Lexicographical Enumeration of Hitting Sets in Hypergraphs

Lexikographische Aufzählung von Hitting Sets in Hypergraphen

Bachelor Thesis

to attain the scientific degree

Bachelor of Science

in

IT-Systems Engineering

handed in by **Julius Sebastian Lischeid**

Supervisor: Prof. Dr. Tobias Friedrich

Advisors: Dr. Thomas Bläsius,
Martin Schirneck

Chair for **Algorithm Engineering**

Potsdam, July 2, 2018

Abstract

In 2018, Bläsius et al. devised an algorithm that enumerates minimal hitting sets of ordered hypergraphs in lexicographical order. It has polynomial delay, provided the transversal hypergraph has a bounded rank k^*. We improve upon the enumeration algorithm by decreasing its worst-case delay bound by factor $|\mathcal{H}|$ to $\mathcal{O}(|\mathcal{H}|^{k^*+1}|V|^2)$. We then evaluate the algorithm in the context of unique column combination discovery and observe that the hypergraphs resulting from real-world problem instances are usually mucher smaller than the original database and that they can be enumerated quickly. Finally, we present a heuristic for finding a vertex order that reduces the total enumeration time in cases where the output order is irrelevant.

Zusammenfassung

Bläsius et al. präsentierten 2018 einen Algorithmus, der minimale Hitting Sets in Hypergraphen lexikographisch geordnet aufzählt. Für eine feste maximale Größe k^* der minimalen Hitting Sets ist die Verzögerung zwischen einzelnen Lösungsausgaben polynomiell in der Eingabegröße begrenzt. Wir entwickeln den Algorithmus weiter und reduzieren die Verzögerung um den Faktor $|\mathcal{H}|$ auf $\mathcal{O}(|\mathcal{H}|^{k^*+1}|V|^2)$. Anschließend wenden wir den Algorithmus an, um in Datenbanken Attributmengen zu finden, die Einträge eindeutig identifizieren. Wir stellen fest, dass die aus realen Datenbanken generierten Probleminstanzen meist deutlich kleiner als ihr Quelldatensatz sind, sodass der Algorithmus schnelle Laufzeiten aufweist. Weiterhin erarbeiten wir eine Heuristik für die Knotenreihenfolge, die die Laufzeit in Szenarien reduziert, in denen die Ausgabereihenfolge nicht relevant ist.

Contents

List of Figures

List of Tables

1 Introduction

Data profiling is the collection and computation of metadata of a given data set to facilitate data management, exploration and analytics. A reoccurring profiling task is the discovery of unique column combinations, sets of attributes that uniquely identify every single record. They find application in database management, data cleansing and database reverse engineering [2].

Unique column combination discovery has been shown to be equivalent to the transversal hypergraph generation problem [6], i. e., the computation of all minimal hitting sets of a hypergraph. While both problems are NP-hard in general [3, 6], real-world data is oftentimes strongly structured and shows very different characteristics than worst-case instances. In these cases, algorithms with exponential worst-case complexity often show tolerable running times.

Bläsius et al. [7] devise an algorithm specifically tailored to the requirements of interactive data profiling software. While maintaining a running time that is polynomial in the output size, it enumerates all minimal hitting sets in lexicographical order with a delay that is polynomial in the input size, provided that minimal hitting sets are bounded in cardinality. This means that the first results can be viewed quickly after start, beginning with the results deemed the most interesting or important. Both features are essential for systems designed around human experts who can apply domain knowledge to decide whether a result is just incidental or portrays valuable insight [7].

Contribution and Outline. This paper is concerned with the applicability of the lexicographical enumeration algorithm by Bläsius et al. [7] to the unique column combination discovery problem. We introduce hypergraphs, the transversal problem and its data profiling sibling, unique column combination discovery, in Section 2. After examining the enumeration algorithm in Section 3, we present some algorithm enhancements in Section 4 and show that these improvements reduce the polynomial delay bound by factor $|\mathcal{H}|$ to $\mathcal{O}(|\mathcal{H}|^{k^*+1}|V|^2)$, where k^* is the size of the largest minimal hitting set. In Section 5, we evaluate the algorithm in the context of unique column combination discovery and find out that real-world datasets result in hypergraphs that can be enumerated quickly. We compare the original algorithm version to its successor and verify that the theoretical improvements translate to a significant speedup in practice. Additionally, we investigate how the vertex or attribute order can be used to improve the running time and present a heuristic for finding an attribute order that reduces the enumeration time when the output order is irrelevant. Finally, we examine the algorithm-specific extension oracle and observe that its average running time is far lower than its exponential worst-case complexity would initially suggest.

Prior Work. Transversal hypergraphs have appeared in various shapes in theoretical computer science. The generation of a transversal hypergraph is, for example, equivalent to the dualization of a monotone Boolean function [9]. Transversal hypergraph recognition, i. e., verifying that a given hypergraph is the transversal hypergraph of a second one, also appears under the name MONET and is covered extensively by Hagen [12]. All in all, the problem has been approached from multiple angles over the last three decades. Fredman and Kachiyan [10] devise an algorithm that generates the transversal hypergraph with a quasi-polynomial running time complexity in input and output. Eiter and Gottlob [8] prove that for a bounded transversal hypergraph rank, recognition is possible in polynomial time. In combination with this result, the equivalence found by Bioch and Ibaraki [5] implies the existence of an incremental-polynomial algorithm for generating such a transversal hypergraph. Bläsius, Friedrich, Meeks and Schirneck [7] extend this result and show that generation is possible with polynomial delay, even under the constraint that the output has to be ordered lexicographically.

2 Preliminaries

We formally introduce hypergraphs and reduce the task of finding unique column combinations to the hypergraph transversal problem. We further consider running time implications of the problem at hand.

2.1 Ordered Hypergraphs and Hitting Sets

Hypergraphs are generalized graphs whose edges can contain an arbitrary number of vertices instead of exactly two. Figure 2.1 shows a hypergraph that will be used as an example throughout the next sections. Hypergraphs are widely studied [4, 18] and find application in data mining [11], natural language processing [14] or machine learning [19].

In formal terms, a *hypergraph* $(V, \mathcal{H})$ consists of a set of *vertices* V and a set of *(hyper-)edges* $\mathcal{H} \subseteq \mathcal{P}(V)$ with cardinalities $n = |V|$ and $m = |\mathcal{H}|$. It may be identified by its set of edges $\mathcal{H}$. Its *rank* is the size of its largest edge $\max\{|E| \mid E \in \mathcal{H}\}$. A *hitting set* or *transversal* is a set of vertices $H \subseteq V$ with $H \cap E \neq \emptyset$ for all $E \in \mathcal{H}$. It can be understood as a hyperedge that intersects all edges in $\mathcal{H}$. We call a hitting set *(inclusion-wise) minimal* if it does not include any other hitting set. The minimal hitting sets of $\mathcal{H}$ form the edges of the *transversal hypergraph* of $(V, \mathcal{H})$ on the vertex set V.

An *ordered hypergraph* $(V, \succcurlyeq, \mathcal{H})$ additionally includes a total ordering $\succcurlyeq$ of the vertex set V that induces a lexicographical order on $\mathcal{P}(V)$. A subset $S \subseteq V$ is *lexicographically smaller* than $R \subseteq V$ if and only if the $\succcurlyeq$-first element in which they differ is in S.

2.2 Unique Column Combinations

A *unique column combination (UCC)* is a set of attributes of a relational dataset whose projection contains no duplicate entry. Therefore, the values of those attributes uniquely identify all records in the dataset. In practice, unique

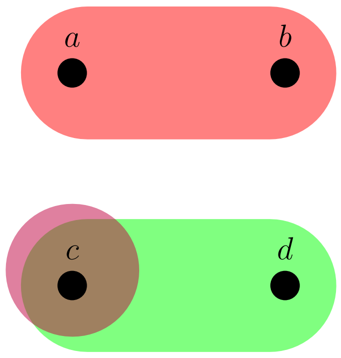


Figure 2.1: Hypergraph $(V, \mathcal{H})$ with $V = \{a, b, c, d\}$ and $\mathcal{H} = \{\{a, b\}, \{c\}, \{c, d\}\}$.

column combinations are used for various data profiling tasks like schema normalization, data cleansing, query optimization, duplicate and anomaly detection or schema reverse engineering [2, 13, 17].

Analogous to the definition of a minimal hitting set, a UCC is called *minimal* if it does not contain any other UCC. Most real-world applications will require only the knowledge of all minimal UCCs from which further non-minimal UCCs can be derived easily [17]. The number of minimal UCCs of a dataset can be exponential in the dataset size [1].

Finding UCCs in a given dataset can be expressed as a hitting set enumeration problem on a generated hypergraph [7]. Each attribute is represented by a vertex. Every pair of non-identical database records is translated into a hyperedge consisting of exactly those vertices that correspond to the attributes where the records differ in value. The record values of any single attribute encoded by the edge allow to distinguish between those records. A hitting set shares at least one vertex with any edge. Therefore, all hitting sets include at least one vertex that represents an attribute that allows to distinguish between any two non-identical records. This makes the attributes corresponding to a hitting set a unique column combination. Likewise, a minimal hitting set corresponds to a minimal UCC.

Some dataset attributes may be especially interesting or relevant to data analysts, e. g., those that are suspected to form a primary key. An algorithm enumerating minimal hitting sets in lexicographical order can be configured to prioritize those attributes by ordering vertices from most to least important. Consequently, all minimal UCCs containing prioritized attributes will be found and returned before all other UCCs.

2.3 Hitting Set Enumeration Complexity

Similar to UCCs, the number of minimal hitting sets of a hypergraph can be exponential in both $|V|$ and $|\mathcal{H}|$ [5]. Therefore, no *input-polynomial* algorithm (an algorithm with a running time that is polynomial in input size) outputting all minimal hitting sets of arbitrary hypergraphs can exist. However, running time complexity can be measured in both input and output size. An algorithm is *output-polynomial* if it has a running time polynomial in both.

Some algorithms do not output a single solution at the end (e. g., a complete transversal hypergraph) but instead emit solution parts throughout their running time (e. g., single transversals). On these algorithms, more fine-grained running time complexity evaluations are possible. An algorithm has *polynomial delay* if it has an input-polynomial upper bound between two consecutive outputs.

3 Lexicographical Minimal Hitting Set Enumeration

Bläsius et al. [7] devise an algorithm that finds all minimal hitting sets with polynomial delay for transversal hypergraphs with a bounded rank. It enumerates minimal hitting sets in lexicographical order.

The algorithm is divided into two parts. The actual enumeration algorithm ENUMORIG traverses an enumeration tree and uses the extension oracle EXTENDORIG to prune branches not containing any minimal hitting sets.

3.1 Extension Oracle

The original extension oracle EXTENDORIG (Algorithm 3.1), given a hypergraph $(V, \mathcal{H})$ and two disjoint sets $X, Y \subseteq V$, outputs TRUE if and only if there exists a minimal hitting set $X \subseteq H \subseteq V \setminus Y$ of $\mathcal{H}$. It makes heavy use of the observation that a hitting set H of a hypergraph $(V, \mathcal{H})$ is minimal if and only if for every $x \in H$, there is an edge $E_x \in \mathcal{H}$ (called *witness* for x) such that $E_x \cap H = \{x\}$ [4, 15].

The underlying concept of the algorithm is to determine a set of potential witnesses $\mathcal{S}_x$ for all $x \in X$ with $\mathcal{S}_x = \{E \setminus Y \mid E \cap X = \{x\}, E \in \mathcal{H}\}$. To extend X to a minimal hitting set, vertices from $V \setminus (X \cup Y)$ have to be added to X until all edges are hit. Adding a vertex v to X eliminates all potential witnesses that contain v. As long as for all x, at least one witness remains after adding enough vertices to hit all edges, an extension to a minimal hitting set is possible. The oracle only determines the possibility of such an extension and does not compute minimal hitting sets itself.

We exemplify the algorithm on the hypergraph $(V, \mathcal{H})$ from Figure 2.1 for $X = \{a\}$ and $Y = \{b\}$. At first, if X is empty, the oracle returns whether $V \setminus Y$ is a hitting set for $\mathcal{H}$ (the existence of any hitting set implies the existence of a minimal hitting set). If X is not empty, the edges of $\mathcal{H}$ are divided into three different categories: Edges that are hit by at least two different vertices from X (those cannot be witnesses), edges hit by exactly one vertex x from X (potential witnesses, collected into $\mathcal{S}_x$, $\mathcal{S}_x \in \mathcal{S}$) and edges not hit by any vertex from x (collected into $\mathcal{T}$). All vertices from Y are removed from these edges. After this step in our example, $\mathcal{S}_a$ is the only set system in $\mathcal{S}$ and contains only $\{a\}$ while $\mathcal{T}$ contains $\{c\}$ and $\{c, d\}$.

If any $x \in X$ has no potential witness, X cannot be extended to a minimal hitting set and the algorithm returns FALSE in line 10.

Following that, every $x \in X$ has a potential witness. If $\mathcal{T}$ is empty, all edges are hit by X, making X a minimal hitting set for $\mathcal{H}$ and EXTENDORIG returns TRUE.

If $\mathcal{T}$ is not empty, there exist edges in $\mathcal{H}$ that are not hit by X. If and only if X is extendable to a minimal hitting set without using vertices from Y, there exists a $Z \subseteq V \setminus (X \cup Y)$ that hits all edges in $\mathcal{T}$ and has an empty intersection with at least one potential witness for any $x \in X$. This ensures that after the addition of a vertex from Z to X, at least one witness remains for all $x \in X$.

The algorithm tests all possible combinations of these remaining witnesses using brute force to find a combination $c \in \mathcal{S}_{x_1} \times \cdots \times \mathcal{S}_{x_{|X|}}$ for which all $T \in \mathcal{T}$ include a vertex that is not included in any witness in c. These vertices make up Z. If no such combination is found, the algorithm returns FALSE. In our example, $\{a\}$ is the only potential witness that has to be checked. Since both $\{c\}$ and $\{c, d\}$ contain vertices that are not in $\{a\}$, the algorithm returns TRUE.

Assuming that all set operations (membership, product, union, intersection and difference) can be done in time proportional to the set size, testing whether $V \setminus Y$ is a hitting set and generating $\mathcal{S}$ and $\mathcal{T}$ can be done in $\mathcal{O}(mn)$. As there are at most $(m/|X|)^{|X|}$ tuples in the brute force loop for which operations in $\mathcal{O}(mn)$ have to be computed to check whether all $T \in \mathcal{T}$ include vertices not included in the remaining witness combination, the total oracle running time complexity is in $\mathcal{O}((m/|X|)^{|X|}mn)$ [7].

As all set systems maintained by the algorithm are disjoint subsets of $\mathcal{H}$, it only requires space linear in m [7].

Algorithm 3.1: Algorithm for EXTENDORIG by Bläsius et al. [7].

```
Data: Hypergraph $(V, \mathcal{H})$
Input: disjoint sets $X, Y \subseteq V$, with $X = \{x_1, x_2, \dots, x_{|X|}\}$.
Output: TRUE iff there is a minimal hitting set $X \subseteq H \subseteq V \setminus Y$ for $\mathcal{H}$
1  Procedure extendOrig($X$,$Y$):
2  if $X = \emptyset$ then
3      if $V \setminus Y$ is a hitting set for $\mathcal{H}$ then return TRUE;
4      else return FALSE;
5  initialize set system $\mathcal{T} = \emptyset$;
6  foreach $x \in X$ do initialize set system $\mathcal{S}_x = \emptyset$;
7  foreach $E \in \mathcal{H}$ do
8      if $E \cap X = \{x\}$ then add $E \setminus Y$ to $\mathcal{S}_x$;
9      if $E \cap X = \emptyset$ then add $E \setminus Y$ to $\mathcal{T}$;
10 if $\exists x \in X : \mathcal{S}_x = \emptyset$ then return FALSE;
11 if $\mathcal{T} = \emptyset$ then return TRUE;
12 foreach $(E_{x_1}, \dots, E_{x_{|X|}}) \in \mathcal{S}_{x_1} \times \dots \times \mathcal{S}_{x_{|X|}}$ do
13     $W \leftarrow \bigcup_{i=1}^{|X|} E_{x_i}$;
14     if $\forall T \in \mathcal{T} : T \not\subseteq W$ then return TRUE;
15 return FALSE;
```

3.2 Enumeration Algorithm

Minimal hitting sets are enumerated by the ENUMORIG procedure in lexicographical order [7]. The algorithm traverses a binary tree depth-first. Each level corresponds to a vertex, starting with the $\succcurlyeq$-first one just below the root down to the last one at leaf level. Each node is characterized by a pair (X, Y), where X includes all vertices that are to be included in a minimal hitting set, while Y includes all vertices that are to be left out. The algorithm starts with $(\emptyset, \emptyset)$ at the root. For each node besides the root, (X, Y) can be assumed to be extendable. Now the next smallest vertex v from the level below is taken into consideration. Using the extension oracle, the algorithm checks whether $(X \cup \{v\}, Y)$ is extendable and descends further down to that node if the oracle returns TRUE. After returning from that branch, the procedure is repeated for $(X, Y \cup \{v\})$. When the tree traversal reaches the bottom, all vertices have been added to either X or Y and (X, Y) is extendable. Thus, X has to be a minimal hitting set and is added to the output [7]. Figure 3.1 shows the complete enumeration tree for the example hypergraph from Figure 2.1.

Lexicographical order is ensured by the pre-order traversal strategy. Let $v \in X_1$ be the first vertex that X_1 and X_2 with $X_2 \nsubseteq X_1$ differ in. Then X_1 is located left of X_2 in the enumeration tree since at their lowest common ancestor node, the algorithm will try to add v to X first before checking the extendability of $(X, Y \cup \{v\})$. Due to pre-order traversal, if X_1 and X_2 are minimal hitting sets, X_1 will be added to the output first [7].

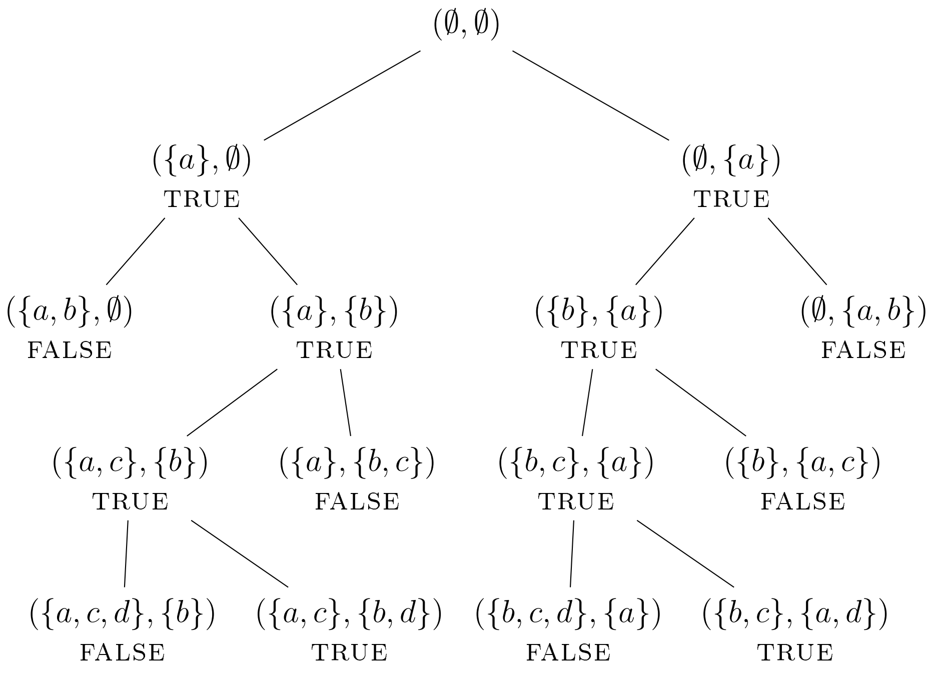
The enumeration algorithm has polynomial delay for a bounded maximum minimal hitting set size. Since each tree level corresponds to a vertex, the tree has height $|V| = n$. Nodes can be entered from above after at most two oracle calls in the parent or from a child below without any oracle call when backtracking. Only branches leading down to a leaf (which holds a solution) are followed. Traversing from one leaf to the next thus requires zero oracle calls when traversing upwards and at most $\mathcal{O}(n)$ oracle calls when traversing downwards. In the process, at most $\mathcal{O}(n)$ nodes are visited.

The argument (X, Y) for an oracle call will always be subject to $|X| \leq k^* + 1$ with k^* being the rank of the transversal hypergraph. The reason is that any call with $|X| = k^* + 1$ will return FALSE, preventing the enumeration algorithm from traversing further down the branch. Thus, with the maximum running time of a single oracle call being $\mathcal{O}(m^{|X|+1}n)$, output delay is at most $\mathcal{O}(n) \cdot \mathcal{O}(m^{(k^*+1)+1}n) \subseteq \mathcal{O}(m^{k^*+2}n^2)$ [7].

The algorithm only requires knowledge of the current node while traversing the enumeration tree. All data generated by an extension oracle call is discarded after the check. Consequently, the enumeration algorithm uses input-linear space [7].

Algorithm 3.2: Recursive enumeration algorithm ENUMORIG by Bläsius et al. [7]. Initial call: `enumOrig`($\emptyset$, $\emptyset$, V).

```
Data: Ordered hypergraph $(V, \succcurlyeq, \mathcal{H})$ with $\mathcal{H} \neq \emptyset$.
Input: Partition $(X, Y, R)$ of the vertex set $V$.
1 Procedure enumOrig($X$,$Y$,$R$):
2 if $R = \emptyset$ then output $X$; return;
3 $v \leftarrow \min_{\succcurlyeq} R$;
4 if extendOrig($X \cup \{v\}$, $Y$) then enumOrig($X \cup \{v\}$, $Y$, $R \setminus \{v\}$);
5 if extendOrig($X$, $Y \cup \{v\}$) then enumOrig($X$, $Y \cup \{v\}$, $R \setminus \{v\}$);
```



$(\emptyset, \emptyset)$
$(\{a\}, \emptyset)$ TRUE
$(\emptyset, \{a\})$ TRUE
$(\{a,b\}, \emptyset)$ FALSE
$(\{a\}, \{b\})$ TRUE
$(\{b\}, \{a\})$ TRUE
$(\emptyset, \{a,b\})$ FALSE
$(\{a,c\}, \{b\})$ TRUE
$(\{a\}, \{b,c\})$ FALSE
$(\{b,c\}, \{a\})$ TRUE
$(\{b\}, \{a,c\})$ FALSE
$(\{a,c,d\}, \{b\})$ FALSE
$(\{a,c\}, \{b,d\})$ TRUE
$(\{b,c,d\}, \{a\})$ FALSE
$(\{b,c\}, \{a,d\})$ TRUE

Figure 3.1: Tree traversal of ENUMORIG on the hypergraph from Figure 2.1. Displayed per node are the arguments to EXTENDORIG and the resulting output. The algorithm will visit a node and traverse further down its branch after its corresponding oracle call returned TRUE.

4 Faster Enumeration

The enumeration algorithm can be made more efficient if the extension oracle returns additional information that is used accordingly. We decrease the output delay bound by factor $|\mathcal{H}|$ to $\mathcal{O}(|\mathcal{H}|^{k^*+1}|V|^2)$, where k^* is the rank of the transversal hypergraph.

4.1 Extension Oracle Return Value

The extendOrig algorithm returns true for all X that are extendable to a minimal hitting set without using vertices from Y, even if the necessary extension is empty, i. e., X is a minimal hitting set in itself. As described, if $\mathcal{T}$ is found to be empty in line 11 of extendOrig, X has to be a minimal hitting set. Thus, the improved extension oracle extendImpr (Algorithm 4.1) returns minimal in this case. If there exists a minimal hitting set $X \subset H \subseteq V \setminus Y$ for $\mathcal{H}$, the oracle returns extendable. If no minimal hitting set $X \subseteq H \subseteq V \setminus Y$ exists, the oracle returns notExtendable.

Lemma 1. Algorithm 4.1 returns minimal if and only if its first input argument X is a minimal hitting set for its hypergraph $(V, \mathcal{H})$ with $\mathcal{H} \neq \emptyset$.

Proof. First, we show that if X is a minimal hitting set for $\mathcal{H}$, the extension oracle returns minimal. Let X be such a minimal hitting set. Since $\mathcal{H} \neq \emptyset$, X cannot be empty. Hence, the algorithm does not enter the branch of line 2. Since X is a minimal hitting set, every $x \in X$ has a witness so that $\mathcal{S}_x \neq \emptyset$ for all $x \in X$. Thus, the algorithm does not return in line 10. Because X is a hitting set, no edge in $\mathcal{H}$ has an empty intersection with X. Consequently, $\mathcal{T}$ is empty and the algorithm returns minimal.

Second, we show that if extendImpr returns minimal, its first argument X is a minimal hitting set for $\mathcal{H}$. The oracle will only return minimal if $\mathcal{T}$ is found to be empty, meaning that X is a hitting set for $\mathcal{H}$. We prove by contradiction that X is indeed minimal. Assume X is a non-minimal hitting set. Then there exists an $x \in X$ that has no witness in $\mathcal{H}$. In this case, EXTENDIMPR must have returned in line 10 since $\mathcal{S}_x$ must have been empty. This contradiction can only be resolved by dropping the initial assumption that X is non-minimal. □

Algorithm 4.1: Algorithm for EXTENDIMPR.

```
   Data: Hypergraph $(V, \mathcal{H})$, $\mathcal{H} \neq \emptyset$
   Input: disjoint sets $X, Y \subseteq V$, with $X = \{x_1, x_2, \dots, x_{|X|}\}$.
   Output: MINIMAL iff $X$ is a minimal hitting set for $\mathcal{H}$,
           EXTENDABLE iff there is a minimal hitting set
           $X \subset H \subseteq V \setminus Y$ for $\mathcal{H}$,
           NOTEXTENDABLE otherwise.
1  Procedure extendImpr($X$,$Y$):
2  if $X = \emptyset$ then
3      if $V \setminus Y$ is a hitting set for $\mathcal{H}$ then return EXTENDABLE;
4      else return NOTEXTENDABLE;
5  initialize set system $\mathcal{T} = \emptyset$;
6  foreach $x \in X$ do initialize set system $\mathcal{S}_x = \emptyset$;
7  foreach $E \in \mathcal{H}$ do
8      if $E \cap X = \{x\}$ then add $E \setminus Y$ to $\mathcal{S}_x$;
9      if $E \cap X = \emptyset$ then add $E \setminus Y$ to $\mathcal{T}$;
10 if $\exists x \in X : \mathcal{S}_x = \emptyset$ then return NOTEXTENDABLE;
11 if $\mathcal{T} = \emptyset$ then return MINIMAL;
12 foreach $(E_{x_1}, \dots, E_{x_{|X|}}) \in \mathcal{S}_{x_1} \times \dots \times \mathcal{S}_{x_{|X|}}$ do
13     $W \leftarrow \bigcup_{i=1}^{|X|} E_{x_i}$;
14     if $\forall T \in \mathcal{T} : T \nsubseteq W$ then return EXTENDABLE;
15 return NOTEXTENDABLE;
```

4.2 Enumeration Algorithm Improvements

While ENUMORIG always traverses to the bottom of the enumeration tree to output a solution, the improved enumeration procedure ENUMIMPR (Algorithm 4.2) makes use of the improved extension oracle to output results as soon as X grows to minimal hitting set. The underlying branch can only lead to a single leaf containing the exact same X while all vertices below will be added to Y. Since the results of all enumeration and oracle calls below are known at this point, the branch is pruned and redundant computation is avoided.

The second enhancement is based on the assumption that for any call of ENUMIMPR, the tuple of the first two input arguments (X, Y) has to be EXTENDABLE. This holds true as the enumeration algorithm will only follow down paths that lead to a minimal hitting set in the enumeration tree. The root $(\emptyset, \emptyset)$ is EXTENDABLE since the hypergraph does not include any empty edges. Given that (X, Y) is EXTENDABLE, if $(X \cup \{v\}, Y)$ is NOTEXTENDABLE, the other branch, $(X, Y \cup \{v\})$, has to be EXTENDABLE. Figure 4.1 shows the tree traversal of ENUMIMPR on the exemplary hypergraph. Note that the algorithm never proceeds to the level of vertex d. Also, the result of EXTENDIMPR for $(\{a\}, \{b\})$ does not have to be computed after $(\{a, b\}, \emptyset)$ turns out to be NOTEXTENDABLE.

The first enhancement further improves the asymptotic output delay over ENUMORIG.

Theorem 1. The delay of Algorithm 4.2 is $\mathcal{O}(m^{k^*+1}n^2)$.

Proof. Let k^* be the rank of the transversal hypergraph. When ENUMIMPR calls EXTENDIMPR with first parameter X, $|X|$ is at most k^* instead of k^*+1. We prove this by contradiction. Assume that EXTENDIMPR is called with $|X| > k^*$. This means that ENUMIMPR had to be called with an X' with $|X'| \geq k^*$ beforehand. This could only have happened if EXTENDIMPR returned EXTENDABLE for X', a contradiction since a vertex set of size greater than or equal to k^* has to be either MINIMAL or NOTEXTENDABLE.

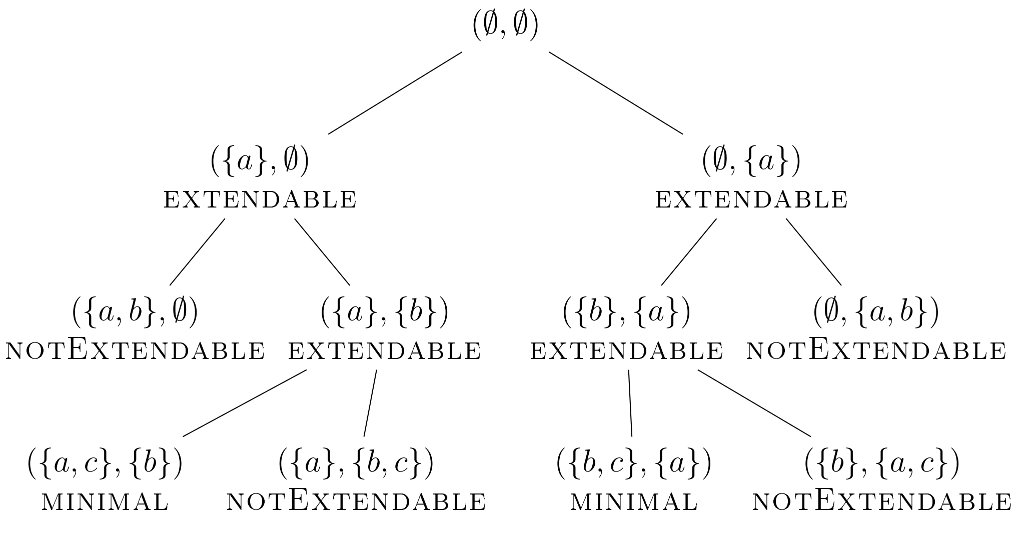
Consequently, a single oracle call has a running time complexity of at most $\mathcal{O}(m^{k^*+1}n)$, resulting in a delay of at most $\mathcal{O}(m^{k^*+1}n^2)$ between outputs, analogous to the $\mathcal{O}(m^{k^*+2}n^2)$ delay of the original algorithm. □

Algorithm 4.2: Improved procedure ENUMIMPR for the transversal hypergraph problem. Initial call: `enumImpr(`$\emptyset$, $\emptyset$, V`)`.

Data: Ordered hypergraph $(V, \succcurlyeq, \mathcal{H})$ with $\mathcal{H} \neq \emptyset$, $\emptyset \notin \mathcal{H}$.

Input: Partition (X, Y, R) of the vertex set V.

```
1  Procedure enumImpr($X$,$Y$,$R$):
2  $v \leftarrow \min_{\succcurlyeq} R$;
3  $left \leftarrow$extendImpr($X \cup \{v\}, Y$);
4  if $left$ = MINIMAL then
5      output $X$;
6  else if $left$ = EXTENDABLE then
7      enumImpr($X \cup \{v\}$, $Y$, $R \setminus \{v\}$);
8  else
9      enumImpr( $X$, $Y \cup \{v\}$, $R \setminus \{v\}$);
10     return;
11 if extendImpr($X$, $Y \cup \{v\}$) = EXTENDABLE then
12     enumImpr($X$, $Y \cup \{v\}$, $R \setminus \{v\}$);
```



$(\emptyset, \emptyset)$
$(\{a\}, \emptyset)$ EXTENDABLE
$(\emptyset, \{a\})$ EXTENDABLE
$(\{a,b\}, \emptyset)$ NOTEXTENDABLE
$(\{a\}, \{b\})$ EXTENDABLE
$(\{b\}, \{a\})$ EXTENDABLE
$(\emptyset, \{a,b\})$ NOTEXTENDABLE
$(\{a,c\}, \{b\})$ MINIMAL
$(\{a\}, \{b,c\})$ NOTEXTENDABLE
$(\{b,c\}, \{a\})$ MINIMAL
$(\{b\}, \{a,c\})$ NOTEXTENDABLE

Figure 4.1: Tree traversal of ENUMIMPR on the hypergraph from Figure 2.1. Displayed per node are the arguments to EXTENDIMPR and the resulting output. The algorithm will visit a node and traverse further down its branch after its corresponding oracle call returned EXTENDABLE.

5 Experimental Results

We use the enumImpr algorithm to find UCCs in real-world datasets. The procedure can be divided into two separate stages: At first, we generate hypergraphs by pairwise record comparison (*generation stage*). Subsequently, we enumerate all minimal hitting sets in lexicographical order using enumImpr (*enumeration stage*).

The main generation stage research topic is the size of hypergraphs generated from real-world data. Our results indicate that a larger number of records does not increase the number of hyperedges after a certain point. To the contrary, adding records to the input set can even decrease the number of hyperedges when the hypergraph is minimized.

For the enumeration stage, we examine the applicability of the algorithm in general and show that on our real-world problem instances, the output delay and the total enumeration time are small. We investigate the magnitude of our performance improvements over enumOrig and find significant running time advantages. Although the lexicographical output order is a unique selling point of the algorithm, an ordered output may not be required in some scenarios. We observe that different attribute (i. e., vertex) orders lead to different enumeration times. We analyze causes of this observation and present a heuristic for finding an attribute order that reduces the enumeration time. This order can be used when the output order is irrelevant. At last, we inspect the behavior of the extension oracle and find out that its running time is highly variable but small on average. We work out the remaining witness combination order in the oracle brute force loop as a topic of future research.

5.1 Implementation Details and Data

The following results are based on an implementation written in C++ running on a Windows 10 system using an Intel Xeon E3-1231 v3 processor. To allow for $\mathcal{O}(1)$ insertion, deletion, union, intersection and difference, vertex sets (i. e., hyperedges) were implemented as bit arrays of length 64. To ensure correctness, results were compared to those of a brute force algorithm. The implementation was tested on publicly available voter data (`ncvoter-temporal` with 19 columns, `ncvoter-allc` with 64 columns)[1] and randomly generated data (`fd-reduced-30` with 30 columns [16]). We restricted all datasets to the first 2^{15} records of each file. If no further information is given, hypergraphs were generated from all those 2^{15} records with attributes ordered from the column containing the most unique values to the column containing the least (*descending uniqueness*). The reverse attribute order, *ascending uniqueness*, is also contained in some benchmarks. Additionally, records were sorted lexicographically after sorting the column order.

5.2 Hypergraph Generation

As described earlier, database records are compared pairwise. Every pair forms an edge that consists of the attributes that the records differ in. This results in a number of edges that is quadratic in the number of records. Apart from eliminating duplicate edges, this number can be greatly reduced by only keeping inclusion-wise minimal edges in the generated set. For two edges X, Y with $X \subseteq Y$, any vertex set intersecting X will also intersect Y. Hence, Y can be discarded during the generation stage without affecting the transversal hypergraph.

5.2.1 Number of Generated Edges

Our results on real-world data show that UCC discovery hypergraphs tend to be much smaller than the dataset they were generated from. Furthermore, their size does not increase with the number of records or even decreases after a certain point (Figure 5.1). Given that real-world data is often strongly structured, we expect records to be similar in what they differ in. This helps to explain why graph size does not grow with record number. Additionally, with a higher number of records, chances increase that there exist pairs of records that differ only in a small number of attributes. The resulting edges are small, leading to a great number of supersets to be omitted. While the second mechanism also has an effect on hypergraphs for randomly generated datasets, the first one does not apply. This could be the reason why the hypergraph for `fd-reduced-30` grows with the record number where the `ncvoter` datasets show the opposite trend.

1 https://www.ncsbe.gov/ncsbe/data-statistics

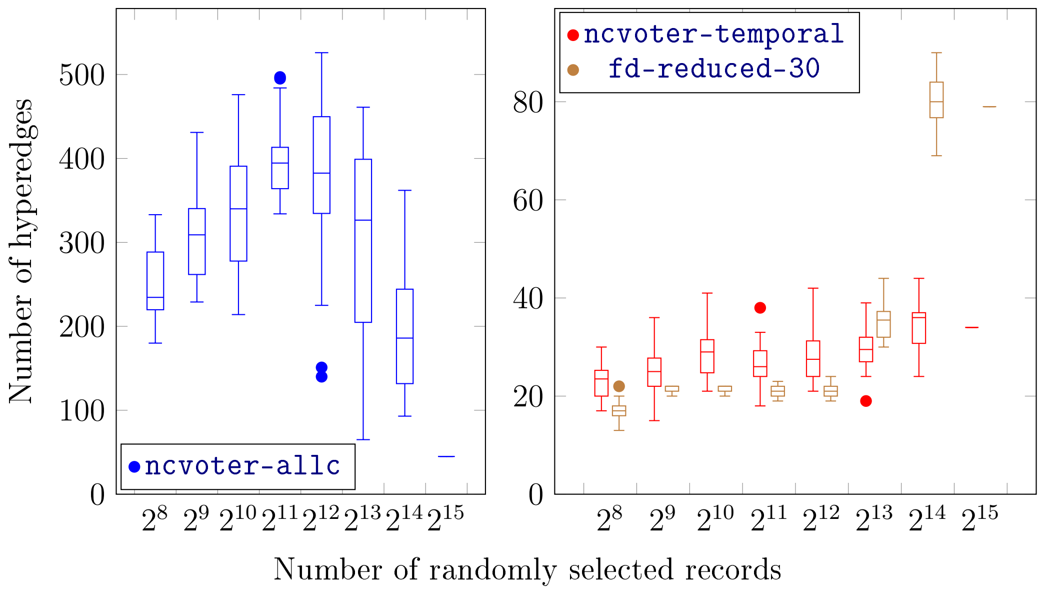


Figure 5.1: Size of the generated UCC hypergraph by number of records for `ncvoter-allc` (64 columns), `ncvoter-temporal` (19 columns) and `fd-reduced-30` (30 columns). Records were randomly sampled from a dataset of size 2^{15} with 20 samples per box plot. All values above 0.75-quantile$+1.5\cdot(0.75$-quantile-0.25-quantile$)$ or below 0.25-quantile$-1.5\cdot(0.75$-quantile-0.25-quantile$)$ are marked as outliers.

One can observe a great variance in the number of edges produced for any fixed number of randomly selected records. We argue that the number of edges is lower whenever pairs of similar records were randomly selected, generating small edges that lead to the omission of a great number of supersets. To test this hypothesis, we used different sampling strategies to generate a hypergraph for `ncvoter-allc` (Figure 5.2). The dataset was sorted by descending or ascending uniqueness at first, then sorted lexicographically. On sorted datasets, similar records tend to be close to each other. For the first 2^{15} records of the dataset sorted by descending uniqueness, out of the 22 pairs of records that differ in seven or less attributes, 14 are direct neighbors (and an additional three pairs have distance two). For ascending uniqueness, where primary keys do not dominate the lexicographical order, 15 are direct neighbors (and again three pairs have distance two). Hypergraphs generated from n consecutive records are therefore significantly smaller than hypergraphs generated from n records with maximal distance from each other or a random sample. Note that for Figure 5.2, records were sampled from a dataset of size 2^{15}, drastically increasing the likelihood that records close to each other are sampled randomly with sample size, while equidistant sampling never selects direct neighbors. This explains why the random sampling curve drops sharper than the equidistant descending uniqueness curve. Additionally, since the ascending uniqueness order places similar records even closer to each other than ordering by descending uniqueness, sampling consecutive ascending records mostly results in even fewer edges than sampling descending ones. The same property causes equidistant ascending uniqueness ordered records to be even more dissimilar than their descending counterparts. Thus, equidistant ascending sampling results in the highest number of hyperedges of all sampling strategies.

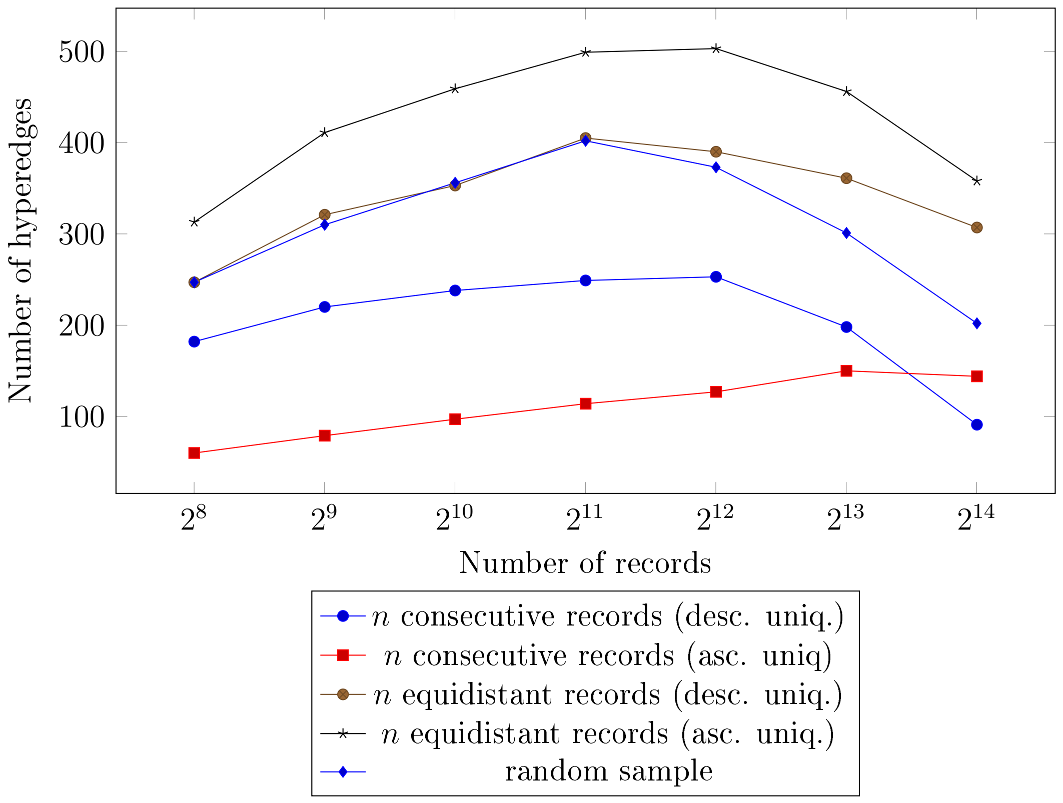


Figure 5.2: Size of the UCC hypergraph by number of records and record selection strategy for 2^{15} ordered rows of `ncvoter-allc` (64 columns). Random data points are an average of 20 samples. Data points for n consecutive or equidistant records display an average of $2^{15}/n$ mutually exclusive samples.

5.2.2 Number of Edges During Generation

Only inclusion-wise minimal edges are kept during hypergraph generation. Every edge that is generated by comparing two records has to be tested for inclusion of any other edge currently in the working set (the new edge is discarded) and inclusion in any other edge (the other edge is discarded). The most simple way to achieve this is to compare the new edge pairwise with all edges in the working set. This approach has a $\mathcal{O}(kn^2)$ running time, where n is the number of records and k is the maximum number of edges that are in the current working set at any given time. In our experiments, k was never more than 5 times larger than the number of edges of the final hypergraph. The row order did not appear to have a significant effect on both k and on the running time. As a result, this trivial approach already yields tolerable running times when all pairs of records are to be compared. Obviously, a quadratic running time is not feasible on larger datasets. Instead, strategic sampling could be used to generate UCC candidates that are verified against the data. However, this is outside the scope of this paper. For details, we direct the reader to Papenbrock and Naumann [17] who use this approach to discover UCCs in datasets, albeit without the explicit use of hypergraphs.

5.3 Lexicographical Enumeration

The second part of the implementation enumerates all minimal UCCs in lexicographical order using ENUMIMPR (Algorithm 4.2). The algorithm was tested on hypergraphs generated from 2^{15} records. Since the hypergraph size does not necessarily grow with the number of records, we expect our results to be applicable to hypergraphs based on larger datasets as well.

On the 64-column `ncvoter-allc` dataset, the total enumeration time is less than 30s for a descending uniqueness ordered hypergraph. The output delay is under 16ms for 99% of all hitting sets, making the algorithm suitable for use in interactive systems designed for human experts. On `ncvoter-temporal` and `fd-reduced-30` , the same attribute order leads to an enumeration time of 0.2s and 1.1s, respectively. Note that these numbers, as all running time measurements in this section, do not include the time necessary for the hypergraph generation.

In the following subsections, we argue why some attributes may be omitted from the hypergraph before enumeration without changing the result, compare the performance of ENUMIMPR and ENUMORIG, present a heuristic for finding an attribute order that reduces the enumeration time and evaluate the running time behavior of the extension oracle.

5.3.1 Eliminating Attributes

For some datasets, the generated hypergraph contains vertices that are not included in any edge due to edge minimization. For example, two addresses differing in the attribute *state* may always differ in the attribute *zip code*. Edges that include both vertices may be eliminated by edges containing only the *zip code* vertex. Since no pair of records differing in *state* but not in *zip code* can exist, this leads to the *state* vertex not being included in any hyperedge. This vertex cannot be included in any minimal hitting set, either. It can therefore be excluded from the graph to improve the enumeration time.

However, a priori knowledge of functional dependencies where attribute A implies attribute B does not allow for removal of B from the dataset before hypergraph generation. It is still possible that another set of attributes $\mathcal{A}$ with $A \notin \mathcal{A}$ forms a UCC together with B.

The `ncvoter-allc` UCC hypergraph contains four vertices that are not included in any edge. Removing these vertices from the graph leads to a modest average speed increase of 3% across 100 random attribute orders. However, this result is highly specific to this dataset. Both `ncvoter-temporal` and `fd-reduced-30` do not contain unused vertices, other datasets may profit a lot more from eliminated attributes. All following results are based on hypergraphs that do not have this optimization applied to them.

5.3.2 enumImpr vs. enumOrig

In Chapter 4, we showed that theoretical improvements over enumOrig can be made and introduced enumImpr. These enhancements show performance increases on all benchmarked datasets, enumImpr outperforms enumOrig on every single tested hypergraph. The average speedup across 100 random attribute orders was 1.8, 1.3 and 3.6 for `ncvoter-allc` , `ncvoter-temporal` and `fd-reduced-30` , respectively (Table 5.1). While these numbers are significant in practice, the improvements are not as strong as the theoretical worst-case delay factor of $|\mathcal{H}|$ would suggest. A closer look shows that the average number of oracle calls decreased by factor 2.2, 1.6 and 5.9. This means that the improvements disproportionately removed cheap oracle calls with far lower than worst-case running time.

dataset	`ncvoter-allc`	`ncvoter-temporal`	`fd-reduced-30`
minimal speedup	1.5	1.1	3.2
average speedup	1.8	1.3	3.6
maximal speedup	2.5	1.6	4.0
oracle call decr.	2.2	1.6	5.9

Table 5.1: Running time improvements and average oracle call number decrease of enumImpr over enumOrig across 100 random attribute orders.

The second conclusion we can draw from these measurements is that the attribute order affects the enumeration time. We inspect this effect next.

5.3.3 Attribute Order

Configurable output order is one of the unique selling points of the enumeration algorithm. The output order depends on the input attribute order. An obvious attribute order is from most interesting to least interesting so that UCCs containing the most important attributes are added to the output set first. In some scenarios however, the output order may be irrelevant. As already suggested by the benchmarks of ENUMIMPR against ENUMORIG, the attribute order affects the enumeration time. Benchmarks across 2500 random orders for `ncvoter-allc` , `ncvoter-temporal` and `fd-reduced-30` demonstrate that the running time for different orders can vary by a factor of more than five (Figure 5.3).

Besides random attribute orders, the benchmarks also include two additional ones. *Descending uniqueness* is the order starting with the column containing the most unique values and ending with the column containing the least uniques. It is a candidate for quick enumeration because records are more likely to differ in their first attributes, leading to these attributes being included in (non-minimized) hyperedges more often. This raises the likelihood of being included in a hitting set. On the `ncvoter-allc` dataset (64 attributes), column-sorted by descending uniqueness, every minimal UCC includes at least one attribute out of the first nine. Consequently, on the same dataset sorted by the opposite order, *ascending uniqueness*, every minimal UCC includes at least one attribute out of the last nine. Thus, in the latter case, the ENUMIMPR procedure has to traverse down to one of the last nine levels to find any minimal hitting set. On average, a minimal hitting set is found on level 52.2 for descending and 59.3 for ascending order.

Our results show that descending uniqueness has one of the lowest enumeration times across all orders. The enumeration of the hypergraph ordered by ascending uniqueness is always significantly slower but not necessarily the slowest amongst all possible permutations (Figure 5.3).

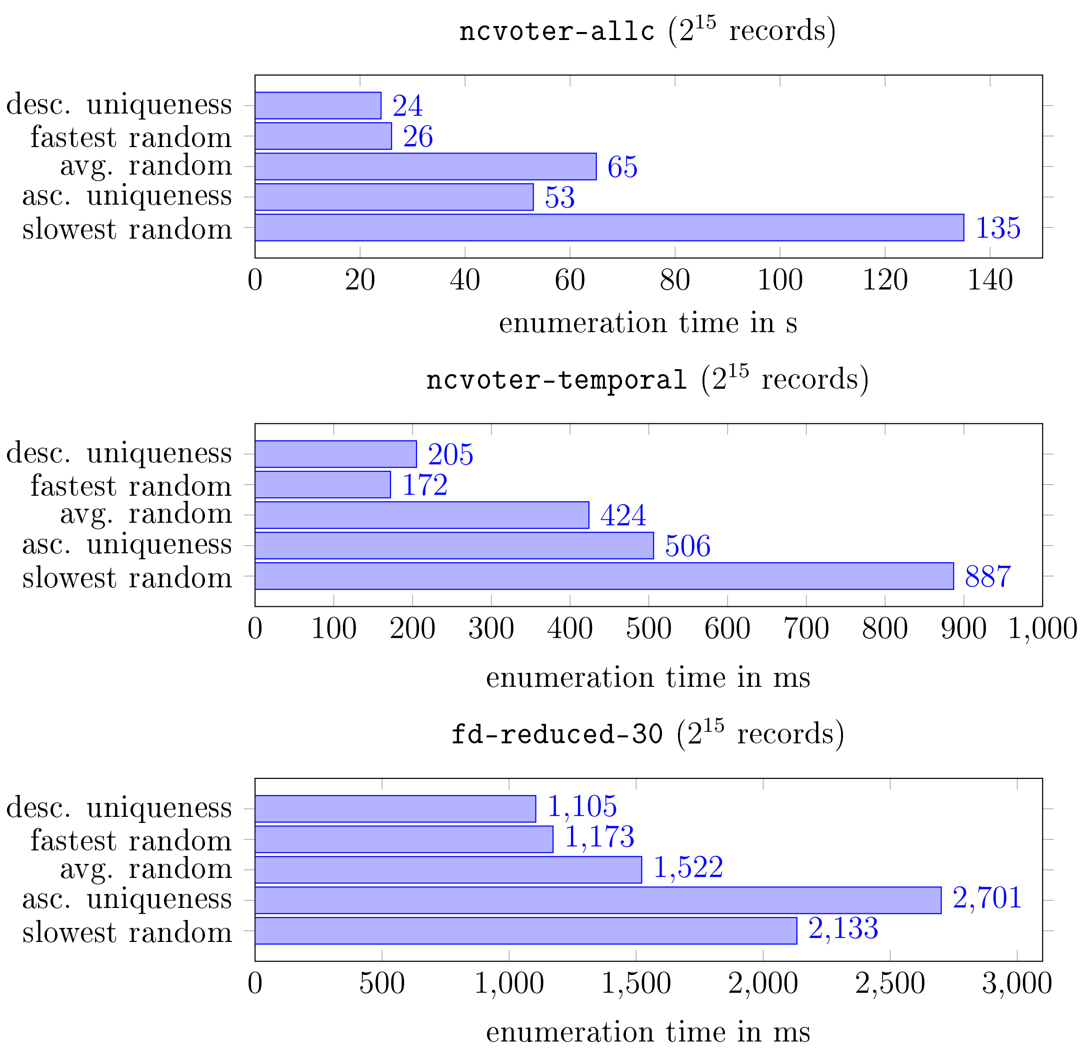


Figure 5.3: Enumeration time by attribute order on hypergraphs for `ncvoter-allc`, `ncvoter-temporal` and `fd-reduced-30`. Random order data points represent an average of 2500 random samples.

However, this enumeration time advantage for descending uniqueness ordered hypergraphs cannot solely be attributed to a small average minimal hitting set depth in the enumeration tree. On `ncvoter-allc` , an attribute order can be found that causes an average depth of just 33.5 (attributes ordered by frequency in the output, resulting in 50,000 oracle calls). Yet, this order almost doubles the enumeration time compared to the descending uniqueness order (78,000 oracle calls). This also shows that a low number of oracle calls is no guarantee for a fast running time, either. While the number of oracle calls does correlate strongly with enumeration running time on the `ncvoter-allc` dataset (Pearson correlation coefficient $r = 0.80$), it does so only weakly on `ncvoter-temporal` ($r = 0.17$) and even negatively on `fd-reduced-30` ($r = -0.19$; compare Figure 5.4).

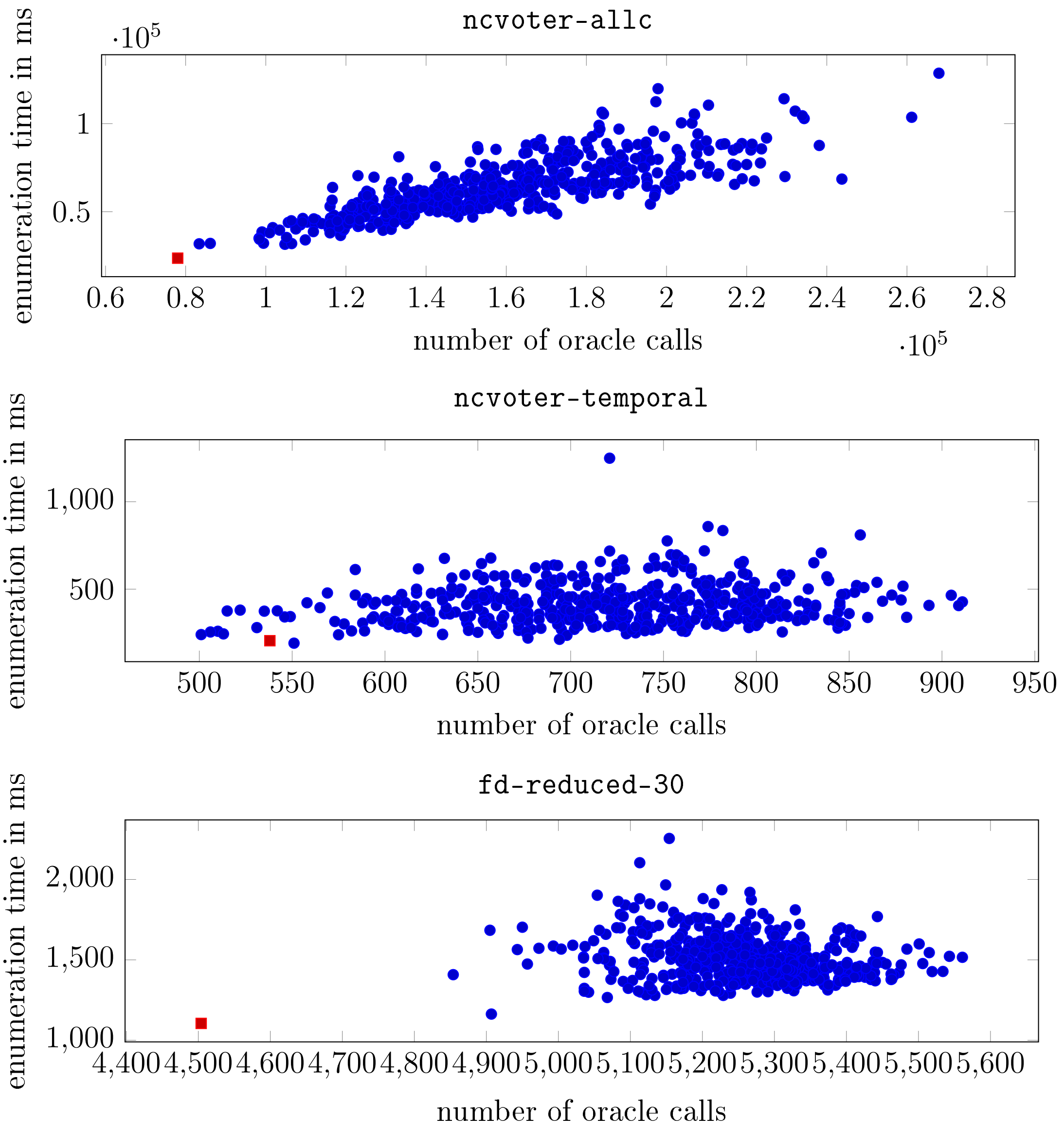


Figure 5.4: Enumeration time vs. number of oracle calls for 500 random attribute orders of `ncvoter-allc`, `ncvoter-temporal` and `fd-reduced-30`. The red square represents the enumeration performance of the descending uniqueness order.

In contrast to that, the total number of oracle brute force iterations is strongly positively correlated with total enumeration time ($r > 0.9$ on all datasets; compare Figure 5.5). This number is the total sum of all iterations of the loop starting in line 12 of Algorithm 4.1 across all oracle calls of a single enumeration.

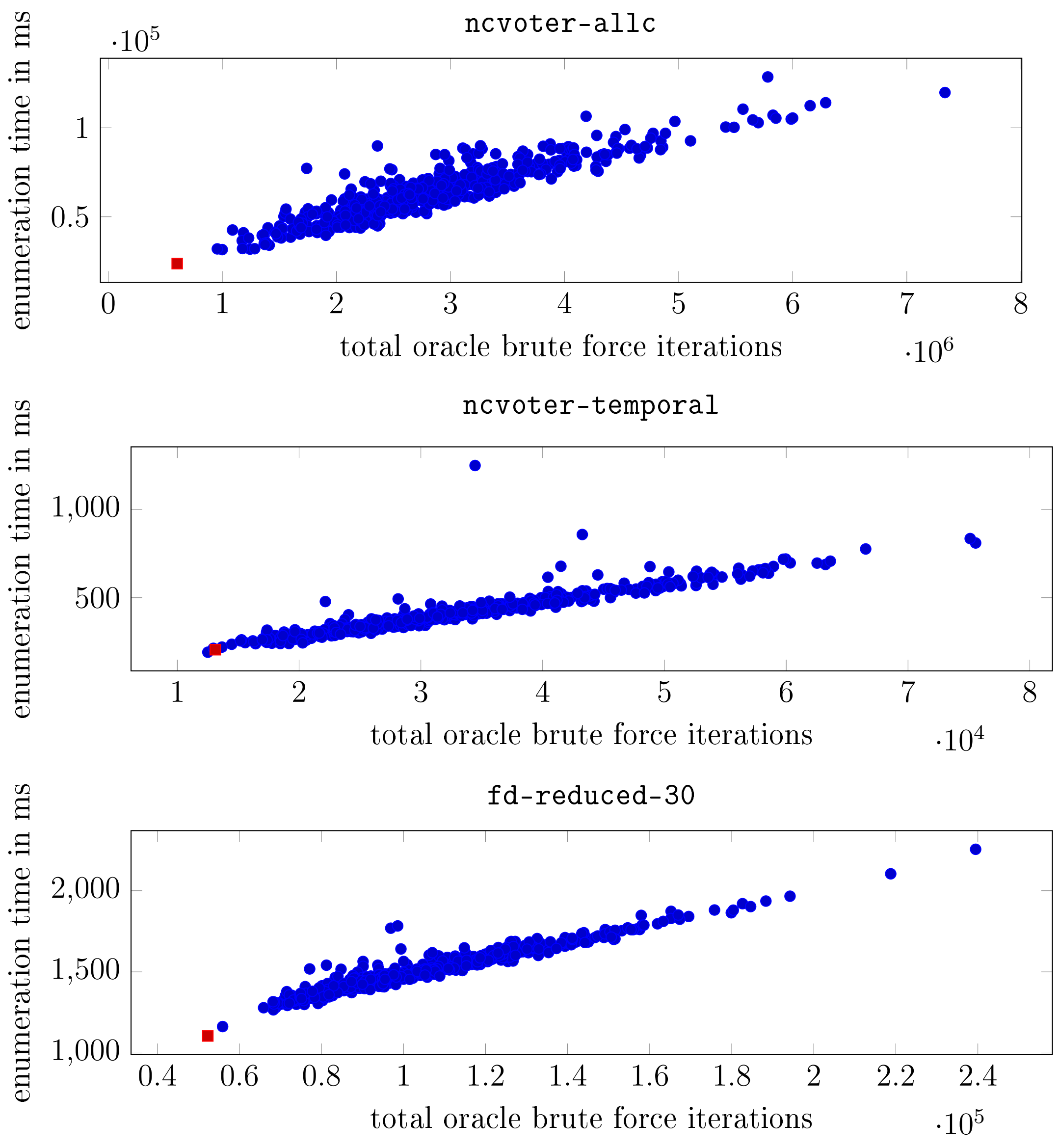


Figure 5.5: Enumeration time vs. total number of oracle brute force iterations for 500 random attribute orders of `ncvoter-allc`, `ncvoter-temporal` and `fd-reduced-30`. The red square represents the enumeration performance of the descending uniqueness order.

Generally, enumeration time differences have multiple reasons. First, the attribute order influences the number of visited nodes in the enumeration tree. If all minimal hitting sets are located close to each other in the tree, large branches may be pruned, reducing the overall number of visited nodes and therefore the total number of oracle calls. Second, oracle calls vary in cost. A call might return before entering the brute force loop at the end, after unsuccessfully testing out all remaining witness combinations or somewhere in between. To optimize the attribute order for enumeration time, one wants to minimize the total number of brute force iterations. This number is a product of the total number of oracle calls and the average brute force iterations per oracle call. Interestingly, these two measures are mostly uncorrelated on `ncvoter-allc` ($r = 0.09$) and `ncvoter-temporal` ($r = -0.18$) but weakly negatively correlated on the random dataset `fd-reduced-30` ($r = -0.38$; compare Figure 5.6). This means that on real-world data, it is possible to have the best of both worlds, a low number of oracle calls and a low average cost per call. Descending uniqueness ordering appears to be a good heuristic for minimizing both and requires only information that is known a priori.

What also follows from Figure 5.6 is that the average number of brute force iterations per oracle call appears to be bounded by a small dataset-specific constant. We have a more detailed look at the oracle running time behavior in the next subsection.

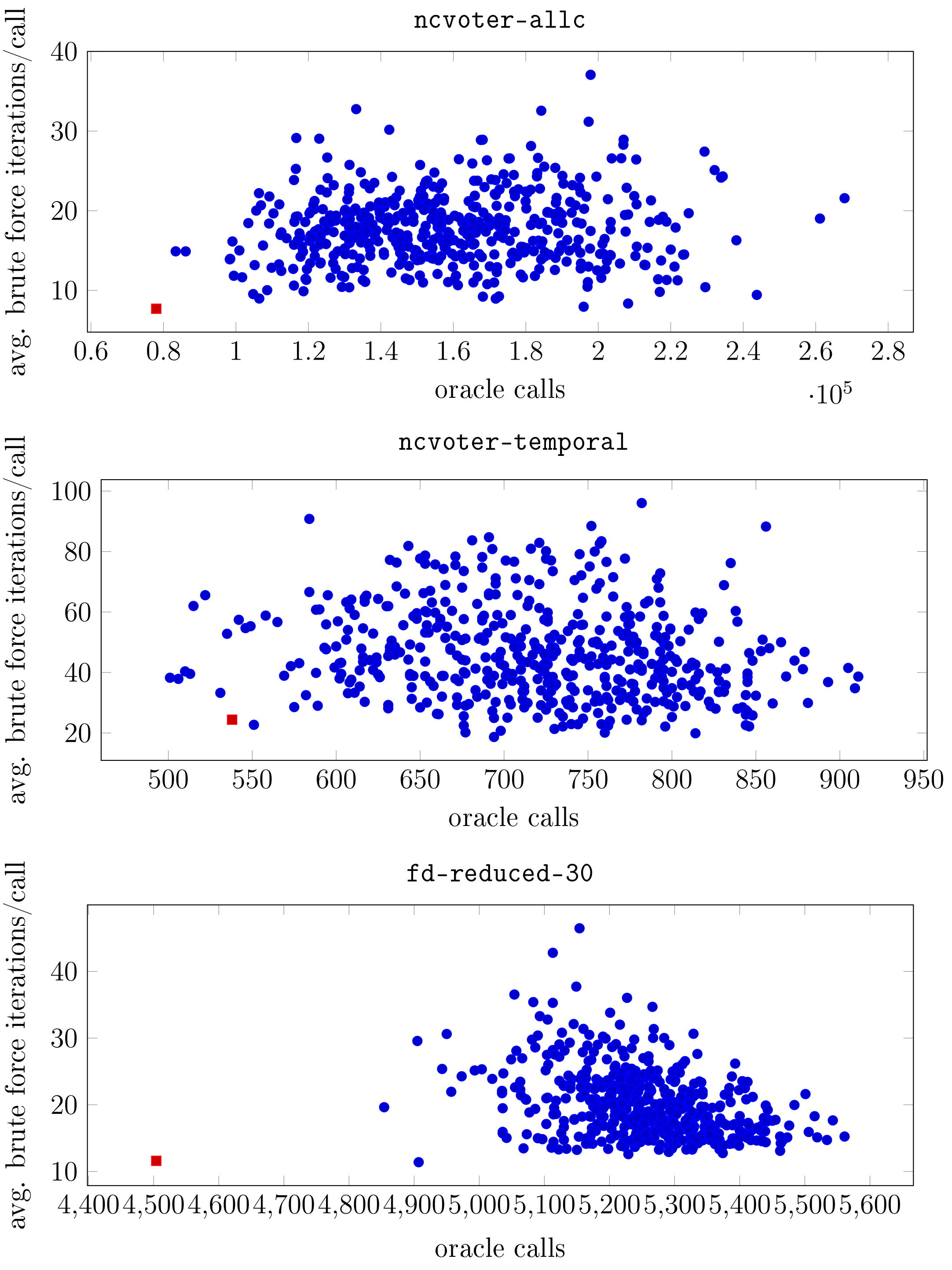


Figure 5.6: Average number of oracle brute force iterations vs. number of oracle calls for 500 random attribute orders of `ncvoter-allc`, `ncvoter-temporal` and `fd-reduced-30`. The red square represents the enumeration performance of the descending uniqueness order.

5.3.4 Extension Oracle Running Time

The extension oracle has a worst-case complexity of $\mathcal{O}((m/|X|)^{|X|}mn)$ when set operations take linear time [7]. Its running time is dominated by the brute force loop at the end. In contrast to the exponential worst-case complexity, the average number of brute force iterations on real-world hypergraphs is small, as visible in Figure 5.6. A large percentage of oracle calls (up to 66% on `ncvoter-allc`, up to 41% on `ncvoter-temporal`, up to 57% on `fd-reduced-30`) even return before entering the loop.

Still, averages do not paint the full picture. While for descending uniqueness ordered hypergraphs, the oracle running time appears to roughly follow a power law distribution (indicated by the almost straight line in the log-log plot), the same does not hold for the distribution of the number of brute force iterations of a single oracle call (Figure 5.7). Instead, a single oracle call seems to either return quickly, i. e., before or right after testing the first remaining witness combination, or it gets drawn out to a much longer brute forcing effort. Interestingly, if the oracle input is EXTENDABLE, it takes only a small number of iterations on average until the oracle call returns (Table 5.2).

However, on both `ncvoter-temporal` and `fd-reduced-30`, not even a single oracle call returns after exactly two iterations, leaving room for improvements. Our implementation tests remaining witness combinations in reverse lexicographical order, an arbitrary implementation choice. It may be possible to choose a brute force order that reduces the average number of iterations per oracle call for inputs that are EXTENDABLE, especially on hypergraphs that are not ordered by descending uniqueness. On these hypergraphs, averaged across 2500 random attribute orders, the number of brute force iterations until the oracle returns EXTENDABLE in the loop is 11.0, 26.2 and 19.7, respectively. We consider finding a faster brute force order a topic for future research.

return value	`ncvoter-allc`	`ncvoter-temporal`	`fd-reduced-30`
EXTENDABLE	2.95	8.79	1.02
NOTEXTENDABLE	55.65	76.51	189.69
total	22.67	40.86	23.18

Table 5.2: Average number of brute force iterations for oracle calls that enter the brute force loop, grouped by return value, for the descending uniqueness ordered hypergraphs of `ncvoter-allc`, `ncvoter-temporal` and `fd-reduced-30`.

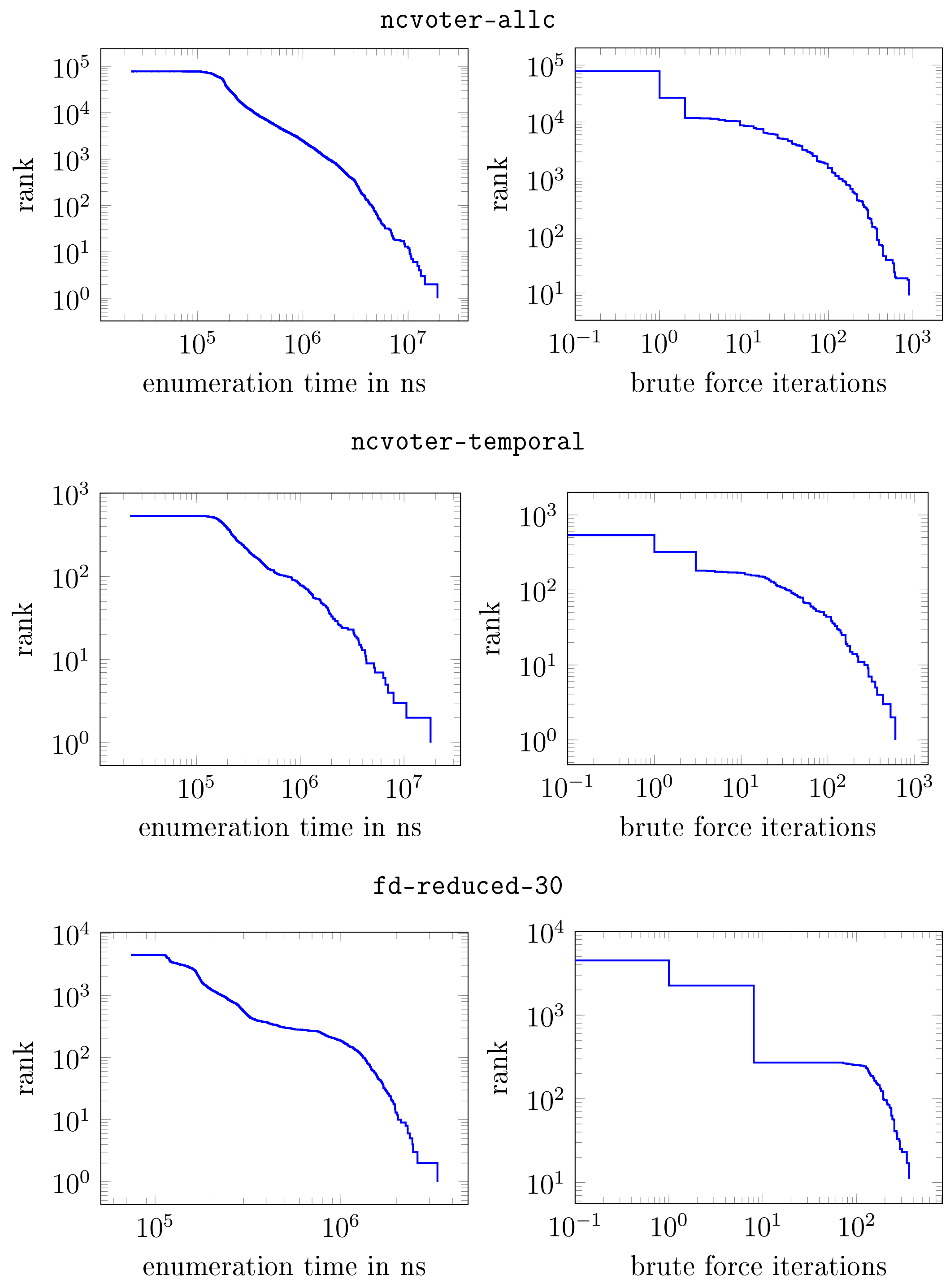


Figure 5.7: Oracle running time and brute force iteration distribution for the descending uniqueness ordered hypergraphs of `ncvoter-allc`, `ncvoter-temporal` and `fd-reduced-30`. The vertical axis displays the number of oracle calls with an enumeration time or iteration count greater or equal to the corresponding value on the horizontal axis.

6 Conclusion and Outlook

Building on the work of Bläsius et al. [7], we proposed an improved enumeration algorithm for lexicographically ordered minimal hitting sets in hypergraphs and subsequently used it to detect unique column combinations in datasets. Our improvements resulted in a lower output delay bound and a lower running time in practice. We showed that large real-world datasets produce small hypergraph problems. On those hypergraphs, the analyzed algorithm performs well despite an exponential worst-case running time complexity. The invoked extension oracle, having an exponential worst-case complexity itself, runs very fast on average. The enumeration requires only input-linear space.

While the attribute order can be used to prioritize important attributes that are added to the output first, allowing the algorithm to choose an order can improve running time more than fivefold. Some attributes may even be omitted due to existing functional dependencies rendering them irrelevant to the UCC detection problem.

Further research could look into optimizing the hypergraph generation stage which, at the moment, has a running time quadratic in input size and presents the main bottleneck in the application of the enumeration algorithm to UCC discovery. Papenbrock and Naumann [17] have shown that hybrid approaches iterating between generation and validation stages can outperform single direction algorithms by a wide margin.

The enumeration stage could profit from advanced heuristics that either prevent the extension oracle from entering the final brute force loop or order the remaining witness combinations in a way that leads to a faster return. While the general enumeration problem is proven to be difficult [7], hypergraphs generated from real-world data show inherent structure and characteristics that may allow for some shortcuts on the way to complete minimal hitting set enumeration.

References

[1] Z. Abedjan and F. Naumann. Advancing the discovery of unique column combinations. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management*, pages 1565–1570, 2011. (Cited on page 4.)

[2] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *VLDB J.*, 24(4):557–581, 2015. (Cited on pages 1 and 4.)

[3] C. Beeri, M. Dowd, R. Fagin, and R. Statman. On the structure of armstrong relations for functional dependencies. *J. ACM*, 31(1):30–46, 1984. (Cited on page 1.)

[4] C. Berge. *Hypergraphs: Combinatorics of Finite Sets*, volume 45 of *North-Holland Mathematical Library*. Elsevier Science, 1984. (Cited on pages 3 and 6.)

[5] J. C. Bioch and T. Ibaraki. Complexity of identification and dualization of positive boolean functions. *Inf. Comput.*, 123(1):50–63, 1995. (Cited on pages 2 and 5.)

[6] T. Bläsius, T. Friedrich, and M. Schirneck. The parameterized complexity of dependency detection in relational databases. In *11th International Symposium on Parameterized and Exact Computation, IPEC 2016*, pages 6:1–6:13, 2016. (Cited on page 1.)

[7] T. Bläsius, T. Friedrich, K. Meeks, and M. Schirneck. On the enumeration of minimal hitting sets in lexicographical order. *CoRR*, abs/1805.01310, 2018. (Cited on pages 1, 2, 4, 6, 7, 8, 9, 10, 27, and 29.)

[8] T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM J. Comput.*, 24(6):1278–1304, 1995. (Cited on page 2.)

[9] T. Eiter, K. Makino, and G. Gottlob. Computational aspects of monotone dualization: A brief survey. *Discrete Applied Mathematics*, 156(11):2035–2049, 2008. (Cited on page 2.)

[10] M. L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *J. Algorithms*, 21(3):618–628, 1996. (Cited on page 2.)

[11] D. Gunopulos, R. Khardon, H. Mannila, and H. Toivonen. Data mining, hypergraph transversals, and machine learning. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 209–216, 1997. (Cited on page 3.)

[12] M. Hagen. *Algorithmic and Computational Complexity Issues of MONET*. Cuvillier, 2008. (Cited on page 2.)

[13] A. Heise, J. Quiané-Ruiz, Z. Abedjan, A. Jentzsch, and F. Naumann. Scalable discovery of unique column combinations. *PVLDB*, 7(4):301–312, 2013. (Cited on page 4.)

[14] D. Klein and C. D. Manning. Parsing and hypergraphs. In *Proceedings of the Seventh International Workshop on Parsing Technologies (IWPT-2001)*. 2001. (Cited on page 3.)

[15] Ø. Ore. *Theory of graphs*, volume 38 of *Colloquium Publications - American Mathematical Society*. American Mathematical Society, 1962. (Cited on page 6.)

[16] T. Papenbrock and F. Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, pages 821–833, 2016. (Cited on page 16.)

[17] T. Papenbrock and F. Naumann. A hybrid approach for efficient unique column combination discovery. In *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, pages 195–204, 2017. (Cited on pages 4, 19, and 29.)

[18] V. Voloshin. *Introduction to Graph and Hypergraph Theory*. Nova Science Publishers, 2009. (Cited on page 3.)

[19] D. Zhou, J. Huang, and B. Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference*

on Neural Information Processing Systems, pages 1601–1608, 2006. (Cited on page 3.)