms**	**90%**
6 mémoires, 6 fenêtres (*6m/6f*)	Oui	28 s	6 s	**50%**

TABLE 1 – Temps et % des instances résolues à l'optimal avec et sans la contrainte Dense Ranking.

On peut donc déterminer une borne inférieure sur chaque variable $mem^s_{i,j}$, $mem^e_{i,j}$ et $r_{i,j}$ en $O(n \log(n))$.

5 Premiers résultats expérimentaux

Pour étudier l'utilité de la contrainte DENSE RANKING, nous avons généré 30 instances aléatoires de taille variable (2 mémoires et 2 fenêtres *2m/2f*, 4 mémoires et 4 fenêtres *4m/4f*, 6 mémoires et 6 fenêtres *6m/6f*). Nous résolvons ces instances avec et sans la contrainte globale DENSE RANKING avec la configuration suivante :

- Solveur : OR-Tools Original CP solver [1].
- Temps limite : 100 secondes.
- Configuration matérielle : 12th Gen Intel Core i7-1265U 100 MHz, 32 GB de RAM.

Les résultats expérimentaux montrent une forte augmentation du taux d'instances résolues à l'optimum. On observe cependant une augmentation globale du temps de résolution avec la contrainte DENSE RANKING.

6 Conclusion

Nous avons présenté deux nouvelles contraintes globales pour l'ordonnancement des transferts de données par priorités, et nous montrons l'efficacité de la contrainte DENSE RANKING pour améliorer la résolution. La contrainte PRIORITY TRANSFER permet un filtrage efficace des variables de priorité, mais nécessite de trouver des bornes supérieures sur le pic d'utilisation maximum pour être vraiment utile. Trouver des solutions de qualité est donc un point crucial pour l'amélioration du modèle. Nous dirigeons aujourd'hui notre recherche vers l'intégration d'heuristiques efficaces dans notre modèle de programmation par contraintes, en particulier celles décrites dans la référence [3], ainsi que sur le calcul de bornes de meilleure qualité. Ces améliorations feront l'objet d'une étude des performances du modèle sur des instances plus grandes. La construction d'une procédure plus efficace que celle présentée dans cet article pour faire le filtrage de bornes des variables de priorité pour la contrainte DENSE RANKING est aussi en cours d'étude. La contrainte sera testée sur des problèmes variés et comparée avec d'autres méthodes pour démontrer son efficacité.

Références

[1] Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, and Toby Walsh. Ranking constraints. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 705–711. IJCAI/AAAI Press, 2016.

[2] Steve Chien, Gregg Rabideau, Daniel Tran, Martina Troesch, Joshua Doubleday, Federico Nespoli, Miguel Perez Ayucar, Marc Costa Sitja, Claire Vallat, Bernhard Geiger, et al. Activity-based scheduling of science campaigns for the Rosetta orbiter. In *Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-15*, Buenos Aires, Argentina, July 2015.

[3] Emmanuel Hebrard, Christian Artigues, Pierre Lopez, Arnaud Lusson, Steve Chien, Adrien Maillard, and Gregg Rabideau. An efficient approach to data transfer scheduling for long range space exploration. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 4635–4641. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[4] Thierry Petit, Jean-Charles Régin, and Christian Bessiere. Specific filtering algorithms for over-constrained problems. In *Principles and Practice of Constraint Programming—CP 2001 : 7th International Conference, CP 2001 Paphos, Cyprus, November 26–December 1, 2001 Proceedings 7*, pages 451–463. Springer, 2001.

[5] Gregg Rabideau, Steve Chien, M. Galer, Federico Nespoli, and Manuel Costa. Managing spacecraft memory buffers with concurrent data collection and downlink. *Journal of Aerospace Information Systems*, 14(12) :637–651, 2017.

[6] Gregg Rabideau, Steve Chien, Federico Nespoli, and Manuel Costa. Managing spacecraft memory buffers with overlapping store and dump operations. In *Workshop on Scheduling and Planning Applications, International Conference on Automated Planning and Scheduling (SPARK, ICAPS 2016)*, London, UK, June 2016.

[7] Gilles Simonin, Christian Artigues, Emmanuel Hebrard, and Pierre Lopez. Scheduling scientific experiments for comet exploration. *Constraints An Int. J.*, 20(1) :77–99, 2015.

1. https ://developers.google.com/optimization/cp/original_cp_solver