Clustering avec la minimisation de la somme des carrés par la programmation par contraintes

Thi-Bich-Hanh Dao, Khanh-Chuong Duong, Christel Vrain
Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022, F-45067, Orléans, France
{thi-bich-hanh.dao, khanh-chuong.duong, christel.vrain}@univ-orleans.fr

Résumé

Le clustering sous contraintes utilisateur a connu un essor important en fouille de données. Dans les dix dernières années, beaucoup de travaux se sont attachés à étendre les algorithmes classiques pour prendre en compte des contraintes utilisateur, mais ils sont en général limités à un seul type de contraintes. Dans de précédents travaux, nous avons proposé un cadre générique et déclaratif, fondé sur la programmation par contraintes, qui permet de modéliser différentes tâches de clustering sous contraintes. L'utilisateur peut spécifier un parmi plusieurs critères d'optimisation et combiner différents types de contraintes. Dans ce papier nous étendons le modèle pour traiter le critère le plus connu de clustering, qui est la minimisation de la somme des distances au carré des objets au centre de leur cluster. C'est un problème difficile et à notre connaissance, il existe une seule méthode exacte permettant d'intégrer des contraintes utilisateurs ; elle est fondée sur la programmation linéaire sur les entiers et la génération de colonnes. Nous développons un algorithme de filtrage pour la nouvelle contrainte spécifiant ce critère. Des expérimentations sur des bases de données classiques montrent que notre modèle obtient une meilleure performance que la méthode exacte fondée sur la programmation linéaire.

1 Introduction

La classification non supervisée (souvent appelée par le terme anglais "clustering") est une tâche importante en fouille de données. Elle vise à regrouper les données en un ensemble de classes ou clusters, de façon à ce que les objets d'une même classe aient une forte similarité entre eux et diffèrent fortement des objets des autres classes. Elle est souvent représentée par un problème d'optimisation. Différents critères d'optimisation existent, comme par exemple minimiser le diamètre maximal des clusters, ou maximiser la marge minimale entre clusters. Depuis les années 2000, des contraintes utilisateur sont introduites dans la tâche de clustering pour guider la recherche et pour atteindre les solutions souhaitées s'il en existe. L'extension avec des contraintes utilisateur se fait soit en adaptant les algorithmes classiques pour gérer les contraintes, soit en modifiant la distance entre données pour traduire les contraintes. Cependant, la plupart des approches ne garantissent pas la satisfaction de toutes le contraintes ou la qualité de la solution. De plus, chaque algorithme est spécifique pour un critère. Dans de précédents travaux [6, 7, 8], nous proposons un cadre général basé sur la programmation par contraintes pour modéliser le clustering sous contraintes utilisateur. Ce cadre intègre différents critères : minimiser le diamètre maximal des clusters, maximiser la marge minimale entre clusters ou minimiser la somme des dissimilarités intra-cluster. Différents types de contraintes utilisateurs sont également intégrés. Le cadre permet de trouver une solution qui est un optimum global et qui satisfait toutes les contraintes utilisateur, s'il en existe une. Dans ce papier, nous développons ce cadre pour intégrer le critère de minimisation de la somme des carrés WCSS (Within-Cluster Sum of Squares), qui est défini par la somme des distances au carré des objets de chaque cluster au centre du cluster.

Ce critère est le plus souvent utilisé dans les tâches de clustering pour définir des clusters homogènes. Sa popularité est dûe au fait que l'algorithme bien connu des k-moyennes optimise ce critère et trouve un optimum local. Trouver un optimum global pour ce critère est un problème NP-Difficile et trouver une bonne borne inférieure est aussi difficile [1]. La meilleure approche exacte pour ce critère est basée sur la programmation linéaire sur des entiers et la génération de colonnes [2]. Elle a été récemment généralisée pour intégrer des contraintes utilisateur [3]. Dans cet ar-

ticle, nous proposons un calcul de borne inférieure et développons un algorithme de filtrage pour une nouvelle contrainte *wcss* qui caractérise ce critère. Des expérimentations sur des bases de données classiques montrent que notre approche basée sur la programmation par contrainte a une meilleure performance que l'approche fondée sur la programmation linéaire sur des entiers et la génération de colonnes. De plus, ce critère s'intègre dans un cadre général, qui permet à l'utilisateur de construire différentes tâches de clustering en combinant différents types de contraintes utilisateur.

Dans la suite du papier, nous présentons, en section 2, des notions préliminaires sur le clustering sous contraintes utilisateur et des travaux connexes. Nous détaillons le modèle en programmation par contraintes dans la section 3 et l'algorithme de filtrage pour la nouvelle contrainte *wcss* dans la section 4. Les expérimentations sont présentées dans la section 5 ; la section 6 contient une discussion sur des travaux futurs et la conclusion.

2 Préliminaires

2.1 Clustering sous contraintes utilisateur

Le clustering vise à regrouper les données en un ensemble de classes ou clusters, de façon à ce que les objets d'une même classe aient une forte similarité entre eux et diffèrent fortement des objets des autres classes. Le clustering est souvent défini par un problème d'optimisation, i.e. trouver une partition des objets qui optimise un critère donné. Considérons une base de données de n objets $\mathcal{O} = \{o_1, \ldots, o_n\}$ et une mesure de dissimilarité $d(o, o')$ entre deux objets $o, o' \in \mathcal{O}$. Une partition Δ des objets en k classes $C_1, \ldots, C_k$ est telle que : (1) pour tout $c \in [1, k]$ [1], $C_c \neq \emptyset$, (2) $\cup_c C_c = \mathcal{O}$ et (3) pour tout $c \neq c'$, $C_c \cap C_{c'} = \emptyset$. Le critère à optimiser peut être entre autres :

- minimiser le diamètre maximal des clusters, où le diamètre maximal d'une partition Δ est défini par
$$D(\Delta) = \max_{c \in [1,k]} \max_{o,o' \in C_c} d(o, o')$$
- maximiser la marge minimale entre clusters, où la marge minimale de Δ est définie par
$$S(\Delta) = \min_{c,c' \in [1,k]} \min_{o \in C_c, o' \in C_{c'}} d(o, o')$$
- minimiser la somme des dissimilarités intra-cluster WCSD (Within-Cluster Sum of Dissimilarities)
$$WCSD(\Delta) = \sum_{c \in [1,k]} \frac{1}{2} \sum_{o,o' \in C_c} d(o, o')$$
- minimiser la somme des carrés intra-cluster WCSS (Within-Cluster Sum of Squares)
$$WCSS(\Delta) = \sum_{c \in [1,k]} \sum_{o \in C_c} ||o - m_c||^2$$
où pour chaque $c \in [1, k]$, m_c est le centre du cluster C_c. Dans l'espace euclidien, lorsque la dissimilarité est définie par $d(o, o') = ||o - o'||^2$, nous avons la relation suivante :
$$WCSS(\Delta) = \sum_{c \in [1,k]} \frac{\frac{1}{2} \sum_{o,o' \in C_c} d(o, o')}{|C_c|}$$

La classification non supervisée est un problème NP-Difficile pour tous ces critères sauf pour la maximisation de la marge minimale. La plupart des algorithmes classiques utilisent des heuristiques et trouvent un optimum local [12]. Par exemple l'algorithme bien connu des k-moyennes (*k-means*) trouve un optimum local pour le critère WCSS. Cependant plusieurs optima peuvent exister et certains pouvant être plus proches de la solution recherchée. Afin de mieux modéliser la tâche d'apprentissage, mais aussi en espérant réduire la complexité, des contraintes définies par l'utilisateur sont ajoutées. On parle alors de la classification non supervisée sous contraintes utilisateur. Ces contraintes peuvent porter sur les classes (contraintes de classe) ou sur les objets (contraintes objet). Deux types de contraintes objets, introduites dans [16], sont couramment utilisés : must-link ou cannot-link. Une contrainte must-link sur deux objets spécifie qu'ils doivent être dans la même classe et une contrainte cannot-link spécifie qu'ils ne doivent pas être dans la même classe.

Les contraintes de classe imposent des restrictions sur les classes. La contrainte de capacité minimale (resp. maximale) exige que chaque classe ait au moins (resp. au plus) un nombre donné α (resp. β) d'objets : $\forall c \in [1, k],\ |C_c| \geq \alpha$ (resp. $\forall c \in [1, k],\ |C_c| \leq \beta$). La contrainte de diamètre maximal spécifie une borne supérieure γ sur le diamètre de chaque cluster : $\forall c \in [1, k], \forall o, o' \in C_c,\ d(o, o') \leq \gamma$. La contrainte de marge minimale, aussi appelée δ-contrainte dans [9], spécifie une borne inférieure δ sur la marge entre deux classes : $\forall c \in [1, k],\ \forall c' \neq c, \forall o \in C_c, o' \in C_{c'}, d(o, o') \geq \delta$. Une contrainte de densité, extension de la ϵ-contrainte introduite dans [9], demande que chaque point o ait dans son voisinage de rayon ϵ au moins m points de la même classe : $\forall c \in [1, k], \forall o \in C_c, \exists o_1, .., o_m \in C_c, o_j \neq o \wedge d(o, o_j) \leq \epsilon$.

2.2 Travaux connexes

Le clustering avec le critère de minimisation de WCSS est un problème NP-Difficile [1]. L'algorithme

1. Nous utilisons $[1, k]$ pour désigner l'ensemble $\{1, .., k\}$.

populaire des k-moyennes trouve un optimum local pour ce critère. Cet algorithme commence avec une partition aléatoire et répète deux étapes : (1) associer chaque objet au cluster dont le centre est le plus proche, (2) calculer les nouveaux centres des clusters. Ces deux étapes sont répétées jusqu'à ce qu'il n'y ait plus de changement d'affectation de points. Le résultat trouvé dépend donc de la partition aléatoire initiale. Quant aux méthodes exactes pour résoudre ce problème, nous pouvons citer l'algorithme de recherche par *branch-and-bound* répétitif RBBA (Repetitive Branch and Bound Algorithm) [5] et l'algorithme basé sur la programmation linéaire sur des entiers et la génération de colonnes [2]. Actuellement les approches exactes les plus efficaces peuvent résoudre ce problème pour des bases d'environ 2300 objets [2].

Lorsque des contraintes utilisateur s'ajoutent, l'algorithme COP-kmeans étend l'algorithme k-means pour traiter des contraintes must-link ou cannot-link [17]. Cet algorithme modifie l'algorithme des k-moyennes en affectant chaque objet au cluster dont le centre est le plus proche, parmi les clusters satisfaisant les contraintes. S'il n'existe pas de cluster satisfaisant les contraintes, l'algorithme s'arrête. Cet algorithme est glouton et très rapide, mais il peut ne pas trouver de solution satisfaisant toutes les contraintes même lorsqu'une telle solution existe [10].

Récemment, un cadre général pour minimiser ce critère WCSS et intégrant différents types de contraintes utilisateur a été développé [3]. Il étend l'approche [2] fondé sur la programmation linéaire sur des entiers et la génération de colonnes. Ce cadre permet de trouver une solution exacte, qui est un optimum global satisfaisant les contraintes utilisateur. Cependant ce cadre ne permet de traiter que le critère WCSS.

Dans des travaux récents, nous avons développé un cadre général pour le clustering sous contraintes utilisateur à base de la programmation par contraintes [6, 7]. Ce cadre permet de traiter différents critères d'optimisation : minimiser le diamètre maximal, maximiser la marge minimale ou minimiser WCSD, tout en intégrant tous les types de contraintes utilisateur populaires. En plus de traiter des problèmes de clustering mono-critère, la généricité du cadre permet de l'utiliser dans des problèmes de clustering bi-critère, comme minimiser le diamètre et maximiser la marge sous contraintes utilisateur [8]. Dans ce papier nous développons ce cadre pour traiter le clustering avec la minimisation de la somme des carrés WCSS sous contraintes utilisateur.

3 Modélisation des tâches de clustering sous contraintes utilisateur

Nous présentons un modèle en programmation par contraintes pour le clustering, intégrant le critère WCSS et des contraintes utilisateur. Ce modèle étend le cadre que nous avons développé pour les critères de diamètre, de la marge ou de WCSD. Pour plus de détails sur ces critères, nous renvoyons le lecteur vers [6, 7, 8]. Le modèle permet de formaliser les tâches de clustering avec ces critères pour un nombre de clusters k variant entre k_{min} et k_{max} clusters ($k_{min} \leq k \leq k_{max}$). Les valeurs k_{min} et k_{max} sont fixées par l'utilisateur. Si l'utilisateur souhaite k clusters, il suffit de fixer $k_{min} = k_{max} = k$.

Dans ce modèle, les clusters sont identifiés par leur indices, qui varient de 1 à k pour une partition en k clusters. Sans perdre de généralité, pour désigner le point o_i nous utilisons son indice i. Pour représenter l'affectation des points aux clusters, nous utilisons des variables entières $G_1, \ldots, G_n$, avec pour domaine $Dom(G_i)$ l'ensemble des entiers $[1, k_{max}]$. Une affectation $G_i = c$ signifie que le point i est affecté au cluster numéro c. Soit $\mathcal{G}$ le tableau des variables $[G_1, \ldots, G_n]$. Pour représenter le critère de la somme des carrés, nous introduisons une variable V ayant pour domaine l'intervalle $[0, \infty)$.

Les contraintes de la PPC permettent d'exprimer les contraintes de partition et les contraintes utilisateur.

3.1 Contraintes de partition

Toute affectation complète des variables de $\mathcal{G}$ définit une partition de points en clusters. Cependant, une partition peut se représenter par plusieurs affectations différentes. Par exemple, à partir d'une affectation des variables de $\mathcal{G}$, si l'on fait un échange où toutes les variables G_i qui ont la valeur c_1 prennent la valeur c_2 et les variables G_j qui ont la valeur c_2 prennent la valeur c_1, nous obtenons une nouvelle affectation mais qui représente toujours la même partition, du point de vue des objets composant les classes. Un autre cas où l'on obtient la même partition avec une affectation différente des variables de $\mathcal{G}$ est lorsque les variables ayant une valeur c_1 prennent une valeur c_3 qui n'est pas utilisée dans l'affectation. De telles situations représentent des symétries de valeurs. Pour casser ces symétries, nous imposons que les clusters formés par les variables G_i soient tels que le numéro 1 est l'indice du premier cluster créé, et un numéro c, avec $c > 1$ est utilisé pour un nouveau cluster si et seulement si le numéro $c - 1$ est déjà utilisé. Une méthode directe pour représenter ces conditions [7] est de poser la contrainte $G_1 = 1$ et les contraintes $G_i \leq \max_{j \in [1, i-1]}(G_j) + 1$, pour $i \in [2, n]$. Cependant, afin d'obtenir de meilleures

interactions et propagations, il est intéressant de présenter ces relations par une contrainte globale avec un bon algorithme de filtrage. La contrainte *precede* [13] permet de réaliser cela :

$$precede(\mathcal{G}, [1, \ldots, k_{max}]).$$

Cette contrainte impose que $G_1 = 1$ et de plus, si $G_i = c$ avec $1 < c \leq k_{max}$, alors il doit exister un indice $j < i$ tel que $G_j = c - 1$.

Le fait qu'il y ait au moins k_{min} clusters signifie que les valeurs de 1 à k_{min} doivent apparaître dans G_i. Avec l'utilisation de la contrainte *precede*, il suffit qu'au moins une variable G_i soit égale à k_{min}. Ceci est représenté par $\#\{i \mid G_i = k_{min}\} \geq 1$, ou par la contrainte

$$atleast(1, \mathcal{G}, k_{min}).$$

Puisque le domaine des variables G_i est $[1, k_{max}]$, il existe au plus k_{max} clusters. Si l'on veut exactement k clusters, il suffit de mettre $k_{min} = k_{max} = k$.

3.2 Contraintes utilisateur

Toutes les contraintes utilisateur populaires peuvent être intégrées dans le modèle.

- Une contrainte must-link sur deux points i, j est représentée par : $G_i = G_j$.
- Une contrainte cannot-link sur i, j est représentée par : $G_i \neq G_j$.
- Taille minimale des clusters α : chaque point doit être dans un cluster avec au moins α points (y compris ce point). Pour chaque $i \in [1, n]$, la valeur de la variable G_i doit donc apparaître au moins α fois dans $\mathcal{G}$, *i.e.* $\#\{j \mid G_j = G_i\} \geq \alpha$. Pour chaque $i \in [1, n]$, on pose la contrainte : $atleast(\alpha, \mathcal{G}, G_i)$. Cette contrainte permet aussi de mettre une borne sur le nombre de clusters. En effet, on peut poser $G_i \leq \lfloor n/\alpha \rfloor$, pour chaque $i \in [1, n]$.
- Taille maximale β des clusters : chaque nombre $c \in [1, k_{max}]$ doit apparaître dans $\mathcal{G}$ au plus β fois (c'est aussi vrai pour les numéros non utilisés $c \in [k_{min} + 1, k_{max}]$, qui apparaissent 0 fois), *i.e.* $\#\{i \mid G_i = c\} \leq \beta$. Donc, pour chaque $c \in [1, k_{max}]$, nous posons : $atmost(\beta, \mathcal{G}, c)$. Cette contrainte implique aussi : $G_i \geq \lceil n/\beta \rceil$, pour tout $i \in [1, n]$.
- Une contrainte de marge impose que la marge entre deux clusters doit être au moins δ. Donc pour chaque couple $i < j \in [1, n]$ tel que $d(i, j) < \delta$, la contrainte $G_i = G_j$ est posée.
- Une contrainte de diamètre impose que le diamètre de chaque cluster ne doit pas dépasser γ. Pour chaque couple $i < j \in [1, n]$ tel que $d(i, j) > \gamma$, on pose : $G_i \neq G_j$.
- Une contrainte de densité exprime que chaque point doit avoir dans son voisinage de rayon ϵ au moins m points dans le même cluster que lui. Donc pour chaque $i \in [1, n]$, l'ensemble de variables dans le voisinage de rayon ϵ est calculé $\mathcal{N}_{i\epsilon} = \{G_j \mid d(i, j) \leq \epsilon\}$ et cette contrainte est posée par : $atleast(m, \mathcal{N}_{i\epsilon}, G_i)$.

3.3 Critère d'optimisation

Pour représenter le fait que V est la somme des carrés des clusters formés par les variables de $\mathcal{G}$, nous développons une contrainte globale $wcss(\mathcal{G}, V, d)$. L'algorithme de filtrage de cette contrainte est présenté dans la section 4. La valeur de V est à minimiser.

3.4 Stratégie de recherche

Le branchement est effectué sur les variables de $\mathcal{G}$. Une stratégie mixte est utilisée. Pour trouver la première solution, une recherche gloutonne est utilisée : à chaque branchement, une variable G_i et une valeur $c \in Dom(G_i)$ telles que l'affectation $G_i = c$ augmente le moins possible V sont choisies. Lorsque la première solution est trouvée, la valeur de V dans cette solution constitue une borne supérieure de la variable V. Après avoir trouvé la première solution, la stratégie de recherche change ensuite en une autre stratégie. Dans cette stratégie, à chaque branchement, pour chaque variable G_i qui n'est pas instanciée, pour chaque valeur $c \in Dom(G_i)$ nous calculons la valeur a_{ic}, qui est la valeur ajoutée à V dans le cas où le point i est affecté au cluster c. Pour chaque variable G_i non instanciée, soit $a_i = \min_{c \in Dom(G_i)} a_{ic}$. La valeur a_i représente donc la valeur minimale ajoutée à V lorsque le point i est affecté à un cluster. Puisque chaque point doit être affecté à un cluster, à chaque branchement, la variable G_i choisie sera celle dont la valeur a_i est maximale, et pour cette variable, la valeur c choisie sera celle dont a_{ic} est minimale. Deux branches sont ensuite créées, une avec $G_i = c$ et une avec $G_i \neq c$. Avec cette stratégie, on tente de détecter rapidement l'échec et dans le cas sans échec, à obtenir une solution avec la valeur V la plus petite possible.

4 Algorithme de filtrage

Nous développons un algorithme de filtrage pour une nouvelle contrainte $wcss(\mathcal{G}, V, d)$. Cette contrainte assure que V représente la somme des carrés des clusters formés par l'instanciation des variables de $\mathcal{G}$. Nous présentons dans cette section le calcul de la borne inférieure de V et l'algorithme pour filtrer les domaines de V et des variables de $\mathcal{G}$, lorsqu'on a une instanciation partielle des variables de $\mathcal{G}$. Puisque la variable

V représente la fonction objectif à minimiser, cette contrainte est de type contrainte globale d'optimisation [11, 15]. L'algorithme de filtrage permet de filtrer non seulement le domaine de la variable objectif V, mais aussi le domaine des variables décisionnelles G_i.

Une instanciation partielle de variables de $\mathcal{G}$ correspond au cas où certain nombre de points sont déjà affectés aux clusters et il reste encore des points non affectés. Soit U l'ensemble des points non affectés et soit $q = |U|$. Soit $\mathcal{C}$ l'ensemble de tous les clusters et soit $k = \max_i\{c \mid c \in Dom(G_i)\}$. L'entier k représente le plus grand numéro de clusters restant dans les domaines de variables de $\mathcal{G}$ et donc le nombre maximum de clusters possible. Pour chaque cluster C_c ($c \in [1, k]$), soit n_c le nombre de points déjà affectés au cluster C_c ($n_c = |C_c|$) et soit $S_1(C_c)$ la somme des dissimilarités des points qui sont déjà affectés au cluster C_c : $S_1(C_c) = \frac{1}{2}\sum_{i,j\in C_c} d(i, j)$.

4.1 Calcul de la borne inférieure de V

L'objectif est de calculer une borne inférieure de la somme des carrés V lorsque l'on affecte tout l'ensemble des points non affectés U dans les clusters $C_1, \dots, C_k$. Ce calcul se fait en deux étapes :

1. Pour chaque $m \in [0, q]$ et pour chaque $c \in [1, k]$, nous calculons une borne inférieure de la somme des carrés $\underline{V}(C_c, m)$ pour le cas où l'on ajoute m points quelconques de U dans le cluster C_c.
2. Pour chaque $m \in [0, q]$ et pour chaque $c \in [2, k]$, nous calculons une borne inférieure de la somme des carrés $\underline{V}(C_1 \dots C_c, m)$ pour le cas où l'on ajoute m points quelconques de U dans c clusters $C_1, \dots, C_c$.

Le calcul de la borne inférieure de V s'effectue par l'algorithme 1. Nous détaillons ci-dessous les deux étapes de calcul.

Ajout de m points de U à un cluster C_c. Si l'on choisit un ensemble $U' \subseteq U$ avec $|U'| = m$ et si l'on ajoute les points de U' dans le cluster C_c, la somme des carrés du cluster sera

$$V(C_c, U') = \frac{S_1(C_c) + S_2(C_c, U')}{n_c + m}$$

Ici $S_1(C_c)$ est la somme des dissimilarités entre les points déjà dans C_c et $S_2(C_c, U')$ est la somme des dissimilarités liées aux points de U'. La valeur de $S_1(C_c)$ est connue. Si l'on connait l'ensemble U' la valeur de $S_2(C_c, U')$ sera aussi connue. Elle est calculée par :

$$S_2(C_c, U') = \sum_{u\in U', v\in C_c} d(u, v) + \frac{1}{2}\sum_{u,v\in U'} d(u, v)$$

Algorithme 1 : Borne_inf()

```
   input  : une instanciation partielle de variables
            de 𝒢, un ensemble
            U = {i | G_i non instanciée}
   output : une borne inférieure pour V
1  foreach x ∈ U do
2      trier u ∈ U dans l'ordre croissant de d(x,u)
3      foreach c ∈ [1,k] do
4          if c ∉ Dom(G_x) then
5              s2[x,c] ← ∞
6          else
7              s2[x,c] ← 0
8      foreach v ∈ [1,n] do
9          if G_v est instanciée et G_v ∈ Dom(G_x)
           then
               s2[x,val(G_v)] = s2[x,val(G_v)] + d(x,v)
10     s3[x,0] ← 0;
11     for m ← 1 to q do
12         s3[x,m] ← s3[x,m−1] + d(x,u_i)/2
13 foreach c ∈ [1,k] do
14     foreach m ∈ [0,q] do
15         foreach x ∈ U do
16             s[x] = s2(x,c) + s3(x,m)
17         trier tableau s dans l'ordre croissant
18         S̲2(C_c,m) ← Σ_{i=1}^{m} s[i]
19         if n_c + m = 0 then
20             V̲(C_c,m) ← 0
21         else
22             V̲(C_c,m) ←
               (S1(C_c) + S̲2(C_c,m))/(n_c + m)
23 for c ← 2 to k do
24     for m ← 1 to q do
25         V̲(C1...C_c,m) ←
           min_{i∈[0,m]}(V̲(C1...C_{c−1},i) + V̲(C_c,m−i))
26 retourner V̲(C1...C_k,q)
```

Mais $S_2(C_c, U')$ devient inconnu lorsque l'ensemble U' n'est pas fixé. Pour tous les sous-ensembles U' de taille m, on peut cependant calculer une borne inférieure $\underline{S_2}(C_c, m)$ comme suit.

Chaque point $x \in U$, dans le cas où il est affecté au cluster C_c avec $m-1$ autres points, va contribuer une somme $s(x, c, m) = s_2(x, c) + s_3(x, m)$, où

- $s_2(x, c)$ représente la somme des dissimilarités entre x et les points de C_c. Si $c \notin Dom(G_x)$ alors $s_2(x, c) = \infty$, car x ne peut pas être affecté au cluster C_c. Sinon :

$$s_2(x, c) = \sum_{v \in C_c} d(x, v)$$

  Cette valeur $s_2(x, c)$ vaut 0 si le cluster C_c est vide. Lignes 3-9 de l'algorithme 1 calculent $s_2(x, c)$.
- $s_3(x, m)$ représente la moitié de la somme des dissimilarités $d(x, z)$, pour tous les $m-1$ autres points z ; si l'on trie tous les points $u \in U$ dans l'ordre croissant des dissimilarités $d(x, u)$, et l'on désigne par u_i le i-eme point dans cet ordre, on a

$$s_3(x, m) \ \geq \ \frac{1}{2} \sum_{i=1}^{m-1} d(x, u_i)$$

  cf. lignes 11-12 de l'algorithme 1.

Pour chaque $c \in [1, k]$ et $m \in [0, q]$, la valeur $s(x, c, m)$ est représentée par $s[x]$ (ligne 15). La borne inférieure $\underline{S_2}(C_c, m)$ est donc la somme des m plus petites contributions $s(x, c, m)$, pour tous les points $x \in U$. La borne inférieure $\underline{V}(C_c, m)$ est calculée par :

$$\underline{V}(C_c, m) \ = \ \frac{S_1(C_c) + \underline{S_2}(C_c, m)}{n_c + m} \qquad (1)$$

Ce calcul est réalisé pour tout $c \in [1, k]$ et pour tout $m \in [0, q]$ (lignes 13 à 21 de l'algorithme 1).

Ajout de m points de U aux c clusters $C_1, .., C_c$. Pour $m \in [0, q]$ et $c \in [1, k]$, calculons une borne inférieure $\underline{V}(C_1..C_c, m)$ de tous les cas possibles où l'on ajoute m points de U aux clusters $C_1, ..., C_c$. Si l'on choisit m points $x_1, \dots, x_m \in U$ et si l'on ajoute i points $x_1, \dots, x_i$ aux $c-1$ clusters $C_1, \dots, C_{c-1}$ et les autres $m-i$ points au cluster C_c, la somme des carrés $V(C_1..C_c, \{x_1, .., x_m\})$ sera égale à :

$$V(C_1..C_{c-1}, \{x_1, ..x_i\}) + V(C_c, \{x_{i+1}, .., x_m\})$$

Lorsque les points $x_1, \dots, x_m$ sont des points quelconques de U, la valeur exacte n'est pas connue. Cependant :

$$V(C_1..C_c, m) \geq \min_{i \in [0,m]} (V(C_1..C_{c-1}, i) + V(C_c, m-i))$$

La borne inférieure $\underline{V}(C_1..C_c, m)$ est donc calculée par :

$$\underline{V}(C_1..C_c, m) = \min_{i \in [0,m]} (\underline{V}(C_1..C_{c-1}, i) + \underline{V}(C_c, m-i)) \qquad (2)$$

Ce calcul est réalisé par un programme dynamique pour tout $c \in [2, k]$ et pour tout $m \in [0, q]$ (lignes 23 à 25 de l'algorithme 1). Soit $[V.lb, V.ub)$ l'intervalle représentant le domaine de la variable V. La borne inférieure $V.lb$ est donc fixée à $\underline{V}(C_1 \dots C_k, q)$, la borne inférieure lorsque l'on affecte l'ensemble de points U aux clusters $C_1, \dots, C_k$. La complexité de l'algorithme est $O(kqn \log q)$.

Notons qu'avec la formule (2), pour tout $c \in [1, k]$, pour tout $m \in [0, q]$, $\underline{V}(C_1..C_c, m)$ est égal à

$$\min_{m_1 + .. + m_c = m} (\underline{V}(C_1, m_1) + \dots + \underline{V}(C_c, m_c)) \qquad (3)$$

4.2 Filtrage

Etant donnée une instanciation partielle des variables de $\mathcal{G}$, la borne inférieure $V.lb$ de la variable V est calculée. Nous présentons le filtrage du domaine des variables G_i qui ne sont pas instanciées dans l'algorithme 2. Cet algorithme permet donc de filtrer le domaine de V qui représente la fonction objectif, mais aussi le domaine des variables décisionnelles G_i.

Dans cet algorithme, pour chaque valeur $c \in [1, k]$, pour chaque variable G_i qui n'est pas instanciée, si $c \in Dom(G_i)$, sous l'hypothèse que le point i soit affecté au cluster C_c, on calcule une nouvelle borne inférieure de V. Soit C'_c le cluster $C_c \cup \{i\}$ et soit $\mathcal{C}' = \{C_l \mid l \neq c\} \cup \{C'_c\}$. La nouvelle borne inférieure V' de V sera la valeur de $\underline{V}(\mathcal{C}', q-1)$, car il reste $q-1$ points dans l'ensemble U à affecter aux k clusters. Suivant le même principe que (2), on a :

$$\underline{V}(\mathcal{C}', q-1) = \min_{m \in [0, q-1]} (\underline{V}(\mathcal{C}' \backslash \{C'_c\}, m) + \underline{V}(C'_c, q-1-m))$$

Il faudra donc réviser les bornes inférieures $\underline{V}(\mathcal{C}' \backslash \{C'_c\}, m)$ et $\underline{V}(C'_c, m)$, pour tout $m \in [0, q-1]$. Ces calculs sont présentés dans le reste de cette sous-section. La nouvelle borne inférieure V' est calculée par la ligne 8 de l'algorithme 2. Ensuite, puisque $Dom(V) = [V.lb, V.ub)$, si $V' \geq V.ub$, la valeur c est inconsistante et sera supprimée du $Dom(G_i)$ (lignes 9-10). La complexité de l'algorithme 2 est aussi de $O(kqn \log q)$.

Calcul de $\underline{V}(C'_c, m)$. Rappelons que C'_c est le cluster C_c auquel on a ajouté le point i, et $\underline{V}(C'_c, m)$ est la borne inférieure de la somme des carrés de C'_c lorsqu'on ajoute m points quelconques de U dans C'_c. D'après (1) :

$$\underline{V}(C'_c, m) = \frac{S_1(C'_c) + \underline{S_2}(C'_c, m)}{n_c + 1 + m}$$

La valeur de $S_1(C'_c)$ est

$$S_1(C'_c) = S_1(C_c) + s_2(i,c)$$

La valeur de $\underline{S_2}(C'_c, m)$ peut être révisée à partir de $\underline{S_2}(C_c, m)$ par :

$$\underline{S_2}(C'_c, m) = \underline{S_2}(C_c, m) + s_3(i,m)$$

Par (1), on a donc (ligne 7 de l'algorithme 2) :

$$\underline{V}(C'_c, m) = \frac{(n_c+m)\underline{V}(C_c, m) + s_2(i,c) + s_3(i,m)}{n_c+m+1}$$

Calcul de $\underline{V}(\mathcal{C}'\backslash\{C'_c\}, m)$. Cette valeur représente la borne inférieure lorsque l'on ajoute m points de $U\backslash\{i\}$ dans les clusters différents de C'_c. D'après (3), pour tout $q' \in [m,q]$ on a :

$$\underline{V}(\mathcal{C}, q') = \min_{m+m'=q'} (\underline{V}(\mathcal{C}\backslash\{C_c\}, m) + \underline{V}(C_c, m'))$$

donc pour tout $q' \in [m,q]$ et avec $m+m' = q'$, on a :

$$\underline{V}(\mathcal{C}, m+m') \leq \underline{V}(\mathcal{C}\backslash\{C_c\}, m) + \underline{V}(C_c, m')$$

ce qui correspond à :

$$\underline{V}(\mathcal{C}\backslash\{C_c\}, m) \geq \underline{V}(\mathcal{C}, m+m') - \underline{V}(C_c, m')$$

Nous avons donc :

$$\underline{V}(\mathcal{C}\backslash\{C_c\}, m) \geq \max_{m'\in[0,q-m]} (\underline{V}(\mathcal{C}, m+m') - \underline{V}(C_c, m'))$$

Nous avons aussi :

$$\underline{V}(\mathcal{C}'\backslash\{C'_c\}, m) \geq \underline{V}(\mathcal{C}\backslash\{C_c\}, m)$$

car $\mathcal{C}'\backslash\{C'_c\}$ et $\mathcal{C}\backslash\{C_c\}$ désigne le même ensemble de clusters, $\underline{V}(\mathcal{C}'\backslash\{C'_c\}, m)$ est calculé pour m points quelconques de $U\backslash\{i\}$, pendant que $\underline{V}(\mathcal{C}\backslash\{C_c\}, m)$ est calculé pour m points quelconques de U. On peut donc fixer une nouvelle borne inférieure (ligne 4 de l'algorithme 2) :

$$\underline{V}(\mathcal{C}'\backslash\{C'_c\}, m) = \max_{m'\in[0,q-m]} (\underline{V}(\mathcal{C}, m+m') - \underline{V}(C_c, m'))$$

Algorithme 2 : Filtrage pour $wcss(\mathcal{G}, V, d)$

1 $V.lb \leftarrow$ Borne_inf()
2 **foreach** $c \in [1,k]$ **do**
3   **foreach** $m \in [0, q-1]$ **do**
4     $\underline{V}(\mathcal{C}'\backslash\{C'_c\}, m) \leftarrow \max_{m'\in[0,q-m]}(\underline{V}(\mathcal{C}, m+m') - \underline{V}(C_c, m'))$
5   **foreach** $i \in [1,n]$ *tel que* $|Dom(G_i)| > 1$ *et* $c \in Dom(G_i)$ **do**
6   **for** $m \leftarrow 0$ **to** $q-1$ **do**
7       $\underline{V}(C'_c, m) \leftarrow ((n_c+m)\underline{V}(C_c, m) + s_2(i,c) + s_3(i,m))/(n_c+m+1)$
8     $V' \leftarrow \min_{m\in[0,q-1]}(\underline{V}(\mathcal{C}'\backslash\{C'_c\}, m) + \underline{V}(C'_c, q-1-m))$
9   **if** $V' \geq V.ub$ **then**
10   supprimer c du $Dom(G_i)$

5 Expérimentations

Notre modèle est implanté en utilisant la librairie Gecode (`http://www.gecode.org`) version 4.2.7. Cette librairie supporte à la fois des variables flottantes et entières. Les expérimentations sont réalisées sur un ordinateur 3.0 GHz Core i7 Intel avec 8 Go de mémoire sous Ubuntu. Tous nos programmes sont disponibles sur `http://cp4clustering.com`. Nous considérons les bases Iris et Wine du dépôt de données UCI (`http://archive.ics.uci.edu/ml`). La base Iris est composée de 150 objets et 3 classes, la base Wine de 178 objets et 3 classes. Nous comparons notre modèle avec l'approche basée sur la programmation linéaire sur des entiers et la génération de colonnes [3] pour le clustering avec le critère WCSS et avec les contraintes utilisateur. Nous comparons également notre modèle avec l'algorithme RBBA [4] dans le cas sans contraintes utilisateur et avec l'algorithme COP-kmeans [17] qui trouve des solutions approchées.

5.1 WCSS avec contraintes utilisateur

Pour générer les contraintes utilisateur must-link et cannot-link, des paires d'objets sont retirées aléatoirement et les classes réelles des objets sont considérées. Si les objets sont de la même classe, une contrainte must-link est générée, sinon une contrainte cannot-link est générée. Les contraintes must-link ou cannot-link sont générées jusqu'à ce que le nombre de contraintes souhaité soit atteint. Pour chaque nombre de contraintes, nous générons 5 ensembles différents pour effectuer le test. Nous considérons pour chaque test la valeur de WCSS, le temps total d'exécution et l'indice Rand de la solution trouvée. Puisque le test est effectué 5 fois pour chaque nombre de contraintes, nous reportons la valeur moyenne et l'écart-type des tests. Une limite de temps est fixée à 30 minutes. Le signe "-" dans les tableaux indique que le temps limite est dépassé sans qu'une solution optimale soit trouvée et prouvée.

Le tableau 1 donne, pour la base Iris, la moyenne sur les 5 tests du temps d'exécution total en secondes ainsi que de la valeur de l'indice de Rand [14], ainsi que le pourcentage de l'écart type par rapport à la moyenne

#c	T moy	% ecart	RI moy	% ecart
0	888,99	0,83	0,879	0
50	332,06	78,96	0,940	1,66
100	7,09	40,18	0,978	1,68
150	0,31	36,39	0,989	0,66
200	0,07	24,83	0,992	0,66
250	0,05	10,63	0,996	0,70
300	0,04	9,01	0,998	0,35

TABLE 1 – Base Iris avec #c contraintes must-link

#c	CP	% ecart	GC	% ecart
0	888,99	0,83	-	-
50	332,06	78,96	-	-
100	7,09	40,18	62	74,24
150	0,31	36,39	0,45	44,55
200	0,06	24,83	0,11	48,03
250	0,05	10,63	0,06	35,56
300	0,04	9,01	0,04	19,04

TABLE 2 – Base Iris avec #c must-link

$(100\sigma/\mu)$. L'indice de Rand mesure la similarité entre une partition P et la partition réelle P^* de la base de données. Il est défini par :

$$Rand = \frac{a+b}{a+b+c+d}$$

où a et b sont les nombres de paires de points pour lesquelles P et P^* s'accordent (a nombre de paires qui se retrouvent dans la même classe dans P et dans P^*, b nombre de paires dans différentes classes), c et d sont les nombres de paires de points pour lesquelles P et P^* ne s'accordent pas (même classe dans P mais différentes classes dans P^* et vice versa). L'indice Rand varie entre 0 et 1 et plus les deux partitions s'accordent, plus il est proche de 1.

On peut remarquer que pour cette base, les contraintes utilisateur ajoutées permettent de réduire le temps d'exécution et de trouver des solutions de très bonne qualité.

Pour la base Iris, notre modèle (CP) peut trouver une solution optimale pour le problème de minimisation de WCSS sans contraintes utilisateur. L'approche par la programmation linéaire sur des entiers et la génération de colonnes (GC) [3] ne peut pas prouver l'optimalité pour ce problème avec moins de 100 contraintes must-link. Les tableaux 2 et 3 reportent les temps d'exécution en moyenne en secondes des tests des deux approches et le pourcentage de l'écart type des tests par rapport au temps moyen. Le tableau 2 présente les cas où seules des contraintes must-link sont ajoutées et le tableau 3 les cas où l'on ajoute le même nombre $\#c$ de contraintes must-link (ML) et cannot-link (CL). On peut constater que notre modèle est le plus efficace dans ces cas.

Pour la base Wine, les deux approches ne peuvent pas prouver l'optimalité en moins de 30 minutes avec moins de 150 contraintes must-link, comme le montre le tableau 4 avec les temps moyen en secondes. On peut constater que pour la base Wine, notre modèle est le plus efficace lorsque des contraintes must-link sont ajoutées.

#c	CP	% ecart	GC	% ecart
25	969,33	51,98	-	-
50	43,85	46,67	-	-
75	4,97	150	-	-
100	0,41	49,8	107	72,35
125	0,09	52,07	4,4	95,85
150	0,06	22,6	0,8	50

TABLE 3 – Base Iris avec #c ML et #c CL

Notre approche à base de programmation par contraintes permet aussi de mieux exploiter les contraintes cannot-link, comme le montre les résultats des tableaux 5 et 6. Ces tableaux présentent le temps en moyen en secondes et le pourcentage des cas résolus parmi tous les tests. Le temps de calcul varie beaucoup en fonction des contraintes ajoutées. Par exemple pour la base Iris, le tableau 5 montre que notre modèle est capable de trouver et de prouver l'optimalité pour environ 60% des cas, alors que GC ne peut résoudre aucun cas. Pour la base Wine, le tableau 6 montre qu'avec 100 contraintes must-link et 100 contraintes cannot-link, notre modèle peut résoudre tous les cas alors que GC ne peut résoudre aucun cas. Lorsque 125 contraintes must-link et 125 contraintes cannot-link sont ajoutées, les deux approches peuvent résoudre tous les cas, mais notre approche prend moins de temps.

#c	CP	GC
150	6,84	12,98
200	0,11	0,32
250	0,08	0,11
300	0,08	0,06

TABLE 4 – Base Wine avec #c must-link

#c	CP	%res CP	GC	%res GC
50	1146,86	20%	-	0%
100	719,53	80%	-	0%
150	404,77	60%	-	0%
200	1130,33	40%	-	0%
250	172,81	60%	-	0%
300	743,64	60%	-	0%

TABLE 5 – Base Iris avec #c contraintes cannot-link

#c	CP	%res CP	GC	% res GC
100	10,32	100%	-	0%
125	0,35	100%	497,6	100%
150	0,12	100%	13,98	100%

TABLE 6 – Base Wine avec #c ML et #c CL

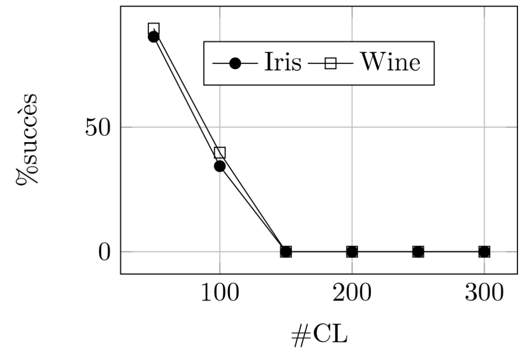

FIGURE 1 – COP-kmeans avec #c cannot-link

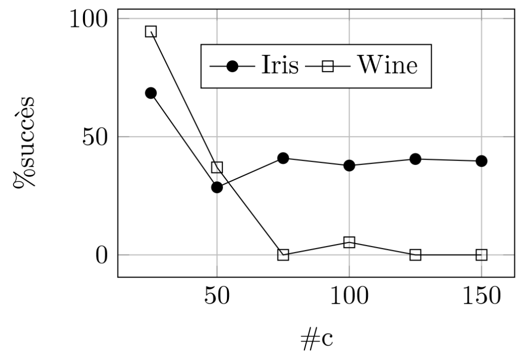

FIGURE 2 – COP-kmeans avec #c ML et #c CL

5.2 Comparaisons avec COP-kmeans et RBBA

L'algorithme RBBA basé sur la recherche *branch-and-bound* répétitive [4] permet de trouver une solution exacte qui optimise globalement le critère WCSS. Cet algorithme prend 0.53s pour trouver et prouver la solution optimale pour la base Iris et 35.56s pour la base Wine. Cependant, il ne permet pas d'intégrer des contraintes utilisateur.

Lorsque les contraintes must-link et cannot-link s'ajoutent, l'algorithme COP-kmeans [17] basé sur une recherche gloutonne cherche une solution approchée pour le critère WCSS satisfaisant toutes les contraintes. Dans le cas où il n'y a que des contraintes must-link, COP-kmeans trouve toujours une partition satisfaisant toutes les contraintes qui optimise localement WCSS. Cependant, avec les contraintes cannot-link, dans plusieurs cas, l'algorithme n'arrive pas à trouver une solution satisfaisant toutes les contraintes, même lorsqu'une telle solution existe. Nous réalisons les mêmes jeux de test mais pour chaque ensemble de contraintes, COP-kmeans est lancé 1000 fois et nous rapportons le nombre de fois où COP-kmeans trouve une partition. La figure 1 montre le pourcentage des cas réussis lorsque les contraintes cannot-link sont ajoutées. Avec les deux bases Iris et Wine, COP-kmeans ne trouve aucune partition lorsque 150 contraintes sont ajoutées. Notre modèle CP trouve toujours des solutions satisfaisant toutes les contraintes et dans plusieurs cas réussit à prouver l'optimalité des solutions pour la base Iris, comme le montre le tableau 5. La figure 2 montre le résultat lorsque l'on ajoute le même nombre $\#c$ de contraintes must-link et cannot-link. Pour la base Wine, COP-kmeans ne peut trouver aucune partition lorsque $\#c =$ 75 ou 125 ou 150. Notre modèle CP trouve des solutions satisfaisant toutes les contraintes pour tous les cas et trouve un optimum global dans tous les cas pour la base Iris, et dans tous les cas pour la base Wine avec $\#c \geq 100$, comme le montre les tableaux 3 et 6.

6 Conclusion

Des cadres génériques et déclaratifs pour modéliser des tâches de clustering attirent de plus en plus l'attention des communautés de la programmation par contraintes et de la fouille de données. Nous avons développé un cadre basé sur la programmation par contraintes pour le clustering sous contraintes utilisateur avec les critères de diamètre, de la marge ou de la somme des dissimilarités [6, 7, 8]. Dans ce papier, nous développons ce cadre pour intégrer le critère bien connu de la somme des carrés WCSS. Trouver un optimum globale pour ce critère est un problème NP-Difficile et trouver une bonne borne inférieure est aussi difficile [1]. La meilleure approche exacte pour ce critère à notre connaissance est basée sur la programmation linéaire sur des entiers et la génération de colonnes [2]. Une extension de cette approche pour traiter des contraintes utilisateur a été développée [3].

Dans ce papier, nous présentons un calcul de borne inférieure pour la somme des carrés. Nous développons une nouvelle contrainte *wcss* et présentons un algorithme pour filtrer le domaine des variables, qui ne sont pas seulement la variable objective, mais aussi des variables décisionnelles. Des expérimentations sur des bases de données classiques montrent que notre approche obtient une meilleure performance que l'approche basée sur la programmation linéaire sur des entiers et la génération de colonnes.

Nous continuons à améliorer l'algorithme de filtrage pour la contrainte *wcss*, en exploitant les contraintes utilisateur qui sont ajoutées aux tâches de clustering. En effet, les contraintes must-link ajoutées peuvent fractionner les points en composants connexes, ce qui peut aider à améliorer la borne inférieure de la somme des carrés.

Références

[1] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. NP-hardness of Euclidean Sum-of-squares Clustering. *Mach. Learn.*, 75(2) :245–248, May 2009.

[2] Daniel Aloise, Pierre Hansen, and Leo Liberti. An improved column generation algorithm for minimum sum-of-squares clustering. *Mathematical Programming*, 131(1-2) :195–220, 2012.

[3] Behrouz Babaki, Tias Guns, and Siegfried Nijssen. Constrained clustering using column generation. In *Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 438–454, 2014.

[4] Michael Brusco and Stephanie Stahl. *Branch-and-Bound Applications in Combinatorial Data Analysis (Statistics and Computing)*. Springer, 1 edition, July 2005.

[5] M.J. Brusco. An enhanced branch-and-bound algorithm for a partitioning problem. *British Journal of Mathematical and Statistical Psychology*, pages 83–92, 2003.

[6] Thi-Bich-Hanh Dao, Kanh-Chuong Duong, and Christel Vrain. A Declarative Framework for Constrained Clustering. In *Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, pages 419–434, 2013.

[7] Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. Clustering sous contraintes en ppc. *Revue d'Intelligence Artificielle*, 28(5) :523–545, 2014.

[8] Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. Constrained clustering by constraint programming. *Artificial Intelligence Journal*, To appear.

[9] Ian Davidson and S. S. Ravi. Clustering with Constraints : Feasibility Issues and the k-Means Algorithm. In *Proceedings of the 5th SIAM International Conference on Data Mining*, pages 138–149, 2005.

[10] Ian Davidson and S. S. Ravi. The Complexity of Non-hierarchical Clustering with Instance and Cluster Level Constraints. *Data Mining Knowledge Discovery*, 14(1) :25–61, 2007.

[11] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 189–203, 1999.

[12] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining : Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[13] Yat Chiu Law and Jimmy Ho-Man Lee. Global constraints for integer and set value precedence. In Mark Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 362–376, 2004.

[14] William M. Rand. Objective Criteria for the Evaluation of Clustering Methods. *Journal of the American Statistical Association*, 66(336) :846–850, 1971.

[15] Jean-Charles Régin. Arc consistency for global cardinality constraints with costs. In *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, pages 390–404, 1999.

[16] K. Wagstaff and C. Cardie. Clustering with instance-level constraints. In *Proceedings of the 17th International Conference on Machine Learning*, pages 1103–1110, 2000.

[17] Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schrödl. Constrained K-means Clustering with Background Knowledge. In *Proceedings of the 18th International Conference on Machine Learning*, pages 577–584, 2001.