A Global Constraint for the Exact Cover Problem:
Application to Conceptual Clustering
Maxime Chabert, Christine Solnon

To cite this version:
Maxime Chabert, Christine Solnon. A Global Constraint for the Exact Cover Problem: Application to Conceptual Clustering. Journal of Artificial Intelligence Research, Association for the Advancement of Artificial Intelligence, 2020, 67, pp.509 - 547. 10.1613/jair.1.11870 . hal-02507186

HAL Id: hal-02507186
https://hal.archives-ouvertes.fr/hal-02507186
Submitted on 12 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
A Global Constraint for the Exact Cover Problem: Application to Conceptual Clustering

Maxime Chabert
Infologic, Université de Lyon, INSA-Lyon, CNRS, LIRIS, F-69621, Villeurbanne, France
mch@infologic.fr

Christine Solnon
Université de Lyon, INSA-Lyon, Inria, CITI, CNRS, LIRIS, F-69621, Villeurbanne, France
christine.solnon@insa-lyon.fr

Abstract

We introduce the exactCover global constraint dedicated to the exact cover problem, the goal of which is to select subsets such that each element of a given set belongs to exactly one selected subset. This \(\mathcal{NP}\)-complete problem occurs in many applications, and we more particularly focus on a conceptual clustering application.

We introduce three propagation algorithms for exactCover, called Basic, DL, and DL+: Basic ensures the same level of consistency as arc consistency on a classical decomposition of exactCover into binary constraints, without using any specific data structure; DL ensures the same level of consistency as Basic but uses Dancing Links to efficiently maintain the relation between elements and subsets; and DL+ is a stronger propagator which exploits an extra property to filter more values than DL.

We also consider the case where the number of selected subsets is constrained to be equal to a given integer variable \(k\), and we show that this may be achieved either by combining exactCover with existing constraints, or by designing a specific propagator that integrates algorithms designed for the NValues constraint.

These different propagators are experimentally evaluated on conceptual clustering problems, and they are compared with state-of-the-art declarative approaches. In particular, we show that our global constraint is competitive with recent ILP and CP models for mono-criterion problems, and it has better scale-up properties for multi-criteria problems.

1. Introduction

The exact cover problem aims at deciding whether it is possible to select some subsets within a given collection of subsets in such a way that each element of a given set belongs to exactly one selected subset. This problem is \(\mathcal{NP}\)-complete (Karp, 1972). It occurs in many applications, and different approaches have been proposed for solving it. In particular, Knuth (2000) has introduced the DLX algorithm that uses a specific data structure called Dancing Links. Also, different declarative exact approaches have been proposed, based on Constraint Programming (CP), Integer Linear Programming (ILP), or Boolean satisfiability (SAT). However, none of these declarative approaches is competitive with DLX.

In this paper, we introduce global constraints and propagation algorithms dedicated to the exact cover problem to improve scale-up properties of CP for solving these problems. We evaluate the interest of these global constraints for solving conceptual clustering problems.
1.1 Contributions and Overview of the Paper

In Section 2, we briefly recall basic principles of Constraint Programming. In Section 3, we describe the exact cover problem, and we describe existing exact approaches for solving this problem. In particular, we describe the DLX algorithm of Knuth (2000). We also describe existing declarative approaches, i.e., the Boolean CP model of Hjort Blindell (2018), the ILP model of Ouali, Loudni, Lebbah, Boizumault, Zimmermann, and Loukil (2016), and the SAT models of Junttila and Kaski (2010).

In Section 4, we define the $\text{exactCover}$ global constraint, and we introduce three propagation algorithms for this constraint, called $\text{Basic}$, $\text{DL}$, and $\text{DL+}$:

- $\text{Basic}$ ensures the same level of consistency as Arc Consistency (AC) on the Boolean CP model of Hjort Blindell (2018), without using any specific data structure;
- $\text{DL}$ ensures the same level of consistency as $\text{Basic}$ but uses Dancing Links to efficiently maintain the data structure that links elements and subsets;
- $\text{DL+}$ also uses Dancing Links, but further propagates a property used by Davies and Bacchus (2011) to filter more values.

We experimentally compare these three algorithms with DLX and with existing declarative exact approaches (SAT, ILP, and CP).

In Section 5, we consider the case where the number of selected subsets is constrained to be equal to a given integer variable $k$, and we show that this may be achieved either by combining $\text{exactCover}$ with existing constraints, or by extending the $\text{DL+}$ propagator of $\text{exactCover}$ in order to integrate algorithms introduced for the $\text{NValues}$ global constraint (Bessièere, Hebrard, Hnich, Kiziltan, & Walsh, 2006).

In Section 6, we introduce conceptual clustering problems and we show how to use our global constraints to solve these problems. We experimentally compare our approach with state-of-the-art declarative exact approaches. We first consider mono-criterion problems, where the goal is to find a clustering that optimizes a single objective function. Finally, we consider bi-criteria problems, where the goal is to compute the Pareto front of all non-dominated solutions for two conflicting objective functions.

2. Background on Constraint Programming

In this section, we briefly recall basic principles of Constraint Programming. We refer the reader to Rossi, Beek, and Walsh (2006) for more details.

A Constraint Satisfaction Problem (CSP) is defined by a triple $(X, D, C)$ such that $X$ is a finite set of variables, $D$ is a function that associates a finite domain $D(x_i) \subset \mathbb{Z}$ to every variable $x_i \in X$, and $C$ is a finite set of constraints.

A constraint $c$ is a relation defined on a sequence of variables $X(c) = (x_{i_1}, \ldots, x_{i_{\#X(c)}})$, called the scheme of $c$, where $\#X(c)$ is the arity of $c$. $c$ is the subset of $\mathbb{Z}^{\#X(c)}$ that contains the combinations of values $\tau \in \mathbb{Z}^{\#X(c)}$ that satisfy $c$. The scheme of a constraint $c$ is a sequence of variables and not a set because the order of values matters for tuples in $c$. However, we use set operators on sequences: $s_1 \subseteq s_2$ denotes that every element in a sequence $s_1$ also appears in another sequence $s_2$, and $e \in s$ denotes that an element $e$ occurs in a sequence $s$. If $\#X(c) = 2$ then $c$ is a binary constraint.
An instantiation \( I \) on \( Y = (x_1, \ldots, x_k) \subseteq X \) is an assignment of values \( v_1, \ldots, v_k \) to the variables \( x_1, \ldots, x_k \). Given a subset of variables \( Z \subseteq Y \), \( I[Z] \) denotes the tuple of values associated with the variables in \( Z \). \( I \) is valid if for all \( x_i \in Y, v_i \in D(x_i) \). \( I \) is partial if \( Y \subset X \) and complete if \( Y = X \). \( I \) is locally consistent if it is valid and for every \( c \in C \) such that \( X(c) \subseteq Y \), \( I[X(c)] \) satisfies \( c \). A solution is a complete instantiation on \( X \) which is locally consistent.

An objective function may be added to a CSP, thus defining a Constrained Optimization Problem (COP). This objective function is defined on some variables of \( X \) and the goal is to find the solution that optimizes (minimizes or maximizes) the objective function.

CSPs and COPs may be solved by generic constraint solvers which are usually based on a systematic exploration of the search space: Starting from an empty instantiation, variables are recursively instantiated until either finding a solution or detecting an inconsistency (in which case the search must backtrack to try other assignments). This exhaustive exploration of the search space is combined with constraint propagation techniques: At each node of the search tree, constraints are propagated to filter variable domains, i.e., remove values that cannot belong to a solution. When constraint propagation removes all values from a domain, the search must backtrack.

Given a constraint, different propagation algorithms may be considered, and they may differ on their filtering strength (i.e., the number of values that are removed) and/or on their time and space complexity. The goal is to find the best trade-off between these criteria. Many propagation algorithms filter domains to ensure arc consistency. A domain \( D \) is AC on a constraint \( c \) for a variable \( x_i \in X(c) \) if for every value \( v \in D(x_i) \) there exists a valid instantiation \( I \) on \( X(c) \) such that \( I \) satisfies \( c \) and \( I[x_i] = v \). A CSP is AC if \( D \) is AC for all variables in \( X \) on all constraints in \( C \).

3. Exact Cover Problem

In this section, we first introduce the exact cover problem and some of its applications. Then, we describe an algorithm and a data structure introduced by Knuth (2000) to solve this problem. Finally, we describe existing declarative models (CP, ILP, and SAT) for this problem.

3.1 Definitions and Notations

**Definition 1.** An instance of the Exact Cover Problem (EC) is defined by a couple \((S, P)\) such that \( S \) is a set of elements and \( P \subseteq \mathcal{P}(S) \) is a set of subsets of \( S \). EC aims at deciding if there exists a subset \( E \subseteq P \) which is a partition of \( S \), i.e., \( \forall a \in S, \#\{u \in E : a \in u\} = 1 \).

Elements of \( S \) are denoted \( a, b, c, \) etc., whereas elements of \( P \) (i.e., subsets) are denoted \( t, u, v, \) etc. For each element \( a \in S \), we denote \( \text{cover}(a) \) the set of subsets that contain \( a \), i.e., \( \text{cover}(a) = \{u \in P : a \in u\} \). Two subsets \( u, v \in P \) are compatible if \( u \cap v = \emptyset \) and, for every subset \( u \in P \), we denote \( \text{incompatible}(u) \) the subsets of \( P \) that are not compatible with \( u \), i.e., \( \text{incompatible}(u) = \{v \in P \setminus \{u\} : u \cap v \neq \emptyset\} \).

**Example 1.** Let us consider the instance \((S, P)\) displayed in Fig. 1. A solution is: \( E = \{v, x, z\}\). We have \( \text{cover}(a) = \{t, u, v\} \), and \( \text{incompatible}(x) = \{w, y\} \).
The maximum cardinality of a subset in $P$ is denoted $n_p$ (i.e., $n_p = \max_{u \in P} \#u$), the maximal number of subsets that cover an element is denoted $n_c$ (i.e., $n_c = \max_{a \in S} \#\text{cover}(a)$), and the maximal number of subsets that are not compatible with another subset is denoted $n_i$ (i.e., $n_i = \max_{u \in P} \#\text{incompatible}(u)$).

Given a set $E \subseteq P$ of selected subsets which are all pairwise compatible, the set of elements that are not covered by a subset in $E$ is denoted $S_E$, i.e.,

$$S_E = \{a \in S : \text{cover}(a) \cap E = \emptyset\}$$

the set of subsets in $P$ that are compatible with every subset in $E$ is denoted $P_E$, i.e.,

$$P_E = \{u \in P : \forall v \in E, u \cap v = \emptyset\}$$

and for every non covered element $a \in S_E$, the set of subsets that cover $a$ and are compatible with every subset in $E$ is denoted $\text{cover}_E(a)$, i.e.,

$$\text{cover}_E(a) = \text{cover}(a) \cap P_E.$$  

Example 2. Let us consider the instance displayed in Fig. 1. If $E = \{x\}$ then $S_E = \{a, b, d, g\}$, $P_E = \{t, u, v, z\}$ and $\text{cover}_E(g) = \{t, u\}$.

3.2 Applications

A classical example of application of EC is the problem that aims at tiling a rectangle figure composed of equal squares with a set of polyominoes: the set $S$ contains an element for each square of the rectangle to tile; each subset of $P$ corresponds to the set of squares that are covered when placing a polyomino on the rectangle (for every possible position of a polyomino on the rectangle); the goal is to select a set of polyomino positions such that each square is covered exactly once (see Knuth 2000 for more details).

Another example of application is the instruction selection problem, that occurs when compiling a source code to generate an executable code: the set $S$ corresponds to the instructions of the source code; each subset of $P$ corresponds to a set of source code instructions that are covered when selecting a processor instruction; the goal is to select a set of processor instructions such that each source code instruction is covered exactly once (see Floch, Wolinski, and Kuchcinski 2010 and Hjort Blindell 2018 for more details).

Our interest for this problem comes from a conceptual clustering application which is described in Section 6.1. Other applications are described, for example, by Junttila and Kaski (2010).

If we add an objective function to the EC in order to minimize the sum of the weights of the selected subsets, we obtain the set partitioning problem. This problem occurs as
Algorithm 1: Algorithm X(S, P, E)

Input: An instance (S, P) of EC and a set E ⊆ P of selected subsets
Precondition: Subsets in E are all pairwise compatible, i.e., ∀{u, v} ⊆ E, u ∩ v = ∅
Postcondition: Output every exact cover E′ of (S, P) such that E ⊆ E′

1 begin
2 if SE = ∅ then Output E;
3 else
4 if ∀a ∈ SE, coverE(a) ≠ ∅ then
5 Choose an element a ∈ SE
6 for each subset u ∈ coverE(a) do X(S, P, E ∪ {u});

3.3 Dedicated Algorithm DLX

Knuth (2000) has introduced an algorithm called X to recursively enumerate all solutions of an instance (S, P) of EC. This algorithm is displayed in Algorithm 1 and has three input parameters: the sets S and P that define the instance of EC to solve, and a partial cover E ⊆ P that contains the subsets that have already been selected in the solution (for the first call to X, we have E = ∅). If the set SE of non-covered elements is empty, then E is a solution and the algorithm outputs it (line 2). If there is an element a ∈ SE such that coverE(a) = ∅, then a cannot be covered by any subset compatible with E and the search must backtrack. Otherwise, we choose a non-covered element a ∈ SE (line 5) and, for each subset u ∈ coverE(a), we recursively try to add u to the partial solution (line 6).

A first key point for an efficient enumeration process is to use an ordering heuristic to choose the next element a (line 5). Knuth shows that this ordering heuristic has a great impact on performance, and that much better results are obtained by selecting an element a ∈ SE for which the number of subsets compatible with E is minimal. Hence, the ordering heuristic used at line 5 chooses an element a ∈ SE such that #coverE(a) is minimal.

A second key point is to incrementally maintain SE and coverE(a) for each element a ∈ SE. To this aim, Knuth introduces Dancing Links and the implementation of Algorithm X with Dancing Links is called DLX. As illustrated in Figure 2, the idea is to use doubly linked circular lists to represent the sparse matrix that links elements and subsets. Each cell γ in this matrix has five fields denoted γ.head, γ.left, γ.right, γ.up, and γ.down, respectively.

For each subset u ∈ P, the matrix has a row which contains a cell γa for each element a ∈ u. This row is a doubly linked circular list, and we can iterate over all elements in u, starting from any cell in the row, by using left fields until returning back to the initial cell. If we use right fields instead of left fields, we also iterate over all elements in u, but we visit them in reverse order.

Besides these #P rows, there is an extra row in the matrix, which is the first row and which contains a cell ha for each non-covered element a ∈ SE. This cell is called the header
and it has an extra field, called size, which is equal to the cardinality of \( \text{cover}_E(a) \). Like the other rows, the first row is a doubly linked circular list and we can iterate over all elements in \( S_E \) by using left or right fields.

Each column of the matrix corresponds to an element \( a \in S_E \) and is composed of \#\( \text{cover}_E(a) + 1 \) cells: the header \( h_a \) plus one cell \( \gamma_{ua} \) for each subset \( u \in \text{cover}_E(a) \). Each cell \( \gamma_{ua} \) in the column can access to its header thanks to the head field (i.e., \( \gamma_{ua}.\text{head} = h_a \)). This column is a doubly linked circular list, and we can iterate over all subsets in \( \text{cover}_E(a) \), starting from the header \( h_a \), by using down fields until returning to \( h_a \). If we use up fields, we also iterate over all subsets in \( \text{cover}_E(a) \), but we visit them in reverse order.

A first advantage of using doubly linked circular lists is that a cell may be removed or restored (when backtracking) very easily. More precisely, to remove a cell \( \gamma \) from a column, we execute: \( \gamma.\text{down}.\text{up} \leftarrow \gamma.\text{up}; \gamma.\text{up}.\text{down} \leftarrow \gamma.\text{down} \). To restore \( \gamma \), we execute: \( \gamma.\text{down}.\text{up} \leftarrow \gamma; \gamma.\text{up}.\text{down} \leftarrow \gamma \). Similarly, to remove \( \gamma \) from a row, we execute: \( \gamma.\text{right}.\text{left} \leftarrow \gamma.\text{left}; \gamma.\text{left}.\text{right} \leftarrow \gamma.\text{right} \). And to restore \( \gamma \), we execute: \( \gamma.\text{right}.\text{left} \leftarrow \gamma; \gamma.\text{left}.\text{right} \leftarrow \gamma \).
A second advantage of using doubly linked lists is that they can be traversed in two directions: This way we can undo a sequence of cell removals by executing the inverse sequence of cell restorations.

Algorithms 2 and 3 describe how to update the matrix with Dancing Links:

- Algorithm 2 is called just before the recursive call (line 6 of Algorithm 1) to remove cells which are incompatible with the selected subset $u$. For each element $a \in u$, it removes the header $h_a$ of the column associated with $a$ (lines 3-4). Then, it iterates over all subsets $v \in \text{cover}_E(a)$ by traversing the column list associated with $a$, starting from its header and using down fields (lines 6-13). Each cell $\gamma_{va}$ in this column corresponds to a subset $v \in \text{cover}_E(a)$ which is incompatible with $u$ (since $a$ is already covered by $u$). Hence, for each element $b \in v$, subset $v$ must be removed from $\text{cover}_E(b)$. To this aim, the row list associated with $v$ is traversed (starting from $\gamma_{va}$ and using right fields) and, for each element $b \in v$, cell $\gamma_{vb}$ is removed from its column list (lines 9-10). Each time a cell $\gamma_{vb}$ is removed, $\gamma_{vb}.\text{head.size}$ is decremented (line 11) to ensure that this field is equal to $\#\text{cover}_E(b)$.

- Algorithm 3 is called just after the recursive call (line 6 of Algorithm 1) to restore the cells removed by Algorithm 2. It performs the same list traversals but in reverse order and restores cells instead of removing them: The elements in $u$ are visited in reverse order, the column list associated with each element $a$ in $u$ is traversed using up fields instead of down fields and row lists are traversed using left fields instead of right fields.

Example 3. Let us consider the EC instance displayed in Figure 1, and let us assume that Algorithm 1 first chooses element $c$ (line 5) and recursively calls DLX with $E = \{x\}$. Before this recursive call, Algorithm 2 iterates on elements in $x = \{c, e, f\}$:

- For element $c$, it removes cell $h_c$ from the first row and then successively removes cells $\gamma_{xc}, \gamma_{xf}$ from their columns (to remove subset $x$), and cells $\gamma_{yb}$ and $\gamma_{yb}$ from their columns (to remove subset $y$).
For element \( e \), it removes cell \( h_e \) from the first row and then successively removes cells \( \gamma_{wg} \) and \( \gamma_{wd} \) from their columns (to remove subset \( w \)).

For element \( f \), it removes the cell \( h_f \) from the headers. The size fields of the headers of the columns in which cells have been removed are updated consequently. The resulting matrix is displayed in Figure 3.

After the recursive call to DLX, Algorithm 3 iterates on elements in \( x \) in reverse order. For element \( f \), it restores cell \( h_f \) in the first row. For element \( e \), it restores cell \( h_e \) in the first row and then successively restores cells \( \gamma_{wd} \) and \( \gamma_{wg} \) in their columns. For element \( c \), it restores cell \( h_c \) in the first row and then successively restores cells \( \gamma_{yb}, \gamma_{yf}, \) and \( \gamma_{xe} \) and \( \gamma_{xe} \) in their columns.

**Property 1.** The time complexity of Algorithms 2 and 3 is \( O(n_p \cdot n_i) \).

**Proof.** Let us first study the time complexity of Algorithm 2. The loop lines 1-13 iterates on every element \( a \in u \) and the loop lines 6-13 iterates on every cell \( \gamma_{va} \) such that \( v \in \text{cover}_E(a) \). If there is a subset \( v \) such that \( u \cap v \) contains more than one element, then the cells of the row associated with \( v \) are only considered once because they are removed when treating the first element common to \( u \) and \( v \) (for example, in Fig. 3, both \( c \) and \( f \) belong to \( x \) and \( y \) but the row associated with \( y \) is traversed only once when treating \( c \)). Hence, the number of considered cells \( \gamma_{va} \) is equal to the cardinality of \( \bigcup_{a \in u} \text{cover}_E(a) \). As
$\cup_{a \in u} cover_E(a) \subseteq \cup_{a \in u} cover(a) = \{u\} \cup incompatible(u)$, the number of considered cells $\gamma_{va}$ is upper bounded by $n_i + 1$. Finally, as the loop lines 8-12 is executed $\#v$ times for each cell $\gamma_{va}$ and $\#v \leq n_p$, the time complexity of Algorithm 2 is $O(n_p \cdot n_i)$.

Algorithm 3 has the same time complexity as Algorithm 2 because it performs the same operations in reverse order.

This time complexity is an upper bound of the number of cell removals because the cardinality of $cover_E(a)$ usually decreases when adding subsets to $E$. In other words, if a cell is removed at some node of the search tree, then it will not be considered in deeper nodes in the same branch of the search tree. Hence, if we consider a whole branch of the search tree explored by Algorithm 2, lines 9-12 are performed at most once per cell in the initial matrix, i.e., $O(\sum_{u \in P} \#u)$ times.

We refer the reader to Knuth (2000) for more details on DLX. An open source implementation of DLX in C, called libexact, is described by Kaski and Pottonen (2008).

3.4 Existing Declarative Exact Approaches

CP Models. Different CP models have been proposed for solving EC. In particular, a model that uses Boolean variables is described by Hjort Blindell (2018); a model that uses a global cardinality constraint (gcc) is described by Floch, Wolinski, and Kuchcinski (2010), and a model that uses set variables is described by Chabert and Solnon (2017). These three models are experimentally compared by Chabert (2018), and these experiments show us that they have rather similar performance. In this paper, we only describe the Boolean model of Hjort Blindell (2018) (denoted BoolDec), and we refer the reader to Chabert (2018) for more details on the other models.

BoolDec uses two different kinds of variables:

- For each element $a \in S$, an integer variable $coveredBy_a$ is used to decide which subset of $P$ covers $a$, and its domain is $D(coveredBy_a) = cover(a)$;
- For each subset $u \in P$, a Boolean variable $selected_u$ indicates if $u$ is selected in the solution.

These variables are channeled by adding, for each subset $u \in P$ and each element $a \in u$, the constraint $C_{ua}$ defined by: $C_{ua} \equiv (coveredBy_a = u) \leftrightarrow (selected_u = true)$.

Property 2. Enforcing AC on BoolDec ensures the following property for each subset $u \in P$ such that $D(selected_u) = \{true\}$:

$$\forall v \in incompatible(u), \, true \notin D(selected_v) \wedge \forall a \in v, \, v \notin D(coveredBy_a).$$

In other words, every subset $v$ incompatible with $u$ cannot be selected and is removed from the domains of $coveredBy$ variables.

Proof. If $D(selected_u) = \{true\}$ then, for each element $b \in u$, the propagation of $C_{ub}$ removes all values but $u$ from $D(coveredBy_b)$. Then, for each subset $v \in incompatible(u)$, there exists at least one element $c \in v \cap u$ such that the propagation of $C_{vc}$ removes $true$ from $D(selected_c)$ (because $D(coveredBy_c) = \{u\}$). Finally, for each subset $v \in incompatible(u)$ and each element $a \in v$, the propagation of $C_{va}$ removes $v$ from $D(coveredBy_a)$ (because $true \notin D(selected_u)$).
ILP Models. Ouali et al. (2016) describe an ILP model for solving an exact cover problem which occurs in a conceptual clustering application. This ILP model associates a binary variable \( x_u \) with every subset \( u \in P \), such that \( x_u = 1 \) iff \( u \) is selected. The set of selected subsets is constrained to define a partition of \( S \) by posting the constraint: \( \forall a \in S, \sum_{u \in \text{cover}(a)} x_u = 1 \).

ILP has also been widely used to solve the set partitioning problem, the goal of which is to find an EC that minimizes the sum of the weights of the selected subsets (Rasmussen, 2011). In particular, Runberg and Larsson (2014) and Zaghrouti, Soumis, and Hallaoui (2014) show how to exploit the quasi-integrality property which implies that all integer extreme points can be reached by making simplex pivots between integer extreme points. In this case, the challenge is to find an efficient way to quickly reach an optimal integer solution. Zaghrouti et al. (2014) use a direction-finding subproblem whereas Runberg and Larsson (2014) use an all-integer column generation strategy. Both approaches are dedicated to the set partitioning problem, and they are very efficient at solving this problem (up to 500000 subsets for the approach of Zaghrouti et al., for example). However, they cannot be easily extended to the case where the goal is to maximize the minimal weight of a selected subset (which is the case of our conceptual clustering application).

Babaki, Guns, and Nijssen (2014) consider a clustering problem which aims at partitioning a set of objects into subsets so that the sum of squared distances between objects within a same subset is minimized. This problem may be formulated as a set partitioning problem with additional constraints and, as the number of subsets is exponential, they use column generation to solve it. Again, this approach assumes that the objective function is a weighted sum and it cannot be easily extended to objective functions that aim at maximizing a minimal weight.

SAT Models. SAT encodings for the exact cover problem are introduced by Junntila and Kaski (2010). Given an instance \((S, P)\) of EC, these models associate a Boolean variable \( x_u \) with every subset \( u \in P \), such that \( x_u \) is assigned to true iff subset \( u \) is selected in the exact cover. The conjunctive normal form (CNF) formula associated with \((S, P)\) is

\[ \bigwedge_{a \in S} \text{exactly-one}(\{x_u : u \in \text{cover}(a)\}) \]

where exactly-one(\(X\)) is a CNF formula which is satisfied iff exactly one variable in \(X\) is assigned to true. Junntila and Kaski describe three different encodings for exactly-one(\(X\)). The first encoding is straightforward and is defined by:

\[ \text{exactly-one}(X) = (\bigvee_{x_u \in X} x_u) \land (\bigwedge_{\{x_u, x_v\} \subseteq X} (\neg x_u \lor \neg x_v)) \]

The two other encodings are less straightforward and use auxiliary variables to reduce the number of clauses in the encoding.

Theorem 1 of Junntila and Kaski (2010) states that if the size of the search tree explored by DLX for solving an instance \((S, P)\) is equal to \(k\), then the CNF formula associated with \((S, P)\) (for any of the three SAT encodings) has, subject to an idealized variable selection heuristic, a DPLL search tree of size at most \(2k\) (where DPLL is the Davis-Putnam-Logemann-Loveland algorithm without clause learning). A consequence of this theorem...
is that modern SAT solvers (that use clause learning and restarts) may explore smaller search trees than DLX. To experimentally evaluate this, several state-of-the-art SAT solvers, especially #SAT solvers, have been compared for enumerating all solutions of EC instances, for the three encodings. These experiments show that the clasp solver (Gebser, Kaufmann, & Schaub, 2012) has the best run time behavior among the DPLL-based approaches tested by Junttila and Kaski (2010), and is also very insensitive to the applied exactly-one encoding scheme. SAT solvers have also been compared with libexact, the C implementation of DLX (Kaski & Pottonen, 2008), showing that SAT solvers actually explore smaller search spaces but do not perform that well in terms of running time: If SAT solvers are faster on some easy instances, they are often outperformed by libexact on harder instances.

4. Propagation of exactCover

In this section, we introduce a global constraint, called exactCover, for modelling EC, and three filtering algorithms for propagating it.

Definition 2. Let \((S, P)\) be an instance of EC and, for each subset \(u \in P\), let \(selected_u\) be a Boolean variable. The global constraint \(exactCover_{S,P}(selected)\) is satisfied iff the set of \(selected\) variables assigned to \(true\) corresponds to an exact cover of \(S\), i.e.,

\[
\forall a \in S, \sum_{u \in \text{cover}(a)} selected_u = 1
\]

assuming that \(true\) is encoded by 1 and \(false\) by 0.

Notations. To simplify the description of the propagators associated with exactCover, we denote \(E\) the set of subsets associated with \(selected\) variables which are assigned to \(true\) (i.e., \(E = \{u \in P : D(selected_u) = \{true\}\}\)), and we use notations introduced in Section 3.1: \(S_E\) denotes the set of elements that are not covered by a subset in \(E\), \(P_E\) denotes the set of subsets in \(P\) that are compatible with every subset in \(E\) and, for each \(a \in S_E\), \(\text{cover}_E(a)\) denotes the set of subsets in \(\text{cover}(a)\) that are compatible with every subset in \(E\).

4.1 Basic Propagator

Let us first introduce a basic propagator which ensures the same level of filtering as AC on BoolDec without using any specific data structure. This propagator called Basic is used as a baseline to evaluate the interest of using Dancing Links.

For each subset \(u \in P\), we compute the set \(\text{incompatible}(u)\) of all subsets of \(P\) that are not compatible with \(u\). These incompatibility sets are computed before starting the search process in \(O(\#P^2 \cdot np)\). They are used to propagate the assignment of a variable \(selected_u\) to \(true\) by removing \(true\) from the domain of every subset \(v\) which is incompatible with \(u\), as described in Algo. 4.

Also, to ensure that each element \(a \in S_E\) can be covered by at least one subset compatible with the selected subsets, we incrementally maintain the cardinality of \(\text{cover}_E(a)\) (without explicitly maintaining \(\text{cover}_E(a)\)) in a counter denoted \(\text{count}_a\). At the beginning of the search, \(\text{count}_a\) is initialized to \(\#\text{cover}(a)\). Then, each time a variable \(selected_v\) is assigned to \(false\), we decrement \(\text{count}_a\) for each element \(a \in v\), and we trigger a failure.
Algorithm 4: \(\text{propagate}(\text{selected}_u = \text{true})\)

1. \textbf{for} each \(v \in \text{incompatible}(u)\) \textbf{do}
2. \quad if \(\text{true} \in D(\text{selected}_v)\) then
3. \quad \quad remove \text{true} from \(D(\text{selected}_v)\)
4. \quad \quad \text{propagate}(\text{selected}_v = \text{false})

Algorithm 5: \(\text{propagate}(\text{selected}_v = \text{false})\)

1. \textbf{for} each \(a \in v\) \textbf{do}
2. \quad decrement \(\text{count}_a\)
3. \quad if \(\text{count}_a = 0\) then trigger failure;

if \(\text{count}_a = 0\), as described in Algo. 5. When backtracking, we restore counter values by performing the inverse operations.

**Property 3.** The Basic propagation algorithm ensures the following properties:

\[\forall u \in P, D(\text{selected}_u) = \{\text{true}\} \Rightarrow \forall v \in \text{incompatible}(u), \text{true} \notin D(\text{selected}_v)\]  
\[\forall a \in S_E, \text{count}_a = \#\text{cover}_E(a)\]

**Proof.** (1) is ensured by Algo. 4, and (2) is ensured by Algo. 5.

This filtering is equivalent to enforcing AC on \(\text{BoolDec}\), and in both cases a failure is triggered whenever there exists an element which cannot be covered:

- Enforcing AC on \(\text{BoolDec}\) removes from the domains of \(\text{coveredBy}\) variables the subsets that are incompatible with any selected subset \(u\) (Property 2), and a failure is triggered whenever the domain of a \(\text{coveredBy}\) variable becomes empty;

- The Basic propagator triggers a failure whenever \(\text{count}_a = \#\text{cover}_E(a) = 0\).

**Property 4.** The time complexity of the Basic propagator (Algo. 4) is \(O(n_p \cdot n_i)\).

**Proof.** The loop of Algo. 4 is executed \(\#\text{incompatible}(u)\) times, with \(\#\text{incompatible}(u) \leq n_i\), and in the worst case (if every \(v \in \text{incompatible}(u)\) is compatible with all subsets in \(E\)), for each \(v \in \text{incompatible}(u)\) the loop of Algo. 5 is executed \(\#v\) times with \(\#v \leq n_p\).

4.2 DL Propagator

In the Basic propagator, incompatibility lists are not incrementally maintained during the search: When \text{true} is removed from the domain of a variable \(\text{selected}_v\), the subset \(v\) is not removed from incompatibility lists. Therefore, Algo. 4 iterates on every subset \(v\) in \(\text{incompatible}(u)\) even if \(D(\text{selected}_v) = \{\text{false}\}\).

We propose a new propagator called DL that incrementally maintains \(\text{cover}_E(a)\) for each element \(a\), so that we only consider subsets that can be selected when propagating the assignment of a variable \(\text{selected}_u\) to \text{true}. To implement this efficiently, we use the Dancing Links described in Section 3.3.

More precisely, Algo. 2 is called each time a variable \(\text{selected}_u\) is assigned to true, and it is modified as follows:

- After line 7, if \(u \neq v\), we remove \text{true} from the domain of \(\text{selected}_v\), where \(v\) is the subset associated with the row of cell \(\gamma_v\).

- After line 11, if \(\gamma_v\.head\.size = 0\), we trigger a failure.

When backtracking from the assignment of \(\text{selected}_u\) to \text{true}, we call Algorithm 3.

**Property 5.** Basic and DL ensure the same level of consistency.
Proof. This is a direct consequence of the fact that Algo. 2 (modified as explained above) removes true from the domain of every subset which is incompatible with a selected subset. Also, for each element \(a \in S_E\), it maintains in \(h_a.\text{size}\) the value of \(#\text{cover}_E(a)\) and it triggers a failure whenever \(h_a.\text{size}\) becomes equal to 0.

Property 6. The time complexity of the DL propagation is \(O(n_p \cdot n_i)\).

Proof. The propagation algorithm has the same complexity as Algo. 2, i.e., \(O(n_p \cdot n_i)\) (see Property 1).

Comparison of Basic and DL. The propagation of the assignment of \(\text{selected}_a\) to true by DL and Basic is very similar when \(E \cap \text{incompatible}(u) = \emptyset\): Both propagators iterate on every subset \(v \in \text{incompatible}(u)\), and for every element \(b \in v\), they decrement a counter (corresponding to \(#\text{cover}_E(b)\)). However, when \(E \cap \text{incompatible}(u) \neq \emptyset\), the two propagators behave differently: Basic still iterates on every subset \(v \in \text{incompatible}(u)\) whereas DL only iterates on every subset \(v \in \bigcup_{a \in u} \text{cover}_E(a)\) (i.e., every subset \(v \in \text{incompatible}(u)\) such that \(v\) is compatible with all subsets in \(E\)). As a counterpart, DL performs more operations than Basic on each element \(b \in v\) such that \(v \in \bigcup_{a \in u} \text{cover}_E(a)\): Basic only decrements a counter whereas DL not only decrements a counter but also removes cell \(\gamma_{vb}\) in order to update \(\text{cover}_E(b)\).

These two propagators are experimentally compared in Section 4.4.

4.3 DL+ Propagator

In this section, we introduce a stronger propagator, that combines DL with an extra-filtering used by Davies and Bacchus (2011) for solving a hitting set problem. This extra-filtering exploits the following property.

Property 7. Let \((S, P)\) be an EC instance, and \(E \subseteq P\) a set of selected subsets that are all pairwise compatible. For each couple of elements \((a, b) \in S_E \times S_E\), we have:

\[
\text{cover}_E(a) \subseteq \text{cover}_E(b) \implies \forall u \in \text{cover}_E(b) \setminus \text{cover}_E(a), \text{cover}_{\{u\} \cup E}(a) = \emptyset
\]

Proof. Let \(a\) and \(b\) be two elements such that \(\text{cover}_E(a) \subseteq \text{cover}_E(b)\), and let \(u\) be a subset that covers \(b\) but not \(a\), i.e., \(u \in \text{cover}_E(b) \setminus \text{cover}_E(a)\). Every subset \(v \in \text{cover}_E(a)\) is incompatible with \(u\) (because \(b \in u \cap v\)), and must be removed from \(\text{cover}_E(a)\) if we add \(u\) to \(E\). Therefore, \(\text{cover}_{\{u\} \cup E}(a) = \emptyset\).

We propose a new propagator called DL+ that exploits this property: This propagator performs the same filtering as DL (as described in Section 4.2), but further filters domains by removing true from \(D(\text{selected}_u)\) for each subset \(u\) such that:

\[
\exists (a, b) \in S_E \times S_E, \text{cover}_E(a) \subseteq \text{cover}_E(b) \land u \in \text{cover}_E(b) \setminus \text{cover}_E(a)
\]

A key point is to efficiently detect \(\text{cover}_E\) inclusions. To this aim, we exploit the following property:

\[
\text{cover}_E(a) \subseteq \text{cover}_E(b) \iff \#(\text{cover}_E(a) \cap \text{cover}_E(b)) = \#\text{cover}_E(a).
\]
Hence, for each pair of uncovered elements \( \{a, b\} \subseteq S_E \), we maintain a counter, denoted \( \text{count}_{a \land b} \), that gives the number of subsets that both belong to \( \text{cover}_E(a) \) and \( \text{cover}_E(b) \), i.e.,

\[
\text{count}_{a \land b} = \#(\text{cover}_E(a) \cap \text{cover}_E(b))
\]

These counters are initialized in \( O(n_p^2 \cdot \#P) \). To incrementally maintain them during the search, we modify Algorithm 2 by calling a procedure before line 13: This procedure decrements \( \text{count}_{b \land c} \) for every pair of elements \( \{b, c\} \subseteq v \), where \( v \) is the subset associated with cell \( \gamma_{va} \). Indeed, as \( v \) has been removed from both \( \text{cover}_E(b) \) and \( \text{cover}_E(c) \), it must also be removed from the intersection of these two sets.

Then, at the end of Algorithm 2 (after the loop lines 1-13), for every pair of elements \( \{a, b\} \subseteq S_E \) such that \( \text{count}_{a \land b} = h_a \cdot \text{size} \), and for every subset \( v \in \text{cover}_E(b) \setminus \text{cover}_E(a) \), we remove \text{true} from \( D(\text{selected}_v) \).

We modify similarly Algorithm 3 to restore \( \text{count}_{a \land b} \) counters when backtracking.

**Property 8.** \( DL^+ \) is stronger than \( DL \).

**Proof.** \( DL^+ \) is at least as strong as \( DL \) since \( DL \) is also applied by \( DL^+ \). Moreover, the instance displayed in Fig. 3 shows us that \( DL^+ \) may filter more values than \( DL \): For example, when \( E = \{x\} \), at the end of Algorithm 2, we have \( \text{count}_{a \land d} = h_d \cdot \text{size} = 2 \) and, therefore, \( \text{selected}_i \) is assigned to \text{false} by \( DL^+ \) (and not by \( DL \)).

**Property 9.** The time complexity of \( DL^+ \) is \( O(n_p^2 \cdot n_i + \#S^2 \cdot n_c) \).

**Proof.** The complexity of the procedure called before line 13 to decrement \( \text{count}_{b \land c} \) for every pair of elements \( \{b, c\} \subseteq v \) is \( O(n_p^2) \) because \( \#v \leq n_p \). As the number of times lines 7-13 of Algorithm 2 are executed is upper bounded by \( n_i + 1 \), the time complexity of lines 1-13 becomes \( O(n_p^2 \cdot n_i) \). The procedure executed at the end of Algorithm 2 to remove \text{true} from \( D(\text{selected}_v) \) for each subset \( v \) such that there exist two elements \( a, b \in S_E \) with \( \text{count}_{a \land b} = h_a \cdot \text{size} \) and \( v \in \text{cover}_E(b) \setminus \text{cover}_E(a) \) is done in \( O(\#S^2 \cdot n_c) \) as \( \#S_E \leq \#S \) and \( \#(\text{cover}_E(b) \setminus \text{cover}_E(a)) \leq n_c \).

### 4.4 Experimental Comparison

We have implemented our three propagators in Choco 4 (Prud’homme, Fages, & Lorca, 2016), and we denote \( EC_{Basic} \) (resp. \( EC_{DL} \) and \( EC_{DL^+} \)) the Choco implementation of \( exactCover \) with the \( Basic \) (resp. \( DL \) and \( DL^+ \)) propagator.

To evaluate scale-up properties of these propagators and compare them with existing approaches, we consider the problem of enumerating all solutions of EC instances built from a same initial instance, called ERP1, which has \( \#S = 50 \) elements and \( \#P = 1580 \) subsets (this instance is described in Section 6.3). As ERP1 has a huge number of solutions, we consider instances obtained from it by selecting \( p\% \) of its subsets in \( P \), with \( p \in \{20, 25, 30, 35, 40\} \). For each value of \( p \), we have randomly generated ten instances and we report average results on these ten instances.

All experiments have been done on an Intel(R) Core(TM) i7-6700 and 65GB of RAM.
Table 1: Comparison of BoolDec, ECBasic, ECDL, ECDL+, libexact, and SAT for enumerating all solutions. For each percentage $p$ of selected subsets in ERP1, we display: the number of solutions ($\#sol$), the maximum size of a subset ($n_p$), the maximum number of subsets that cover an element ($n_e$), the maximum number of subsets that are incompatible with a subset ($n_i$), the number of choice points of BoolDec and ECDL+, and the CPU time of BoolDec, ECBasic, ECBasic, ECDL, ECDL+, libexact, and SAT (average values on ten instances per line). We report `-` when time exceeds 50,000 seconds.

Comparison of BoolDec, ECBasic, ECDL, and ECDL+. Let us first compare our three propagators with the Boolean decomposition BoolDec described in Section 3.4. We have considered the same search strategy for all models, which corresponds to the ordering heuristic introduced in Knuth (2000):

- For BoolDec, this is done by branching on coveredBy variables and using the minDom heuristic to select the next coveredBy variable to assign (as maintaining AC ensures that $D(\text{coveredBy}[a]) = \text{cover}_E(a)$);

- For ECBasic, ECDL, and ECDL+, at each node of the search tree, we search for an element $a \in S_E$ such that $\#\text{cover}_E(a) = h_a$.size is minimal, and we create a branch for each subset $u \in \text{cover}_E(a)$ where the variable $\text{selected}_a$ is assigned to true.

In all cases, we break ties by fixing an order on elements and subsets, and we consider the same order in all implementations.

Table 1 displays the number of choice points performed by BoolDec to enumerate all solutions. ECBasic and ECDL explore the same number of choice points as BoolDec since they achieve the same consistency and they consider the same ordering heuristics.

If BoolDec, ECBasic and ECDL explore the same number of choice points, Table 1 shows us that ECDL is faster than ECBasic which is faster than BoolDec. Also, when increasing $p$ (i.e., the number of subsets in $P$), the difference between EC DL and ECBasic increases: if they have very similar performance when $p = 20\%$, EC DL is nearly twice as fast as ECBasic when $p = 40\%$. Average values for the maximum size of incompatibility lists ($n_i$) are reported in Table 1, and we can see that $n_i$ is very close to the number of subsets in $P$ (when $p = 20\%$ (resp. $40\%$), $\#P = 316$ (resp. $\#P = 632$)). In other words, some subsets are incompatible with nearly all other subsets. As ECBasic exhaustively traverses the incompatibility list of every selected subset $u$ (even if some of these subsets are incompatible with previously selected subsets), it is less efficient than EC DL (which only considers subsets that belong to $\bigcup_{a \in u} \text{cover}_E(a)$).

As expected, EC DL+ explores fewer choice points than BoolDec, ECBasic and EC DL. However, the gap decreases when $p$ increases: The number of choice points explored by
Boolean, ECBasic and ECDL is 3.4 times (resp. 2.3, 1.8, 1.7 and 1.6) as large as the number of choice points explored by ECDL when \( p = 20 \) (resp. \( p = 25, 30, 35, \) and \( 40 \)). This comes from the fact that inclusions of \( \text{cover}_E \) sets become less frequent when increasing the number of subsets in \( P \). Even if the time complexity of DL+ is higher than the time complexity of DL, the reduction of the search space achieved by DL+ pays off. However, if ECDL+ is twice as fast as ECDL for small instances, the gain becomes smaller when increasing \( p \).

**Experimental Comparison with SAT and libexact.** In Table 1, we report results of SAT (using the clasp solver (Gebser et al., 2012) with the ladder encoding of Junntila and Kaski (2010) which obtains the best results), and the libexact (Kaski & Pottonen, 2008) implementation of the dedicated DLX algorithm (Knuth, 2000). libexact is always faster than ECDL+: libexact is 3 times as fast as ECDL+, and this ratio is rather constant when \( p \) increases. The gap between ECDL and libexact is explained (1) by the difference of support languages (Java for ECDL and C for libexact), and (2) by the cost of using a generic CP solver instead of a dedicated algorithm.

ECDL+ is faster than SAT, and the gap between the two approaches increases when increasing \( p \), showing that ECDL+ has better scale-up properties than SAT: ECDL+ is 10 times as fast as SAT when \( p = 20 \) and 31 times as fast when \( p = 35 \). When \( p = 40 \), SAT is not able to enumerate all solutions within the CPU time limit of 50,000 seconds whereas ECDL+ needs 4,036 seconds on average.

### 5. Constraining the Number of Selected Subsets

In some applications, we may need to add constraints on the number of selected subsets. For example, in our conceptual clustering application, the number of selected subsets corresponds to the number of clusters and we may need to constrain this number to be equal to a given value. In this case, we constrain an integer variable \( k \) to be equal to the number of selected subsets. This may be done either by adding new constraints to exactCover (as explained in Section 5.1), or by defining a new global constraint (as proposed in Section 5.2).

#### 5.1 Addition of Existing Constraints to exactCover

In this section, we study how to add constraints to \( \text{exactCover}_{S,P}(\text{selected}) \) in order to ensure that the number of selected subsets is equal to an integer variable \( k \).

A first possibility is to add the constraint: \( \sum_{u \in P} \text{selected}_u = k \). We denote ECDL, sum and ECDL+, sum the Choco implementations that combine this sum constraint with the propagation algorithms of exactCover introduced in Sections 4.2 and 4.3, respectively.

Another possibility is to use the \( N\text{Values}(X,n) \) global constraint (Pachet & Roy, 1999) which constrains the integer variable \( n \) to be equal to the number of different values assigned to variables in \( X \). To combine \( N\text{Values} \) with exactCover, we must introduce new variables such that the number of different values assigned to these variables corresponds to the number of selected subsets: For each element \( a \in S \), we define an integer variable \( \text{coveredBy}_a \) whose domain is \( D(\text{coveredBy}_a) = \text{cover}(a) \) and we channel these variables with \( \text{selected} \) variables like in the boolean model introduced in Section 3.4. In this case, the complete set
A Global Constraint for the Exact Cover Problem

Figure 4: Comparison of the number of choice points (left) and time (right) of $EC_{DL,sum}$, $EC_{DL,NV}$, $EC_{DL+},sum$, and $EC_{DL+},NV$ for enumerating all solutions when $k$ is assigned to $x$, with $x \in [2,49]$ (average on 10 instances obtained from ERP1 by randomly selecting 25% of its subsets).

The number of constraints is:

$$\forall u \in P, \forall a \in u, coveredBy_a = u \leftrightarrow selected_u = true$$

$$NValues(coveredBy,k)$$

$$exactCover_{S,P}(selected)$$

We denote $EC_{DL,NV}$ and $EC_{DL+},NV$ the Choco implementations that combine these constraints with the propagation algorithms introduced in Sections 4.2 and 4.3, respectively. In Choco, $NValues$ is decomposed into two constraints, i.e., $atLeastNValues$ and $atMostNValues$. In our experiments, we consider the strongest propagator for each of these two constraints: The propagator of $atLeastNValues$ ensures AC and the propagator of $atMostNValues$ is described by Fages and Lapègue (2014).

**Experimental Evaluation.** We consider the problem of enumerating all EC solutions when the number of selected subsets $k$ is constrained to be equal to a given value. We consider 10 instances obtained from ERP1 by selecting randomly 25% of the subsets in $P$ (the same 10 instances as in Section 4.4 when $p = 25\%$). These instances have $\#S = 50$ elements and the number of subsets is close to 400. We vary the value assigned to $k$ from 2 to $\#S - 1$. For each point $(x,y)$ in Figure 4, $y$ is the performance measure (time or number of choice points) for enumerating all solutions when $k$ is assigned to $x$ (i.e., for enumerating all exact covers with exactly $x$ selected subsets).

In Figure 4, we compare $EC_{DL,sum}$, $EC_{DL,NV}$, $EC_{DL+},sum$, and $EC_{DL+},NV$. The number of choice points is much smaller when using $NValues$, especially for extremal values of $k$. However, the propagation of $NValues$ is much more time consuming than the propagation of a sum constraint. As a consequence, using $NValues$ does not pay-off, except for very large values of $k$ (i.e., when $k > 40$) for which $NValues$ reduces the number of choice points by several orders of magnitude. Using $DL+$ instead of $DL$ for propagating $exactCover$ reduces the number of choice points, especially when $k$ is larger than 10, and this stronger filtering also reduces the run time, except for very low values of $k$: When $k$ is lower than 5, variants that use $DL$ are slightly faster than variants that use $DL+$. 

525
5.2 Definition of a New Global Constraint exactCoverK

As pointed out in the previous section, \( EC_{DL+,NV} \) explores much fewer choice points than \( EC_{DL+,sum} \), but this strong reduction of the search space pays off only for the largest values of \( k \) because the propagation of \( NValues \) is more time consuming than the propagation of a sum constraint. However, \( NValues \) is decomposed into two global constraints: \( atLeastNValues \), for which AC is ensured in polynomial time, and \( atMostNValues \), for which enforcing AC is an \( NP \)-complete problem. Bessièrè et al. (2006) introduce a propagator for \( atMostNValues \) that exploits an intersection graph. This intersection graph may be easily derived from the counters maintained by \( DL+ \) to detect \( cover_E \) inclusions. Hence, we introduce in this section a new global constraint which better propagates constraints between \( k \) and selected variables by integrating a propagator designed for \( atMostNValues \).

**Definition 3.** Let \((S,P)\) be an instance of EC, \( k \) an integer variable and, for each subset \( u \in P \), \( selected_u \) a Boolean variable. The global constraint \( exactCoverK_{S,P}(selected,k) \) is satisfied iff the number of selected variables assigned to \textit{true} is equal to \( k \) and the subsets associated with these variables define an exact cover of \( S \), i.e.,

\[
\forall a \in S, \sum_{u \in cover(a)} selected_u = 1 \\
\sum_{u \in P} selected_u = k
\]

In the next sections, we describe algorithms for updating upper and lower bounds of \( k \), and filtering domains of \textit{selected} variables when the domain of \( k \) is reduced to a singleton.

### 5.2.1 Updating the Upper Bound of \( k \)

A first simple filtering ensures that \( k \) is upper bounded by the number of selected subsets (i.e., \( \#E \)) plus the number of subsets that are compatible with \( E \) (i.e., \( \#P_E \)). We tighten this bound by taking into account the minimum number of elements that may be covered by one subset. More precisely, we need at most \( \frac{\#S_E}{\min_{u \in P} \#u} \) subsets to cover all elements in \( S_E \). Note that we round the result of the division to the largest integer value no greater than \( \frac{\#S_E}{\min_{u \in P} \#u} \). The number of elements must be an integer value. Therefore, \( k \) is upper bounded by \( \#E + \min\{\#P_E, \lfloor \frac{\#S_E}{\min_{u \in P} \#u} \rfloor \} \). This upper bound is incrementally updated with a constant time complexity.

We have experimentally compared this upper bound with the upper bound computed by propagating the global constraint \( atLeastNValues(coveredBy,k) \), and noticed that the propagation of \( atLeastNValues \) does not pay off because it is much more time consuming and it nearly never reduces the number of choice points.

### 5.2.2 Updating the Lower Bound of \( k \)

Given the set \( E \subseteq P \) of subsets that have already been selected, \( k \) is lower bounded by \( \#E \) plus the minimum number of subsets in \( P_E \) needed to cover all elements in \( S_E \). In this section, we show how to compute a lower bound of this minimum number of subsets by exploiting an algorithm introduced by Bessièrè et al. (2006) for propagating the global
constraint \textit{atMostNValues}. This algorithm exploits independent sets and independence numbers: An independent set of a graph $G = (V,E)$ is a set of vertices $S \subseteq V$ with no edge in common, \textit{i.e.}, $\forall i,j \in S, (i,j) \not\in E$, and the independence number of a graph is the maximal cardinality of its independent sets.

Bessière et al. (2006) show that the minimum number of distinct values of a set $X$ of variables is lower bounded by the independence number of the intersection graph which has a vertex $v_i$ for each variable $x_i \in X$ and an edge between two vertices $v_i$ and $v_j$ iff $D(x_i) \cap D(x_j) \neq \emptyset$. Indeed, the domains of all vertices in a same independent set have empty intersections, and therefore the corresponding variables must be assigned to different values. As a consequence, the independence number of the intersection graph is a lower bound of the minimum number of distinct values of $X$.

The interest of exploiting this property during the propagation of \textit{exactCoverK} (instead of combining \textit{exactCover} with \textit{NValues}) is that the intersection graph can be derived in a straightforward way from the counters we maintain for $DL^+$: In our context, this graph associates a vertex with every non covered element in $S_E$ and an edge with every pair of non covered elements $\{a,b\} \subseteq S_E$ such that $\text{cover}_E(a) \cap \text{cover}_E(b) \neq \emptyset$. As $DL^+$ maintains in $\text{count}_{a\land b}$ the size of $\text{cover}_E(a) \cap \text{cover}_E(b)$, edges of the intersection graph simply correspond to pairs $\{a,b\} \subseteq S_E$ such that $\text{count}_{a\land b} > 0$.

As computing the independence number of the intersection graph is \textit{NP}-hard, we compute a lower bound by constructing an independent set with the greedy algorithm of Halldórsson and Radhakrishnan (1997), as proposed by Bessière et al. (2006). Starting from an empty independent set, this algorithm iteratively adds vertices to it until the graph is empty. At each iteration, it selects a vertex $v$ of minimum degree and removes $v$ and all its adjacent vertices from the graph. The complexity of this algorithm is linear with respect to the number of edges in the intersection graph, provided that buckets are used to incrementally maintain the set of vertices of degree $d$ for every $d \in [0, \#S_E - 1]$.

**Example 4.** In Fig. 5, we display the two intersection graphs associated with the instance displayed in Fig. 1 when $E = \emptyset$ (left) and when $E = \{x\}$ (right).

When $E = \emptyset$, the greedy algorithm builds the independent set $\{a,e,b\}$, and the lower bound computed for $k$ is $\#E + \#\{a,e,b\} = 3$.

When $E = \{x\}$, the greedy algorithm builds the independent set $\{b,a\}$ (or $\{b,d\}$), and the lower bound computed for $k$ is $\#E + \#\{b,a\} = 3$. 

![Figure 5: Intersection graphs associated with the instance displayed in Fig. 1 when $E = \emptyset$ (left) and when $E = \{x\}$ (right).](image-url)
5.2.3 Use of Independent Sets to Filter selected Variable Domains

Bessière et al. (2006) also show how to use independent sets to filter domains when the cardinality of the independent set is equal to the number of different values. In our context, this filtering allows us to assign false to some selected variables. More precisely, when the domain of \( k \) is reduced to the singleton \( \{\#I\} \) where \( I \) is the independent set, for every subset \( u \) that does not cover an element of \( I \) (i.e., \( u \not\in \bigcup_{a \in I} \text{cover}_E(a) \)), we can assign false to \( \text{selected}_u \).

This filtering may be done not only for \( I \), but also for any other independent set that has the same cardinality as \( I \). However, as this is too expensive to compute all independent sets that have the same cardinality as \( I \), we only compute a subset of them using the algorithm described by Beldiceanu (2001). This algorithm computes in linear time with respect to \( \#S_E \) all independent sets that differ from \( I \) by only one vertex: It iterates on every vertex \( a \in I \) and, for every edge \( \{a, b\} \) such that \( b \) is not adjacent to any vertex of \( I \setminus \{a\} \), it adds the independent set \( I \setminus \{a\} \cup \{b\} \).

Let \( I_0 \) be the initial independent set computed with the greedy algorithm, and \( I_1, \ldots, I_n \) be the independent sets derived from \( I_0 \). We remove true from the domain of every variable \( \text{selected}_u \) such that \( u \not\in \bigcap_{j \in [0,n]} \bigcup_{a \in I_j} \text{cover}_E(a) \).

**Example 5.** On our running example, when \( E = \{x\} \), the greedy algorithm builds a first independent set which is either \( \{b, a\} \) or \( \{b, d\} \). Hence, the lower bound of \( k \) is 3. The upper bound of \( k \) is also equal to 3 because \( \#E + \min\{\#P_E, \lceil \frac{\#S_E}{\min_{u \in P} \#u} \rceil \} = 1 + \min\{4, \frac{1}{2}\} = 3 \).

Therefore, the domain of \( k \) is reduced to the singleton \( \{3\} \) and we can apply the filtering on selected domains. We derive from the first independent set (i.e., either \( \{b, a\} \) or \( \{b, d\} \)) a second independent set (i.e., \( \{b, d\} \) if the first independent set is \( \{b, a\} \), and \( \{b, a\} \) otherwise). We have \( \text{cover}_E(b) = \{z\} \), \( \text{cover}_E(a) = \{t, u, v\} \) and \( \text{cover}_E(d) = \{u, v\} \). We can remove true from the domain of every selected variable associated with a subset that does not belong to: \( \{u, v, z\} \cap \{t, u, v, z\} = \{u, v, z\} \). Therefore, we remove true from the domain of selected_1.

**Experimental Evaluation.** We denote ECK the Choco implementation of exactCoverK, which combines the DL+ propagator described in Section 4.3 with the propagator described in this section. We experimentally compare ECK with \( E_{DL+,NV} \) and \( E_{DL+,sum} \) in Fig. 6.

ECK and \( E_{DL+,NV} \) explore a similar amount of choice points: ECK explores slightly fewer choice points than \( E_{DL+,NV} \) when \( k < 16 \) and vice-versa for higher values of \( k \). As expected, the use of NV values is much more time consuming than our propagation algorithm. However, our filtering does not pay off compared with \( E_{DL+,sum} \) when \( 17 \leq k \leq 27 \) even if it explores almost twice fewer choice points.

6. Experimental Evaluation on a Conceptual Clustering Application

Clustering aims at grouping objects into homogeneous and well separated clusters. The key idea of conceptual clustering is that every cluster is not only characterized by its set of objects but also by a conceptual description such as, for example, a set of shared properties (Michalski & Stepp, 1983). Many approaches (such as, for example, COBWEB introduced by Fisher (1987)) build hierarchies of conceptual clusters in an incremental and greedy way.
that does not ensure the optimality of the final hierarchy. Formal Concept Analysis (Ganter & Wille, 1997) is a particular case of conceptual clustering where data are structured by means of formal concepts, i.e., sets of objects that share a same subset of attributes. Formal concepts are partially ordered, and we may compute lattices of formal concepts, as proposed by Carpineto and Romano (1993), for example. Guns, Nijssen, and Raedt (2013) introduce the problem of \( k \)-pattern set mining, concerned with finding a set of \( k \) related patterns under constraints, and they show that this problem may be used to solve a particular case of conceptual clustering problem: The goal of this problem is to find a subset of \( k \) formal concepts which is a partition of the initial set of objects and which maximizes the minimal weight of a selected formal concept. In this section, we experimentally evaluate the interest of our global constraint on this particular case of conceptual clustering problem. Indeed, this problem has been widely studied since its introduction by Guns et al., and different CP and ILP approaches have been recently proposed for solving it. Furthermore, this problem occurs in an industrial application which aims at mining a catalog of configuration parts from existing configurations of an ERP (Enterprise Resource Planning) system (Chabert, 2018), and we consider instances coming from this application in our experimental study.

In Section 6.1, we formally define the problem and describe existing approaches for solving it. As this problem involves optimizing some utility measures associated with the selected subsets, we show how to extend our global constraints in order to add constraints on bounds of these utility measures in Section 6.2. In Section 6.3, we describe the experimental setup. In Sections 6.4, 6.5 and 6.6, we report experimental results on different problems, where the number of clusters is either fixed to a given value (in Section 6.4) or bounded within a given interval of values (in Sections 6.5 and 6.6), and where the goal is either to optimize a single objective function (in Sections 6.4 and 6.5) or to compute the whole Pareto front of non-dominated solutions for two conflicting objective functions (in Section 6.6).

### 6.1 Definition of the Problem

**Definition 4.** Let \( \mathcal{O} \) be a set of objects, and for each object \( o \in \mathcal{O} \), let \( \text{attr}(o) \) be the set of attributes that describes \( o \).
\begin{align*}
\text{attr}(o_1) &= \{a_1, a_2, a_4\} & \text{attr}(o_3) &= \{a_2, a_4\} & \text{attr}(o_5) &= \{a_1, a_3\} \\
\text{attr}(o_2) &= \{a_1, a_3, a_4\} & \text{attr}(o_4) &= \{a_2, a_3\}
\end{align*}

Figure 7: Example of dataset with 5 objects and 4 attributes.

<table>
<thead>
<tr>
<th>$\mathcal{F}$</th>
<th>intent</th>
<th>subset of objects</th>
<th>frequency</th>
<th>size</th>
<th>diameter</th>
<th>split</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_0$</td>
<td>$\emptyset$</td>
<td>${o_1, o_2, o_3, o_4, o_5}$</td>
<td>$5$</td>
<td>$0$</td>
<td>$1/3$</td>
<td>$1/0$</td>
</tr>
<tr>
<td>$c_1$</td>
<td>${a_1}$</td>
<td>${o_1, o_2, o_5}$</td>
<td>$3$</td>
<td>$1$</td>
<td>$3/4$</td>
<td>$1/3$</td>
</tr>
<tr>
<td>$c_2$</td>
<td>${a_2}$</td>
<td>${o_1, o_3, o_4}$</td>
<td>$3$</td>
<td>$1$</td>
<td>$3/4$</td>
<td>$1/2$</td>
</tr>
<tr>
<td>$c_3$</td>
<td>${a_3}$</td>
<td>${o_2, o_4, o_5}$</td>
<td>$3$</td>
<td>$1$</td>
<td>$3/4$</td>
<td>$1/2$</td>
</tr>
<tr>
<td>$c_4$</td>
<td>${a_4}$</td>
<td>${o_1, o_2, o_3}$</td>
<td>$3$</td>
<td>$1$</td>
<td>$3/4$</td>
<td>$2/3$</td>
</tr>
<tr>
<td>$c_5$</td>
<td>${a_1, a_3}$</td>
<td>${o_2, o_5}$</td>
<td>$2$</td>
<td>$2$</td>
<td>$2/3$</td>
<td>$1/2$</td>
</tr>
<tr>
<td>$c_6$</td>
<td>${a_1, a_4}$</td>
<td>${o_1, o_2}$</td>
<td>$2$</td>
<td>$2$</td>
<td>$1/2$</td>
<td>$1/3$</td>
</tr>
<tr>
<td>$c_7$</td>
<td>${a_2, a_3}$</td>
<td>${o_4}$</td>
<td>$1$</td>
<td>$2$</td>
<td>$0$</td>
<td>$2/3$</td>
</tr>
<tr>
<td>$c_8$</td>
<td>${a_2, a_4}$</td>
<td>${o_1, o_3}$</td>
<td>$2$</td>
<td>$2$</td>
<td>$1/3$</td>
<td>$1/2$</td>
</tr>
<tr>
<td>$c_9$</td>
<td>${a_1, a_3, a_4}$</td>
<td>${a_2}$</td>
<td>$1$</td>
<td>$3$</td>
<td>$0$</td>
<td>$1/2$</td>
</tr>
<tr>
<td>$c_{10}$</td>
<td>${a_1, a_2, a_4}$</td>
<td>${o_1}$</td>
<td>$1$</td>
<td>$3$</td>
<td>$0$</td>
<td>$1/3$</td>
</tr>
<tr>
<td>$c_{11}$</td>
<td>${a_1, a_2, a_3, a_4}$</td>
<td>$\emptyset$</td>
<td>$0$</td>
<td>$4$</td>
<td>$0$</td>
<td>$0$</td>
</tr>
</tbody>
</table>

Table 2: The set $\mathcal{F}$ of all formal concepts contained in the dataset described in Table 7.

- The \textit{intent} of a subset of objects $O_i \subseteq \mathcal{O}$ is the set of attributes common to all objects in $O_i$, \textit{i.e.}, $\text{intent}(O_i) = \cap_{o \in O_i} \text{attr}(o)$.

- A subset of objects $O_i \subseteq \mathcal{O}$ is a formal concept if it contains every object whose set of attributes is a superset of its intent, \textit{i.e.}, $O_i = \{o \in \mathcal{O} : \text{intent}(O_i) \subseteq \text{attr}(o)\}$.

- A conceptual clustering is a partition of $\mathcal{O}$ in $k$ formal concepts $O_1, \ldots, O_k$, \textit{i.e.}, $\forall o \in \mathcal{O}, \#\{i \in [1, k] : o \in O_i\} = 1$.

\textbf{Example 6.} In Table 2, we list all formal concepts associated with the dataset displayed in Fig. 7. Examples of conceptual clusterings are $\{c_2, c_5\}$ and $\{c_5, c_7, c_8\}$.

\textbf{Quality Measures Associated with Formal Concepts.} Two classical measures for evaluating the quality of a formal concept $O_i \subseteq \mathcal{O}$ are the \textit{frequency}, which corresponds to its number of objects (\textit{i.e.}, $\text{frequency}(O_i) = \#O_i$), and the \textit{size}, which corresponds to its number of attributes (\textit{i.e.}, $\text{size}(O_i) = \#\text{intent}(O_i)$).

Two other measures that are often used to evaluate the quality of a group of objects for clustering applications are the \textit{diameter} and the \textit{split}. These two measures assume that there exists a distance measure $d : \mathcal{O} \times \mathcal{O} \to \mathbb{R}$ between objects. In our experiments, we consider the distance of Jaccard (1901) which depends on the number of common attributes, \textit{i.e.}, $d(o, o') = 1 - \frac{\#(\text{attr}(o) \cap \text{attr}(o'))}{\#(\text{attr}(o) \cup \text{attr}(o'))}$. Given this distance measure $d$, the \textit{diameter} of a formal concept $O_i \subseteq \mathcal{O}$ evaluates its homogeneity by the maximal distance between objects in $O_i$ (\textit{i.e.}, $\text{diameter}(O_i) = \max_{o, o' \in O_i} d(o, o')$ if $\#O_i > 1$ and $\text{diameter}(O_i) = 0$ otherwise) whereas the \textit{split} evaluates its separation with other objects by the minimal distance between
objects in $O_i$ and objects in $O \setminus O_i$ (i.e., $\text{split}(O_i) = \min_{o \in O_i, o' \in O \setminus O_i} d(o, o')$ if $O_i \neq O$ and $\text{split}(O_i) = 0$ otherwise).

Many other quality measures could be defined, depending on the applicative context. In this section, we report experimental results for these four quality measures which are widely used and rather representative. We denote $Q = \{ \text{frequency}, \text{size}, -\text{diameter}, \text{split} \}$ this set of four quality measures. Note that for each quality measure $q \in Q$, the higher $q(O_i)$, the better the quality of $O_i$ (when the quality measure is the diameter, we define $q(O_i) = -\text{diameter}(O_i)$ as smaller diameter values indicate more homogeneous clusters).

**Optimization Criteria for Conceptual Clustering.** There may exist different solutions to a conceptual clustering problem, and we may add an objective function to search for the best solution. In this paper, we consider the case where we maximize an objective variable denoted $\min_q$ (where $q \in Q$ is a quality measure) which is constrained to be equal to the smallest quality among the selected formal concepts, i.e., $\min_q = \min_{O_i \in \{O_1, \ldots, O_k\}} q(O_i)$. By maximizing $\min_q$, we ensure a minimal quality over all clusters, and this is well suited for many applications such as, for example, the ERP configuration problem addressed by Chabert (2018).

**Example 7.** In Table 2, we give for each formal concept of the dataset the value of each quality measure defined above. For the conceptual clustering $\{c_5, c_7, c_8\}$, we have: $\min_{\text{frequency}} = 1$, $\min_{\text{size}} = 2$, $\min_{-\text{diameter}} = -2/3$, and $\min_{\text{split}} = 1/2$.

**Computation of Formal Concepts.** Formal concepts correspond to closed itemsets (Pasquier, Bastide, Taouil, & Lakhal, 1999) and the set of all formal concepts may be computed by using algorithms dedicated to the enumeration of frequent closed itemsets. In particular, LCM (Uno, Asai, Uchida, & Arimura, 2004) is able to extract all formal concepts in linear time with respect to the number of formal concepts.

Constraint Programming (CP) has been widely used to model and solve itemset search problems (Raedt, Guns, & Nijsen, 2008; Khiari, Boizumault, & Crémilleux, 2010; Guns, Nijsen, & Raedt, 2011; Guns, 2015; Lazaar, Lebhab, Loudni, Maamar, Lemière, Bessiere, & Boizumault, 2016; Schaus, Aoga, & Guns, 2017; Ugarte, Boizumault, Crémilleux, Lepailleur, Loudni, Plantevit, Raïssi, & Soulet, 2017). Indeed, CP allows the user to easily model various constraints on the searched itemsets. The propagation of these constraints reduces the search space and allows CP to be competitive with dedicated approaches such as LCM for extracting constrained itemsets.

**CP for Conceptual Clustering.** The conceptual clustering problem we consider here is a special case of $k$-pattern set mining, as introduced by Guns et al. (2013): This problem is defined by combining a cover and a non-overlapping constraint, and a binary CP model is proposed to solve this problem. Dao, Duong, and Vrain (2017) describe a CP model for clustering problems where a dissimilarity measure between objects is provided, and this CP model has been extended to conceptual clustering by Dao, Lesaint, and Vrain (2015). Experimental results reported by Dao et al. (2015) show that this model outperforms the binary model of Guns et al. (2013). Chabert and Sohon (2017) introduce another CP model, which improves the model of Dao et al. (2015) when the number of clusters is not fixed.
Conceptual Clustering as an Exact Cover Problem. The set $F$ of all formal concepts may be efficiently computed with dedicated tools such as LCM (Uno et al., 2004). Given this set, a conceptual clustering problem may be seen as an exact cover problem, the goal of which is to find a subset of formal concepts $E \subseteq F$ that covers every object exactly once, i.e., $\forall o \in O, \#\{O_i \in E : o \in O_i\} = 1$. This exact cover problem may be solved by using any approach described in Section 3.4. In particular, Ouali et al. (2016) propose to use ILP, and they show that ILP is very convenient and efficient for modeling and solving conceptual clustering problems given the set of all formal concepts.

6.2 Extension of exactCover to exactCoverQ

We propose to use exactCover to solve conceptual clustering problems in a two-step approach: In a first step we use LCM to extract the set $F$ of all formal concepts, and in a second step we use exactCover to select a subset of $F$ which is an exact cover of $O$. However, as pointed out previously, we add an objective function to search for an exact cover $E$ that maximizes $Min_q$, where $q \in Q$ is the measure which evaluates the quality of a formal concept. Furthermore, in some cases it may be useful to add constraints on minimal and/or maximal measures associated with selected formal concepts (this is the case, for example, when considering several quality measures and computing the Pareto front of all non dominated solutions with respect to these measures).

Hence, we extend exactCover and exactCoverK to the case where quality measures are associated with subsets.

Definition 5. Let $(S, P)$ be an instance of EC and, for each subset $u \in P$, let $selected_u$ be a Boolean variable. Let $n$ be the number of different quality measures and, for each $i \in [1, n]$ and each subset $u \in P$, let $q_i(u)$ denote the $i^{th}$ quality measure associated with $u$. For each $i \in [1, n]$, let $MinQ_i$ and $MaxQ_i$ be two integer variables. The global constraint $exactCoverQ_{S,P,q}(selected, MinQ, MaxQ)$ is satisfied iff all $selected_u$ variables assigned to true correspond to an exact cover of $(S, P)$ and $MinQ$ and $MaxQ$ variables are assigned to the minimum and maximum quality associated with selected subsets, i.e.,

$$\forall a \in S, \sum_{u \in cover(a)} selected_u = 1$$

$$\forall i \in [1, n], MinQ_i = \min_{u \in P, selected_u=true} q_i(u)$$

$$\forall i \in [1, n], MaxQ_i = \max_{u \in P, selected_u=true} q_i(u)$$

Similarly, we define the global constraint $exactCoverQK_{S,P,q}(selected, k, MinQ, MaxQ)$ which further ensures that the integer variable $k$ is equal to the number of selected subsets, i.e., $\sum_{u \in P} selected_u = k$.

Propagation of exactCoverQ (resp. exactCoverQK). This constraint is propagated like exactCover (resp. exactCoverK), but before starting the search we remove from $P$ every subset $u$ that does not satisfy the bound constraints, i.e., such that there exists $i \in [1, n]$ for which $q_i(u) \notin [MinQ_i.lb, MaxQ_i.ub]$ (where $x.lb$ and $x.ub$ respectively denote the smallest and greatest value in the domain of a variable $x$). Then, each time a variable $selected_u$ is
Table 3: Benchmark: for each instance, \( \#O \) gives the number of objects, \( \#A \) gives the number of attributes, \( \#F \) gives the number of formal concepts, and \( t \) gives the time (in seconds) spent by LCM to compute the set \( F \) of all formal concepts.

<table>
<thead>
<tr>
<th>Name</th>
<th>#O</th>
<th>#A</th>
<th>#F</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERP1</td>
<td>50</td>
<td>27</td>
<td>1,580</td>
<td>0.01</td>
</tr>
<tr>
<td>ERP2</td>
<td>47</td>
<td>47</td>
<td>8,133</td>
<td>0.03</td>
</tr>
<tr>
<td>ERP3</td>
<td>75</td>
<td>36</td>
<td>10,835</td>
<td>0.03</td>
</tr>
<tr>
<td>ERP4</td>
<td>84</td>
<td>42</td>
<td>14,305</td>
<td>0.05</td>
</tr>
<tr>
<td>ERP5</td>
<td>94</td>
<td>53</td>
<td>63,633</td>
<td>0.28</td>
</tr>
<tr>
<td>ERP6</td>
<td>95</td>
<td>61</td>
<td>71,918</td>
<td>0.45</td>
</tr>
<tr>
<td>ERP7</td>
<td>160</td>
<td>66</td>
<td>728,537</td>
<td>5.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>#O</th>
<th>#A</th>
<th>#F</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCI1 (zoo)</td>
<td>101</td>
<td>36</td>
<td>4,567</td>
<td>0.01</td>
</tr>
<tr>
<td>UCI2 (soybean)</td>
<td>630</td>
<td>50</td>
<td>31,759</td>
<td>0.10</td>
</tr>
<tr>
<td>UCI3 (primary-tumor)</td>
<td>336</td>
<td>31</td>
<td>87,230</td>
<td>0.28</td>
</tr>
<tr>
<td>UCI4 (lymph)</td>
<td>148</td>
<td>68</td>
<td>154,220</td>
<td>0.52</td>
</tr>
<tr>
<td>UCI5 (vote)</td>
<td>435</td>
<td>48</td>
<td>227,031</td>
<td>0.68</td>
</tr>
<tr>
<td>UCI6 (hepatitis)</td>
<td>137</td>
<td>68</td>
<td>3,788,341</td>
<td>13.90</td>
</tr>
</tbody>
</table>

...assigned to \( \text{true} \), for each \( i \in [1, n] \), we propagate:

\[
\begin{align*}
\text{Min}Q_i.ub &= \min \{ \text{Min}Q_i.ub, q_i(u) \}, \\
\text{Max}Q_i.lb &= \max \{ \text{Max}Q_i.lb, q_i(u) \}.
\end{align*}
\]

Also, each time \( \text{Min}Q_i.lb \) or \( \text{Max}Q_i.ub \) is updated, for each subset \( u \) such that \( q_i(u) \notin [\text{Min}Q_i.lb, \text{Max}Q_i.ub] \), we remove \( \text{true} \) from the domain of \( \text{selected}_u \).

6.3 Experimental Setup

**Benchmark.** We describe in Table 3 six classical machine learning instances, coming from the UCI database, and six ERP instances coming from an ERP configuration problem described by Chabert and Solnon (2017).

Let us recall that to solve conceptual clustering problems with a two-step approach, we first compute the set \( F \) of all formal concepts with a dedicated tool such as LCM, and then we solve an exact cover problem \((S, P)\) such that \( S \) is the set of all objects (i.e., \( S = O \)) and \( P \) is the set of all formal concepts (i.e., \( P = F \)). The number of objects \( \#O \) varies from 47 to 630, and the number of formal concepts \( \#F \) varies from 1,580 to 3,788,341. The time spent by LCM to compute \( F \) is smaller than one second for all instances but two. The two harder instances (ERP7 and UCI6) are solved in 5.31 and 13.9 seconds, respectively.

**Considered Implementations of our Global Constraints.** All our global constraints and propagators are implemented with Choco v.4.0.3 (Prud’homme et al., 2016). We consider the following models:

- \( ECQ_{*,\text{sum}} \), with \( * \in \{ DL, DL^+ \} \), which combines the exactCoverQ constraint (propagated with the algorithms described in Sections 4.2 or 4.3 depending on whether \( * = DL \) or \( * = DL^+ \)) with a sum constraint, as described in 5.1;

- \( ECQK \), which is the model that uses the exactCoverQK constraint.

We consider the ordering heuristic introduced by Knuth (2000): At each node, we search for the element \( a \in S_E \) such that \( \#cover_E(a) \) is minimal and, for each subset \( u \in cover_E(a) \), we create a branch where \( \text{selected}_u \) is assigned to \( \text{true} \).
Considered Implementations of Other Declarative Approaches. We compare ECQK and ECQs,\textit{sum} with the following declarative approaches:

- \textit{FCP1}, the full CP model introduced by Dao et al. (2015), and implemented with Gecode (2005);
- \textit{FCP2}, the full CP model introduced by Chabert and Solnon (2017), and implemented with Choco v.4.0.3;
- \textit{ILP}, the hybrid approach introduced by Ouali et al. (2016), and implemented with CPLEX v12.7. Note that ILP approaches dedicated to the set partitioning problem (such as Rasmussen (2011), Runberg and Larsson (2014), or Zaghrouti et al. (2014), for example) cannot be used to solve our problem as the goal is not to minimize a weighted sum but to maximize the minimal weight of a selected subset.

Performance Measures. We consider two different performance measures, \textit{i.e.}, the number of choice points and the CPU time. All experiments were conducted on Intel(R) Core(TM) i7-6700 with 3.40GHz of CPU and 65GB of RAM, using a single thread.

For all hybrid approaches that use LCM to extract all formal concepts in a preprocessing step, and then solve an exact cover problem (\textit{i.e.}, ILP, ECQs,\textit{sum}, and ECQK), CPU times that are reported always include the time spent by LCM to extract all formal concepts (see Table 3 for information on this time).

6.4 Single Criterion Optimization with \(k\) Fixed

In this section, we consider the problem of maximizing \(\text{Min}_q\) (with \(q \in Q\)) when the number of clusters \(k\) is fixed to a given value that ranges from 2 to 10. For this experiment which is rather time-consuming (it involves solving one instance per value of \(k\)), we only report results for six instances, \textit{i.e.}, ERP2 to ERP4, and UCI1 to UCI3.

Fig. 8 reports the number of nodes explored by ECQs,\textit{sum} and ECQK for values of \(k\) ranging between 2 and 10. For \(\text{Min}_{\text{frequency}}\), ECQDL,\textit{sum} usually explores much fewer nodes than ECQDL,\textit{sum} whereas ECQK often explores the same number of nodes as ECQDL,\textit{sum}.

For \(\text{Min}_{\text{split}}\), the three propagators explore rather similar numbers of choice points when \(k\) is small. When \(k\) increases, ECQK often explores fewer choice points than ECQs,\textit{sum}, but the difference is moderate. Finally, for \(\text{Min}_{\text{size}}\) and \(\text{Min}_{\text{diameter}}\), ECQK explores much fewer choice points than ECQDL,\textit{sum}, and ECQDL,\textit{sum} and ECQDL,\textit{sum} nearly always explore the same number of choice points. For these criteria, many instances are not solved within a CPU time limit of 1000 seconds by ECQs,\textit{sum}. ECQK is able to solve more instances, but it fails at solving ERP4 and UCI2 when \(k > 6\) for \(\text{Min}_{\text{size}}\), and UCI3 when \(k > 4\) for \(\text{Min}_{\text{size}}\) and \(\text{Min}_{\text{diameter}}\).

In Fig. 9, we compare CPU times of ECQK (which is the best performing propagator for exactCoverQ), FCP1 and ILP. We do not report CPU times of FCP2 because it is outperformed by FCP1. ECQK, FCP1 and ILP have complementary performance:

- For \(\text{Min}_{\text{split}}\) and \(\text{Min}_{\text{diameter}}\) FCP1 is very efficient and clearly outperforms ECQK and ILP. For these two criteria, ECQK is always faster than ILP, except for UCI3 when \(k = 4\).
A Global Constraint for the Exact Cover Problem

Figure 8: Number of nodes of ECQ_{DL, sum}, ECQ_{DL+, sum}, and ECQK to maximize $Min_{frequency}$, $Min_{split}$, $Min_{size}$, and $Min_{-diameter}$ (from top to bottom), when $k$ is assigned to $x$, with $x \in [2, 10]$. Results are reported only when the time is smaller than 1000 seconds.

- For $Min_{frequency}$ and $Min_{size}$, FCP1 is the fastest approach when $k = 2$, but it does not scale well when $k$ increases, and it is not able to solve instances when $k > 5$. For $Min_{frequency}$, ECQK is faster than ILP (except for UCI1 when $k \in \{9, 10\}$), and
Figure 9: CPU time of ECQK, FCP1, and ILP to maximize $\text{Min}_{\text{frequency}}$, $\text{Min}_{\text{split}}$, $\text{Min}_{\text{size}}$ and $\text{Min}_{\text{diameter}}$ (from top to bottom) when $k$ is assigned to $x$, with $x \in [2, 10]$. Results are reported only when the CPU time is smaller than 1000 seconds.

ECQK is the only approach that is able to solve all instances. For $\text{Min}_{\text{size}}$, ECQK and ILP have rather comparable performance for ERP2, ERP3, and UCI3. However, ECQK is outperformed by ILP for ERP4, UCI1, and UCI2.
As a conclusion, if ECQK is not the best approach on every instance, it is the approach which solves the largest number of instances within the CPU time limit of 1000 seconds: Among the $9 \times 6 \times 4 = 216$ considered instances, ECQK solves 194 instances whereas ILP and FCP1 solve 187 and 147 instances, respectively.

### 6.5 Single Criterion Optimization with $k$ Bounded

In some applicative contexts, we do not know a priori the number of clusters and, therefore, $k$ is not fixed. This is the case, for example, in the application to ERP configuration (Chabert & Solnon, 2017; Chabert, 2018). In this case, we only constrain $k$ to be strictly greater than 1 and strictly smaller than the number of objects, i.e., $D(k) = [2, \#O - 1]$. In other words, we want more than one cluster and at least one cluster must contain two objects.

When $k$ is not fixed, there is a huge number of solutions, and we refine the ordering heuristic in order to favor the construction of good solutions first. As the goal is to maximize $Min_q$, this is done by branching first on subsets $u \in \text{cover}_E(a)$ such that $q(u)$ is maximal (where $a$ is an element in $S_E$ such that $\#\text{cover}_E(a)$ is minimal). However, we apply this ordering heuristic only when $q \in \{\text{size, split, } -\text{diameter}\}$. When the goal is to maximize $Min_{\text{frequency}}$, we use the objectiveStrategy ordering heuristic (Prud’homme et al., 2016) which performs a dichotomous branching over $Min_{\text{frequency}}$. Indeed, in this case, the sum of frequencies of the subsets in an exact cover is equal to the number of objects. As a consequence, better solutions are obtained by favoring the selection of subsets of medium frequencies (instead of large frequencies) because once a first subset $u$ has been selected, we know that $Min_{\text{frequency}}$ is upper bounded by $\#E - \text{frequency}(u)$.

Table 4 displays the results of $ECQ_{DL,\text{sum}}$, $ECQ_{DL+,\text{sum}}$, FCP1, FCP2, and ILP when maximizing $Min_q$ with $q \in \{\text{size, split, } -\text{diameter, frequency}\}$. We do not report results of ECQK because, when $k$ is not fixed, the advanced bound computations and filterings described in Section 5.2 nearly never reduce the number of choice points (compared to $ECQ_{DL+,\text{sum}}$).

$ECQ_{DL+,\text{sum}}$ is the only approach able to solve the 52 instances within the time limit of 1000 seconds. It never spends more than 104 seconds to solve an instance, and its average solving time is equal to 9s. Both $ECQ_{DL,\text{sum}}$ and FCP2 are able to solve all instances but one, and their average solving time on the 51 solved instances are equal to 13.2s and 25.8s, respectively. FCP1 fails at solving four instances, and its average solving time on the 48 solved instances is equal to 68.3s. Finally, ILP fails at solving 21 instances, and its average solving time on the 31 solved instances is equal to 286.1s.

However, if $ECQ_{DL+,\text{sum}}$ is the only approach able to solve all instances within a time limit of 1000s, and if it has the smallest average solving time, there are only 10 instances for which $ECQ_{DL+,\text{sum}}$ is the fastest approach. Indeed, if $ECQ_{DL+,\text{sum}}$ explores much fewer choice points than $ECQ_{DL,\text{sum}}$ for the $Min_{\text{frequency}}$ criterion, $ECQ_{DL,\text{sum}}$ and $ECQ_{DL+,\text{sum}}$ often explore the same number of choice points (and when this is not the case, the difference is very small) for the three other criteria. As a consequence, the stronger propagation of $DL+$ only pays off for the $Min_{\text{frequency}}$ criterion, and for the three other criteria $ECQ_{DL+,\text{sum}}$ is never faster than $ECQ_{DL,\text{sum}}$. 
Table 4: Comparison of $ECQ_{DL,sum}$, $ECQ_{DL+,sum}$, $FCP1$, $FCP2$ and $ILP$ for monocriterion problems $Min_{size}$ (top left), $Min_{frequency}$ (top right), $Min_{split}$ (bottom left), and $Min_{-diameter}$ (bottom right). '-' is reported when time exceeds 1,000s. For each instance, the fastest approach is highlighted in blue.

$FCP1$ is very efficient on most instances for $Min_{split}$ and $Min_{-diameter}$ criteria. However, it has rather poor performance on $Min_{size}$ and $Min_{frequency}$ criteria. $FCP2$ is the fastest approach on some instances for $Min_{size}$ and $Min_{frequency}$ criteria, but on some other instances it has rather poor performance. $ILP$ is not competitive with CP approaches.

In Fig. 10, we plot the evolution of the cumulative number of solved instances with respect to time for the five different approaches. If $FCP1$ is able to solve more instances for time limits smaller than 0.6s, it is outperformed by $ECQ_{DL,sum}$ when the time limit is greater than 0.6s. For time limits greater than 40s, $ECQ_{DL+,sum}$ is able to solve more instances than all other approaches.

### 6.6 Multi-Criteria Optimization

Solutions that maximize $Min_{size}$ or $Min_{-diameter}$ usually have a very large number of clusters (close to $\#O - 1$), whereas solutions that maximize $Min_{frequency}$ or $Min_{split}$ usually have...
Figure 10: Cumulative number of solved instances with respect to time: For each approach $f \in \{ECQ_{DL,\text{sum}}, ECQ_{DL,\text{sum}+}, FCP1, FCP2, ILP\}$, we plot the curve $f(x) = y$ such that $y$ is the number of instances which are solved by $f$ within a time limit of $x$ seconds.

Very few clusters (close to 2). To obtain different kinds of compromise solutions, ranging from solutions that have very few clusters (with high values of $Min_{\text{frequency}}$ and $Min_{\text{split}}$) to solutions that have a lot of clusters (with high values of $Min_{\text{size}}$ and $Min_{\text{diameter}}$), we may compute Pareto fronts: Given two optimization criteria, the Pareto front contains all non-dominated solutions, where a solution $s$ dominates another solution $s'$ if $s$ is at least as good as $s'$ for one criterion, and it is strictly better for the other criterion.

In this section, we evaluate scale-up properties of our global constraints for computing the Pareto front of all non-dominated solutions for two pairs of conflicting criteria, i.e., $Min_{\text{frequency}}$ and $Min_{\text{size}}$ (denoted (frequency,size)) and $Min_{\text{split}}$ and $Min_{\text{diameter}}$ (denoted (split,diameter)).

For this problem, the number of clusters $k$ is not fixed to a given value, and it is only bounded between 2 and $\#O - 1$. Hence, we do not consider $ECQ_K$ and only report results of $ECQ_{*,\text{sum}}$ with $* \in \{DL, DL+\}$.

There exist two main approaches for solving multi-criteria problems with CP. In Section 6.6.1, we consider the static approach of Wassenhove and Gelders (1980) which involves solving a sequence of mono-criterion problems. In Section 6.6.2, we consider the dynamic approach of Gavanelli (2002) which involves solving a single enumeration problem while dynamically adding constraints to prevent the search from enumerating dominated solutions.
<table>
<thead>
<tr>
<th></th>
<th>(Split, Diameter)</th>
<th>(Frequency, Size)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$ECQ_{DL,\text{sum}}$</td>
<td>$ECQ_{DL+,\text{sum}}$</td>
</tr>
<tr>
<td></td>
<td>#s, time, nodes</td>
<td>#s, time, nodes</td>
</tr>
<tr>
<td>ERP 1</td>
<td>1, 0.2, 57</td>
<td>7, 0.3, 1003</td>
</tr>
<tr>
<td>ERP 2</td>
<td>5, 0.4, 165</td>
<td>9, 6.7, 5,593</td>
</tr>
<tr>
<td>ERP 3</td>
<td>2, 0.3, 70</td>
<td>10, 9.7, 4,582</td>
</tr>
<tr>
<td>ERP 4</td>
<td>2, 0.6, 98</td>
<td>13, 69.4, 64,546</td>
</tr>
<tr>
<td>ERP 5</td>
<td>3, 1.4, 128</td>
<td>13, 601.8, 53,349</td>
</tr>
<tr>
<td>ERP 6</td>
<td>3, 3.2, 143</td>
<td>15, 2320.6, 5,447,207</td>
</tr>
<tr>
<td>ERP 7</td>
<td>2, 81.1, 175</td>
<td>17, - - - 826.5, 2,698</td>
</tr>
<tr>
<td>UCI1</td>
<td>3, 0.2, 102</td>
<td>13, 2.5, 5,821</td>
</tr>
<tr>
<td>UCI2</td>
<td>3, 0.8, 650</td>
<td>- - - -</td>
</tr>
<tr>
<td>UCI3</td>
<td>1, 0.8, 250</td>
<td>11, - - - 368.5, 18,523</td>
</tr>
<tr>
<td>UCI4</td>
<td>5, 3.2, 418</td>
<td>14, 946.1, 560,908,932</td>
</tr>
<tr>
<td>UCI5</td>
<td>2, 3.4, 513</td>
<td>8, 2407.1, 9,962,639,915</td>
</tr>
<tr>
<td>UCI6</td>
<td>4, 428.1, 518</td>
<td>- - - - 637.2, 1,098,756,915</td>
</tr>
</tbody>
</table>

Table 5: Time (in seconds) and number of choice points needed by $ECQ_{DL,\text{sum}}$ and $ECQ_{DL+,\text{sum}}$ for (split, diameter) and (frequency, size) to compute the set of non-dominated solutions using the static method of Wassenhove and Gelders (1980). #s gives the number of non-dominated solutions. '-' is reported when time exceeds 1,000s.

### 6.6.1 Static Approach

Given two objective variables $x_1$ and $x_2$ to maximize, we can compute the Pareto front by solving a sequence of mono-criterion optimization problems (Wassenhove & Gelders, 1980; Duong, 2014). The idea is to alternate between the two objectives as follows:

1. Search for a solution $s_1$ that maximizes $x_1$;
2. Search for a solution $s_2$ that maximizes $x_2$ when $x_1$ is assigned to its value in $s_1$ ($s_2$ is a non-dominated solution);
3. Constrain $x_2$ to be greater than its value in $s_2$, and go to step (1), until no more solution can be found.

In Table 5, we report results of $ECQ_{DL,\text{sum}}$ and $ECQ_{DL+,\text{sum}}$ for solving (split, diameter) (resp. (frequency, size)) when the first maximized variable ($x_1$) is $\text{Min}_{\text{split}}$ (resp. $\text{Min}_{\text{frequency}}$) and the second variable to maximize ($x_2$) is $\text{Min}_{\text{diameter}}$ (resp. $\text{Min}_{\text{size}}$). Note that if there is almost no difference between choosing $\text{Min}_{\text{split}}$ and $\text{Min}_{\text{diameter}}$ as first variable to maximize, considering $\text{Min}_{\text{frequency}}$ as first variable significantly reduces solving times (compared to considering $\text{Min}_{\text{size}}$ as first variable).

The number of non-dominated solutions ranges from 1 to 5 for (split, diameter) whereas it ranges from 7 to 19 for (frequency, size). This may come from the fact that frequency and size measures are very conflicting criteria (formal concepts with large frequencies usually have small sizes, and vice versa), whereas (split, diameter) are less conflicting criteria.

For (split, diameter), $ECQ_{DL,\text{sum}}$ is faster than $ECQ_{DL+,\text{sum}}$ because $DL+$ never reduces significantly the number of choice points.
Figure 11: Example of dominated area for two quality measures $q_1$ and $q_2$. Each point $(x, y)$ corresponds to a subset $u_j \in P$ such that $x = q_1(u_j)$ and $y = q_2(u_j)$. Let us assume that $E = \{u_7, u_8, u_{12}\}$ is an exact cover (displayed in red). The area dominated by $E$ is displayed in blue. The variables $\text{selected}_{u_3}$ and $\text{selected}_{u_4}$ can be assigned to $false$ because any exact cover that contains $u_3$ or $u_4$ is dominated by $E$.

For $(\text{frequency}, \text{size})$, $DL^+$ significantly reduces the number of choice points, compared to $DL$, for all instances, and $ECQ_{DL^+, \text{sum}}$ is able to solve four more instances than $EC_{DL, \text{sum}}$ within the time limit.

6.6.2 Dynamic Approach

Gavanelli (2002) introduces an alternative approach to the static approach described in the previous section. The idea is to solve a single enumeration problem: Each time a new solution $s$ is found, the Pareto front is updated by adding $s$ to it and removing from it all solutions dominated by $s$, and a constraint is dynamically added in order to prevent the search from computing a solution which is dominated by $s$. The search stops when no more solution can be found. A Pareto constraint based on this filtering rule has been introduced by Schaus and Hartert (2013) with an efficient filtering algorithm for bi-objective problems.

In this section, we show how to improve this approach for solving a multi-criteria exact cover problem $(S, P)$ when every objective function involves maximizing a variable $\text{Min}_q$ with $q \in Q'$ (where $Q' \subseteq Q$ is the subset of considered quality measures). Indeed, during the search process, when an exact cover $E \subseteq P$ is found, we can discard any subset $u$ such that $\forall q \in Q', q(u) \leq \min_{v \in E} q(v)$, as illustrated in Fig. 11.

Hence, we propose to extend the dynamic approach of Gavanelli (2002) and Schaus and Hartert (2013). More precisely, each time a solution $s$ is found, we dynamically add two constraints:

- The first constraint is the constraint used by Gavanelli (2002) and Schaus and Hartert (2013) to prevent the search from computing a solution dominated by $s$, i.e.,
  \[
  \bigvee_{q \in Q'} \text{Min}_q > s[\text{Min}_q];
  \]

- The second constraint is a new constraint which prevents the search from selecting a subset dominated by $s$, i.e., $\forall u \in P$, $\bigwedge_{q \in Q'} q_i(u) \leq s[\text{Min}_q] \implies \text{selected}_{u} = false$. 

541
where $s[\text{Min}_q]$ denotes the value assigned to $\text{Min}_q$ in $s$, for each quality measure $q \in Q'$.

The second constraint immediately filters the domains of selected variables associated with subsets which are dominated by $s$, whereas the first constraint does not filter any domain when all upper bounds of $\text{Min}_q$ variables are greater than $s[\text{Min}_q]$.

**Example 8.** In Fig. 11, domains of selected variables are not immediately filtered if we only add the first constraint $\text{Min}_{q_1} > 4 \lor \text{Min}_{q_2} > 4$. Indeed, when both upper bounds of $\text{Min}_{q_1}$ and $\text{Min}_{q_2}$ are greater than 4, this disjunctive constraint is not propagated: It is propagated only when the upper bound of one of these variables becomes lower than or equal to 4. As a comparison, the second constraint $\forall u \in P, (q_1(u) \leq 4 \land q_2(u) \leq 4) \Rightarrow \text{selected}_u = false$ allows us to remove true from the domains of selected$_{u_3}$ and selected$_{u_4}$.

**Experimental Evaluation.** In Table 6, we compare the three strategies for computing the Pareto front of non dominated solutions with $ECQ_{*,\text{sum}}$:

- The static approach of Wassenhove and Gelders (1980) denoted Static;
- The dynamic approach of Gavanelli (2002) denoted Dynamic;
- Our extension of Dynamic introduced in this section and denoted Extended.

We report results obtained with the best propagator according to the experimental comparison reported in Table 5, i.e., $ECQ_{DL,\text{sum}}$ for (split,diameter), and $ECQ_{DL+,\text{sum}}$ for (frequency,size). We also report results obtained with ILP, using the static approach of Wassenhove and Gelders (1980). We do not report results of the full CP approaches (FCP1 and FCP2) because they do not scale. For example, for the (size,frequency) criteria, FCP2 is not able to solve ERP1 in less than one day using the Static strategy, whereas this instance is solved in less than one second with our global constraint.

Dynamic and Extended are very sensitive to ordering heuristics because they are very sensitive to the quality of the enumerated solutions: If every new solution is far from the Pareto front and dominates very few solutions then the search space is not much reduced by the dynamically added constraints and a lot of solutions are enumerated. Hence, for Dynamic and Extended, we adapt the ordering heuristic introduced by Knuth: We still search for an element $a \in S_E$ such that $\#\text{cover}_E(a)$ is minimal, but instead of branching first on subsets that maximize the quality measure, we branch first on subsets that maximize the number of dominated subsets in $P$.

Let us first compare Dynamic and Extended to evaluate the interest of adding the second constraint that filters selected variables. Extended explores fewer choice points and is clearly faster than Dynamic. In particular, it is able to solve four more instances than Dynamic.

For (split,diameter), Static is competitive with Extended for the small instances, but it is outperformed for larger instances such as ERP6, ERP7, or UCI6. This may come from the fact that the number of solutions computed by Extended is often close to the number of non dominated solutions: nbSol is equal to $\#s$ for three instances, and never greater than $4 \ast \#s$. This means that ordering heuristics are able to guide the search towards solutions that often belong to the Pareto front. All these solutions are computed by solving a single enumeration problem within a single search. As a comparison, Static always computes $2\ast\#s$ solutions, and each of these solutions is obtained by solving a new optimization problem.
Table 6: Comparison of the three strategies Static, Dynamic, and Extended with ILP to compute the Pareto front. For (split,diameter) (resp. (frequency,size)) we report results of $ECQ_{DL,sum}$ (resp. $ECQ_{DL+,sum}$) for the three strategies. #s is the number of non-dominated clusterings, time is the CPU time in seconds (or '-' when time exceeds 3,600 seconds), nodes is the number of choice points, and nbSol is the number of solutions found.

<table>
<thead>
<tr>
<th></th>
<th>#s</th>
<th>ECQ Static</th>
<th></th>
<th>ECQ Dynamic</th>
<th></th>
<th>ECQ Extended</th>
<th></th>
<th>ILP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>nodes</td>
<td>nbSol</td>
<td>time</td>
<td>nodes</td>
<td>nbSol</td>
<td></td>
</tr>
<tr>
<td>(split,diameter)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ERP1</td>
<td>1</td>
<td>0.2</td>
<td>57</td>
<td>2</td>
<td>0.1</td>
<td>191</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ERP2</td>
<td>5</td>
<td>0.4</td>
<td>165</td>
<td>10</td>
<td>2.0</td>
<td>631</td>
<td>8</td>
<td>1.4</td>
</tr>
<tr>
<td>ERP3</td>
<td>2</td>
<td>0.3</td>
<td>70</td>
<td>4</td>
<td>2.3</td>
<td>459</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>ERP4</td>
<td>2</td>
<td>0.6</td>
<td>98</td>
<td>4</td>
<td>7.7</td>
<td>724</td>
<td>2</td>
<td>20.1</td>
</tr>
<tr>
<td>ERP5</td>
<td>3</td>
<td>1.4</td>
<td>128</td>
<td>6</td>
<td>157.2</td>
<td>3,488</td>
<td>3</td>
<td>27.4</td>
</tr>
<tr>
<td>ERP6</td>
<td>3</td>
<td>3.2</td>
<td>143</td>
<td>6</td>
<td>172.1</td>
<td>2,813</td>
<td>6</td>
<td>268.1</td>
</tr>
<tr>
<td>ERP7</td>
<td>2</td>
<td>81.1</td>
<td>175</td>
<td>6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ERP8</td>
<td>1</td>
<td>0.8</td>
<td>650</td>
<td>6</td>
<td>1.3</td>
<td>2,523</td>
<td>11</td>
<td>231.8</td>
</tr>
<tr>
<td>UCI1</td>
<td>1</td>
<td>0.2</td>
<td>102</td>
<td>6</td>
<td>0.2</td>
<td>203</td>
<td>7</td>
<td>0.9</td>
</tr>
<tr>
<td>UCI2</td>
<td>3</td>
<td>0.8</td>
<td>650</td>
<td>6</td>
<td>1.3</td>
<td>2,523</td>
<td>11</td>
<td>231.8</td>
</tr>
<tr>
<td>UCI3</td>
<td>3</td>
<td>0.8</td>
<td>250</td>
<td>2</td>
<td>2.1</td>
<td>395</td>
<td>3</td>
<td>645.1</td>
</tr>
<tr>
<td>UCI4</td>
<td>5</td>
<td>3.2</td>
<td>418</td>
<td>10</td>
<td>571.2</td>
<td>12,457</td>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>UCI5</td>
<td>2</td>
<td>3.4</td>
<td>513</td>
<td>4</td>
<td>66.8</td>
<td>2,732</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>UCI6</td>
<td>4</td>
<td>428.1</td>
<td>518</td>
<td>8</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ERP1</td>
<td>13</td>
<td>3.1</td>
<td>546</td>
<td>26</td>
<td>8.4</td>
<td>1,169</td>
<td>47</td>
<td>39.4</td>
</tr>
<tr>
<td>ERP2</td>
<td>9</td>
<td>0.7</td>
<td>154</td>
<td>18</td>
<td>2.5</td>
<td>3,239</td>
<td>26</td>
<td>3.8</td>
</tr>
<tr>
<td>ERP3</td>
<td>10</td>
<td>1.1</td>
<td>213</td>
<td>20</td>
<td>2.1</td>
<td>1,694</td>
<td>31</td>
<td>6.3</td>
</tr>
<tr>
<td>ERP4</td>
<td>13</td>
<td>3.1</td>
<td>546</td>
<td>26</td>
<td>8.4</td>
<td>1,169</td>
<td>47</td>
<td>39.4</td>
</tr>
<tr>
<td>ERP5</td>
<td>13</td>
<td>12.9</td>
<td>362</td>
<td>26</td>
<td>76.1</td>
<td>17,288</td>
<td>48</td>
<td>89.0</td>
</tr>
<tr>
<td>ERP6</td>
<td>15</td>
<td>24.0</td>
<td>563</td>
<td>30</td>
<td>185.2</td>
<td>6,619</td>
<td>68</td>
<td>450.4</td>
</tr>
<tr>
<td>ERP7</td>
<td>17</td>
<td>826.5</td>
<td>2,698</td>
<td>34</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UCI1</td>
<td>13</td>
<td>1.0</td>
<td>499</td>
<td>26</td>
<td>2.5</td>
<td>1,489</td>
<td>67</td>
<td>8.1</td>
</tr>
<tr>
<td>UCI2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UCI3</td>
<td>11</td>
<td>368.5</td>
<td>18,523</td>
<td>22</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UCI4</td>
<td>14</td>
<td>322.4</td>
<td>55,457</td>
<td>28</td>
<td>3,077.0</td>
<td>713,914</td>
<td>68</td>
<td>66</td>
</tr>
<tr>
<td>UCI5</td>
<td>8</td>
<td>637.2</td>
<td>1,098,756</td>
<td>16</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UCI6</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

On (frequency,size), Static is the fastest approach for all instances but ERP1, and it scales much better: It is able to solve all instances but UCI2 and UCI6 in less than one hour, whereas Extended reaches the CPU time limit for ERP7, UCI2, UCI5, and UCI6. This may come from the fact that the number of solutions computed by Extended is often much larger than the number of non dominated solutions. For example, for UCI3, Extended computes 99 solutions, whereas the Pareto front only contains 11 non dominated solutions. For this instance, Static solves 22 optimization problems, and it is three times as fast as Extended.

As a conclusion, Dynamic is outperformed by Extended, and Extended and Static are complementary: Extended is more efficient for (split,diameter), and Static for (frequency,size).
In Table 6, we also report results of ILP. For (split,diameter), $ECQ_{DL}^\oplus_{sum}$ is significantly faster than ILP: It is able to solve all instances whereas ILP fails at solving four instances. For (frequency,size), $ECQ_{DL}^\oplus_{sum}$ is also significantly faster than ILP: It is able to solve all instances but UCI2 and UCI6 whereas ILP fails at solving six instances.

7. Conclusion

We have introduced the $exactCover$ global constraint for modelling exact cover problems, and we have introduced the $DL$ propagator, that uses Dancing Links, and the $DL^+$ propagator, that exploits cover inclusions to strengthen $DL$. We have also extended $exactCover$ to the case where the number of selected subsets is constrained to be equal to a given variable, and we have shown how to integrate a propagator designed for $atMostNValues$ within $DL^+$, thus allowing us to take benefit of the fact that the intersection graph is maintained by $DL^+$.

We have experimentally evaluated our propagators on conceptual clustering problems, and we have compared them with state-of-the-art declarative approaches, showing that our approach is competitive with them for mono-criterion problems, and outperforms them for multi-criteria problems.

As further works, we plan to extend our global constraint to allow the user to soften non-overlapping or coverage constraints which may be relevant in some applications, as pointed out by Ouali et al. (2016). A convenient and flexible extension is to add $\#S$ integer variables to the input parameters: Each of these variables is associated with a different element and is constrained to be equal to the number of selected subsets that cover this element. This way, we allow the user to constrain in many different ways the coverage and the overlapping of the selected subsets. For instance, we may easily model the constraint of allowing at most $x\%$ of elements to overlap or allowing few elements not to be covered.

Also, we plan to study the extension of our work to other optimization criteria for conceptual clustering problems. In the experiments reported in Section 6, we have considered the case where we maximize a variable which is constrained to be equal to the smallest quality among the selected formal concepts. This aggregation function ensures a minimal quality over all clusters. Aribi, Ouali, Lebbah, and Loudni (2018) consider other aggregation functions for evaluating the quality of a clustering, and they show that the Ordered Weighted Average function (that returns a weighted sum of qualities) ensures equity by weighting quality measures according to their rank. This kind of aggregation function hardly scales with CP, and it would be interesting to design a specific propagator for it.

Acknowledgments. We thank Jean-Guillaume Fages and Charles Prud’homme for their help on Choco, Dao et al. (2015) for sending us their Gecode code, and Ouali et al. (2016) for sending us UCI instances. We also thank Ian Davidson for an enriching discussion on clustering problems and their relation with exact cover problems.

References


Duong, K.-C. (2014). Constrained clustering by constraint programming. Theses, Université d’Orléans.


A Global Constraint for the Exact Cover Problem


