The minimal hitting set generation problem: algorithms and computation

Andrew Gainer-Dewar* and Paola Vera-Licona†

Center for Quantitative Medicine, UConn Health, Farmington, Connecticut, USA

January 13, 2016

Abstract

Finding inclusion-minimal hitting sets for a given collection of sets is a fundamental combinatorial problem with applications in domains as diverse as Boolean algebra, computational biology, and data mining. Much of the algorithmic literature focuses on the problem of recognizing the collection of minimal hitting sets; however, in many of the applications, it is more important to generate these hitting sets. We survey twenty algorithms from across a variety of domains, considering their history, classification, useful features, and computational performance on a variety of synthetic and real-world inputs. We also provide a suite of implementations of these algorithms with a ready-to-use, platform-agnostic interface based on Docker containers and the AlgoRun framework, so that interested computational scientists can easily perform similar tests with inputs from their own research areas on their own computers or through a convenient Web interface.

1 Introduction

Fix a family $S$ of sets $S_1, S_2, \ldots$. A hitting set $T$ of $S$ is a set which intersects each of the sets $S_i$; $T$ is a minimal hitting set (hereafter “MHS”) if no proper subset of $T$ is a hitting set of $S$.

The problem of generating the collection of MHSes for a given set family is of interest in a wide variety of domains, and it has been explicitly studied (under a variety of names) in the contexts of combinatorics (Section 2.1 [1]), Boolean algebra (Section 2.2 [2, 3, 4]), fault diagnosis (Section 2.3 [5, 6, 7, 8, 9, 10, 11]), data mining (Section 2.5 [12, 13, 14, 15]), and computational biology (Section 2.4 [16, 17, 18, 19, 20, 21]), among others. While some interesting results have been obtained for the associated decision problem, the computational complexity of this problem is currently unknown. Nevertheless, there is an extensive literature of algorithms to generate minimal hitting sets.

In this paper, we survey the state of the art of algorithms for enumerating MHSes, with particular attention dedicated to publicly-available software implementations and their performance on problems derived from various applied domains. We begin in Section 2 with discussion of the cognate problems that emerge in several applied domains, including explanations of how each can be translated back to our MHS generation problem. In Section 3, we survey what is known about the complexity of the problem, both in general and in specialized tractable cases.

The bulk of the paper is dedicated in Section 4 to surveying the history of some twenty algorithms for enumerating MHSes. For each algorithm, we discuss its structure, relevant properties, any known bounds on its complexity, and information about available software implementations.

In Section 5 we present the results of extensive benchmarks run on the available implementations of these algorithms, focusing on running time for large examples derived from specific applications. Additionally, we provide these implementations in a unified, ready-to-use framework of Docker images based on the AlgoRun framework, which are available to download or to use through a Web interface. Interested readers and
computational scientists may use these containers to try out the algorithms on their own input data and incorporate them into processing pipelines.

We conclude in Section 6 with an overview of our results and the state of the art in MHS generation algorithms.

2 Cognate problems

There are many important problems in numerous domains that can be reduced or translated to MHS problems. We survey a few of them here, organized by domain.

2.1 Combinatorics

**Transversal hypergraph problem** Given a set \( V \) of vertices and a set \( E \) of sets \( E_1, E_2, \ldots \subseteq V \) of hyperedges, the pair \( H = (V,E) \) is a hypergraph. \( H \) is finite if \( V \), \( E \), and all the \( E_i \) are finite sets. A hypergraph is simple if none of its edges is a subset of any other edge. The hypergraph \( H = (V,E) \) can naturally be identified with the set family \( E \), so the theory of hypergraphs is similar to that of set families. Readers interested in the full theory of hypergraphs should consult Berge’s 1984 monograph [22] on the subject.

In the finite hypergraph setting, a (minimal) hitting set of \( E \) is called a (minimal) transversal of \( H \). The collection of all transversals of \( H \) is its transversal hypergraph \( \text{Tr}(H) \). \( \text{Tr}(H) \) is also sometimes called the dual of \( H \) because, in the case that \( H \) is simple, \( \text{Tr}(\text{Tr}(H)) = H \). (More generally, \( \text{Tr}(\text{Tr}(H)) = \min H \), the hypergraph obtained from \( H \) by removing all non-inclusion-minimal edges.)

There is an extensive literature on the problem of determining whether two hypergraphs \( H_1 \) and \( H_2 \) are transversal of each other. We will consider a number of algorithms developed for this purpose in Section 4. Interested readers can consult the recent survey of Eiter [1] and Ph.D. thesis of Hagen [2] for more details about this subject.

**Set cover problem** Fix a hypergraph \( H = (V,E) \). A set cover of \( H \) is a set \( E' \subseteq E \) of edges of \( H \) with the property that every vertex in \( V \) is in some edge in \( E' \). Of course, \( E \) is a set cover, but there may be others, and it is of particular interest to compute the inclusion-minimal ones.

We can construct a new hypergraph \( G \) whose vertex set is \( E \) and whose edge set is \( V \) (that is, with a vertex in \( G \) for every edge of \( H \) and an edge in \( G \) for each collection of edges in \( H \) with a common vertex). Then the MHSes of \( G \) are exactly the minimal set covers of \( H \). Thus, an algorithm for generating MHSes can be used to enumerate minimal set covers, and vice versa.

**Independent set problem** Fix a hypergraph \( H = (V,E) \). An independent set of \( H \) is a set \( V' \) of vertices of \( H \) with the property that no edge of \( H \) is a subset of \( V' \). Of course, the empty vertex set \( \emptyset \) is a set cover, but there may be others, and it is of particular interest to compute the inclusion-maximal ones. It is easy to show that an independent set is the complement of a hitting set and thus that a maximal independent set is the complement of a minimal hitting set. Thus, we can enumerate maximum independent sets of \( H \) simply by applying some algorithm for MHS generation and then taking complements.

2.2 Boolean algebra

**Definition 2.1.** A Boolean function is a function \( f = f(x_1, x_2, \ldots, x_k) : \mathbb{B}^k \to \mathbb{B} \), where \( \mathbb{B} = \{0,1\} \) and \( k \) is a non-negative integer.

A \( k \)-ary Boolean function is described by an algebraic expression, called a Boolean expression or Boolean formula, which consists of the binary variables \( x_1, \ldots, x_k \), the binary conjunction operator \( \land \) (often written AND), the binary disjunction operator \( \lor \) (often written OR), and the unary negation operator \( \neg \) (often written NOT).

If a Boolean function \( f \) has the property that \( f(X) \leq f(Y) \) (resp. \( f(X) \geq f(Y) \)) for any inputs \( X \leq Y \), we say that \( f \) is positive (resp. negative). A function which is either positive or negative is monotone. Monotone Boolean functions appear in a wide variety of formal and applied settings. (See [23] for an extensive survey.)
We can easily ensure that this property holds by restricting the formulas considered:

**Definition 2.2.** A formula for a Boolean function is monotone if it contains only variables, conjunctions, and disjunctions (i.e. no negations).

**Theorem 2.3.** Any monotone formula gives a monotone Boolean function.

However, representation in these propositional formulas is not unique; the same function may be given by many formulas, even after commutativity is considered. Accordingly, restrictions are often placed on the formulas to ensure uniqueness.

**Definition 2.4.** A formula for a Boolean function is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals and disjunctive normal form (DNF) if it is a disjunction of conjunctions of literals.

**Example 2.5.** The formula \((x_1 \lor x_2) \land x_3\) is in CNF, while \((x_2 \land x_3) \lor (x_1 \land x_3)\) is in DNF.

Both normal forms are of great interest for computational applications because they are easy to decompose and compare. In addition, a given monotone Boolean function has exactly one each of CNF and DNF formulas satisfying an irredundancy condition.

If two given formulas \(C\) in CNF and \(D\) in DNF represent the same function, we say they are dual. Deciding whether this is true of a given pair \((C,D)\) is widely studied in the literature of complexity theory; see the excellent survey in the Ph.D. thesis of Hagen [2], which gives this problem the picturesque name MONGEN. We may also consider the generation problem MONGEN, which asks for the DNF formula \(D\) equivalent to a given CNF formula \(C\).

Surprisingly, MONGEN is computationally equivalent to the MHS generation problem. To illustrate why, consider the function \((x_2 \lor x_3) \land (x_1 \lor x_3)\), which is evidently in CNF. Its clauses (i.e. the disjunctive components) are formed from the variable sets \(\{x_2, x_3\}\) and \(\{x_1, x_3\}\). Since these clauses are conjoined, to satisfy the formula we must satisfy each clause, which means we must set to 1 at least one variable in each clause; for example, we might take \(x_2\) and \(x_1\) or take \(x_3\) alone.

Indeed, this is exactly equivalent to finding hitting sets of the set family \(\{\{2, 3\}, \{1, 3\}\}\). Moreover, the irredundancy requirement for a DNF is equivalent to requiring the hitting sets to be minimal. The collection \(\{\{2, 1\}, \{3\}\}\) of MHSes corresponds to the DNF \((x_2 \land x_1) \lor x_3\) of the function.

### 2.3 Model-based fault diagnosis

Modern engineered systems may involve incredible numbers of interconnected components; for example, the recently-retired NASA Space Shuttle reportedly had over 2.5 million moving parts. When such a system fails to perform as intended, it is infeasible to check or replace every part and connection; thus, diagnostic procedures are needed to narrow attention to some subset of components which may have caused the observed failure. In a celebrated 1987 paper [5], Reiter developed the foundation for a formal theory of model-based diagnosis (MBD), which we will introduce briefly.

Consider a system made up of some finite set \(V\) of components, each of which may be either active or inactive during any given transaction or activity. We make a series of observations of transactions of the system, recording which components are active and whether the behavior is normal or anomalous. If any of the observed transactions are anomalous, we assume this is due to one or more faulty components. A diagnosis of the faulty system is a set \(D\) of components which, if all are faulty, would explain all the anomalous transactions. Under a parsimony restriction, the interesting diagnoses are those which are inclusion-minimal or irredundant. (Reiter only applies the term “diagnosis” to these minimal examples.)

Let \(F\) be the set of faulty transactions, where each such transaction is represented as the set of components involved. A diagnosis is then a hitting set of \(F\), and the parsimonious diagnoses are exactly the MHSes of \(F\). Thus, any algorithm for generating MHSes is directly applicable to fault diagnosis. Reiter proposed an algorithm for exactly this purpose; we will discuss it and its successors in Section 4.

### 2.4 Computational biology

Minimal hitting sets have appeared as an important combinatorial motif in numerous problems in computational biology. We survey four of them here.
Minimal cut sets in metabolic networks  A metabolic reaction network is the system by which metabolic and physical processes that determine the physiological and biochemical properties of a cell, are represented. As such, these networks comprise the chemical reactions of metabolism, the metabolic pathways and the regulatory interactions that guide these reactions. In its graph representation, a metabolic network is a directed hypergraph in which each of $m$ vertices represents a metabolite and each of $n$ directed hyperedges represents a biochemical reaction. Information about the reactions can be encoded in a stoichiometric matrix $M$ in which each row represents a metabolite, each column represents a biochemical reaction, and the entry $S_{i,j}$ represents the coefficient of metabolite $i$ in reaction $j$. If the coefficient $S_{i,j}$ is positive, metabolite $i$ is produced by reaction $j$; if the coefficient is negative, the metabolite is consumed; and if the coefficient is zero, the metabolite is not involved in the reaction at all and thus is not in its hyperedge. An $m$-dimensional vector may be employed to represent concentrations of metabolites, while an $n$-dimensional vector may represent rates of reactions.

It is typically assumed that some internal metabolites are at steady state and thus must have a net zero rate of change in the system. Such a steady state corresponds to an $n$-dimensional flux vector $\vec{x}$ with the property that $S\vec{x} = 0$; we call a vector satisfying this condition an admissible flux mode. We call such a flux mode elementary if its support (the set of reactions with nonzero fluxes) is inclusion-minimal.

The notion of elementary flux modes (“EFMs”) was introduced by Schuster and Hilgetag in [24]; subsequent work has developed numerous techniques from linear algebra and computational geometry to find the EFMs. see the recent surveys [25, 26] for overview of the problem, the methods and software which are used to solve it, and various applications.

One important area of applications of metabolic network analysis is metabolic engineering, in which the metabolic network of an organism is modified to adjust its production. In [27], Klamt and Gilles focus on blocking a target reaction through cut sets, which they define as a set of reactions whose removal from the network leaves no feasible balanced flux distribution involving the target reaction. To do this, they first compute all the elementary flux modes which involve the target reaction. They then construct a set family whose elements are the reactions of the original network and whose sets are the relevant elementary flux modes. Finally, they compute the minimal hitting sets of this family. Subsequently, Haus et al. developed in [28] a specialized version of the FK-A algorithm (cf. Section 4.2.2) which greatly improves on other methods available at the time.

Optimal combinations of interventions in signal transduction networks  Signal transduction describes the process of conversion of external signals to a specific internal cellular response, such as gene expression, cell division, or even cell suicide. This process begins at the cell membrane where an external stimulus initiates a cascade of enzymatic reactions inside the cell that typically includes phosphorylation of proteins as mediators of downstream processes. A signal transduction network is represented as a signed directed graph in which each of the vertices is a signaling component (such as a protein, gene in a cell) and each edge represents an interaction which the source induces in the target (either positive-signed activation or negative-signed inhibition). Biological signaling networks typically exhibit a natural decomposition into input, intermediate, and output nodes; engineering and control of these networks then typically depends on adjusting the input and intermediate layers to obtain some outcome at the outputs. As an analogous of elementary flux modes (“EFMs”) in metabolic networks, in [30], Wang and Albert introduced the notion of elementary signaling modes (ESMs). ESMs are minimal sets of signaling components which can perform signal transduction from inputs to outputs; ESMs are the natural analogues of EFMs for signal transduction networks. They also provide algorithms for generating the ESMs of a given network.

Once the topology and ESMs of a signal transduction network are known, it may be of interest to control how signal flows from a given set of source nodes to a given set of targets, perhaps avoiding interfering with certain side effect nodes along the way. Vera-Licona et al. introduced the OCSANA framework to study this problem in [16]. They begin by computing the ESMs which pass from the specified sources to the specified targets. They then construct a set family whose elements are the signalling components and whose sets are these ESMs. They next compute MHSes of this family, which they term combinations of interventions (CIs). Finally, they apply their “OCSANA” scoring, which encodes heuristics about control of the targets and influence on the side effects, and use the results to identify the most promising CIs.

Since discovery of hitting sets is a crucial step in the algorithm, the authors of [16] performed an experimental comparison. In particular, they tested Berge’s algorithm (cf. Section 4.1.1) and an approach
similar to MTMiner (cf. Section 4.3.1) which incorporates the OCSANA score into the generation process. They found their new algorithm to improve substantially on Berge’s algorithm.

**Reverse-engineering biological networks from high-throughput data** In some situations, the network structure itself is not known. In this case, it is useful to reconstruct the network from experimental data through reverse engineering. Broadly, the **biological reverse-engineering problem** is that of “analyzing a given system in order to identify, from biological data, the components of the system and their relationships” [31]. This may result in either a topological representation of the network structure (typically expressed as an enriched graph) or a full dynamic model (in terms of a mathematical model). Numerous approaches to the topological reverse-engineering problem have been introduced, including at least two which use MHS generation techniques. See [31] for a comparative survey of these two approaches and the relative performance of the specialized algorithms developed for each; we give here only a brief overview of each.

**Ideker** In 2000, Ideker et al. introduce a method to infer the topology of a network of gene regulatory interactions in [21]. They first perform a series of experiments in which various genes in the system are forced to high or low expression and use standard microarray techniques to measure the expression of the other genes. For each gene $i$, they then consider all pairs $\{a, b\}$ of experiments for which that gene’s expression values differ, then find the set $S_{a,b}$ of all other genes whose expression values also changed.

Under their assumptions, one or more genes in $S_{a,b}$ must cause the change in $i$ between experiments $a$ and $b$. A set of genes which intersects *all* the sets $S_{a,b}$ is a candidate to explain the observed variation in $i$ over the whole suite of experiments, and thus to be the set of genes connected to $i$ in the regulatory network. Thus, they generate a collection of hitting sets for the family $S_{a,b}$; in particular, they develop an algorithm based on the standard branch and bound optimization technique which gives sets of minimal cardinality. Since this collection may be large, they use an entropy-based approach to iteratively generate new experiments which will discriminate among the candidates and re-apply the algorithm to the enlarged data set to improve the accuracy of the predicted networks.

**Jarrah** In 2007, Jarrah et al. introduce another method to infer the topology of a gene regulatory network in [19] which focuses on time series data within a single experiment. They associate to each gene $i$ a variable $x_i$ from some finite field $k$, then represent each time point $t$ in the experiment as an assignment $x^t$. For a given gene $i$, they then consider how the values of $x^t$ determine the value $x^{t+1}_i$ for each $t$.

Under their assumptions, if $x_i$ is observed to take different values at times $t_1$ and $t_2$, it must be due to some other variables being different at times $t_1 - 1$ and $t_2 - 1$. Thus, for each pair $t_1 \neq t_2$ with $x^{t_1}_i \neq x^{t_2}_i$, they construct a set $S_{t_1,t_2} = \{j | j \neq i, x_j^{t_1-1} \neq x_j^{t_2-1}\}$ which encodes the variables which may be responsible for the observed change in $x_i$. Like Ideker et al., they then compute minimal hitting sets of this collection, but their algorithm is formalized in terms of computational algebra and gives a complete enumeration of the family of MHSes. (We discuss this approach in Section 4.5.1.) They also present a heuristic scoring function which may help to select the most viable model from the generated hitting sets.

**Drug cocktail development** Many widely-used antibiotics are effective against some bacterial strains but ineffective against others. Thus, in cases where more than one strain may be present or the specific strain is unknown, it is necessary to deploy a “cocktail” of several drugs to increase the number of strains covered. A similar situation applies in cancer chemotherapy, where different chemotherapeutic agents are known to be effective only against certain cell lines. In either case, it is desirable to minimize the number of drugs used at once, to minimize the cost of the therapy and the risk of emergent multiple-drug-resistant strains.

Given a set of drugs (say, antibiotics) and a set of targets (say, bacterial strains), we can assign to each target the set of drugs that are effective against it. A (minimal) hitting set of this set family is then a (minimal) cocktail of drugs that, taken together, affects all the targets.

This application has been studied in detail by Vazquez in [17], using a greedy algorithm to search for very small effective combinations from the NCI60 collection (32) of 45334 drugs and 60 cancer cell lines. He is then able to recommend some of these small MHSes as targets for further research.
2.5 Data mining

A number of problems in data mining can be formalized in terms of MHS generation. We survey two of them here.

(In)frequent itemset discovery One fundamental problem in data mining, introduced by Agrawal et al. in [33] and developed further in [34], is the discovery of frequent itemsets in a database of transactions. We adopt the formal setting of the problem presented by Boros et al. in [35]. Fix a finite set $A$ of $m$ transactions, each of which is a finite subset of a set $I$ of $n$ items. Fix an integer threshold $1 \leq t \leq m$. A set $C$ of items is $t$-frequent if at least $t$ transactions are supersets of $C$ and $t$-infrequent if no more than $t$ transactions are supersets of $C$. The maximal frequent (resp. minimal infrequent) itemset problem is to enumerate inclusion-maximal (resp. inclusion-minimal) sets $C$ which are $t$-frequent (resp. $t$-infrequent).

Let $F_t$ denote the hypergraph whose edges are maximal $t$-frequent itemsets in $A$ and $I_t$ denote the hypergraph whose edges are minimal infrequent itemsets in $A$. (Both have the common vertex set $I$.) Any element of $I_t$ must intersect the complement of every element of $F_t$, and in fact as hypergraphs we have that $I_t = \text{Tr}(F_t^C)$ exactly. Thus, if either $I_t$ or $F_t$ is known, the other can be computed using any algorithm for MHS generation. This connection is explored by Manilla and Toivonen in [36]; more algorithmic details are given by Toivonen in [37]. An application of these ideas to database privacy is given by Stavropoulos et al. in [38].

Furthermore, so-called “joint-generation” algorithms inspired by the FK algorithms of Fredman and Khachiyan [4] (cf. Section 4.2.2) can generate $I_t$ and $F_t$ simultaneously. This is developed by Gunopulos et al. in [39] and its complexity implications explored by Boros et al. in [40].

Emerging pattern discovery Another important data mining problem is the discovery of emerging patterns, which represent the differences between two subsets of the transactions in a database. We adopt the formal setting of the problem introduced by Bailey et al. in [13]. Consider two sets $A$ and $B$ of transactions, where each transaction is itself a set of items. A minimal contrast is an inclusion-minimal set of items which appears in some transaction in $A$ but no transaction in $B$. Fix a transaction $a \in A$ and construct a set family whose underlying elements are the items in $a$ and with a set $a \setminus b$ for each $b \in B$. Then the minimal contrasts supported by $a$ are exactly the minimal hitting sets of this set family. Thus, any algorithm for MHS generation can be applied to emerging pattern discovery. Indeed, two of the algorithms we study, DL (Section 4.1.4, [12]) and BMR (Section 4.1.5, [13]), were developed for this purpose.

2.6 Minimal Sudoku puzzles

The Sudoku family of puzzles is widely published in newspapers and magazines and is played by millions worldwide. An instance of Sudoku is a $9 \times 9$ grid of boxes, a few of which already contain digits (“clues”) from the range 1–9; a solution is an assignment of digits to the remaining boxes so that each of the nine $3 \times 3$ subgrids and each row and each column of the whole puzzle contains each digit exactly once. An example with seventeen clues is shown in Fig. 1.

Of course, not every possible placement of clues into the grid yields a valid puzzle. There may be inherent contradictions, such as two identical clues in the same column, so that the puzzle has no solutions. There may also be ambiguities, in which more than one solution is possible. A mathematical question of particular interest, then, is: what is the smallest number of clues in an unambiguous valid Sudoku puzzle? Many such puzzles with 17 clues are known, but none with 16 have ever been identified. In [11], McGuire et al. show that an exhaustive search for such puzzles can be formulated in our terms by constructing set families that represent the effects of clues in solved puzzles and then searching for hitting sets of size 16 or less. They ran this search on a supercomputing cluster and proved conclusively that there are no 16-clue Sudoku puzzles. They use an algorithm similar to HST from [8], discussed in detail in Section 1.1.3 for speed, they modify the algorithm to essentially build the set families and their hitting sets simultaneously.
3 Complexity results

3.1 Asymptotic complexity

Before considering specific algorithms for MHS generation, we should consider the current state of knowledge about the asymptotic complexity of the problem. It has been known since Karp’s seminal 1972 paper [42] that the problem of determining whether a given set family has a hitting set of size no greater than some $k$ is NP-complete. However, we are more concerned with the collection of all hitting sets than with the existence of a single one. We therefore consider two other separate but related problems.

3.1.1 Recognition problem

Much of the literature on complexity analysis focuses on “decision problems”, which must have a “yes” or “no” answer. The natural decision variant of the MHS problem is recognition: given two hypergraphs $H$ and $G$, to decide whether $H = \text{Tr} G$. Fredman and Khachiyan present in [4] an algorithm (discussed in Section 4.2.2) which tests this in time $n^{O(\log n)}$ (for $n$ the sum of the number of hyperedges in $F$ and $G$). This time bound is notable in that it is quasi-polynomial—it is worse than a polynomial bound, but better than an exponential bound or even certain sub-exponential bounds. The BM algorithm introduced by Boros and Makino in [43] improves on this bound in parallel cases. It is a long-standing open problem to determine whether recognition is possible in polynomial time.

3.1.2 Generation problem

For many applications, however, we need to generate the MHSes of a given set family rather than recognize them. It is straightforward to show that no algorithm can do this in time polynomial in the size of the input. Consider the example of the matching graph $M_n = \{(1, 2), (3, 4), \ldots, (2n - 1, 2n)\}$ as a set family. A minimal hitting set contains a choice of one of the two elements of each edge; thus, there are evidently $2^n$ of them. Simply writing out this result therefore requires at least $o(2^n)$ time, so no algorithm can in subexponential time in general. This is not necessarily to say that the MHS generation problem is intractable; rather, it suggests that it is inappropriate to analyze its complexity input-polynomiality solely in terms of input size.

Johnson et al. introduced a formalism to deal with this issue in [44]. Instead of considering only the size of the input to the problem, we can incorporate the size of the output as well. If a given set family $S$ has $n$ sets and $m$ MHSes, an algorithm for generating those MHSes is said to be output-polynomial (or to run in polynomial total time) if its running time is $o(\text{poly}(n + m))$. Unfortunately, this is not known to be achieved by any current algorithm, and Hagen showed in [45] that several important algorithms are not output-polynomial.

If we shift our attention to incremental generation, we may instead ask whether an algorithm can generate one MHS at a time with reasonable delay between outputs. Johnson et al. introduced two suitable formalisms in [44]. First, a generation algorithm may run in incremental-polynomial time; in this case, given a set family...
and a set of MHSes for it, it should yield a new MHS in time polynomial in the combined size of both of these inputs. Second, the algorithm may run with polynomial delay; this stronger variant of incremental-polynomial time requires that the time required depend only on the size of the set family and not on the number of MHSes already known. Crucially, if an algorithm runs with polynomial delay, it is guaranteed to run in output-polynomial total time, but incremental-polynomial time gives no such guarantee.

Incremental time analysis is of particular interest in some classes of applications, where we may wish to generate only some MHSes or to perform further processing on each one as it emerges. No algorithm is known to solve the MHS generation problem in incremental polynomial time (much less with polynomial delay) in general, but there are some interesting special cases, considered in Section 3.2.1.

### 3.2 Tractable cases

Crucially, the complexity results in Section 3.1 concern the performance of algorithms in general, which is to say, for the class of all possible hypergraphs or set families. Another thread of research has focused on demonstrating that, in such restricted cases, much better complexity results are possible.

#### 3.2.1 Fixed-parameter tractability

In some cases, algorithms are available which “factor out” some of the complexity of the problem with respect to a particular parameter of the hypergraphs. Specifically, letting \( k \) be the parameter of interest and \( n \) be the number of edges of the hypergraph, we may find an algorithm that generates all MHSes in time \( f(k) \cdot n^{O(1)} \) for some arbitrary function \( f \) (which is to say, the time is polynomial in \( n \) once \( k \) is fixed, though it may depend arbitrarily on \( k \)). In this case, we say that the problem is fixed-parameter tractable (“FPT”) with respect to that parameter \( k \), since fixing \( k \) yields a complexity function that depends polynomially on \( n \).

Fixed-parameter tractability results have been obtained for the transversal hypergraph recognition problem with a wide variety of parameters, including vertex degree parameters ([46, 48, 50]), hyperedge size or number parameters ([49, 48, 50]), and hyperedge intersection or union size parameters ([51, 48]). For a more complete survey, the interested reader may consult [2, §4, §7].

#### 3.2.2 Acyclicity

A graph is acyclic if it contains no cycles—that is, if no non-self-repeating path in the edges leads back to where it starts. Beeri et al. introduced in [52] a notion of acyclicity in hypergraphs, now known as \( \alpha \)-acyclicity, in the context of the study of relational database schemes. Fagin subsequently introduced in [53] the notions of \( \beta \)-acyclicity and \( \gamma \)-acyclicity, which are successively more restrictive and correspond to desirable tractability problems in databases. Eiter showed that the transversal recognition is solvable in polynomial time for \( \beta \)-acyclic hypergraphs in [54] and for \( \alpha \)-acyclic hypergraphs in [49].

### 3.3 Limited nondeterminism

Another important line of inquiry for studies of complexity is how its solution improves with nondeterminism—that is, if the algorithm is allowed access to some “free” information. The crucial question is how many nondeterministic bits are required to achieve a better solution. Kavvadias and Stavropoulos showed in [55] that the recognition problem is in the class co-NP\( [\log^2 n] \) for \( n \) the total number of edges in \( H \) and \( TrH \), meaning that only \( O(\log^2 n) \) nondeterministic bits are required to demonstrate that two hypergraphs are not transversals of each other. Since \( \log^2 n \) is subpolynomial, this suggests that the recognition problem is not as hard as the well-known NP-complete problems.

### 4 Existing algorithms

A wide array of algorithms have been developed to generate MHSes (either explicitly or in the language of various cognate problems). If we strip away the details of the various domains and applications by casting all the algorithms in the language of MHS generation, we find that they fall naturally into a few high-level categories:
set iteration approaches which work through the input set family one set at a time, building MHSes as they go;

divide and conquer approaches which partition the input family into disjoint subfamilies, find their MHSes separately, and then combine them;

MHS buildup approaches which build candidate MHSes one element at a time, keeping track of un-hit sets as they go; and

full cover approaches which improve on the divide-and-conquer approach with a technical hypergraph lemma that allows more efficient recombination.

We survey these categories, including discussions of a few published algorithms for each. A summary of these algorithms, giving their taxonomic classifications, original problem domains, and relevant characteristics is presented in Table 1. Whenever possible, we use the terminology of set families and minimal hitting sets, since this language is typically the most straightforward to understand.

It can be shown (cf. [56]) that an algorithm for the transversal hypergraph recognition problem (cf. Section 2.1) can be used to generate MHSes of a given set family with a number of runs that is polynomial in the size of the transversal hypergraph. This conversion is possible because, if the input hypergraphs $H$ and $G$ are not transversals of each other, any recognition algorithm must return a “witness” of this, which can be translated into an edge which must be added to either $H$ or $G$. Thus, beginning with some given set family $S$ and an empty collection $T$ of MHSes, we can interpret $S$ as a hypergraph and apply any recognition algorithm to find a new MHS to add to $T$, then repeat until eventually the complete collection is generated and the algorithm confirms $S = \text{Tr} T$. As a result, we will consider algorithms for both recognition and generation interchangeably.

For nearly all of these algorithms, software implementations are available to perform the calculations on a computer. We have collected eighteen of these implementations into a public repository at github.com/VeraLiconaResearchGroup/MHSGenerationAlgorithms. Source code and information about the implementations are available there. In addition, we provide a ready-to-use Docker container for each using the AlgoRun framework (cf. [57]) and a Web interface to instances of the software running on the AlgoRun project’s servers at algorun.org. This framework is used for experimental benchmarks which are presented in Section 5.

4.1 Set iteration approaches

One type of approach to computing hitting sets of a set family $S$ is to begin with a small subfamily of $S'$ of $S$, find the MHSes for $S'$, and then iteratively add more sets to $S'$ and update the MHS collection. The methods in this section all follow this approach; they differ in how they select the subfamilies $S'$ and in the details of how they update the MHS collection.

4.1.1 Berge (1984)

The first systematic algorithm for computing transversals of hypergraphs was presented by Berge in [22], a monograph on the theory of hypergraphs. Although this algorithm is called the “Sequential Algorithm” in some literature, we will refer to it as Berge. The core idea of the algorithm is proceed inductively over the hyperedges of the hypergraph, alternately adding a new edge to the intermediate hypergraph under consideration and extending the known transversals.

We first introduce three important operations on hypergraphs.

Definition 4.1. Let $H_1 = (V_1, E_1)$ and $H_2 = (V_2, E_2)$ be two hypergraphs. Their vee $H_1 \lor H_2$ is the hypergraph with vertex set $V = V_1 \cup V_2$ and edge set $E = E_1 \cup E_2$. Their wedge $H_1 \land H_2$ is the hypergraph with vertex set $V = V_1 \cup V_2$ and edge set $E = \{e_1 \cup e_2 | e_1 \in E_1, e_2 \in E_2\}$.

Definition 4.2. Let $H$ be a hypergraph with vertex set $V$ and edge set $E$. Its minimization (or simplification), $\text{min} H$, is the hypergraph with vertices $V$ and edges $\{e \in E | \forall f \in E \setminus e : e \subset f\}$. In other words, $\text{min} H$ retains exactly the inclusion-minimal edges of $H$. $H$ is simple if $H = \text{min} H$.
These two operations interact nicely with the transversal construction:

**Lemma 4.3.** Let $H_1$ and $H_2$ be hypergraphs. Then the following relations hold:

\[
\text{Tr}(H_1 \lor H_2) = \min(\text{Tr} H_1 \land \text{Tr} H_2)
\]

(1)

and

\[
\text{Tr}(H_1 \land H_2) = \min(\text{Tr} H_1 \lor \text{Tr} H_2).
\]

(2)

Algorithm Berge then proceeds as follows:

1. Let $H$ be a hypergraph with edge set $E = \{e_1, e_2, \ldots, e_n\}$ (for an arbitrarily chosen ordering), and for each $i$ let $H_i$ be the subhypergraph of $H$ with all its vertices and its first $i$ edges $e_1, \ldots, e_i$.

2. For each $i$ in order, compute $\text{Tr} H_i$ inductively: $\text{Tr} H_1 = \{v|v \in e_1\}$, and $\text{Tr} H_i = \min(\text{Tr} H_{i-1} \land \text{Tr} e_i)$ by equation (1).

3. When the iteration is finished, $\text{Tr} H_n = \text{Tr} H$ by construction.

Pseudocode for the algorithm (in the language of set families) is given in Algorithm 1.

**Algorithm 1 Berge’s algorithm**

**Input:** A finite set family $S = \{s_1, s_2, \ldots, s_n\}$

**Output:** The set of MHSes of $S$

1: function Berge($S$)
2: $T \leftarrow \{\{e\}|e \in s_1\}$
3: for all $s \in S \setminus s_1$ do
4:   $T \leftarrow \{t \cup \{e\}|t \in T, e \in s\}$
5:   $T \leftarrow \min T$ \hrulefill \> Remove non-minimal elements of $T$
6: end for
7: return $T$
8: end function

As suggested in [18], Berge’s algorithm can be adapted to search only for MHSes of cardinality bounded by some $k$ by simply discarding candidates larger than $k$ at Lines 4 and 5 in each stage of the algorithm.

This algorithm is straightforward to implement in code and to study theoretically. Unfortunately, it also has the potential to be extremely inefficient. The complexity is studied thoroughly by Boros et al. in [58]. In particular, for a set family $S$ with $n$ sets and a total of $m$ underlying elements, if the sets are ordered so that the collection $T$ reaches maximum size $k$ during the algorithm, the running time of this algorithm is $O(kmn \cdot \min(m, k))$.

Accordingly, it is clear that the ordering of the edges matters a great deal, since this determines the value of $k$. Takata showed in [59] that there exists a family of hypergraphs for which no edge ordering yields output-polynomial running time, and thus that Berge is not output-polynomial in general, even if the edge ordering is optimal. Boros et al. demonstrate in [58], however, that the worst case is still sub-exponential.

We provide a C++ implementation of Berge which supports enumeration of small MHSes in the repository.

**4.1.2 Reiter (1987) and Greiner et al. (1989)**

As discussed in Section 2.3, Reiter introduced the formal theory of model-based diagnosis as an application of MHS enumeration in [5]. His approach proceeds through a set family inductively, alternately picking a set which is not yet hit and an element which hits it, until a hitting set for the whole family is found. It then backtracks to the most recent step where another valid choice was available and repeats. The intermediate data are stored in a structure Reiter calls a “hitting set tree”.

However, it was shown by Greiner et al. in [60] that this algorithm is incomplete; the hitting sets it generates are guaranteed to be minimal, but in certain circumstances some MHSes may be missed. They
repair the algorithm, but in the process they sacrifice the acyclicity of Reiter’s hitting set tree; the result is a directed acyclic graph. We will refer to the algorithm as HS-DAG (for Hitting Set Directed Acyclic Graph). It is straightforward to implement and is widely studied and cited in the MBD literature.

The authors are not aware of a formal complexity analysis of this algorithm.

It is possible to search only for hitting sets of bounded cardinality with HS-DAG simply by restricting the depth of the search DAG.

A Python implementation of this algorithm by the authors of [7] is available in the repository.

4.1.3 Wotawa (2001)

Wotawa returned to Reiter’s approach in [8], reviewing the HS-DAG algorithm of [60] (see Section 4.1.2) and adjusting it to reduce the number of set containment checks required. These improvements render the DAG generalization unnecessary, so the underlying data structure is once again a hitting set tree as originally envisioned by Reiter. We will refer to the algorithm as HST (for Hitting Set Tree).

The authors are not aware of a formal complexity analysis of this algorithm.

It is possible to search only for hitting sets of bounded cardinality with HST simply by restricting the depth of the search tree.

A Python implementation of this algorithm by the authors of [7] is available in the repository.

4.1.4 Dong and Li (2005)

Dong and Li considered in [12] the “emerging patterns problem” discussed in Section 2.5. Although their work was essentially independent of the literature on hypergraph transversals, they developed an algorithm thematically very similar to Berge. Their algorithm incorporates some optimizations to the minimization calculation in equation (2) to speed up the loop step. We will refer to the algorithm as DL (for its authors).

However, the running time of Berge is dominated by the need to search the intermediate transversals, not the complexity of generating them, so the DL optimization should not be expected to improve the worst-case behavior of Berge. Hagen shows in [2, 45] that Takata’s time bounds on Berge in [59] apply to DL as well, so it is not output-polynomial. Nevertheless, for families with relatively few sets, DL performs well, so it is useful as a subroutine to be used in base cases of other algorithms.

A C implementation of this algorithm by the authors of [61] is available in the repository.

4.1.5 Bailey et al. (2003)

Continuing with the study of emerging patterns, Bailey et al. developed in [13] an algorithm which decomposes the input set family more carefully than Berge’s algorithm. Rather than simply considering one new set at a time, their approach attempts to partition the set family into components with few sets, then use the DL algorithm of [12] as a subroutine to compute their MHSes before combining them using equation (1). We will refer to their algorithm as BMR (for Bailey, Manoukian, and Ramamohanarao).

Hagen shows in [2, 45] that BMR is not output-polynomial. Furthermore, he shows that its complexity is $n^{\Omega(\log \log n)}$, where the $\Omega$ indicates that this is a lower bound instead of the upper bound indicated by $O$ and where $n = |H| + |\text{Tr } H|$.

A C implementation of this algorithm by the authors of [61] is available in the repository.

4.1.6 Kavvadias and Stavropoulos (2005)

Returning to the explicit study of hypergraph transversals, Kavvadias and Stavropoulos introduced in [62] another algorithm, which seeks to reduce the memory requirements of Berge with two optimizations. First, they preprocess the input set family to combine elements which occur only in the same sets. Second, they carefully reorganize the processing steps so that many intermediate MHSes can be forgotten without jeopardizing the correctness of the algorithm, allowing them to output MHSes early in the algorithm’s run and then discard them. We will refer to this algorithm as KS (for its authors).

The algorithm is designed to run in polynomial memory by avoiding regeneration of candidate hitting sets. Hagen shows in [2, 45] that KS does not run in output-polynomial time. Furthermore, he shows that its
complexity is \( n^{\Omega(\log \log n)} \), where \( n \) is the sum of the number of sets in the family and the total number of minimal hitting sets that it admits.

The organization of the search routine in \( KS \) makes it possible to search for hitting sets of bounded cardinality to save time. The authors are not aware of an implementation that offers this feature.

A Pascal implementation of this algorithm by the authors of [62] is available in the repository.

4.2 Divide and conquer approaches

Another type of approach to computing MHSes of a set family \( S \) is to partition \( S \) into several subfamilies, find their MHSes separately (perhaps recursing until the subfamilies are sufficiently small), and then combine the results. The algorithms in this section all follow this approach; they differ primarily in how they partition \( S \).

4.2.1 Lin and Jiang (2003)

Lin and Jiang return in [9] to the problem of model-based diagnosis. They cast the problem in the Boolean algebra framework, but their algorithm is a straightforward example of the divide-and-conquer approach. We will refer to this algorithm as \( BOOL \) (since Lin and Jiang call it the “Boolean algorithm”). Their recursive decomposition algorithm proceeds as follows.

1. Let \( S \) be a finite set family. If \(|S| < 2\), it is trivial to find the MHSes of \( S \) directly. Thus, we assume that \(|S| \geq 2\).
2. If there is an element \( e \) which is present in every set \( s \in S \), construct a new set family \( S' = \{s \setminus e | s \in S\} \). Recursively find the MHSes of \( S' \), add \( \{e\} \), and return the result.
3. If there is a set \( s \in S \) with \(|s| = 1\), let \( e \) be the unique element of \( s \) and construct a new set family \( S' = S \setminus s \). Recursively find the MHSes of \( S' \), add \( e \) to each, and return the result.
4. Otherwise, choose some \( e \in \bigcup S \) arbitrarily. Let \( S_1 = \{s \setminus e | s \in S, e \in s\} \) and \( S_2 = \{s | s \in S, e \notin s\} \). Recursively find the MHSes of \( S_1 \) and \( S_2 \). Add \( e \) to each MHS of \( S_2 \), then take the union of the results and return.

Pseudocode for the algorithm is given in Algorithm 2. They call this the “Boolean algorithm”; we will denote it hereafter by \( BOOL \).

The Boolean algorithm was subsequently optimized by Pill and Quaritsch in [11]. In particular, improved its performance in cases that only MHSes of size bounded by some \( k \) are desired.

The authors are not aware of a formal complexity analysis of this algorithm.

If desired, this algorithm can search for hitting sets of bounded cardinality to save time.

A Python implementation of this algorithm by the authors of [7] is available in the repository.

4.2.2 Fredman and Khachiyan (1996)

Fredman and Khachiyan introduced two iterative algorithms in [4] to study the recognition version of the MHS problem in the Boolean algebra context. Like \( BOOL \), these two algorithms both proceed by choosing one considering sets that do and do not contain that element separately with recursive calls, and then combining the results. However, they first apply several algebraically-motivated degeneracy tests. If the tests fail, a new hitting set can be found very efficiently. If, however, they succeed, it guarantees that an element can be found which is present in many (specifically, logarithmically many) sets but missing from many others.

Considering the sets which do and do not contain this element separately decomposes the problem into two large disjoint sub-problems, which can be considered recursively; the large size of each subproblem ensures that the recursion does not go too deep. This bound on the recursion depth allows Fredman and Khachian to prove running-time bounds which are the strongest known on any sequential algorithm to date. We will refer to these two algorithms as \( FK-A \) and \( FK-B \) (“FK” for the authors, who use the names \( A \) and \( B \) in [4]).

The first algorithm, \( FK-A \), runs in time \( n^{O(\log^2 n)} \) and is relatively straightforward to implement. The algorithm is modified in \( FK-B \) to improve its runtime slightly and adapt it to MHS generation. An implementation in (compiled) C is provided by those authors and is available in the repository.
The Boolean algorithm (BOOL)

Input: A finite set family \( S = \{s_1, s_2, \ldots, s_n\} \)

Output: The set of MHSes of \( S \)

1: function BOOL(S)
2: \( E \leftarrow \bigcup_{s \in S} s \)
3: if \( |S| = 0 \) then
4: \( T \leftarrow \emptyset \)
5: else if \( |S| = 1 \) then
6: let \( S = \{s\} \)
7: \( T \leftarrow s \)
8: else if there is some \( e \in E \) such that \( e \in s \forall s \in S \) then
9: \( T \leftarrow \{e\} \lor BOOL(\{s \setminus e | s \in S\}) \)
10: else if there is some \( s \in S \) such that \( |s| = 1 \) then
11: \( G \leftarrow s \land BOOL(S \setminus s) \)
12: else
13: choose \( e \in E \) \( \triangleright \) can be arbitrary
14: \( S_1 \leftarrow \{s \setminus e | s \in S, e \in s\} \)
15: \( T_1 \leftarrow BOOL(S_1) \)
16: \( S_2 \leftarrow \{s | s \in S, e \notin s\} \)
17: \( T_2 \leftarrow \{e\} \land BOOL(S_2) \)
18: \( T \leftarrow T_1 \cup T_2 \)
19: end if
20: return \( T \)
21: end function

The second algorithm, FK-B, runs in time \( n^{O(\log n)} \). (More exactly, its time bound is \( n^{\chi(n) + O(1)} \) where \( \chi(n)^{\chi(n)} = n \).) However, its implementation is significantly more complex than that of FK-A. As a result, most authors (including [63]) have disregarded FK-B in comparative studies. However, analysis in [64] suggests that this assumption may be inaccurate. The authors are aware of no publicly-available implementations of FK-B.

If desired, both algorithms can search for hitting sets of bounded cardinality to save time. The authors are not aware of an implementation that supports this feature.

The algorithms also allow for “joint generation” of a set family and its MHSes if subsets of both are known. This can be advantageous in situations where the set family is not known a priori but it is possible to check whether a given set is a member of the family. For example, Haus et al. apply this approach in [28], as discussed in Section 2.4.

4.2.3 Abreu and Gemund (2009)

Model-based diagnosis often involves finding hitting sets of extremely large set families, so approximation algorithms are particularly attractive in this field. Abreu and Gemund presented such an algorithm in [10]. They use a divide-and-conquer approach similar to that of BOOL, but which considers the elements in an order determined by a statistical heuristic. They also define mechanisms to stop the algorithm early to obtain a partial set of approximately minimal hitting sets. We will refer to this algorithm as STACCATO (the name used by its authors in [10]).

The authors of [10] claim that, for a set family with \( N \) sets and \( M \) total elements, the algorithm guarantees to find a hitting set of cardinality \( C \) in \( O \left( (M \cdot (N + \log M))^C \right) \) worst-case time and \( O(C \cdot M) \) space, with improved expected times based on their heuristic and tested experimentally.

If desired, this algorithm can search for hitting sets of bounded cardinality to save time.

A Python implementation of this algorithm by the authors of [7] is available in the repository.
4.2.4 Leiserson et al. (2010)

Some recent research has focused on parallelizing the search for minimal hitting sets. Leiserson et al. cast this issue in a very abstract setting in [65], developing a framework to parallelize any algorithm that searches for minimal elements of a poset and then applying it to the lattice of hitting sets of a set family. We will refer to this algorithm as ParTran (the name used by its authors in [65]).

Treated as a sequential algorithm by running it in a single execution thread, ParTran is similar to BOOL; its primary distinction is that the two subfamilies $S_1$ and $S_2$ are carefully chosen to be of similar size to improve parallel efficiency. The authors are not aware of a formal analysis of its complexity in either sequential or parallel settings.

A Cilk++ implementation of this algorithm by the authors of [65] is available in the repository.

4.2.5 Knuth (2011)

Binary decision diagrams (BDDs) are a graph-based structure for representing boolean functions and hypergraphs originally introduced by Bryant in [66]. Given a set family $S$, it is computationally expensive to compress $S$ into a BDD or to decompress that BDD back into $S$. Nevertheless, BDDs are a powerful data structure for certain combinatorial algorithms. Many logical operations on hypergraphs, such as the $\land$ and $\lor$ operations of Definition 4.1, are inexpensive to perform on their BDDs. In exercises 236 and 237 of [67, §7.1.4], Knuth asks the reader to devise an algorithm for MHS generation using BDD operations, and in the solutions he presents a simple one. We will refer to this algorithm as Knuth.

The authors are not aware of a formal complexity analysis of Knuth, and Knuth asserts that the worst-case runtime is unknown.

A C implementation of this algorithm by the author of [68] is available in the repository.

4.2.6 Toda (2013)

In 2013, Toda improved on the Knuth algorithm in [68] by incorporating a variation on the BDD data structure—the zero-suppressed binary decision diagram (ZDD). After compressing a given set family $S$ into a ZDD, Toda recursively applies a simple divide-and-conquer algorithm to obtain a BDD of all hitting sets of $S$. He then uses a minimization algorithm to obtain a ZDD of the MHSes of $S$, which he finally decompresses. We will refer to this algorithm as HTC-BDD (the name given to it by Toda).

Toda gives a formal complexity analysis of HTC-BDD in [68], but the resulting bounds are expressed in terms of the intermediate BDD and ZDD data structures and are incommensurable with bounds like those known for FK-A and FK-B. One important factor is that the decompression of the output from ZDD format into a list of sets can be very time-consuming. Details are explored in Section 5.3. However, the ZDD intermediate data structure makes it possible to determine the number of MHSes without decompressing, which may be of interest for some applications.

A C implementation of this algorithm by the author of [68] is available in the repository.

4.2.7 Cardoso and Abreu (2014)

Cardoso and Abreu revisited the STACCATO approach in [6]. They present several optimizations to reduce wasted computation. In addition, their new algorithm is distributed using the widely-used Map-Reduce paradigm, so in principle it can be deployed over very large message-passing distributed computing systems. It is also designed so that early termination will return a useful approximate result; a collection of hitting sets will be obtained, although they may not be minimal and some may be missing. We will refer to this algorithm as MHS$^2$ (the name given to it by its authors).

The authors are not aware of a formal analysis of the complexity of MHS$^2$.

A C++ implementation of the algorithm by the authors of [6] is available in the repository.

4.3 MHS buildup approaches

A third type of approach to computing MHSes of a set family $S$ is to construct sets of elements which are expected or guaranteed to be subsets of MHSes, then iteratively add elements until they are hitting sets.
This approach fits into the standard scheme of “backtracking” combinatorial algorithms. The approaches in this section all follow this approach; they differ primarily in the conditions used to identify candidate sub-MHSes and the strategies used to avoid redundant calculation.

### 4.3.1 Hébert et al. (2007)

Hébert et al. take an approach in [69] that brings insights from data mining to bear on the MHS generation problem. We follow the explanation of the algorithm in [70], which avoids the algebraic complexity of the original. We will refer to this algorithm as MTMiner (the name given to its software implementation by its authors); it is also called HBC (for Hébert, Bretto, and Crémilleux) in some literature (e.g. [70]).

Fix a set family \( S = \{s_1, s_2, \ldots, s_n\} \) with underlying element set \( E = \bigcup S = \{e_1, e_2, \ldots, e_m\} \). The MTMiner algorithm is initialized with the set \( C_1 = \{\{e\} \mid e \in E\} \) of element sets of size 1. At each step of the algorithm, the set \( C_i \) of candidate hitting sets of size \( i \) is processed. First, any set in \( C_i \) which is a hitting set is removed and outputted; as will be seen, it is guaranteed to be minimal. The remaining sets in \( C_i \) are extended by combining all pairs \((a, b)\) which overlap in \( i - 1 \) elements into their union \( a \cup b \). For each of these extended sets (of size \( i + 1 \)), the algorithm checks whether more sets are hit by \( a \cup b \) than by \( a \) or \( b \). If so, \( a \cup b \) is added to \( C_{i+1} \). The algorithm terminates no later than \( i = n \), by which time all MHSes have been output.

Pseudocode of the algorithm is given in Algorithm 3.

#### Algorithm 3 MTMiner algorithm

**Input:** A family of sets \( S = \{s_1, s_2, \ldots, s_n\} \)

**Output:** The set of MHSes of \( S \)

1: function MTMiner(S)
2:     \( C_1 \leftarrow \emptyset \) \quad \triangleright \text{initial candidate set}
3:     for all \( e \in \bigcup S \) do
4:         if \( e \in s \) for all \( s \in S \) then
5:             output \( \{e\} \)
6:         else
7:             \( C_1 \leftarrow C_1 \cup \{e\} \)
8:         end if
9:     end for
10:    \( i \leftarrow 1 \) \quad \triangleright \text{Size of candidates under consideration}
11:    while \( C_i \neq \emptyset \) do
12:        \( C_{i+1} \leftarrow \emptyset \) \quad \triangleright \text{candidates of size } i + 1
13:        for all \( a, b \in C_i \) such that \( |a \cup b| = i + 1 \) do
14:            \( c \leftarrow a \cup b \)
15:            if \( c \setminus \{e\} \in C_i \) and \( c \setminus \{e\} \) hits fewer sets than \( c \) for all \( e \in c \) then
16:                if \( c \) is a hitting set of \( S \) then
17:                    output \( c \)
18:                else
19:                    \( C_{i+1} \leftarrow C_{i+1} \cup \{e\} \)
20:                end if
21:            end if
22:        end for
23:        \( i \leftarrow i + 1 \)
24:    end while
25: end function

The authors claim a running time bound of \( O(2^x \cdot y) \) where \( x \) is the size of the largest hitting set and \( y \) is the number of hitting sets of \( S \). However, Hagen shows in [70] that this bound is incorrect. He shows that MTMiner is not output-polynomial and that its complexity is \( n^{\Omega(\log \log n)} \), where \( n = |S| + y \).

It is possible to search only for MHSes of bounded cardinality with MTMiner by discarding any candidate that is too large. The second author and collaborators apply this approach as a “greedy algorithm” in [15] to
study minimal interventions in a biochemical signalling network. They take a different approach to element search than that in Lines 13 and 14; they instead loop over all candidate sets \( a \) of a given size and consider \( a \cup \{ e \} \) for every element \( e \notin a \) which does not form a singleton hitting set. They also consider sets and elements in orders determined by a heuristic score called OCSANA to optimize the quality of approximate results in cases where complete enumeration is infeasible.

A C++ implementation of this algorithm by the authors of [69] is available in the repository.

4.3.2 Murakami and Uno (2014)

Murakami and Uno take a somewhat different approach in [61] in two new algorithms. We will refer to these algorithms as MMCS and RS (the names given to them by their authors). Both rely on a crucial observation which makes possible efficient bottom-up searches for minimal hitting sets.

First, we require two definitions. For a given family of sets \( S \), a sub-MHS is a set \( M \) which is a subset of some MHS of \( S \). For a given set \( E \) of elements of \( S \), an element \( e \in E \) is critical in \( E \) if there is at least one set \( s \in S \) which contains \( e \) but no other elements of \( E \).

Then we have the following proposition, appearing in various forms in cf. [69, 61]:

**Proposition 4.4.** A set \( M \) of elements of a set family \( S \) is a sub-MHS if and only if every \( m \in M \) is critical in \( M \). In this case, we say that \( M \) satisfies the minimality condition.

Thus, the MHSes of a set family are exactly the maximal element sets satisfying the minimality condition. Both algorithms MMCS and RS proceed by building up sets that satisfy the minimality condition until they are hitting sets, making clever use of intermediate data structures to ensure that no redundant checks are performed.

Let \( k = \|S\| \) be the sum of the sizes of the sets in a set family \( S \). Then MMCS runs in \( O(k) \) time per iteration of its main loop, but the authors of [61] do not give bounds for the number of iterations required.

For RS, each iteration also takes \( O(k) \) time, but the number of iterations can be bounded explicitly: it is \( O(\sum y_i) \) for \( y_i \) the number of MHSes of the subfamily \( S_{\leq i} = \{s_1, \ldots, s_i\} \). Thus, the total running time is \( O(k \cdot \sum y_i) \).

It is possible to search only for MHSes of bounded cardinality with MMCS or RS by simply discarding any candidate that is too large. Furthermore, it is straightforward to parallelize the algorithm using the task model. However, the shd program distributed by the authors of [61] does not support either of these modes.

A C implementation of MMCS and RS by the authors of [61] is available in the repository. A C++ implementation of the parallel versions pMMCS and pRS which supports efficient enumeration of small MHSes is also included.

4.4 Full cover approaches

A fourth type of approach to computing the MHSes of a set family \( S \) is to decompose the underlying elements into several subsets such that every set in \( S \) lies entirely in one of them. Formally, a full cover of \( S \) is another set family \( C \) with the property that every \( s \in S \) is a subset of some \( c \in C \).

For any dual cover, we have the following decomposition result, given in cf. [43, Lemma 3], which we express in the algebraic language of hypergraphs:

**Lemma 4.5.** Let \( H \) be a simple hypergraph and let \( C \) be a full cover of \( H \). Then the transversal hypergraph \( \text{Tr} H \) of \( H \) satisfies

\[
\text{Tr} H = \bigwedge_{c \in C} \text{Tr}(H_c).
\]

where \( \wedge \) is the wedge operation defined in Definition 4.1, and \( H_c \) is the subhypergraph of \( H \) containing only the edges that are subsets of \( c \).

Given a set family \( S \) and a full cover \( C \) of \( S \), we can use equation (3) to break down the dualization computation into several independent computations which can be run in parallel, then merge the results using the hypergraph wedge operation. (Each of these computations can in turn be decomposed recursively.)
There are two easy-to-find full covers of any set family $S$: the family $S$ itself and the singleton family \{\bigcup S\}. The approaches given below use more refined full covers to ensure that the recursion of Lemma 4.5 is efficient.

Pseudocode of this approach is given in Algorithm 4 in the language of set families.

**Algorithm 4 Full cover algorithm**

**Input:** A set family $S = \{s_1, s_2, \ldots, s_n\}$ and a full cover $C$ of $S$

**Output:** The set of MHSes of $S$

1: function FullCoverDualize($S, C$)
   2:     for all $c \in C$ do \hspace{1cm} \Comment{can be considered in parallel}
   3:         $S_c \leftarrow \min(\{s|s \subseteq c\})$
   4:         $C_c \leftarrow \text{some full cover of } S_c \hspace{1cm} \Comment{details vary by algorithm}$
   5:         $T_c \leftarrow \text{FullCoverDualize}(S_c, C_c)$
   6:     end for
   7:     $T \leftarrow \min(\{\bigcup_{c \in C} t_c | t_c \in T_c\}) \hspace{1cm} \Comment{hypergraph wedge}$
   8:     return $T$
   9: end function

Of course, the efficiency of this algorithm depends on the choice of full cover in Line 4. In particular, the procedure to choose this full cover $C$ should have three properties:

1. $C$ should have many components, to spread the load over many processors;
2. the individual computations FullCoverDualize($S_c, C_c$) should be substantially smaller in scale than the full computation, so no one processor has too much work to do; and
3. the merge operation in Line 7 (and in particular the minimization step) should not be too complex, so the sequential part of the algorithm does not dominate the running time.

Several published algorithms fit into this scheme; they differ primarily in how they approach the construction of $C$.

**4.4.1 Khachiyan et al. (2007)**

Khachiyan et al. introduced the full cover decomposition approach in [50, 46]. They focus on hypergraphs with a curious property: the restriction of the hypergraph to any vertex subset $V'$ admits a full cover with the property that each covering edge has size less than $(1 - \epsilon)|V'|$ for a fixed threshold parameter $0 < \epsilon < 1$. They show that using such a collection of full covers in Lemma 4.5 yields an efficient recursive algorithm. They then are able to show that this procedure runs in polylogarithmic time on polynomially many processors, with coefficients determined by the value of $\epsilon$. We will refer to this algorithm as pKBEG (for the Parallel algorithm of Khachiyan, Boros, Elbassioni, and Gurvich).

Of course, this analysis only applies if such a family of full covers can be found. They demonstrate constructions (and give explicit values of $\epsilon$) for several important families: hypergraphs with bounded edge size (“dimension”), bounded “dual-conformality” (a condition related to intersections in the transversal), or bounded edge-transversal intersection size.

The authors are not aware of a public implementation of this algorithm. Since it does not apply in generality, we will not study it in Section 5.

**4.4.2 Elbassioni (2008)**

Following up on [50, 46], Elbassioni presents in [71] two parallel decomposition approaches for the transversal recognition problem. The first is essentially a rearrangement of FK-B to make the search tree broader and shallower so parallel computation is efficient. The second is a variant of a full-cover decomposition algorithm; given a transversal $T$ of a hypergraph $H$ with vertex set $V$, it uses

$$C(T) = \{V \setminus \{i\}|i \in T\} \cup \{T\} \quad (4)$$
as a full cover of $\text{Tr} H$ to decompose the problem. (He also incorporates a special divide-and-conquer case similar to the FK algorithms under certain circumstances.) We will refer to this algorithm as pELB (for the Parallel algorithm of Elbassioni).

Elbassioni shows that this algorithm runs in polylogarithmic time on quasipolynomially many processors in polynomial space for any hypergraph; in particular, letting $n$ be the number of vertices, $x$ be the number of edges of $H$, and $y$ be the size of $\text{Tr} H$, the running time is bounded by both $n^2 x^{o(\log y)}$ and $n^2 y^{o(\log x)}$, so any asymmetry in the sizes of $H$ and $\text{Tr} H$ reduces the runtime. (The exact bounds are cumbersome to state but may be found in [71].)

The authors are not aware of any public implementation of this algorithm.

4.4.3 Boros and Makno (2009)

Boros and Makino present in [43] a full cover algorithm which improves on the asymptotic complexity bounds of [71] for transversal recognition. To do this, they introduce another full cover in addition to that in equation (4), which they incorporate into an FK-like recursive duality-testing framework. Fix a hypergraph $H$ and an edge $e \in H$; then

\[ C(e) = \{(V \setminus f) \cup \{i\}|f \in H, i \in f \cap e\} \tag{5} \]

is a full cover of Tr $A$. By carefully choosing when to use a full cover from equation (4) or equation (5), Boros and Makino are able to obtain very strong bounds on parallel runtime. We will refer to this algorithm as pBM (for the Parallel algorithm of Boros and Makino).

Fix a hypergraph $H$ with $n$ vertices and $x$ edges for which $\text{Tr} H$ has $y$ edges. Then pBM runs in $O(\log n + \log x \log y)$ time using $O(nx^{1+\log x})$ processors.

A C++ implementation of this algorithm for MHS generation, written by the first author, is available in the repository.

4.5 Other

Some authors have used approaches that translate the MHS generation problem into other domains for which specialized algorithms already exist. We outline these below.

4.5.1 Primary decomposition of squarefree monomial ideals

The MHS generation problem can be translated into a problem in computational algebra. Fix a set family $S = \{s_1, s_2, \ldots, s_n\}$ with underlying element set $E = \bigcup_i s_i = \{e_1, \ldots, e_m\}$. To each element $e_i$ associate a variable $x_i$ in a polynomial ring over $\mathbb{Q}$. To each set $s_i$, associate a monomial $m_i = \prod_{j \in s_i} x_j$. (For example, the set $\{1, 2, 5\}$ becomes the monomial $x_1x_2x_5$). We can then construct a monomial ideal $I_S$ generated by the monomials $m_i$, which encodes the set family algebraically. By construction, $I_S$ is squarefree. It then turns out that the generators of the associated primes of $I_S$ correspond exactly to the minimal hitting sets of $H$.

We will refer to this approach as PrimDecomp.

This approach was used by Jarrah et al. in [19] for an application in computational biology. They calculate the associated primes of $I_S$ using Alexander duality [72] as provided in Macaulay2 [73].

A container which uses Macaulay2 to perform this calculation is provided in the repository.

4.5.2 Integer programming

The MHS generation problem can be interpreted as an integer programming problem. Fix a set family $S = \{s_1, s_2, \ldots, s_n\}$ with underlying element set $E = \bigcup_i s_i = \{e_1, \ldots, e_m\}$. We declare $n$ variables $x_i$, each of which may take values from $\{0, 1\}$. A subset $T$ of the vertices then corresponds to an assignment $x$ of the $x$-variables. For each set $s_i$, we impose a constraint $\sum_{j \in s_i} x_j \geq 1$; an assignment $x$ corresponds to a hitting set if it satisfies all these constraints. Enumeration of inclusion-minimal assignments that satisfy the constraints is then exactly the MHS generation problem. (Indeed, it was shown by Boros et al. in [74] that MHS generation is equivalent to the general problem of enumerating minimal solutions to the linear system $Ax = b$ for $0 \leq x \leq c$ where $A$ is a binary matrix, $x$ is a binary vector, and $b$ and $c$ are all-ones vectors.) We will refer to this approach as IntProg.
Because linear programming solvers are so diverse and many widely-used ones are proprietary, we do not provide an implementation of this approach.

## 4.6 Feature comparison

We summarize in Table 1 the salient features of the algorithms introduced in Section 4.

### Table 1: Feature comparison of MHS generation algorithms

| Algorithm          | Domain           | Published | Eval. | Parallel | Cutoff |
|--------------------|------------------|-----------|-------|----------|--------|
| **Set iteration**  |                  |           |       |          |        |
| Berge (§4.1.1)     | Hypergraphs      | 1984      | ✓     |          |        |
| HS-DAG (§4.1.2)    | Fault diagnosis  | 1989      | ✓     |          |        |
| HST (§4.1.3)       | Fault diagnosis  | 2001      | ✓     |          |        |
| BMR (§4.1.5)       | Data mining      | 2003      | ✓     |          |        |
| DL (§4.1.4)        | Data mining      | 2005      | ✓     |          |        |
| KS (§4.1.6)        | Hypergraphs      | 2005      | ✓     |          |        |
| **Divide and conquer** |                |           |       |          |        |
| FK-A (§4.2.2)      | Boolean algebra  | 1996      | ✓     |          |        |
| FK-B (§4.2.2)      | Boolean algebra  | 1996      | ✓     |          |        |
| BOOL (§4.2.1)      | Fault diagnosis  | 2003      | ✓     |          |        |
| STACCATO (§4.2.3)  | Fault diagnosis  | 2009      | ✓     |          |        |
| ParTran (§4.2.4)   | Poset theory     | 2010      | ✓     | ✓        | ✓      |
| Knuth (§4.2.5)     | Boolean algebra  | 2011      | ✓     |          |        |
| HTC-BDD (§4.2.6)   | Boolean algebra  | 2013      | ✓     |          |        |
| MHS² (§4.2.7)      | Fault diagnosis  | 2014      | ✓     | ✓        | ✓      |
| **MHS buildup**    |                  |           |       |          |        |
| MTMiner (§4.3.1)   | Data mining      | 2007      | ✓     |          |        |
| MMCS (§4.3.2)      | Hypergraphs      | 2014      | ✓     | ✓ c      | ✓      |
| RS (§4.3.3)        | Hypergraphs      | 2014      | ✓     | ✓ c      | ✓      |
| **Full covers**    |                  |           |       |          |        |
| pKBE (§4.4.1)      | Boolean algebra  | 2007      | ✓     |          |        |
| pELB (§4.4.2)      | Boolean algebra  | 2008      | ✓     |          |        |
| pBM (§4.4.3)       | Boolean algebra  | 2009      | ✓     | ✓        |        |
| **Other**          |                  |           |       |          |        |
| PrimDecomp (§4.5.1)| Comp. algebra    | 2007      | ✓     |          |        |
| IntProg (§4.5.2)   | Optimization     | –         |       |          |        |

\[\text{Eval.}^a\] indicates that an algorithm is evaluated in Section 5.

\[\text{Cutoff}^b\] indicates that an algorithm can generate only small MHSs to save time.

\[\text{Separate parallel implementation by the first author}\]

## 4.7 Algorithm miscellany

A genetic algorithm for finding many (but not necessarily all) small (but not necessarily minimal) hitting sets is studied by Li and Yunfei in [15]. Vinterbo and Öhrn study in [14] the more refined problem of finding weighted \(r\)-approximate hitting sets, which are sets which hit some fraction \(0 \leq r \leq 1\) of the target sets according to assigned weights; they also apply a genetic algorithm with promising results.

Jelassi et al. consider the efficacy of pre-processing methods in [75]. They find that, for many common classes of set families, it is worthwhile to compute from the family \(S\) a new family \(S'\) which combines elements...
which occur only in the same sets into so-called generalized nodes. (This optimization was also used by Kavvadias and Stavropoulos in [62] for their algorithm KS.) Their algorithm Irred-Engine performs this preprocessing, applies a known MHS algorithm (in their case, MMCS from [61]) to the resulting family $S'$, and then expands the results into MHSes for the original $S$. We will not study this approach separately here, but it may be of interest for applications where many vertices may be redundant.

5 Time-performance comparison of the algorithms

Numerous previous papers have included experimental comparisons of some algorithms, including [61, 65, 62, 69, 64, 12, 6, 13]. However, we find that there is need for a new, comprehensive survey for several reasons:

1. Each published comparison involves only a few algorithms, and differences in data sets and environment make the results incompatible. Thus, it is not possible to assemble a systematic overview of the relative performance of these algorithms.

2. Many existing comparisons overlook published algorithms in domains far from the authors’ experience. An algorithm designed for monotone dualization may prove to be useful for data mining, for example, but authors in that field may be unaware of it due to translational issues in the literature.

3. Most existing comparisons are not published alongside working code and do not provide methodological details so that the results can be reproduced or extended. (Murakami and Uno’s work in [61] is a notable exception, and indeed their publicly-available implementations are used for several algorithms here.)

5.1 Methodology

We have assembled a repository of software implementations of existing algorithms. Each is wrapped in a Docker container using the Algorun framework and using standardized JSON formats for input and output. Details, code, and containers are available from https://github.com/VeraLiconaResearchGroup/MHSGenerationAlgorithms including complete instructions for reproducing the experimental environment and running new experiments. These containers are easy to deploy on any computer supporting the Docker container environment; they do not require compiling any code or downloading libraries. Interested readers are encouraged to run similar experiments on their own data sets.

We have run each implemented algorithm on a variety of input set families (discussed in detail in Section 5.2). Each was allowed to run for up to one hour (3600 s) before termination; at least one algorithm ran to termination on every data set with this timeout. Algorithms which did not time out were run a total of three times and the median runtime used for analysis, presented in Section 5.3. Algorithms which support cutoff enumeration (that is, finding only hitting sets of size up to some fixed $c$) were run with $c = 5, 7,$ and 10 as well as full enumeration. Algorithms which support multiple threads were run with $t = 1, 2, 4, 6, 8, 12,$ and 16 threads. All experiments were performed on a workstation with an Intel Xeon E5-2630v3 processor with eight cores at 2.4 GHz (with Hyperthreading enabled, allowing 16 concurrent threads) and 32 GB of ECC DDR4 RAM.

In all cases, the generated hitting sets were compared to ensure that the algorithms were running correctly. This revealed errors in several published implementations, which are discussed in Section 4 and Table 1. In cases where an algorithm’s results are only slightly incorrect, we have included its benchmark timing in the results below, since we believe these still give a useful impression of the relative performances of these algorithms.

5.2 Data sets used for time-performance comparison

We apply each algorithm to a variety of set families derived from real-world data. We briefly discuss each data set here. We have focused on data sets that provide large, heterogenous set families, since these cases highlight the performance differences among algorithms; for smaller families, the differences may be negligible in practice.

20
accident

Anonymized information about several hundred thousand accidents in Flanders during the period 1991–2000. Originally published in [76]. Converted by the authors of [61] into a set family whose sets are the complements of maximal frequent itemsets with specified threshold $1000\theta$ for $\theta \in \{70, 90, 110, 130, 150, 200\}$; MHSes of this set family then correspond to minimal infrequent itemsets. All of the set families have 441 underlying elements; numbers of sets range from 81 (for $\theta = 200$) to 10968 (for $t = 70$). This formulation was downloaded from [77].

ecoli

Metabolic reaction networks from *E. coli*. Reaction networks for producing acetate, glucose, glycerol, and succinate, along with the combined network, were analyzed to find their “elementary modes” using Metatool [78], which are given as set families. MHSes of these set families correspond to “minimal cut sets” of the original networks, which are of interest in studying and controlling these networks. Statistics for these set families are given in Table 2.

Table 2: Statistics for *E. coli* network data sets

| Network  | elements | sets   | avg. set size |
|----------|----------|--------|---------------|
| Acetate  | 103      | 266    | 23.7          |
| Glucose  | 104      | 6387   | 30.4          |
| Glycerol | 105      | 2128   | 27.2          |
| Succinate| 103      | 932    | 22.3          |
| Combined | 109      | 27503  | 30.6          |

ocsana

Interventions in cell signalling networks. Two cell signalling networks (EGFR from [79] and HER2+ from [20]) were analyzed to find their “elementary pathways” using OCSANA, which are given as set families. MHSes of these set families correspond to “optimal combinations of interventions” in the original networks, which are of interest in studying and controlling these networks. Each network has been preprocessed to find these elementary pathways using three different algorithms of increasing resolution: shortest paths only (SHORT), including “suboptimal” paths (SUB), and including all paths up to length 20 (ALL). This results in six set families. Statistics for these families are given in Table 3.

Table 3: Statistics for OCSANA network data sets

| Network | Method  | elements | sets   | avg. set size |
|---------|---------|----------|--------|---------------|
| EGFR    | SHORT   | 49       | 125    | 8.9           |
|         | SUB     | 55       | 234    | 9.9           |
|         | ALL     | 63       | 11050  | 16.5          |
| HER2+   | SHORT   | 122      | 534    | 15.2          |
|         | SUB     | 171      | 2538   | 20.3          |
|         | ALL     | 318      | 69805  | 19.1          |

These set families demonstrate particularly effectively the problem of combinatorial explosion in MHS generation. For example, the HER2+.SHORT set family has just 122 underlying elements and 534 sets, but we have computed that it has 128833310 MHSes. Even storing the collection of MHSes in memory is difficult because of its size. As a result, none of the algorithms tested were able to complete the full enumeration of MHSes for any of the HER2+ data sets. However, the cutoff enumeration is much more manageable; for example, HER2+.SHORT has just 26436 MHSes of size $c \leq 7$, which are found easily by several of the algorithms under study.
5.3 Results

We present below the results of the benchmarking experiments on the data sets described in Section 5.2. All experiments were performed on a workstation with an Intel Xeon E5-2630v3 processor with eight cores at 2.4 GHz (with Hyperthreading enabled, allowing 16 concurrent threads) and 32 GB of ECC DDR4 RAM. Each algorithm was allowed to run for up to 3600 s; algorithms that did not complete in this time are marked with –, while algorithms that crashed due to memory exhaustion are marked with !.

5.3.1 Full enumeration

We first consider the general problem of enumerating all MHSes of a given set family. Timing results for the full enumeration cases are given in Tables 4 to 6, with algorithms sorted in approximately increasing order of speed.

5.3.2 Multithreaded full enumeration

Although many of the published algorithms are serial, a few can be parallelized. For the algorithms for which multithreaded implementations were available, we have run tests with $t \in \{1, 2, 4, 6, 8, 12, 16\}$ threads on our workstation with eight true cores and Hyperthreading support. Timing results for selected full enumeration cases with various numbers of threads are shown in Tables 7 and 8.

5.3.3 Cutoff enumeration

In many applications, only small MHSes are relevant. We consider here the enumeration of MHSes of size no greater than some “cutoff” $c$; we have run benchmarks for $c \in \{5, 7, 10\}$ using the algorithms which support cutoff mode. Timing results for selected cutoff enumeration cases are given in Tables 9 and 10.

5.4 Discussion

As shown in Section 5.3, the algorithms MMCS and RS from [61] and HTC-BDD from [68] are far faster than their competitors across a variety of input set families.

HTC-BDD is extremely fast on inputs for which it terminates, outperforming its closest competitors by a factor of 4 to 10 on many inputs. However, it frequently exhausted the 32 GB available memory on our workstation. In addition, it does not support cutoff enumeration. Thus, we recommend HTC-BDD for situations where all the MHSes of moderately-sized set families must be found quickly—for example, when many such families must be processed. Since the core algorithm takes a ZDD representation of the input set family and returns either a BDD or a ZDD of its hitting sets, it is also very suitable for processing pipelines where BDDs are already used.

MMCS and RS are also very fast, and they support both cutoff and full enumeration. They have the additional benefit of very low memory requirements—in principle, the space required for a run depends only on the size of the input set family. This is especially useful for inputs like HER2+.SHORT where $S$ is small ($\approx$ 500 sets) but has an enormous collection of MHSes ($\approx$ 128 million). Thus, we recommend these algorithms for situations where very large set families are studied or where only the small MHSes are required.

We note, however, that the provided implementations (both those by Murakami and Uno and by the first author) store the result MHSes in memory before writing them to disk, which did result in memory exhaustion for some inputs in our experiments. It would be straightforward to modify the implementations of MMCS or RS to stream the result MHSes to disk rather than storing them in memory or to count them without storing them at all, as we did to compute the number of MHSes of size $\leq 10$ for HER2.short and HER2.all in Table 9. In addition, the implementations mmcs and rs of Uno and pMMCS and pRS of the first author varied dramatically in performance depending on the input, highlighting the importance of implementation. Researchers planning to use any of these algorithms should certainly benchmark all the available implementations on data drawn from their application before adopting one.

We also find that parallel algorithms for MHS generation can be highly effective. For example, the MHS$^2$ algorithm (cf. Section 4.2.7) of Cardoso and Abreu [6] shows a 2.33× improvement in running time.

1Supported by the first author’s implementations pMMCS and pRS.
Algorithm |\textbf{accident} (threshold $\theta$ in thousands of incidents, smaller $\theta$ gives larger data set) | $\theta = 200$ | $\theta = 150$ | $\theta = 130$ | $\theta = 110$ | $\theta = 90$ | $\theta = 70$ | $\theta = 50$ | $\theta = 30$
|---|---|---|---|---|---|---|---|---|
m MCS | 0.00 | 0.01 | 0.02 | 0.06 | 0.23 | 0.66 | 2.28 | 20.26 |
rs | 0.00 | 0.01 | 0.03 | 0.06 | 0.23 | 0.65 | 2.27 | 19.84 |
pMMCS | 0.01 | 0.02 | 0.03 | 0.06 | 0.22 | 0.56 | 1.82 | 17.32 |
pRS | 0.01 | 0.02 | 0.03 | 0.07 | 0.26 | 0.75 | 3.05 | 46.81 |
nntminer | 0.01 | 0.02 | 0.04 | 0.08 | 0.32 | 0.94 | 3.45 | 28.59 |
bmr | 0.01 | 0.05 | 0.10 | 0.18 | 1.40 | 3.44 | 13.60 | 108.56 |
h tcbdd | 0.38 | 0.39 | 0.46 | 0.44 | 0.59 | 0.83 | ! | ! |
kmuth | 0.32 | 0.35 | 0.38 | 0.41 | 0.73 | 1.34 | ! | ! |
mhs2 | 0.01 | 0.04 | 0.12 | 0.29 | 2.67 | 14.72 | 125.86 | – |
dl | 0.01 | 0.07 | 0.23 | 0.70 | 3.06 | 17.38 | 146.44 | 2763.59 |
fka-begk | 0.26 | 1.12 | 2.88 | 7.08 | 64.23 | 321.60 | 2271.22 | – |
bool | 0.04 | 0.24 | 1.26 | 3.08 | 111.49 | 493.72 | – | – |
hst | 0.12 | 2.73 | 12.26 | 54.80 | 283.95 | 1855.83 | – | – |
primdecomp | 0.53 | 1.19 | 3.66 | 7.38 | 272.63 | 888.39 | – | ! |
hsdag | 0.26 | 2.11 | 11.91 | 41.30 | 2344.98 | – | – | – |
berge | 0.38 | 6.71 | 52.71 | 290.00 | – | – | – | – |
partran | 0.84 | 16.46 | 167.95 | 727.06 | – | – | – | – |
pbm | 3.20 | 1758.71 | – | – | – | – | – | – |
staccato | 45.59 | – | – | – | – | – | – | – |

|\textbf{vertices} | 64 | 64 | 81 | 81 | 335 | 336 | 336 | 442 |
|\textbf{edges} | 81 | 447 | 990 | 2000 | 4322 | 10968 | 32207 | 135439 |
|\textbf{|MHSes|} | 253 | 1039 | 1916 | 3547 | 7617 | 17486 | 47137 | 185218 |

Table 4: Computation time (in s) to enumerate all MHSes for \textbf{accident} with various cutoff values $\theta$

(– indicates timeout. ! indicates memory exhaustion.)
| Algorithm  | EGFR |   |   |
|------------|------|---|---|
|            | short | sub | all |
| mmcs       | 0.01  | 0.05 | 1.34 |
| rs         | 0.01  | 0.05 | 1.33 |
| pMMCS      | 0.08  | 0.46 | 3.21 |
| pRS        | 0.04  | 0.37 | 4.67 |
| mtminer    | !     | !     | !   |
| bmr        | 0.22  | 2.99 | 22.26 |
| htcbbdd    | 0.40  | 0.50 | 0.51 |
| knuth      | 0.37  | !     | !   |
| mhs2       | –     | –     | !   |
| dl         | 0.14  | 4.25 | 39.66 |
| fka-begk   | 10.39 | 152.44 | 705.07 |
| bool       | 12.18 | 3115.06 | 2865.42 |
| hst        | –     | –     | !   |
| primdecomp | 1.01  | 10.00 | 13.91 |
| hsdag      | –     | –     | !   |
| berge      | 0.31  | 9.89  | 2065.67 |
| partran    | 289.41 | –   | –   |
| pbm        | 1544.00 | –   | –   |
| staccato   | –     | –     | –   |
| |vertices| 51   | 57   | 65   |
| |edges   | 125  | 234  | 11050 |
| |MHSes  | 1340 | 11765 | 13116 |

Table 5: Computation time (in s) to enumerate all MHSes for ocsana-egfr with the path-finding strategies short, sub, and all
(– indicates timeout. ! indicates memory exhaustion.)
### Table 6: Computation time (in s) to enumerate all MHSes for ecoli networks

| Algorithm | mcmc | pMMCS | pRS | mntminer | bmr | htcbdd | kmuth | mhs2 | dl | fka-begk | bool.iterative | hst | primdecomp | lsdag | berge | partran | pbm | staccato | \( |vertices| | \|edges| | \|\|MHSes| |
|-----------|------|-------|-----|----------|-----|--------|-------|------|----|----------|---------------|-----|-----------|-------|-------|--------|-----|----------|----------|
| acetate   | 0.05 | 0.07  | 0.33| 149.86   | 0.56| 0.47   |       | 140.50| 3.15| 25.80    | 5.11          |     | 3.46      | 2945.19| 52.90 |        |     |          |          |
| succinate | 0.86 | 0.46  | 3.95| 682.34   | 12.13| 0.55   |       | 737.68| 225.51| 514.18   | 438.79        |     | 22.42     |       |       | 1400.99|     |          |          |
| glycerol  | 2.51 | 1.13  | 15.49| !        | 33.20| 0.69   |       | 2447.53| 1771.77| 1421.68  |             |     | 53.24     |       |       | !       |     |          |          |
| glucose   | 4.72 | 4.43  | 26.31| !        | 26.74| 0.85   |       | !     | 1749.88| !         |     | 62.57     | !      |       | !       |     |          |          |
| combined  | 157.08| 202.65| 1989.85| !        | 2209.63| !     |       | !     | !     | !         | !           |     | !        | !      | !     | !       |     |          |          |

(– indicates timeout. ! indicates memory exhaustion.)

### Table 7: Computation time (in s) to enumerate all MHSes with \(t\) threads for ecoli-acetate

| Algorithm | pMMCS | pRS | mhs2 | pbm | \(t = 1\) | \(t = 2\) | \(t = 4\) | \(t = 6\) | \(t = 8\) | \(t = 12\) | \(t = 16\) |
|-----------|------|-----|------|-----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| acetate   | 0.07 | 0.33| 140.50|     | 0.06      | 0.23      | 0.04      | 0.20      | 0.04      | 0.14      | 0.12      | 0.07      |
| succinate | 0.04 | 0.14| 90.99 |     | 0.04      | 0.13      | 0.04      | 0.14      | 0.06      | 0.14      | 0.12      |          |
| glycerol  | 0.04 | 0.14| 62.52 |     | 0.04      | 0.13      | 0.04      | 0.14      | 0.06      | 0.14      | 0.12      |          |
| glucose   | 0.04 | 0.14| 61.36 |     | 0.04      | 0.13      | 0.04      | 0.14      | 0.06      | 0.14      | 0.12      |          |
| combined  | 0.04 | 0.14| 60.60 |     | 0.04      | 0.13      | 0.04      | 0.14      | 0.06      | 0.14      | 0.12      |          |

(– indicates timeout. ! indicates memory exhaustion.)

### Table 8: Computation time (in s) to enumerate all MHSes with \(t\) threads for ecoli-combined

| Algorithm | pMMCS | pRS | mhs2 | pbm | \(t = 1\) | \(t = 2\) | \(t = 4\) | \(t = 6\) | \(t = 8\) | \(t = 12\) | \(t = 16\) |
|-----------|------|-----|------|-----|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| acetate   | 202.65| 1989.85| 202.65|     | 118.70    | 1072.64   | 591.16    | 431.82    | 364.22    | 348.78    | 334.61    |
| succinate | 72.05 | 591.16| 72.05 |     | 60.01     | 431.82    | 364.22    | 348.78    | 334.61    |            | !         |
| glycerol  | 60.01 | 431.82| 60.01 |     | 50.76     | 364.22    | 348.78    | 334.61    |            | !         | !         |
| glucose   | 50.76 | 364.22| 50.76 |     | 49.16     | 348.78    | 334.61    |            | !         | !         | !         |
| combined  | 49.16 | 348.78| 49.16 |     | !         | !         | !         | !         | !         | !         | !         |

(– indicates timeout. ! indicates memory exhaustion.)
| Algorithm | \(c = 5\) | \(c = 7\) | \(c = 10\) | \(c = 5\) | \(c = 7\) | \(c = 10\) |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| pMMCS     | 0.10      | 1.08      | !         | 5.71      | 64.89     | !         |
| pRS       | 0.08      | 2.23      | !         | 1.88      | 94.69     | !         |
| mhs2      | 649.81    | –         | –         | –         | –         | –         |
| bool      | 1.23      | 42.18     | !         | 201.71    | !         | !         |
| hst       | 26.93     | –         | –         | 10.00     | –         | –         |
| hsdag     | 0.84      | 1218.23   | –         | 2.21      | 666.13    | –         |
| berge     | 5.18      | –         | –         | 42.02     | –         | –         |
| staccato  | –         | –         | –         | –         | –         | –         |
| \(|\text{vertices}|\)   | 124       | 124       | 124       | 320       | 320       | 320       |
| \(|\text{edges}|\)  | 534       | 534       | 534       | 69805     | 69805     | 69805     |
| \(|\text{MHSes}|\) | 88        | 26436     | 8744333   | 40        | 1892      | 2853026   |

Table 9: Computation time (in s) to enumerate all MHSes up to size \(c\) for ocsana-HER2+ with path-finding strategies short and all (– indicates timeout. ! indicates memory exhaustion.)
| Algorithm | acetate |  |  | ecoli | combined |  |  |
|-----------|---------|---|---|-------|----------|---|---|
|           | \(c = 5\) | \(c = 7\) | \(c = 10\) | \(c = 5\) | \(c = 7\) | \(c = 10\) |
| pMMCS     | 0.01    | 0.01 | 0.03 | 0.40  | 1.90     | 17.94 |
| pRS       | 0.01    | 0.03 | 0.10 | 0.27  | 4.12     | 118.49 |
| mhs2      | 1.32    | 12.14| 78.93| 870.36| !        | !      |
| bool      | 0.02    | 0.09 | 0.73 | 21.45 | 198.68   | !      |
| hst       | 0.02    | 8.23 | –   | 8.92  | !        | !      |
| hsdag     | 0.02    | 0.55 | 45.94| 6.96  | 546.50   | !      |
| berge     | 0.20    | 0.93 | 3.80 | 280.45| –        | –      |
| staccato  | –       | –    | –   | –     | !        | !      |
| vertices  | 103     | 103  | 103 | 110   | 110      | 110    |
| edges     | 266     | 266  | 266 | 27503 | 27503    | 27503  |
| MHSes     | 39      | 195  | 735 | 937   | 9212     | 49061  |

Table 10: Computation time (in s) to enumerate all MHSes up to size \(c\) for ecoli networks acetate and combined
(- indicates timeout. ! indicates memory exhaustion.)
when passing from one thread to eight on the ecoli-acetate set family, while the first author’s parallel implementation of the MMCS algorithm (cf. Section 4.3.2) of Murakami and Uno [61] shows a 4.04× speedup when passing from one thread to eight on the ecoli-combined set family. Unfortunately, the first author’s implementation of the EM full-cover-based parallel algorithm (cf. Section 4.4.3) of Boros and Makino [43] was too slow to yield useful results.

6 Conclusion

In this paper, we have surveyed the history and literature concerning the problem of generating minimal hitting sets. The computational complexity of this task is a long-standing open problem. However, since many applications (cf. Section 2) depend on generating MHSes, a variety of algorithms (cf. Section 4) have been developed to solve it across numerous pure and applied research domains.

We have presented extensive benchmarks (cf. Section 5.3) comparing the computation time required by nearly two dozen of these algorithms on a variety of inputs derived from real-world data. These experiments consistently show that the MMCS and RS algorithms (cf. Section 4.3.2) of Murakami and Uno [61] and the HTC-BDD algorithm (cf. Section 4.2.6) of Toda [68] are far faster than other available algorithms across a variety of inputs. We have provided our benchmarking framework and code in easy-to-install Docker containers (cf. Section 5.1), so researchers wishing to analyze the performance of these algorithms on their own inputs can do so easily. Further details are available on our software repository at https://github.com/VeraLiconaResearchGroup/MHSGenerationAlgorithms.

References

[1] T. Eiter, K. Makino, G. Gottlob, Computational aspects of monotone dualization: A brief survey, Discrete Applied Mathematics 156 (11) (2008) 2035–2049. doi:10.1016/j.dam.2007.04.017.

[2] M. Hagen, Algorithmic and computational complexity issues of MONET, Dr. rer. nat., Friedrich-Schiller-Universität Jena (2008).

[3] C. Domingo, N. Mishra, L. Pitt, Efficient read-restricted monotone CNF/DNF dualization by learning with membership queries, Machine learning 37 (1) (1999) 89–110. doi:10.1023/A:1007627028578.

[4] M. L. Fredman, L. Khachiyan, On the complexity of dualization of monotone disjunctive normal forms, Journal of Algorithms 21 (3) (1996) 618–628. doi:10.1006/jagm.1996.0062.

[5] R. Reiter, A theory of diagnosis from first principles, Artificial intelligence 32 (1) (1987) 57–95. doi:10.1016/0004-3702(87)90062-2.

[6] N. Cardoso, R. Abreu, Mhs2: A map-reduce heuristic-driven minimal hitting set search algorithm, in: J. a. M. LourenAgo, E. Farchi (Eds.), Multicore Software Engineering, Performance, and Tools, Vol. 8063 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 25–36. doi:10.1007/978-3-642-39955-8_3.

[7] T. Quaritsch, I. Pill, PyMBD: A library of MBD algorithms and a light-weight evaluation platform, Proceedings of Dx-2014.

[8] F. Wotawa, A variant of reiter’s hitting-set algorithm, Information Processing Letters 79 (1) (2001) 45–51. doi:10.1016/S0020-0190(00)00166-6.

[9] L. Lin, Y. Jiang, The computation of hitting sets: Review and new algorithms, Information Processing Letters 86 (4) (2003) 177–184. doi:10.1016/S0020-0190(02)00506-9.

[10] R. Abreu, A. J. van Gemund, A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis., in: SARA, Vol. 9, 2009, pp. 2–9.

[11] I. Pill, T. Quaritsch, Optimizations for the boolean approach to computing minimal hitting sets., in: ECAI, 2012, pp. 648–653. doi:10.3233/978-1-61499-095-7-648.
[12] G. Dong, J. Li, Mining border descriptions of emerging patterns from dataset pairs, Knowledge and Information Systems 8 (2) (2005) 178–202. doi:10.1007/s10115-004-0178-1

[13] J. Bailey, T. Manoukian, K. Ramamohanarao, A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns, in: Proceedings of the Third IEEE International Conference on Data Mining, IEEE, 2003, p. 485. doi:10.1109/ICDM.2003.1250958.

[14] S. Vinterbo, A. Öhrn, Minimal approximate hitting sets and rule templates, International Journal of approximate reasoning 25 (2) (2000) 123–143. doi:10.1016/S0888-613X(00)00051-7

[15] L. Li, J. Yunfei, Computing minimal hitting sets with genetic algorithm, Tech. rep., DTIC Document (2002).

[16] P. Vera-Licona, E. Bonnet, E. Barillot, A. Zinovyev, OCSANA: optimal combinations of interventions from network analysis, Bioinformatics 29 (12) (2013) 1571–1573. doi:10.1093/bioinformatics/btt195

[17] A. Vázquez, Optimal drug combinations and minimal hitting sets, BMC systems biology 3 (1) (2009) 81. doi:10.1186/1752-0509-3-81

[18] O. Hädicke, S. Klamt, Computing complex metabolic intervention strategies using constrained minimal cut sets, Metabolic engineering 13 (2) (2011) 204–213. doi:10.1016/j.mynb.2010.12.004

[19] A. S. Jarrah, R. Laubenbacher, B. Stigler, M. Stillman, Reverse-engineering of polynomial dynamical systems, Advances in Applied Mathematics 39 (4) (2007) 477–489.

[20] P. Vera-Licona, A. Zinovyev, E. Bonnet, I. Kuperstein, O. Kel, A. Kel, T. Dubois, G. Tucker, E. Barillot, A pathway-based design of rational combination therapies for cancer, european journal of cancer 48 (2012) S154. doi:10.1016/S0959-8049(12)71297-2

[21] T. E. Ideker, V. Thorsson, R. M. Karp, Discovery of regulatory interactions through perturbation: inference and experimental design, in: Pacific symposium on biocomputing, Vol. 5, 2000, pp. 302–313.

[22] C. Berge, Hypergraphs: combinatorics of finite sets, Vol. 45, Elsevier, 1984.

[23] A. D. Korshunov, Monotone boolean functions, Russian Mathematical Surveys 58 (5) (2003) 929–1001. doi:10.1070/RM2003v058n05ABEH000667

[24] S. Schuster, C. Hilgetag, On elementary flux modes in biochemical reaction systems at steady state, Journal of Biological Systems 2 (02) (1994) 165–182. doi:10.1142/S0218339094000131

[25] C. T. Trinh, A. Wlaschin, F. Srin, Elementary mode analysis: a useful metabolic pathway analysis tool for characterizing cellular metabolism, Applied microbiology and biotechnology 81 (5) (2009) 813–826. doi:10.1007/s00253-008-1770-1

[26] J. Zanghellini, D. E. Ruckerbauer, M. Hauscho, C. Jungreuthmayer, Elementary flux modes in a nutshell: Properties, calculation and applications, Biotechnology journal 8 (9) (2013) 1009–1016. doi:10.1002/biot.201200269

[27] S. Klamt, E. D. Gilles, Minimal cut sets in biochemical reaction networks, Bioinformatics 20 (2) (2004) 226–234. doi:10.1093/bioinformatics/btg395

[28] U.-U. Haus, S. Klamt, T. Stephen, Computing knock-out strategies in metabolic networks, Journal of Computational Biology 15 (3) (2008) 259–268. doi:10.1089/cmb.2007.0229

[29] I. Zevedei-Oancea, S. Schuster, A theoretical framework for detecting signal transfer routes in signalling networks, Computers & Chemical Engineering 29 (3) (2005) 597–617. doi:10.1016/j.compchemeng.2004.08.026

[30] R.-S. Wang, R. Albert, Elementary signaling modes predict the essentiality of signal transduction network components, BMC systems biology 5 (1) (2011) 44. doi:10.1186/1752-0509-5-44
[31] B. DasGupta, P. Vera-Licona, E. Sontag, Reverse engineering of molecular networks from a common combinatorial approach, in: M. Elloumi, A. Y. Zomaya (Eds.), Algorithms in Computational Molecular Biology: Techniques, Approaches and Applications, Wiley Online Library, 2010, pp. 941–953. doi:10.1002/9780470892107.ch40.

[32] R. H. Shoemaker, The nci60 human tumour cell line anticancer drug screen, Nature Reviews Cancer 6 (10) (2006) 813–823. doi:10.1038/nrc1951.

[33] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, in: ACM SIGMOD Record, Vol. 22-2, ACM, 1993, pp. 207–216.

[34] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo, Fast discovery of association rules, in: U. M. Fayyad, G. Piatestky-Shapiro, P. Smyth, R. Uthursamy (Eds.), Advances in knowledge discovery and data mining, AAAI/MIT Press, 1996, pp. 307–328.

[35] E. Boros, V. Gurvich, L. Khachiyan, K. Makino, On maximal frequent and minimal infrequent sets in binary matrices, Annals of Mathematics and Artificial Intelligence 39 (3) (2003) 211–221. doi:10.1023/A:1024605820527.

[36] H. Mannila, H. Toivonen, On an algorithm for finding all interesting sentences extended abstract, in: Proceedings of the 13th European Meeting on Cybernetics and Systems Research, Citeseer, 1996.

[37] H. Toivonen, et al., Sampling large databases for association rules, in: VLDB, Vol. 96, 1996, pp. 134–145.

[38] E. C. Stavropoulos, V. S. Verykios, V. Kagklis, A transversal hypergraph approach for the frequent itemset hiding problem, Knowledge and Information Systems (2015) 1–21.

[39] D. Gunopulos, H. Mannila, R. Khardon, H. Toivonen, Data mining, hypergraph transversals, and machine learning, in: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, ACM, 1997, pp. 209–216.

[40] E. Boros, V. Gurvich, L. Khachiyan, K. Makino, On the complexity of generating maximal frequent and minimal infrequent sets, in: STACS 2002, Springer, 2002, pp. 133–141.

[41] G. McGuire, B. Tugemann, G. Civario, There is no 16-clue Sudoku: Solving the Sudoku minimum number of clues problem via hitting set enumeration, Experimental Mathematics 23 (2) (2014) 190–217. doi:10.1080/10586458.2013.870056.

[42] R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher, J. D. Bohlinger (Eds.), Complexity of Computer Computations, The IBM Research Symposia Series, Springer US, 1972, pp. 85–103. doi:10.1007/978-1-4684-2001-2_9.

[43] E. Boros, K. Makino, A fast and simple parallel algorithm for the monotone duality problem, in: Automata, Languages and Programming, Springer, 2009, pp. 183–194.

[44] D. S. Johnson, M. Yannakakis, C. H. Papadimitriou, On generating all maximal independent sets, Information Processing Letters 27 (3) (1988) 119–123.

[45] M. Hagen, Lower bounds for three algorithms for transversal hypergraph generation, Discrete Applied Mathematics 157 (7) (2009) 1460–1469. doi:10.1016/j.dam.2008.10.004.

[46] L. Khachiyan, E. Boros, V. Gurvich, K. Elbassioni, Computing many maximal independent sets for hypergraphs in parallel, Parallel processing letters 17 (02) (2007) 141–152.

[47] N. Mishra, L. Pitt, Generating all maximal independent sets of bounded-degree hypergraphs, in: Proceedings of the tenth annual conference on Computational learning theory, ACM, 1997, pp. 211–217. doi:10.1145/267460.267500.

[48] K. Elbassioni, M. Hagen, I. Rauf, Some fixed-parameter tractable classes of hypergraph duality and related problems, in: Parameterized and Exact Computation, Springer, 2008, pp. 91–102.
T. Eiter, G. Gottlob, K. Makino, New results on monotone dualization and generating hypergraph transversals, SIAM Journal on Computing 32 (2) (2003) 514–537. doi:10.1137/S009753970240639X.

L. Khachiyan, E. Boros, K. Elbassioni, V. Gurvich, A global parallel algorithm for the hypergraph transversal problem, Information Processing Letters 101 (4) (2007) 148–155. doi:10.1016/j.ipl.2006.09.006.

L. Khachiyan, E. Boros, K. Elbassioni, V. Gurvich, A new algorithm for the hypergraph transversal problem, in: COCOON, Springer, 2005, pp. 767–776.

C. Beeri, R. Fagin, D. Maier, M. Yannakakis, On the desirability of acyclic database schemes, Journal of the ACM (JACM) 30 (3) (1983) 479–513.

R. Fagin, Degrees of acyclicity for hypergraphs and relational database schemes, Journal of the ACM (JACM) 30 (3) (1983) 541–550.

T. Eiter, G. Gottlob, Identifying the minimal transversals of a hypergraph and related problems, SIAM Journal on Computing 24 (6) (1995) 1278–1304. doi:10.1137/S0097539793250299.

D. J. Kavvadias, E. C. Stavropoulos, Monotone boolean dualization is in co-NP $[\log 2^n]$, Information Processing Letters 85 (1) (2003) 1–6. doi:10.1016/S0020-0190(02)00346-0.

J. C. Bioch, T. Ibaraki, Complexity of identification and dualization of positive boolean functions, Information and Computation 123 (1) (1995) 50–63. doi:10.1006/inco.1995.1157.

A. I. Hosny, P. Vera-Licona, R. Laubenbacher, T. Favre, Algorun, a docker-based packaging system for platform-agnostic implemented algorithms, preprint.

E. Boros, K. Elbassioni, K. Makino, Left-to-right multiplication for monotone boolean dualization, SIAM Journal on Computing 39 (7) (2010) 3424–3439. doi:10.1137/080734881.

K. Takata, A worst-case analysis of the sequential method to list the minimal hitting sets of a hypergraph, SIAM Journal on Discrete Mathematics 21 (4) (2007) 936–946. doi:10.1137/060653032.

R. Greiner, B. A. Smith, R. W. Wilkerson, A correction to the algorithm in Reiter’s theory of diagnosis, Artificial Intelligence 41 (1) (1989) 79–88. doi:10.1016/0004-3702(89)90079-9.

K. Murakami, T. Uno, Efficient algorithms for dualizing large-scale hypergraphs, Discrete Applied Mathematics 170 (2014) 83–94. doi:10.1016/j.dam.2014.01.012.

D. J. Kavvadias, E. C. Stavropoulos, An efficient algorithm for the transversal hypergraph generation., J. Graph Algorithms Appl. 9 (2) (2005) 239–264. doi:10.7155/jgaa.00107.

L. Khachiyan, E. Boros, K. Elbassioni, V. Gurvich, An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals and its application in joint generation, Discrete Applied Mathematics 154 (16) (2006) 2350–2372. doi:10.1016/j.dam.2006.04.012.

M. Hagen, P. Horatschek, M. Mundhenk, Experimental comparison of the two Fredman-Khachiyan-algorithms., in: ALENEX, SIAM, 2009, pp. 154–161. doi:10.1137/1.9781611972894.15.

C. E. Leiserson, M. Moreno Maza, L. Li, Y. Xie, Parallel computation of the minimal elements of a poset, in: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation, ACM, 2010, pp. 53–62. doi:10.1145/1837210.1837221.

R. E. Bryant, Graph-based algorithms for boolean function manipulation, Computers, IEEE Transactions on 100 (8) (1986) 677–691. doi:10.1109/TC.1986.1676819.

D. E. Knuth, Combinatorial Algorithms: Part 1, Vol. 4A of The Art of Computer Programming, Addison-Wesley, Boston, 2011.
[68] T. Toda, Hypergraph transversal computation with binary decision diagrams, in: 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013, Springer, 2013, pp. 91–102. doi:10.1007/978-3-642-38527-8_10.

[69] C. Hébert, A. Bretto, B. Créminilleux, A data mining formalization to improve hypergraph minimal transversal computation, Fundamenta Informaticae 80 (4) (2007) 415–434.

[70] K. M. Elbassioni, M. Hagen, I. Rauf, A lower bound for the hbc transversal hypergraph generation., Fundam. Inform. 130 (4) (2014) 409–414. doi:10.3233/FI-2014-997.

[71] K. M. Elbassioni, On the complexity of monotone dualization and generating minimal hypergraph transversals, Discrete Applied Mathematics 156 (11) (2008) 2109–2123. doi:10.1016/j.dam.2007.05.030.

[72] E. Miller, Alexander duality for monomial ideals and their resolutions, available from http://arxiv.org/abs/math/9812095 (1998). arXiv:9812095.

[73] D. R. Grayson, M. E. Stillman, Macaulay2, a software system for research in algebraic geometry, Available at http://www.math.uiuc.edu/Macaulay2/.

[74] E. Boros, K. Elbassioni, V. Gurvich, L. Khachiyan, K. Makino, Dual-bounded generating problems: All minimal integer solutions for a monotone system of linear inequalities, SIAM Journal on Computing 31 (5) (2002) 1624–1643. doi:10.1137/S0097539701388768.

[75] M. N. Jelassi, C. Larger, S. Ben Yahia, Concise representation of hypergraph minimal transversals: Approach and application on the dependency inference problem, in: Research Challenges in Information Science (RCIS), 2015 IEEE 9th International Conference on, IEEE, 2015, pp. 434–444.

[76] K. Geurts, G. Wets, T. Brijs, K. Vanhoof, Profiling of high-frequency accident locations by use of association rules, Transportation Research Record: Journal of the Transportation Research Board 1840 (2003) 123–130. doi:dx.doi.org/10.3141/1840-14.

[77] K. Murakami, T. Uno, Hypergraph dualization repository (2014). URL http://research.nii.ac.jp/~uno/dualization.html.

[78] A. Von Kamp, S. Schuster, Metatool 5.0: fast and flexible elementary modes analysis, Bioinformatics 22 (15) (2006) 1930–1931.

[79] R. Samaga, J. Saez-Rodriguez, L. G. Alexopoulos, P. K. Sorger, S. Klamt, The logic of EGFR/ErbB signaling: theoretical properties and analysis of high-throughput data, PLoS Comput Biol 5 (8) (2009) e1000438. doi:10.1371/journal.pcbi.1000438.