MPRO – ECMA
Rapport Exercice 2

Léa Blaise & Alexis Reymann

March 8, 2018

Introduction

La deuxième partie du projet concernant le Problème d'Affectation Généralisée (PAG) consistait en l'implémentation du modèle compact, d'une heuristique, d'une génération de colonnes, et d'un algorithme de résolution exact du problème. Excepté pour le modèle compact, que nous avons fait en OPL, nous avons choisi le langage de programmation C++ pour tout implémenter. Les deux membres de notre binôme ne codant pas sous le même système d'exploitation, nous avons du rendre notre projet multi-plateforme, ce qui signifie que le projet peut aussi bien tourner sous Windows que sous Linux. Le projet peut être ouvert et lancé par Visual Studio ; et nous avons créé un Makefile pour compiler sous Linux.

Nous avons également décidé d'implémenter une fonction `parse(string s)` qui prend en argument le nom du fichier `.dat` contenant l'instance souhaitée. Cette fonction copie toutes les données de l'instance dans des variables globales, donc accessibles par toutes les fonctions. Par ailleurs, que ce soit pour l'heuristique ou pour la génération de colonnes, l'utilisateur doit choisir une durée pendant laquelle le programme va s'exécuter.

Notre binôme a utilisé Git pour mieux organiser le projet. La liste des commits peut être trouvée ici : GitHub-Commits. La liste des releases peut être trouvée ici : GitHub-Releases. Le dernier release correspond à l'état final du projet tel que nous vous l'avons rendu par mail.

Recommandations :

- Nous avons créé un `Makefile` qui permet de compiler aisément nos programmes sous Linux. Utilisez les commandes `make heuristique`, `make colGen`, `make branchCut` ou bien `make all` pour compiler toutes les parties du projet en une fois.
- Notre algorithme de Branch&Cut est très gourmand en mémoire. Visual Studio est capable d'arrêter l'exécution du programme s'il n'y a plus de place dans la RAM pour ajouter des coupes. Par contre, sous Linux, le système d'exploitation va tenter de faire de la place dans la RAM en copiant des données sur le disque dur (SWAP), ce qui est extrêmement lent et peut monopoliser la puissance de calcul de la machine. Nous vous conseillons donc de faire tourner cet algorithme sous Windows, ou d'utiliser la commande `$ timeout 5s ./branchCut` afin de limiter à 5 secondes le temps d'execution du programme.
- Dans le fichier `colGen.cpp`, nous avons commenté une fonction `main` alternative qui execute l'algorithme de génération de colonnes sur toutes les instances les unes à la suite des autres. Pour executer cette version de notre programme, décommentez cette fonction et recompilez.

1 Résolution du modèle compact avec CPLEX

Nous avons implémenté le modèle compact fourni dans l'énoncé du projet en utilisant OPL.

```
int n = ...;
int m = ...;

int c[1..m][1..n] = ...;
int a[1..m][1..n] = ...;
int b[1..m] = ...;

dvar int x[1..m][1..n] in (0..1);


minimize
  sum (i in (1..n)) sum (j in (1..m)) c[j][i] * x[j][i];
subject to {
  forall (i in (1..n))
    sum (j in (1..m)) x[j][i] == 1;
  forall (j in (1..m))
    sum (i in (1..n)) a[j][i] * x[j][i] <= b[j];
}
```

On peut constater, comme attendu, que la relaxation continue du problème ne fournit pas une borne inférieure d'excellente qualité.

Instance	Valeur optimale : solution entière (ou estimation si le temps de calcul est trop long)	Relaxation continue
GAP-a05100.dat	1698	1697.72
GAP-a20100.dat	1158	1157.08
GAP-b10100.dat	1407	1400.67
GAP-b10200.dat	2827	2815.05
GAP-b20100.dat	1166	1155.18
GAP-c05100.dat	1931	1923.97
GAP-c10100.dat	1402	1387.01
GAP-c40400.dat	4244	4231.98
GAP-c401600.dat	$17142 \leq x \leq 17148$	17139.33
GAP-d20100.dat	$6162 \leq x \leq 6224$	6142.53
GAP-d20200.dat	$12224 \leq x \leq 12279$	12217.69
GAP-d401600.dat	$97105 \leq x \leq 97300$	97105.00
GAP-d801600.dat	$97034 \leq x \leq 97491$	97034.00

Table 1: Résultats du modèle compact

2 Heuristique

Notre heuristique se décompose en deux parties : trouver une solution réalisable, puis l'améliorer en appliquant une procédure de recherche locale.

1. Trouver une solution réalisable : nous appliquons une simple heuristique gloutonne, consistant à affecter chaque tâche à la machine sur laquelle cette tâche coûte le moins cher en ressources. Si à une étape de l'algorithme, aucune machine n'a suffisamment de ressources restantes pour accepter une tâche i, celle-ci est affectée à une machine virtuelle, numérotée $m + 1$.

   Pour avoir une meilleure probabilité de trouver ainsi une solution immédiatement réalisable, nous trions les tâches au préalable : celles qui sont potentiellement les plus coûteuses (les i qui ont les plus grandes valeurs de $\max_j(a_{i,j})$) sont traitées en premiers. Les situations similaires à la suivante sont alors évitées :

   - Les $n - 1$ premières tâches (non triées) nécessitent une faible quantité de ressources ($a_{i,j} = 1$ ou 2 par exemple) sur chaque machine, et sont affectées à la machine sur laquelle elles consomment le moins de ressources possible, saturant certaines machines.
   - La dernière tâche a un coût en ressources de 1 sur les machines saturées, et un coût très élevé sur les autres : elle doit être affectée à la machine virtuelle.

   Les instances données dans le cadre du projet sont assez "simples" pour qu'une solution réalisable soit obtenue directement à chaque fois, sans avoir à réparer la solution pour n'avoir aucune tâche affectée à la machine virtuelle. Nous avons toutefois implémenté une heuristique de recherche locale qui, tant qu'il existe des tâches affectées à la machine virtuelle, tente de faire des échanges et des déplacements de tâches (déjà affectées à des machines réelles), afin de libérer de l'espace sur les machines, et de pouvoir y placer les tâches non affectées jusqu'à présent.

2. Nous améliorons ensuite la solution obtenue à la suite de l'application de l'algorithme glouton avec un algorithme de recuit simulé : pendant un temps déterminé à l'avance, nous réalisons des déplacements de certaines tâches d'une machine vers une autre, ainsi que des échanges entre deux tâches affectées à deux machines différentes (en s'assurant que la nouvelle affectation reste réalisable), afin de diminuer le coût.

Le tableau suivant présente quelques résultats obtenus par notre heuristique (algorithme glouton seul, puis algorithme glouton suivi d'une seconde de recherche locale, et algorithme glouton suivi de cinq secondes de recherche locale), comparés aux solutions optimales, obtenues par résolution de la formulation compacte (ou comparés aux bornes inférieure et supérieure obtenues par le début de la résolution de la formulation compacte, lorsque celle-ci prenait trop de temps). Les résultats complets sont disponibles dans le fichier `Resultats.xlsx`.

Instance	Solution optimale (ou bornes)	Algorithme glouton seul	Recherche locale 1s	Recherche locale 5s
GAP-a05100.dat	1698	3319	1733	1743
GAP-a20100.dat	1158	2871	1179	1167
GAP-b10100.dat	1407	2799	1558	1542
GAP-b10200.dat	2827	6023	3730	3673
GAP-b20100.dat	1166	2902	1251	1285
GAP-c05100.dat	1931	3293	2308	2492
GAP-c10100.dat	1402	3185	1876	1719
GAP-c40400.dat	4244	12372	5010	4816
GAP-c401600.dat	$17142 \leq x \leq 17148$	47257	21995	22285
GAP-d20100.dat	$6162 \leq x \leq 6224$	10630	6897	6848
GAP-d20200.dat	$12224 \leq x \leq 12279$	21251	13717	13808
GAP-d401600.dat	$97105 \leq x \leq 97300$	172654	108448	108491
GAP-d801600.dat	$97034 \leq x \leq 97491$	175078	107381	106936

Table 2: Résultats de l'heuristique de recuit simulé

On constate que la première solution issue de l'algorithme glouton est extrêmement mauvaise, cela s'explique par le fait qu'elle est construite sans aucune considération de coût. Par ailleurs, exécuter l'heuristique 5 secondes au lieu d'une seule ne produit pas toujours une meilleure solution. Cela signifie que le recuit simulé tombe rapidement dans un minimum local duquel il est difficile de s'extirper.

3 Génération de colonnes

Nous avons décidé de démarrer l'algorithme, pour chaque instance, à partir d'une solution réalisable donnée par notre heuristique en 1 seconde. Le problème maître restreint commence donc avec une variable par machine : l'affectation trouvée pour chacune par notre heuristique.

La génération de colonnes se lance ensuite pour une durée déterminée par l'utilisateur (ou s'arrête lorsque tous les coûts réduits sont positifs).

A chaque étape de la procédure de génération de colonnes, m problèmes auxiliaires (correspondant à ceux de la modélisation faite lors de l'exercice 1) sont résolus. Chacun correspond à un problème de sac-à-dos, et renvoie une affectation réalisable pour la machine correspondante : il s'agit de la colonne de coût réduit minimal. m nouvelles variables sont alors ajoutées au problème maître restreint, correspondant aux m affectations trouvées par la résolution des problèmes auxiliaires. Leurs coefficients dans l'objectif et dans les différentes contraintes sont ensuite calculés en fonction des tâches présentes dans les affectations auxquelles elles correspondent. Pour éviter une consommation inutile de mémoire, nous ne gardons pas d'autre trace de l'expression explicite des affectations ainsi ajoutées (nous ne conservons pas l'expression des $s_{i,j}^k$ correspondant à ces affectations, pour reprendre les notations de l'exercice 1).

Les bornes inférieures obtenues à partir de notre algorithme de génération de colonnes sur quelques instances sont présentées dans le tableau suivant (les résultats complets sont disponibles dans le fichier `Resultats.xlsx`) :

Instance	Borne inférieure de la génération de colonnes (après 5 minutes)	Borne inférieure de la relaxation continue du modèle compact
GAP-a05100.dat	1699	1697.72
GAP-a20100.dat	1158	1157.08
GAP-b10100.dat	1406	1400.67
GAP-b10200.dat	2956	2815.05
GAP-b20100.dat	1166	1155.18
GAP-c05100.dat	1929	1923.97
GAP-c10100.dat	1399	1387.01
GAP-c40400.dat	4422	4231.98
GAP-c401600.dat	17638	17139.33
GAP-d20100.dat	6176	6142.53
GAP-d20200.dat	12230	12217.69
GAP-d401600.dat	102411	97105.00
GAP-d801600.dat	102279	97034.00

Table 3: Résultats de la génération de colonnes

4 Branch-and-Cut

Pour la dernière partie du projet, concernant la résolution exacte du problème, nous avons choisi d'implémenter un algorithme de Branch-and-Cut, basé sur la formulation compacte du problème.

Stratégie de branchement : A chaque nœud k de l'arbre de résolution du Branch-and-Cut, nous choisissons de brancher sur la variable dont la valeur est la plus proche de 0,5. Notons $x_{i,j}$ cette variable. Deux sous-problèmes sont alors ajoutés à partir du nœud k à la liste des sous-problèmes à résoudre : pour le premier, on impose $x_{i,j} = 1$, c'est-à-dire que l'on force la tâche i à être affectée à la machine j, et pour le deuxième, on impose $x_{i,j} = 0$, c'est-à-dire que l'on force la tâche i à être affectée à n'importe quelle machine sauf j.

Coupes : Nous avons décidé d'utiliser des inégalités de couvertures en tant que coupes à utiliser dans notre algorithme de Branch-and-Cut. Ces coupes s'expriment, pour chaque machine j, sous la forme suivante :

$$\sum_{i \in C} x_{i,j} \leq |C| - 1 \qquad \text{où } C \text{ est un ensemble de tâches tel que } \sum_{i \in C} a_{i,j} > b_j$$

Algorithme de séparation : En remarquant que l'inégalité $\sum_{i \in C} x_{i,j} \leq |C| - 1$ peut aussi s'écrire $\sum_{i \in C} (1 - x_{i,j}) \geq 1$, on peut déduire une méthode de séparation des inégalités de couvertures.

On résout :

$$(P_{\text{couv}_j}) \quad \begin{cases} \min\limits_{z} & \sum\limits_{i=1}^{n} (1 - \hat{x_{i,j}}) z_i \qquad \text{avec } \hat{x_{i,j}} \text{ la valeur de la variable } x_{i,j} \text{ au nœud courrant } k \\ \text{s.c.} & \sum\limits_{i=1}^{n} a_{i,j} z_i > b_j \end{cases}$$

Si la valeur de la solution optimale de (P_{couv_j}) est strictement inférieure à 1, il existe une couverture C sur la machine j, donnée par la valeur des z_i, telle que l'inégalité de couverture qui lui correspond est violée : on peut l'ajouter à l'ensemble des contraintes pour améliorer la qualité de la relaxation continue du modèle.

Sinon, toutes les inégalités de couvertures sont respectées pour la machine j, et il est donc inutile d'en ajouter une.

Etapes de notre algorithme : Nous avons implémenté notre algorithme de Branch-and-Cut de la façon suivante :

- Un problème maître est défini : il correspond à la formulation compacte du problème d'affectation généralisée.
- L'ensemble des sous-problèmes (nœuds de l'arbre du Branch-and-Cut) est initialisé : il contient uniquement le problème maître défini au point précédent. L'ensemble de ces sous-problèmes est implémenté comme une pile : l'arbre est parcouru en profondeur, afin de trouver une solution réalisable rapidement.
- Lorsqu'on explore un nœud de l'arbre :
  - On résout le sous-problème correspondant à ce nœud.
  - S'il n'admet pas de solution réalisable en variables continues, le nœud est abandonné.
  - Sinon, si la solution optimale en ce nœud est entière, on met à jour la borne supérieure du problème : on a obtenu une solution réalisable.
  - Sinon, si la solution optimale (en variables continues) a une valeur supérieure à la meilleure borne supérieure (solution réalisable en variables entières) rencontrée jusqu'à présent : on abandonne le nœud car on sait qu'il ne permettra pas de trouver la solution optimale.

– Sinon, on branche : on ajoute deux sous-problèmes, tels que décrits ci-dessus dans le paragraphe "stratégie de branchement", et on tente d'ajouter une coupe.
On choisit pour cela un indice $j \leq m$ au hasard, et on résout (P_{couv_j}). On ajoute l'inégalité valide ainsi obtenue (sauf si la résolution de (P_{couv_j}) a montré que toutes les inégalités de couverture étaient déjà respectées sur la machine j) à l'ensemble des contraintes. Cette coupe sera donc présente dans tous les sous-problèmes que l'on devra résoudre par la suite (même ceux qui sont déjà présents dans la pile des sous-problèmes).

Nous avons constaté que notre approche Branch-and-Cut pour la résolution exacte du problème à partir de sa formulation compacte était bien moins rapide que celle faite automatiquement par Cplex. Ce résultat n'est pas étonnant : en effet, on peut penser qu'un solveur comme Cplex utilise plus de coupes, plus efficaces, que notre algorithme, et que son implémentation, améliorée sur plusieurs années, est bien plus efficace que la nôtre.

Sur certaines instances, nous avons rencontré des problèmes de mémoire dus à l'ajout de trop nombreuses coupes, bien que celles-ci soient ajoutées au moyen de la méthode "addCuts", spécialement conçue pour l'ajout d'inégalités valides et plus optimisée du point de vue de la mémoire selon la documentation officielle de Cplex. Pour pallier à ce problème, nous avons choisi de ne pas ajouter une nouvelle coupe à chaque nœud visité, mais de ne le faire qu'avec une certaine probabilité. Les problèmes résolus sont alors moins contraints, ce qui impose de rencontrer plus de nœuds et brancher plus longtemps sur ces nœuds, mais permet de résoudre plus vite chacun des sous-problèmes rencontrés, et de ne pas saturer la mémoire.

Malheureusement, certaines instances restent trop longues à résoudre, et nécessitent trop de mémoire. Nous n'avons donc pu résoudre que les instances de type "a". Ces instances sont toutefois peu intéressantes du point de vue du Branch-and-Cut : en effet, aucune coupe n'est ajoutée car toutes les inégalités de couvertures sont immédiatement vérifiées.

Résultats : Les résultats que nous avons obtenus sont les suivants :

Instance	Meilleure solution entière	Borne inférieure (à la racine)	Nombre de nœuds explorés	Nombre solutions entières rencontrées	Nombre de de nœuds abandonnés
GAP-a05100	1698	1697,72	19	2	8
GAP-a05200	3235	3234,74	3	1	1
GAP-a10100	1360	1358,56	25	1	12
GAP-a10200	2623	2623,00	9	1	4
GAP-a20100	1158	1157,08	11	1	5
GAP-a20200	2339	2337,38	6403	31	3171

Table 4: Résultats du Branch & Cut

Conclusion

- En ce qui concerne notre heuristique, on remarque qu'un recuit simulé assez simple à implémenter suffit à obtenir des solutions très correctes sur la plupart des instances. On remarque également qu'une execution de 1 ou 5 secondes ne change pas beaucoup la valeur de la solution finale, et peut même être moins bonne lors d'une exécution plus longue.
- Pour la génération de colonnes, nous avons fait tourner l'algorithme sur toutes les instances pendant 5 minutes au maximum. Dans certaines instances, un tel temps d'execution ne permet pas d'obtenir une borne inférieure du problème. Les valeurs que nous donnons sont donc peu significatives, il aurait fallu laisser l'algorithme s'excuter jusqu'à atteindre les conditions d'arrêt sur les coûts réduits. Or, même sur les plus petites instances cela peut prendre plus d'une dizaine de minutes.
- Notre algorithme de Branch&Cut est vraisemblablement moins bon que celui de Cplex. Même si notre algorithme est capable d'obtenir la solution exacte du problème, il lui faut pour cela beaucoup de temps et beaucoup de coupes qui prennent trop de place dans la mémoire vive d'un ordinateur. Par ailleurs, les coupes de couverture sont assez peu performantes, notamment sur les petites instances où toutes les solutions réalisables les vérifient.