Annales Mathematicae et Informaticae
58 (2023) pp. 147–159
DOI: 10.33039/ami.2023.08.007
URL: https://ami.uni-eszterhazy.hu

An incremental algorithm for computing the transversal hypergraph

Laszlo Szathmary

University of Debrecen, Faculty of Informatics
Debrecen, Hungary
szathmary.laszlo@inf.unideb.hu

Abstract. In this paper we present an incremental algorithm for computing the transversal hypergraph. Our algorithm is an optimized version of Berge's algorithm [2] for solving the transversal hypergraph problem. The original algorithm of Berge is the simplest and most direct scheme for generating all minimal transversals of a hypergraph. Here we present an optimized version of Berge's algorithm that we call *BergeOpt*. We show that *BergeOpt* can significantly reduce the number of expensive inclusion tests.

1. Basic concepts

Here we recall the basic notions of hypergraph theory, frequent itemset mining, and we also point out the relation between itemsets and hypergraphs.

1.1. Hypergraphs

In this subsection we mainly rely on [3]. Hypergraph theory [2] is an important field of discrete mathematics with many relevant applications in applied computer science. A hypergraph is a generalization of a graph, where edges can connect arbitrary number of vertices. Formally:

Definition 1.1 (hypergraph)**.** A *hypergraph* is a pair $(V, \mathcal{E})$ of a finite set $V = \{v_1, v_2, \ldots, v_n\}$ and a family $\mathcal{E}$ of subsets of V. The elements of V are called vertices, the elements of $\mathcal{E}$ edges.

Note that some authors, e.g. [2], state that the edge-set as well as each edge must be non-empty and that the union of all edges results in the vertex set.

Submitted: July 30, 2023
Accepted: August 7, 2023
Published online: August 19, 2023

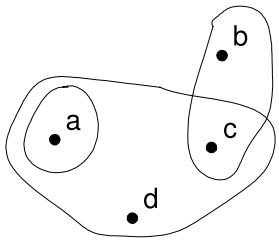

Figure 1. A sample hypergraph $\mathcal{H}$, where $V = \{a, b, c, d\}$ and $\mathcal{E} = \{\{a\}, \{b, c\}, \{a, c, d\}\}$.

Definition 1.2 (partial hypergraph)**.** Let $\mathcal{H} = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_m\}$ be a hypergraph. The partial hypergraph $\mathcal{H}_i$ of $\mathcal{H}$ $(i = 1, \ldots, n)$ is the hypergraph that contains the first i edges of $\mathcal{H}$, i.e. $\mathcal{H}_i = \{\mathcal{E}_1, \ldots, \mathcal{E}_i\}$.

A hypergraph is *simple* if none of its edges is contained in any other of its edges. Formally:

Definition 1.3 (simple hypergraph)**.** A hypergraph is called *simple* if it satisfies $\forall \mathcal{E}_i, \mathcal{E}_j \in \mathcal{E} : \mathcal{E}_i \subseteq \mathcal{E}_j \Rightarrow i = j$.

Example. The hypergraph $\mathcal{H}$ in Figure 1 is not simple because the edge $\{a\}$ is contained in the edge $\{a, c, d\}$.

Definition 1.4. Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph. Then $min(\mathcal{H})$ denotes the set of minimal edges of $\mathcal{H}$ w.r.t. set inclusion, i.e. $min(\mathcal{H}) = \{E \in \mathcal{E} \mid \nexists E' \in \mathcal{E} : E' \subset E\}$, and $max(\mathcal{H})$ denotes the set of maximal edges of $\mathcal{H}$ w.r.t. set inclusion, i.e. $max(\mathcal{H}) = \{E \in \mathcal{E} \mid \nexists E' \in \mathcal{E} : E' \supset E\}$.

Clearly, for any hypergraph $\mathcal{H}$, $min(\mathcal{H})$ and $max(\mathcal{H})$ are simple hypergraphs. Moreover, every partial hypergraph of a simple hypergraph is simple, too.
Example. In the case of hypergraph $\mathcal{H}$ in Figure 1, $\min(\mathcal{H}) = \{\{a\}, \{b, c\}\}$ and $\max(\mathcal{H}) = \{\{b, c\}, \{a, c, d\}\}$.

The problem that is of high interest for us concerns hypergraph transversals. A transversal of a hypergraph $\mathcal{H}$ is a subset of the vertex set of $\mathcal{H}$ which intersects each edge of $\mathcal{H}$. A transversal is *minimal* if it does not contain any transversal as proper subset. Formally:

Definition 1.5 (transversal)**.** Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph. A set $T \subseteq V$ is called a *transversal* of $\mathcal{H}$ if it meets all edges of $\mathcal{H}$, i.e. $\forall E \in \mathcal{E} : T \cap E \neq \emptyset$. A transversal T is called *minimal* if no proper subset T' of T is a transversal.

Note that Pfaltz and Jamison call transversal (resp. minimal transversal) as *blocker* (resp. *minimal blocker*) in [8]. Outside hypergraph theory, a transversal is usually called a *hitting set*.

EXAMPLE. The hypergraph $\mathcal{H}$ in Figure 1 has two minimal transversals: $\{a, b\}$ and $\{a, c\}$. For instance, the sets $\{a, b, c\}$ and $\{a, c, d\}$ are transversals but they are not minimal.

Definition 1.6 (transversal hypergraph)**.** The family of all minimal transversals of $\mathcal{H}$ constitutes a simple hypergraph on V called the *transversal hypergraph* of $\mathcal{H}$, which is denoted by $Tr(\mathcal{H})$.

EXAMPLE. Considering the hypergraph $\mathcal{H}$ in Figure 1, $Tr(\mathcal{H}) = \{\{a, b\}, \{a, c\}\}$.

The following propositions capture important relations between a hypergraph and its transversal hypergraph (for proofs see [2]).

Proposition 1.7. *Let $\mathcal{H} = (V, \mathcal{E})$ be a hypergraph. Then $Tr(\mathcal{H})$ is a simple hypergraph, and $Tr(\mathcal{H}) = Tr(min(\mathcal{H}))$.*

Proposition 1.8. *Let $\mathcal{G}$ and $\mathcal{H}$ be two simple hypergraphs. Then $\mathcal{G} = Tr(\mathcal{H})$ if and only if $\mathcal{H} = Tr(\mathcal{G})$.*

Corollary 1.9. *Let $\mathcal{G}$ and $\mathcal{H}$ be two simple hypergraphs. Then $Tr(\mathcal{G}) = Tr(\mathcal{H})$ iff $\mathcal{G} = \mathcal{H}$.*

Corollary 1.10 (duality property)**.** *Let $\mathcal{H}$ be a simple hypergraph. Then $Tr(Tr(\mathcal{H})) = \mathcal{H}$.*

Corollary 1.10 states that calculating the transversal hypergraph $\mathcal{H}'$ of a simple hypergraph $\mathcal{H}$, and calculating once again the transversal hypergraph $\mathcal{H}''$ of $\mathcal{H}'$, we get back the original hypergraph $\mathcal{H}$, i.e. $\mathcal{H}'' = \mathcal{H}$.

EXAMPLE. Consider the hypergraph $\mathcal{H}$ in Figure 1. Since $\mathcal{H}$ is not simple, let $\mathcal{G} = \min(\mathcal{H}) = \{\{a\}, \{b, c\}\}$. Then,

$$\mathcal{G}' = Tr(\mathcal{G}) = Tr(\{\{a\}, \{b, c\}\}) = \{\{a, b\}, \{a, c\}\}$$
$$\mathcal{G}'' = Tr(\mathcal{G}') = Tr(\{\{a, b\}, \{a, c\}\}) = \{\{a\}, \{b, c\}\}.$$

That is, $\mathcal{G}'' = \mathcal{G}$.

1.2. Frequent itemsets

Consider the following 5×5 sample dataset: $\mathcal{D} = \{(1,\ ACDE),\ (2,\ ABCDE),\ (3,\ AB),\ (4,\ D),\ (5,\ B)\}$. Throughout the paper, we will refer to this example as **"dataset $\mathcal{D}$"**.

Below we use standard definitions of data mining. We consider a set of *objects* or *transactions* $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$, a set of *attributes* or *items* $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$, and a relation $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$, where $\mathcal{R}(o, a)$ means that the object o has the attribute a. In formal concept analysis [4] the triple $(\mathcal{O}, \mathcal{A}, \mathcal{R})$ is called a *formal context*. A set of items is called an *itemset* or a *pattern*. Each transaction has a unique identifier

(*tid*), and a set of transactions is called a *tidset*.[1] The *length* of an itemset is the cardinality of the itemset, i.e. the number of items included in the itemset. An itemset of length i is called an i-long itemset, or simply an i-itemset[2]. An itemset P is said to be *larger* (resp. *smaller*) than Q if $|P| > |Q|$ (resp. $|P| < |Q|$). We say that an itemset $P \subseteq \mathcal{A}$ is *included* in an object $o \in \mathcal{O}$, if $(o, p) \in \mathcal{R}$ for all $p \in P$. Let f be the function that assigns to each itemset $P \subseteq \mathcal{A}$ the set of all objects that include P: $f(P) = \{o \in \mathcal{O} \mid o \text{ includes } P\}$. The set of objects including the itemset is also known as the *image* of the itemset.[3] The (absolute) *support* of an itemset P indicates how many objects include the itemset, i.e. $supp(P) = |f(P)|$. The support of an itemset P can also be defined in relative value, which corresponds to the proportion of objects including P, with respect to the whole population of objects. An itemset P is called *frequent*, if its support is not less than a given *minimum support* (denoted by *min_supp*), i.e. $supp(P) \geq min\_supp$.

Definition 1.11 (generator)**.** An itemset G is called *generator* if it has no proper subset H ($H \subset G$) with the same support.

Definition 1.12 (closed itemset)**.** An itemset X is called *closed* if it has no proper superset Y ($X \subset Y$) with the same support.

The *closure* of an itemset X (denoted by $\gamma(X)$) is the largest superset of X with the same support. Naturally, if $X = \gamma(X)$, then X is closed. The task of frequent (closed) itemset mining consists of generating all (closed) itemsets with supports greater than or equal to a specified *min_supp*.

Equivalence classes. Two itemsets $P, Q \subseteq \mathcal{A}$ are said to be *equivalent* ($P \cong Q$) iff they belong to the same set of objects (i.e. $\gamma(P) = \gamma(Q)$). From this definition it follows that *equivalent itemsets have the same support values*. The set of itemsets that are equivalent to an itemset P (P's *equivalence class*) is denoted by $[P] = \{Q \subseteq \mathcal{A} \mid P \cong Q\}$. Generators are *minimal elements* in their equivalence classes (w.r.t. set inclusion), i.e. a generator $G \in [G]$ has no proper subset in $[G]$. An equivalence class has at least one generator. Closed itemsets are *maximal elements* in their equivalence classes (w.r.t. set inclusion), i.e. a closed itemset $X \in [X]$ has no proper superset in $[X]$. An equivalence class has exactly one closed itemset, which means that closed itemsets are unique elements in their equivalence classes. If an equivalence class has only one element, then the equivalence class is called *singleton*. The only element of a singleton equivalence class is closed as well as generator.

1.3. Relation between itemsets and hypergraphs

Here we show that a family of itemsets can be treated as a hypergraph, and vice versa. As seen in Def. 1.1, a hypergraph $\mathcal{H}$ is a pair $(V, \mathcal{E})$, where V is a finite

[1] For convenience, we will use separator-free set notations throughout the paper, e.g. AB stands for $\{A, B\}$, 13 stands for $\{1, 3\}$, etc.

[2] For instance, ABE is a 3-itemset.

[3] For instance, in dataset $\mathcal{D}$, the image of AB is 23.

set $\{v_1, v_2, \ldots, v_n\}$ and $\mathcal{E}$ is a family of subsets of V. The elements of V are called vertices, the elements of $\mathcal{E}$ edges. In the previous subsection we saw that a formal context is a triple $(\mathcal{O}, \mathcal{A}, \mathcal{R})$, where $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ is a set of objects, $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ is a set of items, and $\mathcal{R} \subseteq \mathcal{O} \times \mathcal{A}$ is a relation between $\mathcal{O}$ and $\mathcal{A}$, where $\mathcal{R}(o, a)$ means that the object o has the item a. A set of items is called an itemset.

The set $\mathcal{A}$ can be considered as a set of vertices V. An itemset corresponds to an edge $E \in \mathcal{E}$. From this it follows that a set of itemsets can be considered as a family of edges $\mathcal{E}$.

EXAMPLE. Consider the hypergraph $\mathcal{H}$ in Figure 1, where $V = \{a, b, c, d\}$ and $\mathcal{E} = \{\{a\}, \{b, c\}, \{a, c, d\}\}$. This hypergraph corresponds to the following set of itemsets: $\{\{a\}, \{b, c\}, \{a, c, d\}\}$. For convenience, we will use separator-free set notations, and we will indicate itemsets with capital letters. That is, the hypergraph $\mathcal{H}$ can be considered as the following set of itemsets: $\{A, BC, ACD\}$. This holds in the other direction too, i.e. the hypergraph representation of the family of itemsets $\{A, BC, ACD\}$ is depicted in Figure 1.

In the rest of the paper, we will treat a family of itemsets as a hypergraph and vice-versa if there is no danger of ambiguity. Thus for a set of itemsets $\{A, BC, ACD\}$, we write "the *hypergraph* $\{A, BC, ACD\}$", etc.

2. The algorithm of Berge

In this section we review the basic algorithm of Berge [2], which is the most simple and direct scheme for generating all minimal transversals of a hypergraph. First, let us see two useful operations on hypergraphs:

Definition 2.1. Let $\mathcal{H} = \{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ and $\mathcal{G} = \{\mathcal{E}'_1, \ldots, \mathcal{E}'_{n'}\}$ be two hypergraphs. Then,

$$\mathcal{H} \cup \mathcal{G} = \{\mathcal{E}_1, \ldots, \mathcal{E}_n, \mathcal{E}'_1, \ldots, \mathcal{E}'_{n'}\}, \text{ and}$$
$$\mathcal{H} \vee \mathcal{G} = \{\mathcal{E}_i \cup \mathcal{E}'_j,\ i = 1, \ldots, n,\ j = 1, \ldots, n'\}.$$

The first operation is the *union* of $\mathcal{H}$ and $\mathcal{G}$, i.e. the hypergraph whose edges are the edges of both hypergraphs. The second operation is very similar to the *Cartesian product*, i.e. the union of all possible pairs of edges, where one element of a pair is from the first hypergraph, and the other element is from the second hypergraph.

Proposition 2.2 ([2])**.** *Let $\mathcal{H}$ and $\mathcal{G}$ be two simple hypergraphs. Then,*

$$Tr(\mathcal{H} \cup \mathcal{G}) = min(Tr(\mathcal{H}) \vee Tr(\mathcal{G})).$$

Let $\mathcal{H}_i = \{\mathcal{E}_1, \ldots, \mathcal{E}_i\},\ i = 1, \ldots, n$ be the partial hypergraph of the hypergraph $\mathcal{H}$. It holds that $\mathcal{H}_i = \mathcal{H}_{i-1} \cup \{\mathcal{E}_i\}$, for all $i = 2, \ldots, n$, where $\mathcal{H}_1 = \{\mathcal{E}_1\}$ and $\mathcal{H}_n = \mathcal{H}$. Thus, $Tr(\mathcal{H}_i) = Tr(\mathcal{H}_{i-1} \cup \{\mathcal{E}_i\})$, and by Prop. 2.2,

Equation 2.3.

$$\begin{aligned} Tr(\mathcal{H}_i) &= min(Tr(\mathcal{H}_{i-1}) \vee Tr(\{\mathcal{E}_i\})) \\ &= min(Tr(\mathcal{H}_{i-1}) \vee \{\{v\},\ v \in \mathcal{E}_i\}). \end{aligned}$$

The algorithm of Berge is based on this equation. The algorithm computes all minimal transversals of a given hypergraph $\mathcal{H}$ in two steps. First, it computes the minimal transversals of the partial hypergraph $\mathcal{H}_{i-1}$ and then it calculates the Cartesian product of the set $Tr(\mathcal{H}_{i-1})$ by the i^{th} edge $\mathcal{E}_i$ of $\mathcal{H}$. Finally, non-minimal elements are removed. Thus, the algorithm starts with the computation of $Tr(\mathcal{H}_1)$, which is a trivial case ($\mathcal{H}_1$ has one edge only, $\mathcal{E}_1$, whose minimal transversals are its vertices). Then, the algorithm adds one by one the rest of the edges, computing at each step the set of minimal transversals of the new partial hypergraph. The algorithm terminates when the last edge $\mathcal{E}_n$ is added. The algorithm of Berge outputs at the end all minimal transversals of the input hypergraph $\mathcal{H}$ [2].

3. An optimized version of Berge's algorithm

In the previous section we reviewed the algorithm of Berge, which implements the most simple and direct approach for calculating the minimal transversals of a hypergraph. Here we present an optimized version of Berge's algorithm that we call *BergeOpt*.

In [7], Le Floc'h *et al.* presented an algorithm called *JEN* whose goal is to efficiently extract generators from a concept lattice [4] for mining exact and approximate association rules [1]. As part of *JEN*, the aforementioned authors presented a simple algorithm without a name for calculating all the minimal transversals of a hypergraph. In the rest of this section we present this algorithm in an extended and completed way. In addition to [7], **(i)** we show that this algorithm is actually an optimization of Berge's original algorithm (hence the name *BergeOpt*), and **(ii)** we provide a proposition (see Prop. 3.1) and its proof.

Optimization idea. One drawback of Berge's algorithm is that after calculating the Cartesian product of the set $Tr(\mathcal{H}_{i-1})$ by the i^{th} edge $\mathcal{E}_i$ of $\mathcal{H}$ (see Equation 2.3), it stores the resulting elements together in the same set, i.e. it has no information whether an element is minimal or not. As a consequence, the filtering of non-minimal elements can be quite expensive when the resulting set has a large number of elements because the algorithm must test the minimality of all elements, including also such elements that are actually minimal.

Our optimization is based on the idea to separate minimal and potentially minimal transversals in two different lists L_1 and L_2, respectively. This way, our optimized algorithm only has to check the minimality of the potentially minimal elements in L_2. As a result, the number of expensive subset checks can be reduced.

The *BergeOpt* algorithm exploits the following proposition:

Proposition 3.1. *In the* BergeOpt *algorithm, the potentially minimal transversals stored in the list L_2 form a simple hypergraph, i.e. L_2 has no two elements e_i and e_j such that $e_i \subseteq e_j$.*

Proof. Assume $X, Y \subseteq V$ are two distinct subsets in $Tr(\mathcal{H}_{i-1}) - Tr(\mathcal{H}_i)$, i.e., they are minimal transversals of $\mathcal{H}_{i-1}$ that lost this status in the i-th partial hypergraph. Assume also that $X \cup \{a\}$ and $Y \cup \{b\}$ are two candidates for $Tr(\mathcal{H}_i)$ produced by the algorithm (i.e., $\{a, b\} \subseteq \mathcal{E}_i$ whereby $a \notin X$ and $b \notin Y$).

Notice that any element of the L_2 list will have the form $X \cup \{a\}$ for some X and a.

Now, without loss of generality we can hypothesize $X \cup \{a\} \subseteq Y \cup \{b\}$, and show this leads to a contradiction. First, notice that $a \neq b$, otherwise we would have $X \subseteq Y$ hence a contradiction with the minimal transversal status. Next, we deduce that $Y = \bar{X} \cup \bar{Y}$ where $\bar{X} = X - \{b\}$ hence $X = \bar{X} \cup \{b\}$. Yet this means that $b \in X \cap \mathcal{E}_i$ which contradicts $X \notin Tr(\mathcal{H}_i)$. □

Pseudo code. The pseudo code of the algorithm is given in Algorithm 1. Let $\mathcal{H}_i = \{\mathcal{E}_1, \ldots, \mathcal{E}_i\}$, $i = 1, \ldots, n$ be the partial hypergraph of the hypergraph $\mathcal{H}$. It holds that $\mathcal{H}_i = \mathcal{H}_{i-1} \cup \{\mathcal{E}_i\}$, for all $i = 2, \ldots, n$, where $\mathcal{H}_1 = \{\mathcal{E}_1\}$ and $\mathcal{H}_n = \mathcal{H}$. Let $\mathcal{MT}_{\mathcal{H}_i}$ denote the set of all minimal transversals of the partial hypergraph $\mathcal{H}_i$.

As input, we have a set of itemsets that we treat as a hypergraph (see Section 1.3). The goal is to compute all the minimal transversals of this hypergraph. The algorithm performs this task in an incremental way. First, the algorithm takes the first itemset $\mathcal{E}_1$ of the input and it calculates its minimal transversals. This is a trivial case; we only have to decompose the itemset into its 1-long subsets. For instance, the itemset ABC has three minimal transversals namely A, B, and C. Then, the algorithm takes the next itemset $\mathcal{E}_i$ of the input and it updates the list of minimal transversals $\mathcal{MT}_{\mathcal{H}_{i-1}}$ if necessary. This is done the following way. Each minimal transversal m found so far, i.e. each element of $\mathcal{MT}_{\mathcal{H}_{i-1}}$, is tested if it has a common part with the current itemset $\mathcal{E}_i$. If it has, then m is a minimal transversal of $\mathcal{H}_i$ too, thus m is added to the list L_1. In the list L_1 we collect those itemsets that are minimal transversals of the partial hypergraph processed so far, including the current itemset $\mathcal{E}_i$ too. Prop. 1.7 guarantees that L_1 has no two elements e_1 and e_2 such that $e_1 \subseteq e_2$. If the test was negative, i.e. m has no common part with the current itemset $\mathcal{E}_i$, then it means that m is not a transversal of $\mathcal{E}_i$, thus m must be extended to have an intersection with $\mathcal{E}_i$ (in other words, m is a transversal of $\mathcal{H}_{i-1}$, but not a transversal of $\mathcal{H}_i$). This can be done by decomposing $\mathcal{E}_i$, and generating the one-size larger supersets of m using the 1-long subsets of $\mathcal{E}_i$ (Cartesian product of m with the vertices of $\mathcal{E}_i$). For instance, if $\mathcal{E}_i = BCH$, and the minimal transversal to be updated is AD, then the following potentially minimal transversals are generated: ABD, ACD, and ADH. We call these itemsets "potentially minimal transversals", because with this extension it is guaranteed that they became transversals of $\mathcal{H}_i$, but it is not sure that they are *minimal*, thus they are put in another list L_2. It can be possible that they have

Algorithm 1 ("getMinTransversals" function):

Description: BergeOpt algorithm
Input: a hypergraph ($\mathcal{H}$)
Output: all minimal transversals of $\mathcal{H}$ ($\mathcal{MT}$)

```
 1)  MT ← ∅;   // initialisation; no minimal transversals are found yet
 2)  loop over the elements of H (E_i)   // an element of H is an edge (an itemset)
 3)  {
 4)      if (E_i is the first element of H) {
 5)          MT ← {vertices of E_i};   // decomposition (1-itemsets of E_i)
 6)      }
 7)      else
 8)      {
 9)          L_1 ← ∅; L_2 ← ∅;   // two empty lists
10)          loop over the elements of MT (m)
11)          {
12)              if (m ∩ E_i ≠ ∅) {   // m has a common vertex with E_i
13)                  L_1 ← L_1 ∪ m;   // m is a minimal transversal of E_i
14)              }
15)              else {
16)                  S ← {one-size larger supersets of m using
                             the vertices of E_i};
17)                  L_2 ← L_2 ∪ S;
18)              }
19)          }
20)          if (L_1 ≠ ∅ and L_2 ≠ ∅) {
21)              cleanSupersets(L_1, L_2);   // removing non-minimal ...
22)          }                               // ... transversals from L_2
23)          MT ← L_1 ∪ L_2;
24)      }
25)  }
26)
27)  return MT;
```

subsets among the minimal transversals in L_1. When all elements of $\mathcal{MT}_{\mathcal{H}_{i-1}}$ are tested against the current itemset $\mathcal{E}_i$, the lists L_1 and L_2 are filled. At this point, there are three possibilities (lines 20–23 of Algorithm 1): **(1)** L_1 is non-empty and L_2 is empty, or **(2)** L_1 is empty and L_2 is non-empty, or **(3)** both L_1 and L_2 are non-empty. In the *first case*, L_1 contains all the minimal transversals of $\mathcal{H}_i$. By Prop. 1.7, L_1 is a simple hypergraph. In the *second case*, L_2 contains all the minimal transversals of $\mathcal{H}_i$. Since L_1 is empty, all elements in L_2 are minimal. Moreover, from Prop. 3.1 it follows that L_2 is a simple hypergraph. In the *third case*, the list L_2 must be cleaned first, i.e. if an element e_1 in L_1 is a subset of an element e_2 in L_2, then e_2 must be removed because e_2 is *not minimal*. Prop. 3.1

guarantees that the elements of L_2 are not comparable w.r.t. set inclusion. Then, taking the union of the lists L_1 and L_2, we have all the minimal transversals of $\mathcal{H}_i$.

The algorithm continues by taking the next itemset of the input set (next current itemset) and it updates again the list of minimal transversals. The algorithm terminates when all elements of the input set are processed. At this point, the algorithm collected all the minimal transversals of the input set, i.e. it calculated the transversal hypergraph of the input hypergraph.

`cleanSupersets` procedure: this method removes non-minimal transversals from the list L_2, i.e. itemsets that have subsets in L_1. The procedure works as follows. It enumerates all elements of L_2. If the current element e_2 in L_2 has a subset in L_1, then e_2 is removed from L_2. When the procedure terminates, L_2 only contains *minimal* transversals.

Running example. Consider the following hypergraph $\mathcal{H} = \{ACD, ACH, BCD, DF, FH\}$. Let $\mathcal{E}_i$ denote the i^{th} element (edge) of the hypergraph, i.e. $\mathcal{E}_1 = ACD, \mathcal{E}_2 = ACH, \ldots, \mathcal{E}_5 = FH$. Let $\mathcal{H}_i$ denote the partial hypergraph that contains the first i elements of $\mathcal{H}$, i.e. $\mathcal{H}_1 = \{ACD\}$, $\mathcal{H}_2 = \{ACD, ACH\}$, $\ldots$, $\mathcal{H}_5 = \{ACD, ACH, BCD, DF, FH\} = \mathcal{H}$. The notation $\mathcal{MT}_{\mathcal{H}_i}$ denotes the set of all minimal transversals of the partial hypergraph $\mathcal{H}_i$.

Table 1. Incremental computation of the transversal hypergraph of $\mathcal{H} = \{ACD, ACH, BCD, DF, FH\}$ with the *BergeOpt* algorithm.

$\mathcal{E}_1 = ACD$	$\mathcal{MT}_{\mathcal{H}_1} = \{A, C, D\}$
$\mathcal{E}_2 = ACH$	$L_1 = \{A, C\}$
	$L_2 = \{\sout{AD}, \sout{CD}, DH\}$
	$\mathcal{MT}_{\mathcal{H}_2} = \{A, C, DH\}$
$\mathcal{E}_3 = BCD$	$L_1 = \{C, DH\}$
	$L_2 = \{AB, \sout{AC}, AD\}$
	$\mathcal{MT}_{\mathcal{H}_3} = \{C, DH, AB, AD\}$
$\mathcal{E}_4 = DF$	$L_1 = \{DH, AD\}$
	$L_2 = \{CD, CF, \sout{ABD}, ABF\}$
	$\mathcal{MT}_{\mathcal{H}_4} = \{DH, AD, CD, CF, ABF\}$
$\mathcal{E}_5 = FH$	$L_1 = \{DH, CF, ABF\}$
	$L_2 = \{ADF, \sout{ADH}, \sout{CDF}, \sout{CDH}\}$
	$\mathcal{MT}_{\mathcal{H}_5} = \{DH, CF, ABF, ADF\} = \mathcal{MT}_{\mathcal{H}} = Tr(\mathcal{H})$

The execution of the algorithm is depicted in Table 1. First, the algorithm takes $\mathcal{E}_1$ (ACD) and computes its minimal transversals that are A, C, and D. The algorithm continues with processing $\mathcal{E}_2$ (ACH). Each time when a new element of $\mathcal{H}$ is handled, the already found minimal transversals are tested. The itemsets A and C have common parts with $\mathcal{E}_2$, thus they are minimal transversals of ACH, so they are added to the list L_1. However, D has no common part with $\mathcal{E}_2$, which means that D is a minimal transversal of $\mathcal{H}_1$, but not a transversal of $\mathcal{H}_2$. In order to make D a transversal of $\mathcal{H}_2$, D is extended with the 1-long subsets of

ACH, thus the following candidates are generated: AD, CD, and DH. These three itemsets are put in the list L_2. Then, the algorithm removes itemsets from L_2 that have subsets in L_1 since they are not minimal transversals (AD and CD). The union of L_1 and L_2, which is stored in the list $\mathcal{MT}_{\mathcal{H}_2}$, gives all the minimal transversals of $\mathcal{H}_2$. The same steps are repeated with the other elements of $\mathcal{H}$ ($\mathcal{E}_3$, $\mathcal{E}_4$, and $\mathcal{E}_5$). When the algorithm terminates, all minimal transversals of the hypergraph $\mathcal{H}$ are discovered. In this example, the transversal hypergraph of $\mathcal{H}$ is $Tr(\mathcal{H}) = \{DH, CF, ABF, ADF\}$.

4. Experimental results

The *BergeOpt* algorithm was implemented in Java in the Coron data mining platform [9].[4] The experiments were carried out on an Intel Core i7 3.5 GHz machine with 16 GB RAM running under Manjaro GNU/Linux. All times reported are real, wall clock times.

Our algorithm *BergeOpt* was used as part of another algorithm called *Snow* that we presented in [10]. In [10] we just mentioned *BergeOpt* without giving any details. A detailed presentation of *Snow* is out of the scope of this paper, but we give a short summary. Frequent closures (FCIs) and frequent generators (FGs) as well as the precedence relation on FCIs are key components in the definition of a variety of association rule bases (see [6] for a survey). The goal of the *Snow* algorithm is to extract the precedence relation from a more common mining output, i.e. closures and generators. Thus, the idea is the following. First, we extract FCIs and their associated generators, i.e. we get the frequent equivalence classes (see Section 1.2). In each equivalence class, we consider the set of FGs to be a simple hypergraph. Using *BergeOpt*, we calculate the transversal hypergraph of the generators. With this result, the order among the FCIs can be obtained very efficiently. For a detailed description of the *Snow* algorithm, please refer to [10]. To conclude, in our experiments *BergeOpt* was used to calculate the transversal hypergraph of the generators in each equivalence class in a dataset.

For the experiments, we used several real and synthetic dataset benchmarks. Database characteristics are shown in Table 2 (top). The chess and connect datasets are derived from their respective game steps. The Mushrooms database describes mushrooms characteristics. These three datasets can be found in the UC Irvine Machine Learning Database Repository.[5] The pumsb, C20D10K, and C73D10K datasets contain census data from the PUMS sample file. The synthetic datasets T20I6D100K and T25I10D10K, using the IBM Almaden generator, are constructed according to the properties of market basket data. Typically, real datasets are very dense, while synthetic data are usually sparse.

Table 2 (bottom left and right) provides a summary of the experimental results. The first column specifies the various minimum support values for each of the datasets (low for the sparse dataset, higher for dense ones). The second and third

[4] http://coron.loria.fr
[5] https://archive.ics.uci.edu

Table 2. Top: database characteristics. **Bottom:** response times of *BergeOpt*.

database name	# records	# non-empty attributes	# attributes (in average)	largest attribute
T20I6D100K	100,000	893	20	1,000
T25I10D10K	10,000	929	25	1,000
chess	3,196	75	37	75
connect	67,557	129	43	129
pumsb	49,046	2,113	74	7,116
MUSHROOMS	8,416	119	23	128
C20D10K	10,000	192	20	385
C73D10K	10,000	1,592	73	2,177

min_supp	# concepts (including top)	*BergeOpt* (seconds)
T20I6D100K		
0.75%	4,711	0.03
0.50%	26,209	0.21
0.25%	149,218	1.10
T25I10D10K		
0.40%	83,063	0.56
0.30%	122,582	0.86
0.20%	184,301	1.33
chess		
65%	49,241	0.34
60%	98,393	0.68
55%	192,864	1.28
connect		
65%	49,707	0.29
60%	68,350	0.46
55%	94,917	0.56

min_supp	# concepts (including top)	*BergeOpt* (seconds)
pumsb		
80%	33,296	0.57
78%	53,418	0.99
76%	82,539	2.04
MUSHROOMS		
20%	1,169	0.01
10%	4,850	0.04
5%	12,789	0.15
C20D10K		
0.50%	132,952	1.12
0.40%	151,394	1.18
0.30%	177,195	1.45
C73D10K		
65%	47,491	0.53
60%	108,428	1.26
55%	222,253	2.70

columns comprise the number of FCIs and the execution time of *BergeOpt* (given in seconds). The CPU time does not include the cost of computing FCIs and FGs since they are assumed as given.

As can be seen, *BergeOpt* is able to calculate the transversal hypergraph of the generators in the equivalence classes very efficiently in both sparse and dense datasets. To find out why the algorithm *BergeOpt* performs so well, we investigated the size of its input data. Figure 2 shows the distribution of hypergraph sizes in the datasets T20I6D100K, MUSHROOMS, chess, and C20D10K.[6] Note that we obtained similar hypergraph-size distributions in the other four datasets too. Figure 2 indicates that most hypergraphs only have 1 edge, which is a trivial case, whereas large hypergraphs are relatively rare. As a consequence, *BergeOpt* can perform very efficiently.

We interpret the above results as an indication that the good performance of *BergeOpt* is independent of the density of the dataset. In other terms, provided

[6]For instance, the dataset T20I6D100K by *min_supp* = 0.25% contains 149,019 1-edged hypergraphs, 171 2-edged hypergraphs, 25 3-edged hypergraphs, 0 4-edged hypergraphs, 1 5-edged hypergraph, and 1 6-edged hypergraph.

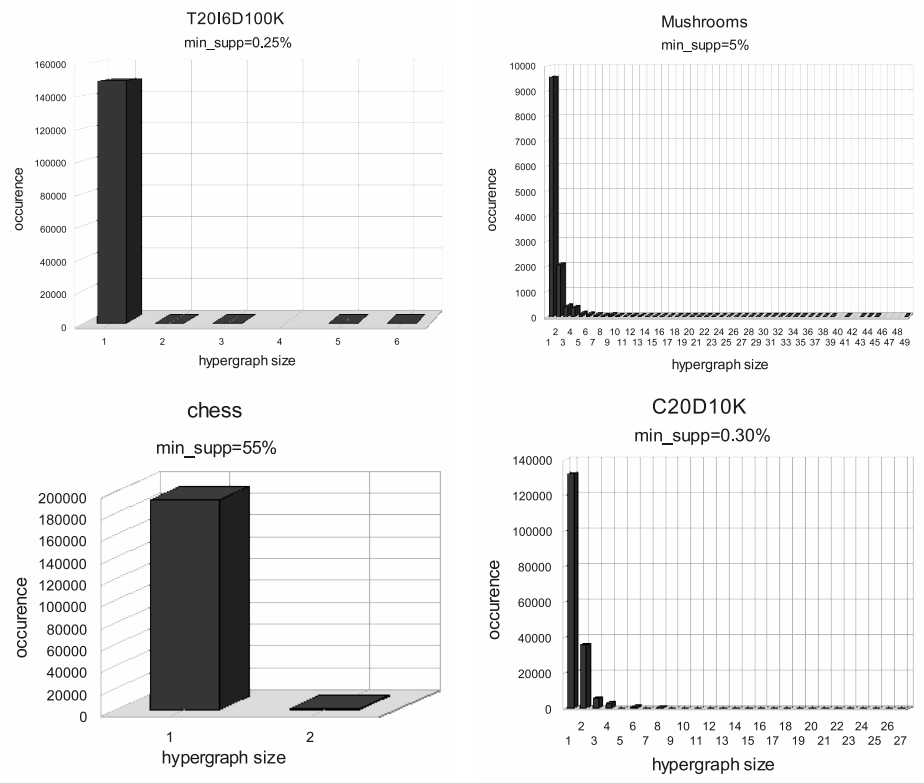

Figure 2. Distribution of hypergraph sizes.

that the input hypergraphs do not contain too many edges, i.e. there are only few FGs per FCIs, the computation is very fast. A natural question arises with this observation: does the modest number of FGs in each class hold for all realistic datasets in the literature? If not, could one profile those datasets which meet this condition?

5. Conclusion

In this paper we presented an optimization of Berge's original algorithm [2] called *BergeOpt* that can significantly reduce the number of expensive inclusion tests. Since Berge's algorithm several other, more efficient algorithms have been introduced. As pointed out in [5] for instance, the simple method of Berge needs exponential many steps to produce the whole output. It generates the first minimal transversal near the end of the procedure and its high memory requirements make it suitable only for small problem cases. However, as we pointed out in the previous section, our hypergraphs are usually very small, thus we did not have to face these

efficiency problems. Experimental results show that *BergeOpt* provides a very efficient solution for the problem instance that we had to deal with, i.e. when we used *BergeOpt* as part of the *Snow* algorithm [10] to discover the precedence relation among FCIs.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo: *Fast discovery of association rules*, in: Advances in knowledge discovery and data mining, American Association for Artificial Intelligence, 1996, pp. 307–328, isbn: 0-262-56097-6.

[2] C. Berge: *Hypergraphs: Combinatorics of Finite Sets*, Amsterdam: North Holland, 1989.

[3] T. Eiter, G. Gottlob: *Identifying the Minimal Transversals of a Hypergraph and Related Problems*, SIAM Journal on Computing 24.6 (1995), pp. 1278–1304, issn: 0097-5397, doi: 10.1137/S0097539793250299.

[4] B. Ganter, R. Wille: *Formal concept analysis: mathematical foundations*, Berlin/Heidelberg: Springer, 1999, p. 284, isbn: 3540627715.

[5] D. J. Kavvadias, E. C. Stavropoulos: *An Efficient Algorithm for the Transversal Hypergraph Generation*, Journal of Graph Algorithms and Applications 9.2 (2005), pp. 239–264.

[6] M. Kryszkiewicz: *Concise Representations of Association Rules*, in: Proc. of the ESF Exploratory Workshop on Pattern Detection and Discovery, 2002, pp. 92–109.

[7] A. Le Floc'h, C. Fisette, R. Missaoui, P. Valtchev, R. Godin: *JEN : un algorithme efficace de construction de générateurs pour l'identification des règles d'association*, Spec. num. of Revue des Nouvelles Technologies de l'Information 1.1 (2003), pp. 135–146.

[8] J. L. Pfaltz, R. E. Jamison: *Closure Systems and their Structure*, Information Sciences 139.3–4 (2001), pp. 275–286.

[9] L. Szathmary: *Symbolic Data Mining Methods with the Coron Platform*, PhD Thesis in Computer Science, Univ. Henri Poincaré – Nancy 1, France, Nov. 2006.

[10] L. Szathmary, P. Valtchev, A. Napoli, R. Godin, A. Boc, V. Makarenkov: *A fast compound algorithm for mining generators, closed itemsets, and computing links between equivalence classes*, Annals of Mathematics and Artificial Intelligence (AMAI) 70.1–2 (2014), pp. 81–105, issn: 1012-2443, doi: 10.1007/s10472-013-9372-8.