Minimization in Cooperative Response to Failing Database Queries

Parke Godfrey
godfrey@cs.umd.edu

Department of Computer Science,
University of Maryland,
College Park, MD 20742, USA

Abstract

When a query fails, it is more cooperative to identify the cause of failure, rather than just to report the empty answer set. If there is not a *cause* for the query's failure, it is worthwhile to report the part of the query which failed. To identify a *minimal failing subquery* (MFS) of the query is the best way to do this. (This MFS is not unique; there may be many of them.) Likewise, to identify a *maximal succeeding subquery* (MSS) can help a user to recast a new query that leads to a non-empty answer set.

Database systems do not provide the functionality of these types of cooperative responses. This may be, in part, because algorithmic approaches to finding the MFSs and the MSSs to a failing query are not obvious. The search space of subqueries is large. Despite work on MFSs in the past, the algorithmic complexity of these identification problems had remained uncharted.

This paper shows the complexity profile of MFS and MSS identification. It is shown that there exists a simple algorithm for finding a MFS or a MSS by asking N subsequent queries, in which N is the length of the query. To find more MFSs (or MSSs) can be hard. It is shown that to find $\mathcal{O}(N)$ MFSs (or MSSs) is **NP**-*hard*. To find k MFSs (or MSSs), for a fixed k, remains polynomial.

An optimal algorithm for enumerating MFSs and MSSs, ISHMAEL, is developed and presented. The algorithm has ideal performance in enumeration, finding the first answers quickly, and decaying toward intractability in a predictable manner as more answers are found.

The complexity results and the algorithmic approaches given in this paper should allow for the construction of MFS and MSS facilities for database systems. These results are relevant to a number of problems outside of databases too, and may find further application.

This work was supported by the Air Force Office of Scientific Research under grant AFOSR-91-0350 and by the National Science Foundation under grant IRI-9300691.

1 Introduction

A query is said to *fail* whenever its evaluation produces the empty answer set. An empty answer set is uninformative to the user. It is presumed that the user expects there to be answers to the query asked.[1] So when a query fails, it often is a surprise to the user. A system could be more cooperative by helping to trace the *reason* for the query's failure, or at least to pinpoint the failure.

The type of queries considered in this paper are conjunctive query formulas.[2] Let $\mathcal{Q}$ be such a query.

$$\mathcal{Q} \equiv \mathcal{A}_1 \wedge \ldots \wedge \mathcal{A}_k$$

Each of the $\mathcal{A}$'s is a literal. Call $\mathcal{Q}'$ a *subquery* of $\mathcal{Q}$ *iff*

$$\mathcal{Q}' \equiv \mathcal{A}_{s_1} \wedge \ldots \wedge \mathcal{A}_{s_j}, \text{ and } \{s_1, \ldots, s_j\} \subset \{1, \ldots, k\}$$

If a subquery fails, then the query itself must fail. Therefore, it is a stronger statement to report the failure of the subquery than to report the failure of the query itself. The best possible response is to report a *minimal* failing subquery (MFS).[3]

Consider that the following query fails:

$$\leftarrow \textit{ward}(\textit{patient: P, ward\_name: maternity}),\ \textit{infected}(\textit{patient: P, infection: I}), \qquad (\mathcal{Q}_1)$$
$$\textit{contagious}(\textit{name: I}),\ \textit{staph}(\textit{name: I}).$$

This query asks if there are any patients on the maternity ward with a contagious staphylococcus infection. Say the answer is *no*. The response that the subquery

$$\leftarrow \textit{infected}(\textit{patient: P, infection: I}),\ \textit{staph}(\textit{name: I}).$$

fails would be more informative. Coupled with the knowledge that this is a *minimal* failing subquery, it is even more informative; for instance, it states implicitly that there *are* patients with infections.

Let **DB** be a database in DATALOG [37]. Let $\vec{q}$ be the variables in the query formula $\mathcal{Q}$. (This is the tuple template for the answer set.) A query fails when

$$\mathbf{DB} \not\vdash \exists \vec{q}.\mathcal{Q} \qquad (\mathcal{F}_1)$$

In such cases, it is said that the query contains a *false presupposition*; the query itself, or one of its subqueries—a logical *presupposition* of the query—evaluates to *false*.

Likewise, a database system can respond to a failing query with a *maximally succeeding subquery* (MSS). Consider query $\mathcal{Q}_1$ again. The system could respond that the query fails, but that the subquery

$$\leftarrow \textit{infected}(\textit{patient: P, infection: I}),\ \textit{contagious}(\textit{name: I}).$$

succeeds. In other words, there are patients with contagious infections, but none have a contagious staphylococcus infection, and none are on the maternity ward.

Sometimes a query must fail, given the semantics of the database. Consider the query

$$\leftarrow \textit{ward}(\textit{patient: P, ward\_name: maternity}),\ \textit{infected}(\textit{patient: P, infection: I}), \qquad (\mathcal{Q}_2)$$
$$\textit{patient}(\textit{name: P, gender: male}).$$

[1] Otherwise, why ask it? Of course, there are cases when a user is attempting to confirm that a given query fails, but this is not the majority case.

[2] This is without loss of generality.

[3] A failing subquery is *minimal* *iff* no subquery of it fails.

There may be an integrity constraint associated with the database which ensures that any patient assigned to the maternity ward is female (hence, not male). Thus this query *cannot* have answers, unless the rules of the database change. To assume an answer to the query would lead to a contradiction.

When a query necessarily fails, it is said to contain a *misconception*.[4] That a query contains a misconception is a stronger statement than that a query fails. In misconception cases, an explanation of *why* the query fails can be produced, based upon a contradiction proof. A query $\mathcal{Q}$ is said to have a misconception when

$$\mathbf{DB} \vdash \neg \exists \vec{q}. \mathcal{Q} \qquad (\mathcal{F}_2)$$

One will see that this is strictly stronger logically than $\mathcal{F}_1$, the definition for when a query contains a false presupposition.

Again, it is useful to identify a minimal subquery that leads to contradiction, or a *minimal conflicting subquery* (MCS). For example, an MCS for query $\mathcal{Q}_2$ would be

$$\leftarrow \mathit{ward}(\mathit{patient{:}}\ P,\ \mathit{ward\_name{:}}\ \mathit{maternity}),\ \mathit{patient}(\mathit{name{:}}\ P,\ \mathit{gender{:}}\ \mathit{male}).$$

It would be worthwhile to extend database systems with facilities to find minimal failing, maximal succeeding, and minimal conflicting subqueries. Such cooperative features would make database systems easier to use. All three of these cooperative behaviors for better response to failing database queries require search for a minimal subquery. It is this search for a minimal that is the focus of this paper.

There has been much research on the topic, in particular on finding MFSs [7, 24, 26, 27, 28, 31, 32]. This work has not formally addressed the complexity of finding MFSs to queries. The implicit assumption has been that it may cost exponential time in worst case even to find one MFS. Much work has been devoted to heuristics and other means to reduce the search time to find MFSs, so that the search could be limited, in most cases, to render the problem tractable in practice.

No commercial database systems today offer the above cooperative features. Very few academic systems have prototyped such. In this paper, the previous algorithms for finding MFSs to queries are shown to be intractable, even in "average" case, despite the measures taken to avoid this. This apparent intractability may be the reason why such capabilities remain unimplemented in mainstream information systems today.

The complexity profile of these search problems is quite surprising. This paper addresses the inherent complexity of an MFS, MSS, or MCS facility for relational and deductive database systems. This paper offers both good and bad news for the possibility of such a facility. Certain MFS facilities are quite tractable and easy to provide; for instance, finding *an* MFS of a query. Other facilities are intractable; for instance, finding *all* MFSs of a query. Meanwhile, a facility to find *some* MFSs of a query warrants more discussion, and is addressed in this report.

To find an MFS or MSS of a query is shown to cost at most N subsequent queries to the database, where N is the number of conjuncts in the original query. To find a number of subsequent MFSs or MSSs of a query can be expensive. It is shown that to find $\mathcal{O}(N)$ MFSs or MSSs of a query of size N is **NP**-*complete*.

The next section reviews previous work done with the MFS problem, work that preceded, and the accomplishments. Section 3 defines an abstraction of these minimization problems. The abstraction involves finding minimal elements with respect to some test which is monotonic with respect to subset (if a set tests *true*, then any superset of it will test *true*) over a finite boolean lattice. Call a *Minimal Element in the Lattice with respect to a test* an MEL. A complexity analysis and algorithms for finding *an* MEL are presented. Section 4 extends the problem to enumerating MELs. The complexity of the enumeration is established, and the enumeration problems for MFSs, MSSs, and MCSs are shown to be equivalent in complexity. A general algorithm to enumerate MELs, ISHMAEL, is developed in Section 5. Heuristics, caching, and other techniques are evaluated for their potential to improve performance. Section 6 considers other theoretical

[4] This terminology arises from the fact that when a user asks a query that necessarily fails, it indicates that the user has a misconception with regards to the semantics of the database; the user thought the query *could* result in answers.

issues raised by the MEL enumeration problem. Many of these issues remain as future work. In particular, the probability distribution of finding given MELs is considered, and it is shown why the MEL algorithm does not offer a probabilistic attack on **NP**-*completeness*. In Section 7, related problems and applications that may benefit by the analysis, techniques, and algorithms for MEL (and MFS) presented in this paper are considered, and concluding remarks made.

2 Background

A student asks an appropriate university database

"Who passed CMSC 420 in the fall semester of 1991?"

The database returns with the answer "No one," leaving the student possibly to think that *CMSC 420* was a very hard course. The student then asks

"Who failed CMSC 420 in the fall semester of 1991?"

Again, the database returns with the answer "No one." Finally, the student is suspicious and asks

"Who taught CMSC 420 in the fall semester of 1991?"

The database answers again "No one."

Kaplan [26, 27] called this behavior *stonewalling*. If the initial question had been asked to a person instead, he or she would have probably answered immediately with a reply such as "Oh, there was no such course taught last semester." Databases stonewall. They will answer a *yes/no* question with a *yes* or a *no* regardless of whether the answer is misleading.

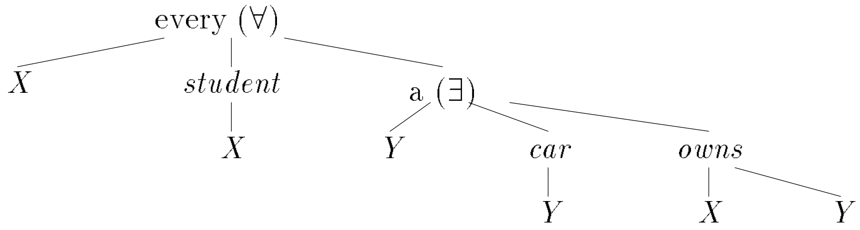
There has been prior interest in such stonewalling behavior (and in avoiding it) within the domain of natural language dialog. Strawson [36] claimed that for a statement to have a truth value, it should be necessary that all of its *presuppositions* be *true*. A presupposition of a statement is any statement entailed by the original. Consider the question "Is the king of France bald?" The statement cannot be answered *yes* or *no* according to Strawson, because the presupposition that there *is* a king of France is *false*.

Belnap and Steel considered such issues as related to information systems. They state "A question, Q, presupposes a statement, A, if and only if the truth of A is a logically necessary condition for there being some true answer to Q" [1]. This is not as rigid a condition as Strawson's; the query Q is considered to be *false* if it has any false presuppositions.

Grice [23] enumerated a number of maxims to which one ought to adhere in conversation in order to be cooperative. He states that an answer to a query should be *correct*, *non-misleading*, and *relevant*. Guarantees can be made that databases answer queries correctly. However, when databases stonewall, their answers are misleading. This commonly happens whenever a query has *false* presuppositions.

Colmerauer and Pique addressed the problem of false presuppositions in their work to translate natural language queries into a logical formalism [6]. They employ a three-valued logic that allows a sentence to be marked as *undefined* when such false presuppositions occur. They translate natural language sentences into a recursive datastructure they call a *three branch quantifier tree* (3BQ tree). Fig. 1 shows this representation for the statement "Every student owns a car." In their work, however, they do not develop a means to identify false presuppositions to the user.

Kaplan [26, 27] may have been the first to note the relevance of false presuppositions to databases. Any subquery of a conjunctive relational query may be considered a presupposition to the query, at least in Belnap's and Steel's view. If any subquery fails (evaluates *false*), then the query necessarily fails too.



"Every student owns a car."

$$\forall X.[student(X) \longrightarrow \exists Y.[car(Y) \wedge owns(X, Y)]]$$

Figure 1: Quantifier Tree Representation.

Within the domain of databases, Kaplan equates the false presupposition problem to that of finding the minimal failing subqueries. He notes that this problem is independent of natural language; it is an issue for any formal query language and query answering system.

Kaplan built a system called CO-OP (A Cooperative Query System), which couples a natural language query system and a CODASYL database management system, SEED [37]. *CO-OP* provides cooperative responses to simple natural language questions, requesting the relevant data from the database. The system was used and tested over a real database from the National Center for Atmospheric Research by both users and programmers.

In related work, Lee [28] developed a CODASYL database system which detects and presents false presuppositions to database queries. The query language employed is called HI-IQ (HIerarchical Interactive Query language). He too noted the independence of the MFS problem from natural language.

Janas [24] studied the computational feasibility of reporting the smallest subqueries that fail. If one considers a conjunctive query as a set of atoms to be satisfied, then the subqueries are the elements of the power set. In all, there are $2^N - 2$ subqueries for a conjunctive query with N atoms, disregarding the query itself (which has already been seen to fail) and $\emptyset$, the empty query. The query $\mathcal{Q}_1$ (on page 1) has 4 atoms, so there are 14 subqueries to consider. The naïve approach is to test all of them. This will incur exponential cost over the length of the query.

Janas introduces an algorithm for finding the minimal failing subqueries. He recognizes the inherent intractability of the algorithm, but surmises that the connectivity of the query can be exploited to reduce the number of subqueries in the lattice that need to be considered. Fig. 2 shows this lattice for the query $\mathcal{Q}_1$ from the introduction. Here only twelve of the subqueries need to be considered; the absent two are *disjoint* and do not have to be evaluated.[5] (See Section 5.6 for more on this.) Janas also proposes that the integrity constraints associated with the database can be used to reduce the number of subqueries that need to be evaluated.

Kaplan [27] devises an algorithm similar to Janas's that operates over a query translated into MQL, the query language CO-OP employs internally, but he does not consider the computational issues involved. He points out that his algorithm to find false presuppositions is independent of domain specific knowledge (as is Janas's). These are techniques that are applicable over any domain. Kaplan also introduces the notion of generalizing a failing query into a query which succeeds; if a query fails due to a failing subquery, somehow the failing parts are removed, resulting in a new query that does have answers. This can serve as a tool in correcting possible errors in users' queries, and to give the user information *related* to the query asked.[6]

Corella et al. [7] considered finding the MFSs to conjunctive boolean queries for library searches. Their

[5] A disjoint query is equivalent to the union of several independent queries.

[6] However, this is not as powerful as identifying a maximally succeeding subquery, as seen in examples in the introduction.

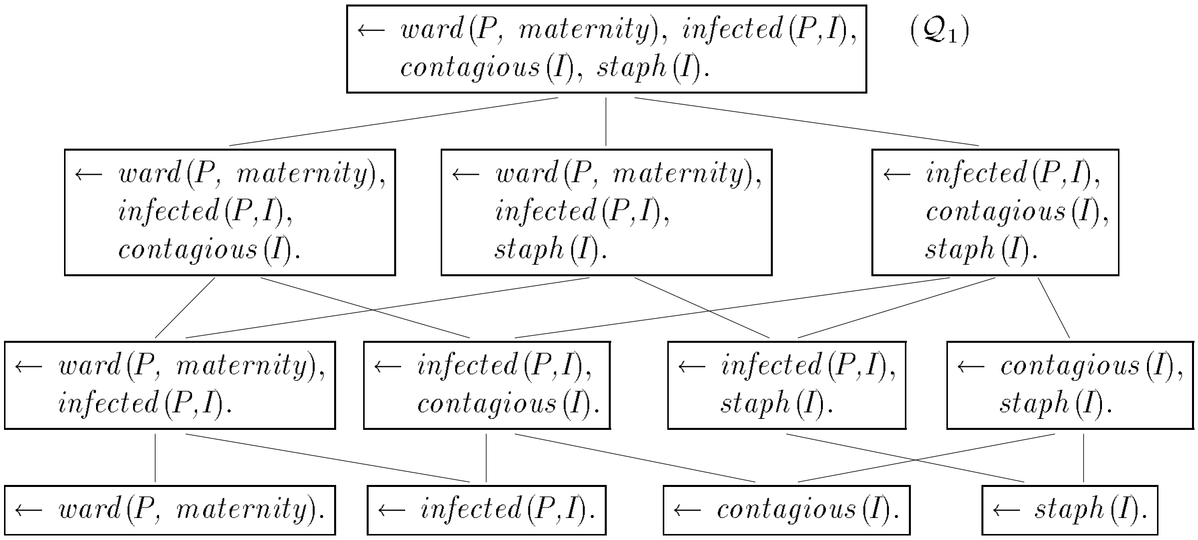


Figure 2: A lattice of subqueries.

system also reports the number of matches found for each subquery and displays the results graphically. The MFSs then are just the subqueries which had zero matches. This is intended to help the user to choose among the subqueries to pursue.

Motro [31, 32] extended on the notion of false presuppositions. Instead of considering only the subqueries of a query, as defined above, he considers certain generalizations of the query as well, which are logical presuppositions to the query. The query generalizations are obtained by *relaxing* to a degree some of the conditions in the query. A subquery is an extreme generalization: some of the conditions have been completely removed (and, hence, are vacuously *true*).

For example, if part of a query is that someone's salary is greater than or equal to forty thousand dollars a year, this condition could be relaxed to that the salary is greater than or equal to thirty-nine thousand. This necessitates that a function for each predicate that can be relaxed be supplied. The step size of the relaxation step must also be decided. For instance in the above example, salary is relaxed in one thousand dollar increments. Eventually a literal may relax to *true* and be removed.

This method combines the notion of relaxing queries into more general queries and that of searching for false presuppositions. Now instead of finding minimal failing subqueries, a system could return *maximally generalized* failing queries (MGQs). Such responses have the potential to be more informative than reporting just the MFSs.

Motro's method extends the lattice of presuppositions compared with the subquery lattice. Consider the query

$$\begin{array}{l}\leftarrow \mathit{employee}(\mathit{age{:}}\ A,\ \mathit{gender{:}}\ G,\ \mathit{salary{:}}\ S),\\ \quad A \leq 30,\ G = \text{female},\ S \geq 40k.\end{array} \qquad (\mathcal{Q}_3)$$

Fig. 3 shows the start of a generalization lattice for the query, in which the three conditions are being relaxed.[7] This paper focuses on finding MFSs; the search for MGQs is considered briefly in Section 6.

One may argue that even if the cost of finding minimal failing subqueries remains high, it may well be offset by the benefits that these cooperative responses offer. When informed of the part of the query that fails, the user will not waste time asking follow-up questions which also necessarily fail, while he or she continues

[7] A generalization lattice for a query is not necessarily finite, as is the subquery lattice.

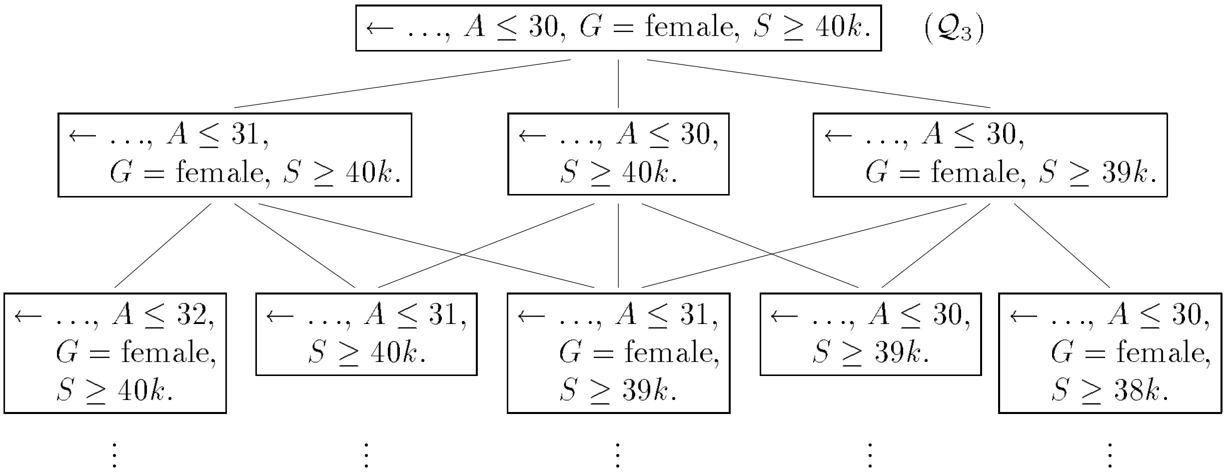


Figure 3: An extended lattice of subqueries.

to look for the information originally desired. The overall reduction in cumulative query-answering costs could be significant, especially if the user would have asked many spurious queries otherwise, as in Kaplan's stonewalling examples. Furthermore, by preventing stonewalling, the user is less likely to become frustrated, dissuaded from his or her goal, or even misinformed; so the database system becomes more effective.

A number of people have been concerned with how to detect misconceptions in queries. Mays [29] explored how to employ the schema information of a relational database in order to *correct* misconceptions, whenever possible. McCoy [30] used world (or general) knowledge to correct *object related* misconceptions that a user might have about the properties of a given object or class. As stated above, Janas considered the use of a database's integrity constraints to eliminate subqueries from consideration; however, he did not consider providing an explanation of a query's failure.

The realization that a query's failure assured by integrity constraints is more meaningful than just exhaustive failure motivated the work in cooperative answering by Gal and Minker [16, 17, 18]. The cooperative answering system developed by Gal [16] identifies the integrity constraints that guarantee failure, and provides a response to the user based on them. Gal also recognized the need to provide an MCS, but did not address how to identify the minimal subqueries with misconceptions. See the survey [14] for more background.

In [15, 21] the design and implementation of a cooperative database system are discussed, in which misconception detection and explanation are key components. In this system, all three problems, MFS, MSS, and MCS, are to be addressed.

3 Analysis—Finding *a* Minimal Failing Subquery

At first consideration, it may seem that finding *one* MFS for a query is potentially intractable. None of the work in the literature states the case any differently. Some further consideration, however, will show that this is not the case. A recursive descent algorithm which finds *an* MFS for a query via N subsequent queries to the database is presented in Section 3.5.

On the other hand, to find *all* MFSs for a query is hard. It requires exponential time in worst case. This is true due to the simple fact that there can be an exponential number of MFSs of a query, so it can require exponential time to report all of them. To see this, assume that all subqueries of size $N/2$ are MFSs of a

	breadth-first	depth-first
top-down	$\mathcal{O}(2^N)$	$\mathcal{O}(N)$
bottom-up	$\mathcal{O}(2^N)$	?

Table 1: Search strategies for finding an MEL

query of N literals.[8] There exist $\binom{N}{N/2}$ such subqueries. $\mathcal{O}(\binom{N}{N/2}) = \Omega(2^N)$ so it is $\mathcal{O}(2^N)$.

The next natural question then is how much does it cost to enumerate the MFSs of a query? That is, once one MFS is known, how much does it cost to find a second MFS, a third, and so forth? There exist several possibilities for the complexity of such an enumeration algorithm. In the first case, the time required to find each subsequent answer might be bounded to within some fixed polynomial. So it would require only some fixed polynomial time with respect to N to find a subsequent MFS. The second case is worse, but still offers decent behavior. It might require fixed polynomial time with respect to N *plus* k, where k is the number of answers found as of this point. After an "exponential" number of answers have been found, the time to find the next answer appears to be "exponential" with respect to N. However, the time needed between answers would grow slowly, and predictably. In the third case, after finding k MFSs, to find another may be intractable. In this case, it is important to determine k, and whether it is a constant or is related to N, the size of the query.

3.1 An abstraction

Let us present an abstraction of the MFS problem. Consider the finite boolean algebra formed over the set $\mathcal{S} = \{e_1, \ldots, e_N\}$ with respect to containment; that is, the powerset of $\mathcal{S}$. This lattice consists of 2^N elements, with $\mathcal{S}$ as the top element and $\emptyset$ as the bottom.

Let the test $\mathcal{T}$ be a unary relation over $2^{\mathcal{S}}$ ($\mathcal{T} \subseteq 2^{\mathcal{S}}$) such that $\mathcal{T}$ is *monotonic with respect to* subset; that is, if $\mathcal{A} \in \mathcal{T}$, then for any superset $\mathcal{B}$ of $\mathcal{A}$, $\mathcal{B} \in \mathcal{T}$. For the sake of this discussion, assume that evaluating $\mathcal{T}$ is of unit cost.

Call a minimal element in the lattice with respect to the test $\mathcal{T}$ an MEL. Thus $\mathcal{A}$ is an MEL *iff* $\mathcal{A} \in \mathcal{T}$ and $\forall \mathcal{B} \subset \mathcal{A}.\ \mathcal{B} \notin \mathcal{T}$.

The MFS problem easily maps into this MEL problem. The $\mathcal{T}$ in this case is to ask the query to the database system. If the query *fails*, then the test returns *true*. Otherwise, if the query has a non-empty answer set, the test returns *false*.[9]

Given an $\mathcal{S}$ and a monotonic $\mathcal{T}$ over $\mathcal{S}$, finding *whether* there exists an answer is trivial. Ask if $\mathcal{S} \in \mathcal{T}$. If (and only if) *yes*, there exists an answer. This indicates that finding an MEL, a minimal answer, may also be simple.

It may seem counterintuitive initially to divide the MFS problem in this way, into two parts, a test and a search space. Any resulting strategy would seem to have to be *generate-and-test*, which rarely is the most efficient approach algorithmically. This abstraction will, however, help to shed light on the inherent complexity of the problem. Furthermore, it will be seen that there is not, in fact, a more efficient algorithmic approach to be had.

[8] Assume that N is even, without loss of generality.
[9] This notation may be initially confusing! Remember that queries resulting in *empty* answer sets test *true*.

```
boolean a_mel(Top, var Mel) {
    if test(Top) then {
        a_mel_true(Top, Mel)
        return true
    } else
        return false
}
a_mel_true(Top, var Mel) {
    Set := Top
    Minimal := true
    while Minimal and Set ≠ [] {
        choose Ele ∈ Set Set := Set − [Ele]
        if test(Top − [Ele]) then {
            a_mel_true(Top − [Ele], Mel)
            Minimal := false
        }
    }
    if Minimal then
        Mel := Top
}
```

Algorithm 1: An algorithm for finding an MEL in N^2 steps.

3.2 Finding an MEL

To find an MEL of a set $\mathcal{S}$ given a monotonic test $\mathcal{T}$ requires searching the complete boolean lattice over $2^{\mathcal{S}}$ with respect to containment, so the search space is very large. The search can be done either breadth-first or depth-first; it can be directed either top-down (starting with $\mathcal{S}$) or bottom-up (starting with $\emptyset$).

The breadth of this lattice is very large, so breadth-first search cannot fare well. The depth of the lattice, however, is fixed at N, so depth-first search should fare well. Table 1 shows how these search strategies fare.

To find all MELs was seen to be of order exponential time, simply because there can be an exponential number of MELs. The number of subsets of $\mathcal{S}$ of length $N/2$ is $\binom{N}{N/2}$. Therefore, breadth-first search always will be in order exponential time. If the search is top-down, assume that all the MELs are smaller than $N/2$. If the search is bottom-up, assume that all the MELs are larger than $N/2$. In either case, all the subsets of length $N/2$ will be explored before the first MEL is found.

Better results can be obtained with depth-first search. Any MEL is only at most N steps away from $\mathcal{S}$ or $\emptyset$ in the lattice. It is not clear how the search could proceed bottom-up in an intelligent fashion. The test $\mathcal{T}$ will not help to decide which edge of the lattice to traverse next. However, if the search proceeds top-down, the test can be used to advantage.

The procedure *amel* (Alg. 1) is an algorithm for finding an MEL of a set with respect to a given test. It proceeds depth-first, top-down. Set $\mathcal{S}$ is initially tested; if it fails ($\mathcal{S} \notin \mathcal{T}$), there is no need to proceed.[10] Otherwise, an element is removed from $\mathcal{S}$ and the resulting subset tested. If the subset fails, another element is chosen and tested instead. If all the possibilities fail, then it is known that $\mathcal{S}$ passed the test ($\mathcal{S} \in \mathcal{T}$) and that none of its subsets did; hence, it is minimal so it is an MEL. If, on the other hand, one of the subsets passed, recursively proceed. Any MEL of the subset is also an MEL of $\mathcal{S}$. It is assumed that *test*($\emptyset$) *fails*.

Consider *a_mel* called on the set $\{a, b, c, d\}$, and that the set has a single MEL, $\{a\}$. Let *amel*'s selection

[10] In the case that we are checking a query for MFSs, if the query evaluated to a *non-empty* answer set, there is no need to proceed; a non-empty query has no MFSs.

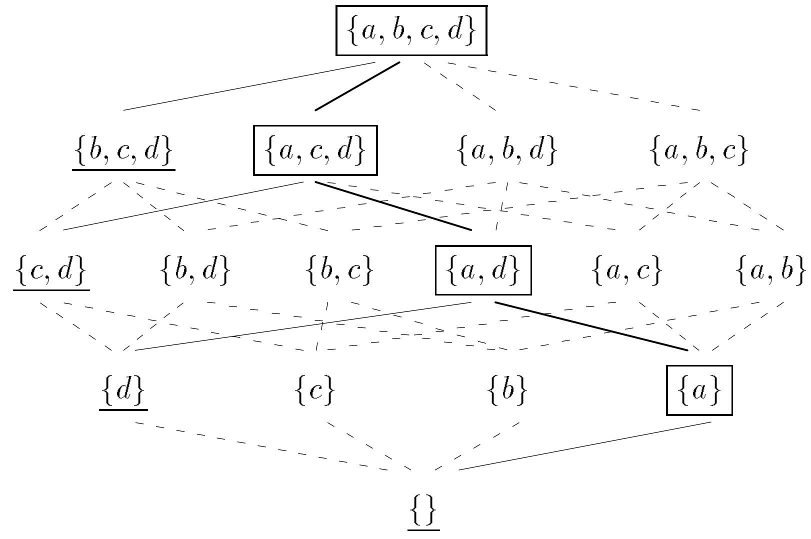


Figure 4: Routine *a_mel* run on $\{a,b,c,d\}$ to find MEL $\{a\}$.

rule in the *choose* statement be to choose the first element from the set as written. (Assume that the set is *ordered.*) Fig. 4 demonstrates *a_mel*'s search. The solid lines show the edges of the lattice which *a_mel_true* traverses. The underlined sets *fail* the test, so the search does not proceed under them. The boxed sets succeed, and *a_mel* is called recursively.

Theorem 1. The worst-case running time of *a_mel*($\mathcal{S}$) is $\Theta(N^2)$, for which N is the length of set $\mathcal{S}$, and each invocation of *test* costs unit time.[11] The running time of *a_mel*($\mathcal{S}$) is bound by $N(N-1)/2$ steps.

Proof. Algorithm *a_mel* is $\mathcal{O}(N^2)$, and always completes within $N(N-1)/2$ steps.

To determine whether a subset is an MEL requires, in worst case, that all its immediate subsets be tested. There are at most N of these. Assume that, in worst case, all but the last tests *false*. The recursive descent to the MEL takes at most N steps. On each recursive call, the size of the input set is reduced by one. The number of tests performed is thus $N + (N-1) + \ldots + 1$. So, in worst case, at most $N(N-1)/2$ tests are performed.

Algorithm *a_mel* has a worst-case running time of $\Omega(N^2)$.

Consider $\mathcal{S} = \{e_1, \ldots, e_n\}$. Let $\mathcal{S}$ have one MEL: $\{e_1, \ldots, e_{n/2}\}$. (Assume that n is even, without loss of generality.) Let the selection rule of *a_mel* in the *choose* statement select the first element of the set. (Consider the sets to be ordered sets.)

Consider the invocation of *a_mel_true*($\{e_1, \ldots, e_n\}$). First e_1 is removed and the remaining subset tested; then e_2 is removed, e_3, and so forth. So the sets $\{e_2, \ldots, e_n\}$, $\{e_1, e_3, \ldots, e_n\}$, through $\{e_1, \ldots, e_{n/2-1}, e_{n/2}, \ldots, e_n\}$ are all tested with each one failing. Finally $\{e_1, \ldots, e_{n/2}, e_{n/2+1}, \ldots, e_n\}$ is tested and succeeds. This constitutes $(n/2)+1$ tests before *a_mel_true*($\{e_1, \ldots, e_{n/2}, e_{n/2+1}, \ldots, e_n\}$) is recursively called.

This call to *a_mel_true* will likewise perform $(n/2)+1$ tests before the next recursive invocation. There are $n/2$ such recursive stages before *a_mel_true*($\{e_1, \ldots, e_{n/2}\}$) is called. This last call performs $n/2$ tests to verify the MEL.

[11] The definitions for the *asymptotic upper bound* ($\mathcal{O}$), the *asymptotic lower bound* (Ω), and the *asymptotic tight bound* (Θ) employed here are the same as presented in [8].

```
array mel(set)
integer Last

boolean all_mels(Top) {
      Last := 0
      if test(Top) then {
            all_mels_true(Top)
            return true
      } else
            return false
}

all_mels_true(Set) {
      Minimal := true
      foreach Ele ∈ Set {
            if test(Set − [Ele]) then {
                  all_mels_true(Set − [Ele])
                  Minimal := false
            }
      }
      if Minimal then {
            Last := Last + 1
            mel(Last) := Set
      }
}
```

Algorithm 2: A naïve algorithm for finding all MELs.

The initial call to *a_mel* executed one test before calling *a_mel_true*. In total, this results in $n^2/4+n/2+1$ calls to test. Hence the worst-case running time of *a_mel* is $\Omega(N^2)$.

Algorithm *a_mel* is worst-case running time of $\Theta(N^2)$. □

The recursive descent method employed by *a_mel* is certainly not a new observation. Bylander et al. [2] show this algorithm for certain abduction problems, closely related to the MEL problem here, and they show that it is $\mathcal{O}(N^2)$ to find *an* answer.

3.3 Finding all MELs

It is still not clear, however, what is the best search strategy for finding all the MELs of a set (with respect to a monotonic test); or at least what is a good search strategy for finding a *number* of MELs efficiently. However, the algorithm *a_mel* can be modified in a simple manner to find all MELs, in principle.

The algorithm *all_mels* (Alg. 2) shows how the algorithm 1 can be reworked for this. The *while* loop of *a_mel* is replaced with a *foreach* loop in *all_mels* so that each of the possible paths to an MEL is explored in turn. Two global variables are used to store the MELs as they are found: the array *mel* is an array of type set, and as each MEL is found, it is added to the next free slot in the array;[12] the integer *Last* points to the last array position in *mel* to have been filled. When *Last* is 0, no MEL has yet been found.

The algorithms that have been suggested for MFS search are essentially isomorphic to *all_mels*. Janas presents such an algorithm in [24]. His algorithm works top-down and depth-first, and the algorithm continues until all MFSs are found. The procedural control is that of *all_mels*.

[12] Assume that the array *mel* is dynamically allocated and is not of fixed length.

Motro presents an algorithm [31] to search his extended lattice for a query to find all maximal failing generalizations of the query. In the trivial case (when all the relaxation operators return *true* for any input), his search is over the same lattice space as for *all_mels*, and the maximal failing generalizations are equivalent to just the MFSs. His algorithm reduces to *all_mels*. It proceeds top-down, depth-first, and the control is the same as that for *all_mels*.

3.4 Finding some MELs

Neither Janas nor Motro stated that their algorithms would find the first MFS in $\mathcal{O}(N^2)$ time, although the algorithms should have this performance. Unfortunately, the performance of *all_mels* (and, hence, these other algorithms) breaks down immediately. The time to find a *second* (unique) MEL can be—and usually is—exponential.

Consider a $\mathcal{S} = \{e_1, \ldots, e_N\}$ and a test $\mathcal{T}$ which yields two MELs with respect to $\mathcal{S}$: $\{e_1, e_2\}$ and $\{e_{N-1}, e_N\}$. The algorithm *all_mels* will find the second of these first, because it will throw out e_1 in the first stage. At the second stage (recursive invocation), the algorithm is called to find all the MELs of $\{e_2, \ldots, e_N\}$.

Unfortunately, throwing away any one of $\{e_2, \ldots, e_{N-2}\}$ will yield a subset which tests *true*. The *foreach* statement will recursively invoke *all_mels* on each of these. This same redundancy will exist in each recursive invocation. The invocation of *all_mels* on $\{e_2, \ldots, e_N\}$ will arrive at $\{e_{N-1}, e_N\}$ $(N-3)!$ number of times. So *all_mels* called on $\{e_1, \ldots, e_N\}$ will spend at least $(N-3)!$ steps between finding the MEL $\{e_{N-1}, e_N\}$ and finding the next MEL $\{e_1, e_2\}$.

In fact, the way that *all_mels* is written, it will insert the same MEL $\{e_{N-1}, e_N\}$ into the array *mel* at least an exponential number of times. This may be considered an oversight, and could easily be corrected. (A check can be added in the last *if* clause to assure that the set is not already in the *mel* array before adding it.) Of course, this will not change the time complexity of the algorithm.

As is, the algorithm does find each subsequent MEL in $\mathcal{O}(N^2)$ time, but each subsequent MEL is not unique, a necessary criterion. Once the algorithm is modified to report each MEL once, the time to the second MEL is exponential in worst case. This might seem to indicate the possibility that finding *one* MEL is easy, but finding two or more is intractable. However, this would be paradoxical. If the order of presentation of the "set" $\mathcal{S}$ were reversed, $\{e_N, \ldots, e_1\}$, then the MEL $\{e_1, e_2\}$ would be found first, and found quickly. (The two MELs are symmetrically indistinguishable.)

In Section 4, it is shown that to find $\mathcal{O}(N)$ MELs is the equivalent of an **NP**-*complete* problem. To find a second or third MEL can be shown to be polynomial, and, in fact, still within N^2 calls to the test. This indicates that a much better algorithm for enumerating MELs (and, hence, MFSs, MSSs, and MCSs) can be devised.

3.5 A faster algorithm for finding *an* MEL

Before continuing, an improvement on the algorithm *a_mel* (Alg. 1) can be made for finding a single MEL. The search for an MEL can be shown to be within $\mathcal{O}(N)$. Algorithm *a_mel_fast*(Alg. 3) runs in $\mathcal{O}(N)$ time.

In *a_mel_fast* (Alg. 3, just one call to *test* is needed per element in the input set. If the result is *true*, the element is thrown away. Otherwise, if the result is *false*, the element must be a member of the MEL being constructed. The following observation makes this improvement possible.

Lemma 2. If $\mathcal{S} \in \mathcal{T}$ and $(\mathcal{S} - \{e\}) \notin \mathcal{T}$, then any MEL of $\mathcal{S}$ must contain e.

Proof. Assume that $\mathcal{R}$ is an MEL of $\mathcal{S}$ which does not contain e. Then $\mathcal{R}$ is a subset of $(\mathcal{S} - \{e\})$. So $\mathcal{R} \notin \mathcal{T}$, since $\mathcal{T}$ is monotonic with respect to subset. Thus $\mathcal{R}$ cannot be an MEL. This contradicts the

```
boolean a_mel_fast(Top, Mel) {
      if test(Top) then {
            a_mel_true(Top, Mel, [])
            return true
      } else
            return false
}
a_mel_true(Set, Mel, Core) {
      if Set = [] then
            Mel := Core
      else {
            choose Ele ∈ Set
            if test((Set − [Ele]) ∪ Core) then
                  a_mel_true(Set − [Ele], Mel, Core)
            else
                  a_mel_true(Set − [Ele], Mel, Core ∪ [Ele])
      }
}
```

Algorithm 3: An algorithm for finding a MEL in N steps.

original assumption. □

Theorem 3. The algorithm *a_mel_fast*($\mathcal{S}$) has a running time of N steps, in which N is the size of $\mathcal{S}$, and each invocation of *test* costs unit time.

Proof. In algorithm *a_mel_fast*, only one call to *test* per element of the input set $\mathcal{S}$ is needed. If $(\mathcal{S}-\{e\}) \in \mathcal{T}$, then the element e can be thrown away; else e must show up in the final answer. In this latter case, e is added to an accumulator, *Core*. Thus *a_mel_fast* will complete within N calls to *test*. □

4 The Complexity of Enumeration

The complexity of finding an MEL has now been established. Next, the complexity of enumerating MELs is to be considered. It shall be shown that to find $\mathcal{O}(|\mathcal{S}|)$ MELs of $\mathcal{S}$ is **NP**-*complete*. To find k MELs, for any fixed k, can be done in polynomial time.

4.1 An Abstraction—Enumerating Minimals

Let us present a set-theoretic version of MEL in order to consider its algorithmic complexity.

Definition 4. Define **MEL** as follows. Consider any pair $\langle \mathcal{S}, \mathcal{T} \rangle$, in which $\mathcal{S}$ is a set and $\mathcal{T}$ is a Turing machine, $\mathcal{S}$ is represented via an enumeration of its elements, and $\mathcal{T}$ has the following properties:

- $\mathcal{T}$ is defined over the domain of $2^{\mathcal{S}}$;
- $\mathcal{T}$ halts and returns *yes* or *no* on any input in its domain;
- $\mathcal{T}$ runs in polynomial time (for some fixed polynomial) on any input in its domain; and
- if $\mathcal{T}$ returns *yes* on input $\mathcal{A}$, then for all $\mathcal{B} \subseteq S$ such that $\mathcal{A} \subset \mathcal{B}$, $\mathcal{T}$ returns *yes* on input $\mathcal{B}$; in other words, $\mathcal{T}$ is *upwardly closed* over $2^{\mathcal{S}}$.

In this case we say that the Turing machine $\mathcal{T}$ *decides* a set. Let us take liberty with the notation and write $\mathcal{A} \in \mathcal{T}$ if machine $\mathcal{T}$ returns *yes* on input $\mathcal{A}$, and $\mathcal{A} \notin \mathcal{T}$ otherwise.

Consider the pair $\langle\langle\mathcal{S},\mathcal{T}\rangle,\mathcal{A}\rangle$ such that $\mathcal{A}\subseteq\mathcal{S}$, and $\mathcal{A}$ is represented via an enumeration of its elements.

$$\langle\langle\mathcal{S},\mathcal{T}\rangle,\mathcal{A}\rangle\in\mathbf{MEL}\ \textit{iff}\ \mathcal{A}\subseteq\mathcal{S},\ \mathcal{A}\in\mathcal{T},\text{ and }\forall\mathcal{B}\subset\mathcal{A}.\ \mathcal{B}\notin\mathcal{T}.$$

It can be seen that **MEL** is **P**, in *polynomial time*. To evaluate whether $\mathcal{A}\in\mathcal{T}$ can be done in polynomial time. Each immediate subset of $\mathcal{A}$ must be tested too (and each must evaluate *false*). This is polynomial in total.

Definition 5. Define $\mathit{exists}(\mathcal{M})$ as follows:

$$\mathcal{X}\in\mathit{exists}(\mathcal{M})\ \textit{iff}\ \exists\mathcal{C}.\ \langle\mathcal{X},\mathcal{C}\rangle\in\mathcal{M}.$$

Here $\mathcal{C}$ is called a *certificate*, and certifies that $\mathcal{X}$ is in $\mathit{exists}(\mathcal{M})$.

We are interested in the class *exists*(**MEL**). Clearly *exists*(**MEL**) is **NP** since **MEL** is in **P**. In fact, *exists*(**MEL**) is **P** also, as demonstrated by the algorithm *a_mel* (Alg. 1) and Theorem 1.

Let us expand the notion of *exists* to that of *enumerate*. We are interested to know how difficult it is to identify a *number* of certificates of **MEL**, instead of just one, as with *exists*.

Definition 6. Define $\mathit{enumerate}(\mathcal{M},\ \mathbf{h})$ as follows:

$$\mathcal{X}\in\mathit{enumerate}(\mathcal{M},\ \mathbf{h})\ \textit{iff}\text{ there exist }\mathcal{C}_1,\ldots,\mathcal{C}_{\mathbf{h}(|\mathcal{X}|)}\text{ distinct certificates such that}$$
$$\bigwedge_{i=1}^{\mathbf{h}(|\mathcal{X}|)}\langle\mathcal{X},\mathcal{C}_i\rangle\in\mathcal{M}.$$

There are some problems for which the enumeration version is intractable, while the existence problem is not. **MEL** is such a problem. In the next section, it is shown that for any linear function **linear**, such that $\mathcal{O}(\mathbf{linear}(N))=\mathcal{O}(N)$, *enumerate*(**MEL**, **linear**) is **NP**-*complete*. In Section 5.3, it is shown that for any constant function **constant**, such that $\mathcal{O}(\mathbf{constant}(N))=\mathcal{O}(N^0)$, *enumerate*(**MEL**, **constant**) is **P**. (Notice that this is a stronger result than just the result that *exists*(**MEL**) is **P**.)

4.2 Enumeration is hard

We shall show that the enumeration problem for MELs is hard. The proof will encode a **CNF** propositional theory into a **MEL** enumeration problem. To determine whether an arbitrary **CNF** propositional theory is *satisfiable*, known as the problem **SAT**, is **NP**-*complete* [20]. These terms are defined as follows.

Definition 7. A propositional theory **C** is in **CNF** *form* when it is of the form $\mathcal{C}_1\wedge\ldots\wedge\mathcal{C}_k$. Each clause $\mathcal{C}_i$ is of the form $L_{1,i}\vee\ldots\vee L_{m_i,i}$. Each $L_{i,j}$ represents a given propositional variable or the negation thereof.

We shall consider a **CNF** theory as represented by a set of sets. Each inner-set represents a clause, a disjunction of the literals in the set. The set of sets then is considered to represent a conjunction of the clauses.

Call a propositional theory in which no propositional variable appears negated a *positive* theory.

Definition 8. Given a disjoint pair of sets of propositional variables $\langle\mathcal{T},\mathcal{F}\rangle$, let the *truth table* $\mathbf{T}_{\mathcal{T},\mathcal{F}}$ denote the following truth assignment:

For each p_i, assign p_i
true if $p_i\in\mathcal{T}$,
false if $p_i\in\mathcal{F}$, and

undefined otherwise.

$\mathbf{T}_{\mathcal{T},\mathcal{F}}$ is defined only when $\mathcal{T} \cap \mathcal{F} = \emptyset$. Let $\mathbf{C}$ be a propositional theory. Let $\mathbf{T}_{\mathcal{T},\mathcal{F}} \models \mathbf{C}$ denote that the truth table described by $\mathbf{T}_{\mathcal{T},\mathcal{F}}$ satisfies $\mathbf{C}$. $\mathbf{T}_{\mathcal{S}}$ is shorthand for $\mathbf{T}_{\mathcal{S},\emptyset}$.

Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ represent the set of *propositional variables*[13] that appear in $\mathbf{C}$. Call $\mathcal{M}$, a subset of $\mathcal{P}$, a *model* of $\mathbf{C}$ *iff* $\mathbf{T}_{\mathcal{M},\mathcal{P}-\mathcal{M}} \models \mathbf{C}$. Let $\mathcal{M}$ *models* $\mathbf{C}$, or $\mathcal{M} \models \mathbf{C}$, be shorthand for this.

A **CNF** propositional theory, $\mathbf{C}$, is in **SAT** (is *satisfiable*) *iff* there exists a truth assignment that satisfies it; that is, there exists an $\mathcal{M}$ such that $\mathcal{M} \models \mathbf{C}$.

The tests we consider for MEL problems are upwardly closed. Unfortunately, models of **CNF** theories are not upwardly closed. That is, the fact that $\mathbf{T}_{\mathcal{M}}$ models $\mathbf{C}$ does not imply that $\mathbf{T}_{\mathcal{N}}$ models $\mathbf{C}$, given $\mathcal{N} \supset \mathcal{M}$. Therefore we shall consider only *positive* **CNF** theories, for which the property of upward closure does hold. This is the inherent reason why *enumerate*(**MEL, linear**) is **NP** while *exists*(**MEL**) is **P**.

Definition 9. Define the following transformation of a **CNF** propositional theory into a positive **CNF** propositional theory.

Let $p_1, \ldots, p_n$ be the propositional variables of the theory $\mathbf{C}$. Introduce new propositional variables $p_{n+1}, \ldots, p_{2n}$. For each negative occurrence of p_i in a clause in $\mathbf{C}$, replace it by a positive occurrence of p_{n+i}. Call the resulting positive theory $pos(\mathbf{C})$.

Lemma 10. Let $\mathbf{C}$ be a positive propositional theory. If $\mathbf{T}_{\mathcal{A}} \models \mathbf{C}$ and $\mathcal{A} \subset \mathcal{B}$, then $\mathbf{T}_{\mathcal{B}} \models \mathbf{C}$. In other words, $\{\mathcal{M} \mid \mathcal{M} \models \mathbf{C}\}$ is upwardly closed.

Proof. Assume $\mathbf{T}_{\mathcal{A}} \models \mathbf{C}$ and $\mathcal{A} \subset \mathcal{B}$. For each clause in $\mathbf{C}$ there exists at least one atom in the clause that is in $\mathcal{A}$. This assures the *truth* of the clause. Each such atom is in $\mathcal{B}$ too, so $\mathbf{T}_{\mathcal{B}} \models \mathbf{C}$ by the same argument. □

Theorem 11. For any given function **linear** such that $\mathcal{O}(\mathbf{linear}(N)) = \mathcal{O}(N)$, there exists a polynomial $\mathbf{t}$ such that *enumerate*(**MEL, linear**), restricted over the domain of pairs $\langle \mathcal{S}, \mathcal{T} \rangle$ such that $|\mathcal{T}| \leq \mathbf{t}(|\mathcal{S}|)$, is **NP**-*complete*.

Proof. Consider the tuple $\langle \mathcal{S}, \mathcal{T} \rangle$. (Assume $|\mathcal{S}|$ is even, without loss of generality.)

We show *enumerate*(**MEL, linear**) is **NP**.

A certificate $\mathcal{A}$ such that $\langle\langle \mathcal{S}, \mathcal{T} \rangle, \mathcal{A}\rangle \in \mathbf{MEL}$ is polynomial to verify. Check $\mathcal{A} \in \mathcal{T}$ and $\forall e \in \mathcal{A}.\ \mathcal{A} - \{e\} \notin \mathcal{T}$. Since $\mathcal{T}$ is polynomial, this can be checked in polynomial time. There are only $\mathcal{O}(N)$ of these $\mathcal{A}$'s to check, so the total check is polynomial.

A reduction of **SAT** to *enumerate*(**MEL, linear**) is presented.

Let $\mathbf{C}$ be a **CNF** propositional theory and $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the set of its propositional variables, where $n = |\mathcal{S}|/2$. Without loss of generality, the size of $\mathbf{C}$ can be restricted to within a constant factor of n.

Assume, without loss of generality, that for all p_i, $\mathbf{T}_{\{p_i\},\emptyset} \not\models \mathbf{C}$ and $\mathbf{T}_{\emptyset,\{p_i\}} \not\models \mathbf{C}$. Also assume $\mathbf{T}_{\emptyset,\mathcal{P}} \not\models \mathbf{C}$. These constitute $2n+1$ truth assignments to pre-check, hence this subproblem of **SAT** is still **NP**-*complete*.

The following tuple $\langle \mathcal{S}, \mathcal{T} \rangle$ is constructed with respect to $pos(\mathbf{C})$.

Let $\mathcal{S} = \{p_1, \ldots, p_{2n}\}$.

[13] Propositional variables are also called *atoms* in the text.

Let $\mathbf{C}_i = (p_i \wedge p_{n+i})$ for $1 \leq i \leq n$. Each of these is a positive propositional theory too.

Let $\mathcal{T}$ be a Turing machine that decides the set $\{\mathcal{A} \mid \mathbf{T}_\mathcal{A} \models pos(\mathcal{S}) \vee \mathbf{T}_\mathcal{A} \models \mathbf{C}_1 \vee \ldots \vee \mathbf{T}_\mathcal{A} \models \mathbf{C}_n\}$.

The combined test above as constructed is within a fixed polynomial size with respect to the size of $\mathbf{C}$, the input problem from **SAT**. The size of $\mathcal{S}$ is within a fixed polynomial size with respect to the size of $\mathbf{C}$. Thus, the transformation from $\mathbf{C}$ to $\langle \mathcal{S}, \mathcal{T} \rangle$ preserves the size of the input. It can be also assumed that the Turing machine $\mathcal{T}$, which decides the test, is smaller than $\mathbf{t}(|\mathcal{S}|)$, without loss of generality.

Each individual test ($\{\mathcal{A} \mid \mathbf{T}_\mathcal{A} \models \mathbf{C}_i\}$, $0 \leq i \leq n$), is monotonic over $\mathcal{S}$ by Lemma 10. Hence, the union of these tests, $\mathcal{T}$, is monotonic over $\mathcal{S}$.

We show $\langle \mathcal{S}, \mathcal{T} \rangle \in enumerate(\mathbf{MEL},\ n+1) \implies \mathbf{C} \in \mathbf{SAT}$.

If $\langle \mathcal{S}, \mathcal{T} \rangle \in enumerate(\mathbf{MEL},\ n+1)$, then one of the $n+1$ answers must have passed because it models $pos(\mathbf{C})$, since there are only n possible answers that can model $\mathbf{C}_1$ through $\mathbf{C}_n$; namely, the sets $\{p_1, p_{n+1}\}, \ldots, \{p_n, p_{2n}\}$. Let $\mathcal{A}$ be this answer, so $\mathbf{T}_A \models pos(\mathbf{C})$. Note that $\nexists i.\ p_i, p_{n+i} \in \mathcal{A}$. Otherwise, $\mathcal{A}$ would not be minimal. Given this $\mathcal{A}$, a truth assignment is constructed that will satisfy $\mathbf{C}$. Let $\mathcal{A}^+ = \{p_i \in \mathcal{P} \mid p_i \in \mathcal{A}\}$. Let $\mathcal{A}^- = \{p_i \in \mathcal{P} \mid p_{n+i} \in \mathcal{A}\}$. The truth assignment $\mathbf{T}_{\mathcal{A}^+, \mathcal{A}^-}$ is defined since $\mathcal{A}^+ \cap \mathcal{A}^- = \emptyset$. $\mathbf{T}_{\mathcal{A}^+, \mathcal{A}^-} \models \mathbf{C}$.

The direction $\mathbf{C} \in \mathbf{SAT} \implies \langle \mathcal{S}, \mathcal{T} \rangle \in enumerate(\mathbf{MEL},\ n+1)$ follows in a similar manner. □

The above theorem was devised independently to address the MFS problem. It should be noted that a similar theorem for a given class of abduction problems is presented by Bylander et al. in [2]. The proof of that theorem follows in a very similar manner.

The reason we restrict the domain of **MEL** in the above theorem to pairs $\langle \mathcal{S}, \mathcal{T} \rangle$ such that $|\mathcal{T}| \leq \mathbf{t}(|\mathcal{S}|)$, is to show that the MEL enumeration problem is intractable with respect to the size of the input set $\mathcal{S}$, not just with respect to the size of the input set $\langle \mathcal{S}, \mathcal{T} \rangle$. Naturally, that the more general case of **MEL**, not restricted over this domain, is **NP**-*complete* follows.

4.3 Enumerating MFSs, MSSs, and MCSs is also hard

4.3.1 MFSs

It is easy to see how the MFS problem can be mapped to the MEL problem discussed above. The top element in the lattice is the (conjunctive) query. Given the query consists of N literals, the lattice is a complete 2^N boolean lattice. Our test is to evaluate the query against the database. Call this test based on the database $\mathcal{E}_{DB}$. Consider a query $\mathcal{Q}$.

$$\mathcal{Q} \in \mathcal{E}_{DB} \textit{ iff } \mathcal{Q} \text{ evaluates to the empty set against } \mathbf{DB}.$$

This $\mathcal{E}_{DB}$ is monotonic with respect to subset. If a subquery evaluates to the empty set, the query must too.

Algorithm *a_mel_fast* (Alg. 3) demonstrates how an MFS can be found for a query, asking N queries to the database. However, it has not been shown that finding $\mathcal{O}(N)$ MFSs for a query is as hard as **MEL**. One might imagine that there can be a smarter algorithm for finding MFSs than there can be for MELs, through some intelligent exploitation of the database. It shall be shown that there cannot be. To do this, it is necessary to show that sufficiently *hard* databases exist. The proof of theorem 11 can be modified to show an encoding into relational tables.

Definition 12. The set **MFS** is defined as follows:

$$\mathbf{MFS} = \{\langle\langle \mathcal{Q}, \mathcal{E}_{DB}\rangle, \mathcal{M}\rangle \mid \mathcal{M} \subseteq \mathcal{Q} \text{ and } \mathcal{Q} \text{ has the MEL } \mathcal{M} \text{ with respect to } \mathcal{E}_{DB}\}$$

Theorem 13. For any given function **linear** such that $\mathcal{O}(\mathbf{linear}(N)) = \mathcal{O}(N)$, *enumerate*(**MFS**, **linear**) is **NP**-*hard.*

Proof. The proof is constructed in the same manner as Proof 11. The difference is that $\mathcal{E}_{DB}$ needs to be constructed instead of $\mathcal{T}$.

Let $\mathbf{C}$ be a **CNF** propositional theory and $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the set of propositional variables appearing in $\mathbf{C}$. As before, assume, without loss of generality, that for all p_i, $\mathbf{T}_{\{p_i\},\emptyset} \not\models \mathbf{C}$ and $\mathbf{T}_{\emptyset,\{p_i\}} \not\models \mathbf{C}$. Also assume $\mathbf{T}_{\emptyset,\mathcal{P}} \not\models \mathbf{C}$.

Consider $pos(\mathbf{C})$. The following $\mathcal{E}_{DB}$ is constructed: for each atom p_i for $1 \leq i \leq 2n$ define a binary database relation. Call these relations by the names p_i also.

Next, **DB** is constructed by defining its tables, which tuples are in each relation.

- Assume $pos(\mathbf{C})$ has k clauses, $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$.
- For each relation $p_i \in \{p_1, \ldots, p_n\}$ construct its table in the following way:
  for each clause $\mathcal{C}_j \in \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$
    if the atom corresponding to the relation's name does not appear in clause $\mathcal{C}_j$, insert the tuple $\langle j, \text{positive}\rangle$ in the table.
    (Thus, $p_i(j, \text{positive})$ is considered *true*.)
- Likewise, for each relation $p_i \in \{p_{n+1}, \ldots, p_{2n}\}$ construct its table in the following way:
  for each clause $\mathcal{C}_j \in \{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$
    if the atom corresponding to the relation's name does not appear in clause $\mathcal{C}_j$, insert the tuple $\langle j,\ \text{negative}\rangle$ in the table.
- These are the only entries in each table.

Let the query $\mathcal{Q}$ be

$$\leftarrow p_1(X, Y_1),\ \ldots,\ p_n(X, Y_n),\ p_{n+1}(X, Y_1),\ \ldots,\ p_{2n}(X, Y_n).$$

$\mathcal{Q}$ is of length $2n$ literals. Note that $\mathcal{Q}$ is empty. It contains a $p_i(X, Y_i)$ and a $p_{n+i}(X, Y_i)$. These cannot join over Y_i, given the tables as constructed.

No subquery of $\mathcal{Q}$ of length one is empty, since no single atom is in every clause. Otherwise, there would be a truth assignment satisfying $\mathbf{C}$ of the form $\mathbf{T}_{\{p_i\},\emptyset} \models \mathbf{C}$ or $\mathbf{T}_{\emptyset,\{p_i\}} \models \mathbf{C}$. By assumption, there are no such assignments. Thus, every subquery of the form $\langle \leftarrow p_i(X, Y_i), p_{n+i}(X, Y_i).\rangle$ is a MFS. There are n of these MFSs.

We show $\langle \mathcal{Q}, \mathcal{E}_{DB}\rangle \in$ *enumerate*(**MFS**, **linear**) $\Longrightarrow$ $\mathbf{C} \in \mathbf{SAT}$.

If $\langle \mathcal{Q}, \mathcal{E}_{DB}\rangle \in$ *enumerate*(**MFS**, **linear**), then one of the $n+1$ answers must have passed because it models $pos(\mathbf{C})$. Call this MFS $\mathcal{Q}'$.

If a subset $\mathcal{Q}''$ of $\mathcal{Q}$ has some answer, say $X = j$, it can only be because for any atom which would satisfy $\mathcal{C}_j$, the corresponding literal in $\mathcal{Q}$ is missing from $\mathcal{Q}''$. Therefore, since $\mathcal{Q}'$ has no answers, for each $\mathcal{C}_i$, there is a literal in $\mathcal{Q}'$ for which the corresponding atom satisfies $\mathcal{C}_i$. So for each literal in $\mathcal{Q}'$, assign the corresponding atom the value *true*. This constitutes a model of $pos(\mathbf{C})$. This implies there is a model of $\mathbf{C}$, as before.

The direction $\mathbf{C} \in \mathbf{SAT} \Longrightarrow \langle \mathcal{Q}, \mathcal{E}_{DB}\rangle \in$ *enumerate*(**MFS**, **linear**) follows in a similar manner. □

4.3.2 MSSs

It may seem at first consideration that once one knew the minimally failing subqueries MFSs of a query, there would be a simple procedure to determine the maximally succeeding subqueries (MSSs) of the query. Of course, any subquery of an MFS succeeds, by definition. However, there is no reason to assume that these subqueries will be maximal. Quite surprisingly, there does not appear to be any direct procedure to determine the MSSs from the MFSs.[14] Knowing the MFSs does not help to ascertain easily the MSSs, or vice versa.

The MSS problem, however, can also be reduced to the MEL problem, and solved accordingly. Let query $\mathcal{Q}$ be the query for which MSSs are to be found. Construct the test $\mathcal{N}_{DB}$ as follows:

$$\mathcal{N}_{DB} = \{\mathcal{S} \subseteq \mathcal{Q} \mid \mathbf{DB} \vdash \exists.(\mathcal{Q} - \mathcal{S})\}$$

The test $\mathcal{N}_{DB}$ is monotonic with respect to subset, for the same reason that the test $\mathcal{E}_{DB}$ from above is monotonic. The MELs found for set $\mathcal{Q}$ and test $\mathcal{N}_{DB}$ (for a database $\mathbf{DB}$) will be the inverses (with respect to $\mathcal{Q}$) of the query $\mathcal{Q}$'s MSSs. Thus, an MEL search can be used for finding MSSs.

It is to be proven that the search for MSSs is as difficult as the search for MELs. As in the case of MFSs, it is necessary to show that there are sufficiently hard databases for the MSS search.

Definition 14. The set **MSS** is defined as follows:

$$\begin{aligned}\mathbf{MSS} = \{\langle\langle\mathcal{Q}, \mathcal{N}_{DB}\rangle, \mathcal{M}\rangle \mid\ & \mathcal{M} \subseteq \mathcal{Q} \text{ and} \\ & \mathcal{Q} \text{ has the MEL } \mathcal{M} \text{ with respect to } \mathcal{N}_{DB}\}\end{aligned}$$

Theorem 15. For any given function **linear** such that $\mathcal{O}(\mathbf{linear}(N)) = \mathcal{O}(N)$, *enumerate*(**MSS**, **linear**) is **NP**-*hard.*

Proof. The proof is constructed in the same manner as Proof 11. The difference is that $\mathcal{N}_{DB}$ needs to be constructed instead of $\mathcal{T}$.

Let $\mathbf{C}$ be a **CNF** propositional theory and $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the set of propositional variables appearing in $\mathbf{C}$. As before, assume, without loss of generality, that for all p_i, $\mathbf{T}_{\{p_i\},\emptyset} \not\models \mathbf{C}$ and $\mathbf{T}_{\emptyset,\{p_i\}} \not\models \mathbf{C}$. Also assume $\mathbf{T}_{\emptyset,\mathcal{P}} \not\models \mathbf{C}$.

Consider $pos(\mathbf{C})$. Let $\mathcal{P}' = \{p_1, \ldots, p_{2n}\}$, the propositional variables of $pos(\mathbf{C})$. $\mathbf{DB}$ is constructed by defining its tables, which tuples are in each relation.

- Assume $pos(\mathbf{C})$ has k clauses, $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$.
- For each clause $\mathcal{C}_i$ in $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$
  For each $p_j \in (\mathcal{P}' - \mathcal{C}_i)$
  For each $q \in (\mathcal{C}_i - \{p_j\}))$
  add $\langle j, i\rangle$ to table q'
- Construct the following view for each $p_i \in \mathcal{P}$:
  Let $\{\mathcal{C}_{s_{i,1}}, \ldots, \mathcal{C}_{s_{i,r_i}}\}$, $\{s_{i,1}, \ldots, s_{i,r_i}\} \subseteq \{1, \ldots, k\}$, be the set of clauses in which p_i appears.
  The table will be $(r_i + 1)$-ary.
  $p_i(X_i, \ldots, X_{r_i}, 0) \leftarrow p_i'(X_1, 1), \ldots, p_i'(X_{r_i}, r_i)$.
  For each $j \in (\{1, \ldots 2n\} - \{i, (n+i)\})$
  Add $p_i(0, \ldots, 0, j)$
- These are the only entries in each table.

Let the query $\mathcal{Q}$ be

$$\leftarrow p_1(X_{s_{1,1}}, \ldots X_{s_{1,r_1}}, Y), \ldots, p_{2n}(X_{s_{2n,1}}, \ldots X_{s_{2n,r_{2n}}}, Y).$$

[14] This conversion is shown to be **NP**-*hard* in Section 6.1.

$\mathcal{Q}$ is of length $2n$ literals.

We show $\langle \mathcal{Q}, \mathcal{N}_{DB} \rangle \in$ *enumerate*(**MSS**, **linear**) $\Longrightarrow$ $\mathbf{C} \in \mathbf{SAT}$.

The answer tuple of $\mathcal{Q}$ is of the form $\langle X_1, \ldots, X_k, Y \rangle$.

Exactly n of the MSSs of $\mathcal{Q}$ are of the form $\mathcal{Q} - \{p_i(\ldots), p_{n+i}(\ldots)\}$. Such a subquery has a non-empty answer set: namely, the tuple $\langle 0, \ldots, 0, i \rangle$ is in the answer set by construction.

One of the $n+1$ MSSs must be of a different form, where $Y = 0$. Call this MSS $\mathcal{Q}'$. The inverse of this query cannot contain both a p_i and a p_{n+i} literal; if it did, it would be a subquery of some $\mathcal{Q} - \{p_i(\ldots), p_{n+i}(\ldots)\}$, but it is known that this subquery is maximal.

For $i \in \{1, \ldots, n\}$, if $p_i(\ldots) \notin \mathcal{Q}'$, then $p_{n+i}(\ldots) \in \mathcal{Q}'$, and if $p_{n+i}(\ldots) \notin \mathcal{Q}'$, then $p_i(\ldots) \in \mathcal{Q}'$. So an answer tuple for $\mathcal{Q}'$ will also be of the form $\langle X_1, \ldots, X_k, Y \rangle$.

Consider a tuple in the answer set of $\mathcal{Q}'$. For each $i \in \{1, \ldots k\}$, check X_i's value in the answer tuple. Say its value is j. Assigning p_j the value *true* satisfies clause i in *pos*($\mathbf{C}$), by construction. Note that $p_j(\ldots) \notin \mathcal{Q}'$; otherwise X_i could not have value j by construction. Thus each clause is satisfied, and there exists a model for *pos*($\mathbf{C}$). The model found for *pos*($\mathbf{C}$) can be directly translated into a model for $\mathbf{C}$.

The direction $\mathbf{C} \in \mathbf{SAT} \Longrightarrow \langle \mathcal{Q}, \mathcal{N}_{DB} \rangle \in$ *enumerate*(**MSS**, **linear**) follows in a similar manner. □

4.3.3 MCSs

Any linear enumeration of MCSs of a query will necessarily be **NP**-*hard.*

Say that a query *leads necessarily to failure* whenever a specially designated predicate called *contradiction* (written often as $\bot$) is derivable, assuming an answer tuple to the query. The MCS problem then reduces to an *abduction* problem: find a smallest *set-of-support*—a minimal subset of a base set of facts—which support a given conclusion.

Definition 16. The set **MCS** is defined as follows:

$$\mathbf{MCS} = \{\langle\langle \mathcal{Q}, \mathcal{T} \rangle, \mathcal{M}\rangle \mid \mathcal{M} \subset \mathcal{Q} \text{ and } \mathcal{T} \cup \mathcal{M}\theta \vdash \bot\}$$

where $\mathcal{T}$ is a theory written in some given logic, $\bot$ is an atom appearing in $\mathcal{T}$, $\vdash$ is some given monotonic proof theory applicable to the logic, and θ grounds $\mathcal{M}$ with constants that do not appear in $\mathcal{T}$.

Such abduction problems have been explored and classified. (See [2].) We present a proof here for the edification of the reader.

Theorem 17. For any given function **linear** such that $\mathcal{O}(\textbf{linear}(N)) = \mathcal{O}(N)$, *enumerate*(**MCS**, **linear**) is **NP**-*hard.*

Proof. The proof is constructed in the same manner as Proof 11.

Let $\mathbf{C}$ be a **CNF** propositional theory and $\mathcal{P} = \{p_1, \ldots, p_n\}$ be the set of propositional variables appearing in $\mathbf{C}$. As before, assume, without loss of generality, that for all p_i, $\mathbf{T}_{\{p_i\},\emptyset} \not\models \mathbf{C}$ and $\mathbf{T}_{\emptyset,\{p_i\}} \not\models \mathbf{C}$. Also assume $\mathbf{T}_{\emptyset,\mathcal{P}} \not\models \mathbf{C}$.

Consider *pos*($\mathbf{C}$). Let $\mathcal{P}' = \{p_1, \ldots, p_{2n}\}$, the propositional variables of *pos*($\mathbf{C}$). A theory $\mathcal{T}$ is constructed based on *pos*($\mathbf{C}$).

- Assume $pos(\mathbf{C})$ has k clauses, $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$.
- For each clause $\mathcal{C}_i$ in $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$
  Introduce a new propositional variable c_i.
  For each $p_j \in \mathcal{C}_i$
  Add the rule $c_i \leftarrow p_j$.
- Add the rule $\bot \leftarrow c_1, \ldots, c_k$.
- For each $p_i \in \{1, \ldots, n\}$
  add the rule $\bot \leftarrow p_i, p_{n+i}$.

We show $\langle \mathcal{P}', \mathcal{T} \rangle \in enumerate(\mathbf{MCS}, \mathbf{linear}) \Longrightarrow \mathbf{C} \in \mathbf{SAT}$.

At most, there are n MCSs of $\mathcal{P}'$ of the form $\{p_i, p_{n+i}\}$. Thus one of the $n+1$ MCSs contains at most either p_i or p_{n+i} for $i \in \{1, \ldots, n\}$, and $\bot$ is derivable from the MCS set. This means each of $c_1, \ldots, c_k$ is derivable. Consider the rule by which c_i can be derived: it must be of the form $\langle c_i \leftarrow p_j . \rangle$ for some j. This p_j is in the MCS. Assign p_j *true*. The resulting model satisfies $pos(\mathbf{C})$. A model can be constructed directly from this which satisfies $\mathbf{C}$.

The direction $\mathbf{C} \in \mathbf{SAT} \Longrightarrow \langle \mathcal{P}', \mathcal{T} \rangle \in enumerate(\mathbf{MCS}, \mathbf{linear})$ follows in a similar manner. $\square$

5 An Enumeration Algorithm

5.1 Criteria for an Enumeration Algorithm

In Section 3, it is shown that the standard algorithm for MEL, *all_mels* in Alg. 2, is highly intractable for enumerating MELs. In Section 4, it is shown that to find $\mathcal{O}(N)$ MELs is, in principle, intractable. Still, there is much middle ground.

An enumeration algorithm for MELs should have the following properties:

1. finds subsequent MELs as quickly as is possible
   - runs and halts quickly if there are *a priori* few MELs
   - can be stopped early with partial results
     (some MELs were found)
2. exploits the decomposability of the MEL problem
3. is optimal with respect to the *test*
   (calls *test* only as often as is necessary)

It was shown that to find an MEL is within $\mathcal{O}(N)$. It can be shown that to find two MELs is still within $\mathcal{O}(N)$, to find three MELs is bounded by $\mathcal{O}(N^2)$, four by $\mathcal{O}(N^3)$, and so forth. (See Section 5.3). An enumeration algorithm should run within these bounds when enumerating MELs sequentially. The problem of finding the MELs of a set with respect to a test is decomposable: the MELs of any subset that passes the test are also MELs of the set. The time to find the MELs of a subset is diminished because the complexity factor is over a smaller input set. Finally, since the test may be expensive (in the case of MFSs, the test is to evaluate a query against a database), the number of calls to test should be minimized.

5.2 Factoring the Lattice

The problem with the search strategy of *all_mels* (Alg. 2) is that the same MEL is encountered repeatedly. This arises from the fact that a lattice is being searched. The algorithm *all_mels* searches the lattice as if it were a tree. A way around this problem might be to prune the search space (the lattice) after each MEL is found, so that the previously found MELs are not rediscovered.

One way to do this is as follows: let $\mathcal{D}$ be the collection of MELs already seen; the set of sublattices, $\mathcal{F}$, of the lattice, $\mathcal{L}$, should be found such that[15]

1. any MEL $\mathcal{N}$ of $\mathcal{L}$ such that $\mathcal{N} \notin \mathcal{D}$ is an MEL of some $\mathcal{S} \in \mathcal{F}$, and
2. for all $\mathcal{M} \in \mathcal{D}$, $\mathcal{M}$ is not an MEL of any $\mathcal{S} \in \mathcal{F}$.

If it is possible to construct such a set of sublattices, then it is only necessary to explore each sublattice to find the rest of the MELs, with no danger of rediscovering any previously seen MEL. This construction will be possible; let us call this operation *factoring* the lattice.

Definition 18. Define that $\mathcal{A}$ and $\mathcal{B}$ are *incomparable iff* $\mathcal{A} \not\supseteq \mathcal{B}$ and $\mathcal{A} \not\subseteq \mathcal{B}$.

Define *factors*$(\mathcal{L}, \mathcal{D})$ as follows:

$$factors(\mathcal{L}, \mathcal{D}) = \{\mathcal{A} \subseteq \mathcal{L} \mid \forall \mathcal{S} \in \mathcal{D}.\ \ \mathcal{A} \text{ and } \mathcal{S} \text{ are incomparable}\}$$

$\mathcal{A}$ is a *factor* of $\mathcal{L}$ with respect to $\mathcal{D}$ *iff* $\mathcal{A} \in factors(\mathcal{L}, \mathcal{D})$.

Define *max_factors*$(\mathcal{L}, \mathcal{D})$ as follows:

$$max\_factors(\mathcal{L}, \mathcal{D}) = \{\mathcal{A} \in factors(\mathcal{L}, \mathcal{D}) \mid \neg\exists \mathcal{S} \in factors(\mathcal{L}, \mathcal{D}).\ \mathcal{S} \subset \mathcal{A}\}$$

$\mathcal{A}$ is a *maximal factor* of $\mathcal{L}$ with respect to $\mathcal{D}$ *iff* $\mathcal{A} \in max\_factors(\mathcal{L}, \mathcal{D})$.

Theorem 19. Let $\mathcal{D}$ be a collection of MELs of $\mathcal{L}$. Let $\mathcal{M}$ be any MEL of $\mathcal{L}$ not in $\mathcal{D}$. Then $\mathcal{M}$ is an MEL of some set in *max_factors*$(\mathcal{L}, \mathcal{D})$.

Proof. For each $\mathcal{S} \in \mathcal{D}$, $\mathcal{M}$ and $\mathcal{S}$ are incomparable since both sets $\mathcal{M}$ and $\mathcal{S}$ are MELs. Therefore, $\mathcal{M}$ is a factor of $\mathcal{L}$ with respect to $\mathcal{D}$, by definition.

If $\mathcal{M}$ is a maximal factor, the proof is complete; else there must be some $\mathcal{N} \in max\_factors(\mathcal{L},\mathcal{D})$ such that $\mathcal{N} \supset \mathcal{M}$. (Otherwise, $\mathcal{M}$ would be a maximal factor after all.) $\mathcal{N} \in \mathcal{T}$ since $\mathcal{M} \in \mathcal{T}$. So, $\mathcal{M}$ is an MEL of $\mathcal{N}$. □

By construction, no previously seen MELs, $\mathcal{D}$, are contained in any of the sets of *max_factors*$(\mathcal{L},\mathcal{D})$.

The approach to enumerate MELs is now as follows. Say that $\mathcal{D}$ is the collection of MELs already known. Find a set $\mathcal{S}$ such that $\mathcal{S} \in max\_factors(\mathcal{L},\mathcal{D})$ and $\mathcal{S} \in \mathcal{T}$. Once such an $\mathcal{S}$ is known, it takes $|\mathcal{S}|$ steps to find the next MEL, employing *a_mel_fast* (Alg. 3).

Even if factoring were simple to do, it would not contradict the results of Theorem 11. This is because there may be many (maximal) factors for any given $\langle \mathcal{L}, \mathcal{D} \rangle$. In fact, it can be shown that the number of possible factors is on the order of $\mathcal{O}(|\mathcal{L}|^{|\mathcal{D}|})$. So, to find a factor that satisfies $\mathcal{T}$ might require that one examine an exponential number of factors that do not satisfy $\mathcal{T}$ first.

Factoring, however, is intractable. It will be shown to be **NP**-*complete* with respect to $|\mathcal{D}|$. Note, however, that the intractability is with respect to the number of previously seen MELs, not with respect to the size of the lattice.

5.2.1 Complexity of Factoring the Lattice

Definition 20. Define the class **SET-SAT**, *set saturation*, as follows. Given a set $\mathcal{L}$ and a collection $\mathcal{D}$ of subsets of $\mathcal{L}$, then $\langle \mathcal{L}, \mathcal{D} \rangle \in$ **SET-SAT** *iff* $|factor(\mathcal{L}, \mathcal{D})| = 0$.

[15] The lattices are represented via their top elements, as sets.

This is a set-theoretic abstraction of the factoring problem introduced in the previous section. **SET-SAT** does not appear in Garey and Johnson's catalog of **NP**-*complete* problems [20], or elsewhere to the author's knowledge. Thus a proof is provided here.

Theorem 21. **SET-SAT** is *co***NP**-*complete.*

Proof. By reduction of **SAT** to ¬**SET-SAT**.

We show **SET-SAT** is *co***NP**.

$\langle \mathcal{L}, \mathcal{D} \rangle \notin$ **SET-SAT** *iff* $\exists \mathcal{S} \subseteq \mathcal{L}.\ \mathcal{S} \in factor(\mathcal{L},\ \mathcal{D})$. The size of $\mathcal{S}$ is at most the size of $\mathcal{L}$.

Given an arbitrary **CNF** propositional theory $\mathcal{C}$ as a set of clauses $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$, with each clause represented as a set of propositional variables (and negated propositional variables), the following transformation is made:

- Let $\mathcal{P}$ be the set of the propositional variables which appear in $\mathcal{C}$.
- Construct $\mathcal{A} = \{a_1, \ldots, a_k\}$ such that $\mathcal{A} \cap \mathcal{P} = \emptyset$.
- For each $\mathcal{C}_i$, $i \in \{1, \ldots, k\}$
  Separate the positive occurrences of propositional variables in $\mathcal{C}_i$ into $\mathcal{C}_i^+$ and the negative occurrences into $\mathcal{C}_i^-$.
  $\mathcal{S}_i^+ = (\mathcal{P} - \mathcal{C}_i^+) \cup (\mathcal{A} - \{a_i\})$
  $\mathcal{S}_i^- = \mathcal{C}_i^- \cup \{a_i\}$

Let $\mathcal{L} = \mathcal{P} \cup \mathcal{A}$. Let $\mathcal{D} = \{\mathcal{S}_i^+, \mathcal{S}_i^- \mid i \in \{1, \ldots, k\}\}$.

We show $\langle \mathcal{L}, \mathcal{D} \rangle \notin$ **SET-SAT** $\Longrightarrow \mathcal{C} \in$ **SAT**.

There is an $\mathcal{I} \subseteq \mathcal{L}$ such that for all $\mathcal{S} \in \mathcal{D}$, $\mathcal{I} \not\subset \mathcal{S}$ and $\mathcal{I} \not\supset \mathcal{S}$. Let $\mathcal{M} = \mathcal{I} \cap \mathcal{P}$.

For each $i \in \{1, \ldots, k\}$, consider $\mathcal{S}_i^+$. $\mathcal{I} - \mathcal{S}_i^+ \neq \emptyset$. Choose $p \in (\mathcal{I} - \mathcal{S}_i^+)$. Then $p \in \mathcal{C}_i^+ \cup \{a_i\}$. Cases:

1. $p \in \mathcal{C}_i^+$. Then $\mathcal{M}$ models $\mathcal{C}_i$. (Recall definition 8.)
2. $p = a_i$. Consider $\mathcal{C}_i^-$. $\mathcal{C}_i^- - \mathcal{I} \neq \emptyset$. Choose $q \in (\mathcal{C}_i^- - \mathcal{I})$. Note that $q \neq a_i$, since $a_i \in \mathcal{I}$. So q appears as a negated propositional variable in $\mathcal{C}_i$, but not in $\mathcal{I}$. Therefore, $\mathcal{M}$ models $\mathcal{C}_i$.

$\mathcal{M} \models \mathcal{C}$.

We now show the other direction $\mathcal{C} \in$ **SAT** $\Longrightarrow \langle \mathcal{L}, \mathcal{D} \rangle \notin$ **SET-SAT**.

$\mathcal{C}$ has a model. Thus there is a set $\mathcal{M} \subseteq \mathcal{P}$ such that $\mathcal{M} \models \mathcal{C}$.

Let $\mathcal{B} = \{a_i \in \mathcal{A} \mid \mathcal{C}_i^+ \cap \mathcal{M} = \emptyset\}$. Let $\mathcal{I} = \mathcal{M} \cup \mathcal{B}$.

For each $\mathcal{S}_i^+$, $\mathcal{I} \not\subseteq \mathcal{S}_i^+$ and $\mathcal{I} \not\supseteq \mathcal{S}_i^+$. Cases:

1. $\mathcal{C}_i^+ \cap \mathcal{M} \neq \emptyset$. $\exists p \in (\mathcal{C}_i^+ \cap \mathcal{M})$. $p \notin \mathcal{S}_i^+$. Therefore, $\mathcal{I} \not\subseteq \mathcal{S}_i^+$.
   Without loss of generality, say that $\exists a_j.\ a_j \neq a_i \wedge a_j \notin \mathcal{I}$. However, $a_j \in \mathcal{S}_i^+$. So $\mathcal{S}_i^+ \not\subseteq \mathcal{I}$.
2. $\mathcal{C}_i^+ \cap \mathcal{M} = \emptyset$. So $a_i \in \mathcal{I}$, but $a_i \notin \mathcal{S}_i^+$. Therefore, $\mathcal{I} \not\subseteq \mathcal{S}_i^+$.
   Again, without loss of generality, say that $\exists a_j.\ a_j \neq a_i \wedge a_j \in \mathcal{I}$. So $\mathcal{S}_i^+ \not\subseteq \mathcal{I}$.

For each $\mathcal{S}_i^-$, $\mathcal{I} \not\subseteq \mathcal{S}_i^-$ and $\mathcal{I} \not\supseteq \mathcal{S}_i^-$. Cases:

1. $\mathcal{C}_i^+ \cap \mathcal{M} \neq \emptyset$. $\exists p \in (\mathcal{C}_i^+ \cap \mathcal{M})$. $p \notin \mathcal{C}_i^-$, without loss of generality. Therefore, $\mathcal{I} \not\subseteq \mathcal{S}_i^-$.
   $a_i \notin \mathcal{I}$, but $a_i \in \mathcal{S}_i^-$. So $\mathcal{S}_i^- \not\subseteq \mathcal{I}$.
2. $\mathcal{C}_i^+ \cap \mathcal{M} = \emptyset$. Thus $\mathcal{C}_i^- - \mathcal{M} \neq \emptyset$ since $\mathcal{M}$ models $\mathcal{C}_i$. Therefore, $\mathcal{S}_i^- \not\subseteq \mathcal{I}$.
   $\mathcal{I} \not\subseteq \mathcal{S}_i^-$ since, without loss of generality, say that $\exists a_j.\ a_j \neq a_i \wedge a_j \in \mathcal{I}$.

For every $\mathcal{S} \in \mathcal{D}$, $\mathcal{I} \not\subseteq \mathcal{S}$ and $\mathcal{I} \not\supseteq \mathcal{S}$. □

Theorem 21 is too weak to imply that the factoring problem for MEL enumeration is **NP**-*complete*. This is because MELs are always *pairwise incomparable*—none is a subset of another. This is a subclass of **SET-SAT**. We show that this subclass is *co*$\mathbf{NP}$-*complete* too, employing the result of theorem 21 in the proof.

Definition 22. Define the class **INC-SET-SAT**, *incomparable set saturation*, as follows. Given a set $\mathcal{L}$ and a collection $\mathcal{D}$ of subsets of $\mathcal{L}$, such that the subsets in $\mathcal{D}$ are pairwise incomparable, then $\langle \mathcal{L}, \mathcal{D} \rangle \in$ **INC-SET-SAT** *iff* $\mathit{factor}(\mathcal{L}, \mathcal{D}) = \emptyset$.

Theorem 23. **INC-SET-SAT** is *co*$\mathbf{NP}$-*complete.*

Proof. By reduction of ¬**SET-SAT** to ¬**INC-SET-SAT**.

We show **INC-SET-SAT** is *co*$\mathbf{NP}$.

$\langle \mathcal{L}, \mathcal{D} \rangle \notin$ **INC-SET-SAT** *iff* $\exists \mathcal{S} \subseteq \mathcal{L}.\ \mathcal{S} \in \mathit{factor}(\mathcal{L}, \mathcal{D})$. The size of $\mathcal{S}$ is at most the size of $\mathcal{L}$.

Given the input $\mathcal{D}$ to **SET-SAT**, transform it as follows. Let $|\mathcal{D}| = k$. Let $\mathcal{E} = \{e_1, \ldots, e_k\}$ such that $\mathcal{E} \cap \mathcal{L} = \emptyset$.

- for each $\mathcal{S}_i \in \mathcal{D}$
  for each $j \in \{1, \ldots, k\}$
  $\mathcal{S}_{i,j} = \mathcal{S}_i \cup \{e_j\}$

Note that all the $\mathcal{S}_{i,j}$ are pairwise incomparable.

Let $\mathcal{D}' = \{\mathcal{S}_{i,j} \mid i, j \in \{1, \ldots, k\}\} \cup \{\{e_i, e_j\} \mid i, j \in \{1, \ldots, k\} \wedge i \neq j\} \cup \mathcal{L}$.

We show $\langle \mathcal{L}, \mathcal{D}' \rangle \in$ ¬**INC-SET-SAT** $\Longrightarrow$ $\langle \mathcal{L}, \mathcal{D} \rangle \in$ ¬**SET-SAT**.

$\exists \mathcal{I}'.\ \forall \mathcal{J} \in \mathcal{D}'.\ \mathcal{I}' \not\subseteq \mathcal{J} \wedge \mathcal{I}' \not\supseteq \mathcal{J}$.
$\mathcal{I}'$ contains exactly one of $e_i \in \mathcal{E}$. If it were to contain two, say e_i and e_j, then $\mathcal{I}' \supseteq \{e_i, e_j\}$. However, this set is in $\mathcal{D}'$, as constructed. If it were not to contain any, then $\mathcal{I}' \subseteq \mathcal{L}$. However, $\mathcal{L}$ is in $\mathcal{D}'$, as constructed.

Assume, without loss of generality, $e_1 \in \mathcal{I}'$. Let $\mathcal{I} = \mathcal{I}' - \{e_1\}$. Consider $\mathcal{S}_{i,1}$. $\mathcal{I}' - \mathcal{S}_{i,1} \neq \emptyset$, so choose $p \in (\mathcal{I}' - \mathcal{S}_{i,1})$. Note that $p \neq e_1$ because $e_1 \in \mathcal{S}_{i,1}$. Therefore, $p \in (\mathcal{I} - \mathcal{S}_i)$, so $\mathcal{I} \not\subseteq \mathcal{S}_i$.

$\mathcal{S}_{i,1} - \mathcal{I}' \neq \emptyset$. Choose $p \in (\mathcal{S}_{i,1} - \mathcal{I}')$. Note that $p \neq e_1$ since $p \in \mathcal{I}'$. Therefore, $p \in (\mathcal{S}_i - \mathcal{I})$, so $\mathcal{I} \not\supseteq \mathcal{S}_i$.

We now show the other direction $\langle \mathcal{L}, \mathcal{D} \rangle \in$ ¬**SET-SAT** $\Longrightarrow$ $\langle \mathcal{L}, \mathcal{D}' \rangle \in$ ¬**INC-SET-SAT**.

$\exists \mathcal{I}.\ \forall \mathcal{J} \in \mathcal{D}.\ \mathcal{I} \not\subseteq \mathcal{J} \wedge \mathcal{I} \not\supseteq \mathcal{J}$.
Let $\mathcal{I}' = \mathcal{I} \cup \{e_1\}$.

It is shown that each member of $\mathcal{D}'$ is incomparable with $\mathcal{I}'$:

- Consider any $\mathcal{S}_{i,1} \in \mathcal{D}'$.
  $\mathcal{S}_i - \mathcal{I} \neq \emptyset$. Choose $p \in (\mathcal{S}_i - \mathcal{I})$. Thus $p \in (\mathcal{S}_{i,1} - \mathcal{I}')$. Therefore, $\mathcal{I}' \not\supseteq \mathcal{S}_{i,1}$.
  $\mathcal{I} - \mathcal{S}_i \neq \emptyset$, so $\mathcal{I}' \not\subseteq \mathcal{S}_{i,1}$.
- Consider any $\mathcal{S}_{i,j} \in \mathcal{D}', j \neq 1$.
  Since $e_j \in \mathcal{S}_{i,j}$ and $e_j \notin \mathcal{I}'$, $\mathcal{I}' \not\supseteq \mathcal{S}_{i,j}$.
  Since $e_1 \notin \mathcal{S}_{i,j}$ and $e_1 \in \mathcal{I}'$, $\mathcal{I}' \not\subseteq \mathcal{S}_{i,1}$.
- Consider $\mathcal{L}$.
  Since $e_1 \notin \mathcal{L}$ and $e_1 \in \mathcal{I}'$, $\mathcal{I}' \not\subseteq \mathcal{L}$.

Assume, without loss of generality, $\exists p \in \mathcal{L}.\ p \notin \mathcal{I}$. Therefore, $p \notin \mathcal{I}'$, so $\mathcal{I}' \not\supseteq \mathcal{L}$.

- Consider any $\{e_i, e_j\} \in \mathcal{D}'$.
  Assume, without loss of generality, that $\mathcal{I}$ is not empty. Therefore $\mathcal{I}' \not\subseteq \{e_i, e_j\}$.
  $\mathcal{I}' \cap \mathcal{E} = \{e_1\}$, by construction. Thus $\mathcal{I}' \not\supseteq \{e_i, e_j\}$

$\forall \mathcal{S} \in \mathcal{D}'.\ \mathcal{I}' \not\subseteq \mathcal{S} \wedge \mathcal{I}' \not\supseteq \mathcal{S}.$ □

Clearly $\mathit{factors}(\mathcal{L}, \mathcal{D}) = \emptyset$ *iff* $\mathit{max\_factors}(\mathcal{L}, \mathcal{D}) = \emptyset$. Thus theorem 23 demonstrates that to find a new sublattice given a list of already seen MELs is inherently hard.

5.2.2 An Algorithm for Factoring

With the use of factoring, one shifts the computational workload in the enumeration of MELs from repeated search through the lattice to dividing the lattice into sublattices (factoring the lattice). Even though factoring is hard, this offers a computationally better approach to the enumeration problem. The factoring approach yields good performance for enumerating a number of MELs, with the performance decaying slowly. This is because the complexity of factoring depends on the size of $\mathcal{D}$, the factoring set. When enumerating MELs, $\mathcal{D}$ is the set of previously found MELs. Say that the factoring routine is called before each MEL search: on the first call to the factor routine, this is empty; on the next call, after a single MEL has be found, the size is one; and so forth. Thus, factoring does not become a computational bottle-neck until a reasonable number of MELs have been accumulated. Meanwhile, it was seen that the approach of repeated search through the lattice becomes intractable after the first MEL is found.[16]

The algorithm *factor* (Alg. 4) finds a maximal factor of the set *Top* with respect to the previously found MELs, stored in the array *mel* as in *all_mels* (Alg. 2). As written, any maximal factor it returns is also guaranteed to pass *test*.

The approach is as follows. Each previously seen MEL is considered via a recursive call to *factor*. If *Top* is a superset of the MEL, an element of *Top* which is also in the MEL is removed in order to *guarantee* that the new set being constructed is not a superset of that MEL. The new set is passed as *Top* on a recursive call to satisfy the rest of the MELs. The recursive invocations end when all the MELs have been checked, and the remaining set returned as the new factor. Thus *factor* guarantees that a factor returned is not a superset of, or equivalent to, any of the MELs.

It is also necessary for *factor* to guarantee that the set returned is not a subset of any of the MELs. This is accomplished by keeping a list of inverses, *Top* minus the current MEL each stage. At each invocation, if *factor* removes an element from *Top*, it also removes this element from all the inverses in the inverse list. If any become empty, this indicates that the factor set being constructed is a subset of one of the MELs; thus the routine *fails*—halts and returns *false*. The routine *reduce_invs* (Alg. 5) implements this routine.

If it were not required that the routine *factor* return a *maximal* factor, then it would be possible to dispense with the routines *justified* (Alg. 6) and *shift_justs*(Alg. 7). These routines are necessary to ensure that the factor is maximal. It is preferable that the routine *factor* find only maximal factors. Clearly, any MELs sought will be found under the maximal factors, so non-maximal ones need not be considered. Thus, only maximal factors should be returned for exploration. Each factor candidate is tested in *factor* via the test. The candidate is only returned if it passed.

A list of *justifications* is kept. There is a justification for each element that has been removed from *Top* towards constructing the factor set. The justification is a list of MELs which contain this element. Removing this element from *Top* is therefore *justified*, since it guarantees that the resulting set cannot be a superset of those MELs. If the justification for an element were empty, then its removal would not have been justified. Removing the element would still result in a valid factor, but it just would not be maximal.

[16] The search can be revised so a second MEL can be found before the search becomes intractable.

```
array mel(set)
integer Last

boolean factor(Top, Factor) {
    return factor(Top, Factor, 1, [], [])
}

boolean factor(Top, Factor, Index, Justs, Invs) {
    if Index ≤ Last then {
        Inv := Top − mel(Index)
        if Inv ≠ [] then
            if justified(mel(Index), Justs, Justs_Mel) then
                return factor(Top, Index + 1, Justs_Mel, [Inv|Invs])
            else {
                Choices := mel(Index) ∩ Top
                Found := false
                while Choices ≠ [] and not Found {
                    Ele ∈ Choices
                    Choices := Choices − [Ele]
                    if reduce_invs(Invs, Ele, Invs_Ele) and
                      shift_justs(Ele, mel(Index), Justs, Justs_Ele) then
                        Found := factor(Top − [Ele], Index + 1, Justs_Ele, [Inv|Invs_Ele])
                }
                return Found
            }
    } else if test(Top) then {
        Factor := Top
        return true
    } else
        return false
    }
}
```

Algorithm 4: Factoring the lattice.

```
boolean reduce_invs(Invs, Ele, Invs_Ele) {
    if Invs = [] then {
        Invs_Ele := []
        return true
    } else {
        Invs = [Inv|Invs_T]
        Inv_Ele := Inv − [Ele]
        if Inv_Ele = [] then
            return false
        else if reduce_invs(Invs_T, Ele, Invs_Ele_T) then {
            Invs_Ele := [Inv_Ele|Invs_Ele_T]
            return true
        } else
            return false
    }
}
```

Algorithm 5: Reducing the inverses.

```
boolean justified(Factor, Justs, Justs_Mel) {
    if Justs = [] then {
        Justs_Mel := []
        return false
    } else {
        Justs = [[E|E_Just]|Justs_T]
        if E ∈ Factor then {
            Justs_Mel := [[E, Factor|E_Just]|Justs_T]
            return true
        } else {
            return justified(Factor, Justs_T, Justs_Mel_T)
            Justs_Mel := [[E|E_Just]|Justs_Mel_T]
        }
    }
}
```

Algorithm 6: Checking whether the MEL is already justified.

The routine *justified* checks for the current MEL whether there already has been an element removed which is in the current MEL. If so, the current MEL is added to that element's justification, and *factor* is recursively called for the next MEL. Otherwise, the current MEL is not already justified. In this case, an element both in *Top* and the MEL must be removed. The MEL will be added to this new element's justification. Also, any MELs which contain this element and are stored in previous justifications must be removed from those justifications and added to the new one. If this removal process leaves any of the previous justifications empty, this construction is illegal and *factor* fails.

5.3 Enumeration

With the routine *factor* in place, it is easy to design the rest of an MEL enumeration program. The algorithm *enumerate_mels* (Alg. 8) does this. While there still remain (maximal) factors (which test *true*), continue to generate MELs. The algorithm finds one MEL per factor. Note that on the first pass through the *while* loop, *Top* itself is returned as the factor.

Algorithm *enumerate_mels* satisfies the first of the desired criteria for an enumeration algorithm, as outlined in Section 5.1: it finds MELs as quickly as is possible. In fact, if *enumerate_mels* is limited to finding k MELs, for any fixed k, it will perform in polynomial time.

Lemma 24. $|\mathit{max\_factors}(\mathcal{L},\mathcal{D})| \leq |\mathcal{L}|^{|\mathcal{D}|}$.

Proof. Any maximal factor of $\mathcal{L}$ with respect to $\mathcal{D}$ is pairwise incomparable with each $\mathcal{A} \in \mathcal{D}$. Consider the maximal factor $\mathcal{M}$. This means $\forall \mathcal{A} \in \mathcal{D}.\ \exists e \in \mathcal{A}.\ e \notin \mathcal{M}$. At most, there are $|\mathcal{D}|$ such e's, one for each set in $\mathcal{D}$. The set $\mathcal{A}$ is equivalent to $\mathcal{L}$ minus those e's. Nothing else is missing from $\mathcal{A}$; otherwise, it would not be maximal. Therefore, $|\mathcal{A}| \geq |\mathcal{L}| - |\mathcal{D}|$.

This limits the number of potential maximal factors to the number of subsets of $\mathcal{L}$ that are of that size. This number is

$$\sum_{i=|\mathcal{L}|-|\mathcal{D}|}^{|\mathcal{L}|} \binom{|\mathcal{L}|}{i}$$

This is less than $|\mathcal{L}|^{|\mathcal{D}|}$. □

Theorem 25. The algorithm *enumerate_mels*($\mathcal{S}$) spends at most $\mathcal{O}(|\mathcal{S}| + |\mathcal{S}|^{k-1})$ time to find the first k

```
boolean shift_justs(Ele, Factor, Justs, New_Justs) {
    if steal_justs(Ele, Justs, Ele_Just, Justs_Ele) then {
        New_Justs = [[Ele, Factor|Ele_Just]|Justs_Ele]
        return true
    } else
        return false
}
boolean steal_justs(Ele, Justs, Ele_Just, Justs_Ele) {
    if Justs = [] then {
        Ele_Just := []
        Justs_Ele := []
        return true
    } else {
        Justs = [[Tag|Just]|Justs_T]
        selector(Ele, Just, With, Without)
        steal_justs(Ele, Justs_T, Ele_Just_T, Justs_Ele_T)
        append(With, Ele_Just_T, Ele_Just)
        Justs_Ele := [[Tag|Without]|Justs_Ele_T]
    }
}
selector(Ele, Sets, With, Without) {
    if Sets = [] then {
        With := []
        Without := []
    } else {
        Sets = [Set|Sets_T]
        if Ele ∈ Set then {
            selector(Ele, Sets_T, With_T, Without)
            With := [Set|With_T]
        } else {
            selector(Ele, Sets_T, With, Without_T)
            Without := [Set|Without_T]
        }
    }
}
```

Algorithm 7: Shifting justifications.

```
array mel(set)
integer Last
enumerate_mels(Top) {
    Last := 0
    while factor(Top, Factor) {
        a_mel_true(Factor, Mel, [])
        Last := Last + 1
        mel(Last) := Mel
    }
}
```

Algorithm 8: Enumerating MELs.

```
ishmael_factor(Top, Core, Index) {
    ishmael_factor(Top, Core, Index, [], [])
}
ishmael_factor(Top, Core, Index, Justs, Invs) {
    if Index ≤ Mlast then {
        Inv := (Top ∪ Core) − mel(Index)
        if Inv = [] then
            if justified(mel(Index), Justs, Justs_Mel) then
                ishmael_factor(Top, Core, Index + 1, Justs_Mel, [Inv|Invs])
            else
                foreach Ele ∈ mel(Index) ∩ Top
                    if reduce_invs(Invs, Ele, Invs_Ele) and
                      shift_justs(Ele, mel(Index), Justs, Justs_Ele) then
                        ishmael_factor(Top − [Ele], Core, Index + 1, Justs_Ele, [Inv|Invs_Ele])
    } else if cache_test(Top ∪ Core) then
        ishmael_true(Top, Core)
}
```

Algorithm 9: Factoring the lattice for ISHMAEL.

MELs, assuming that each call to *test* costs unit time.

Proof. Each time *a_mel_true* is called, only $|\mathcal{S}|$ steps are spent at most to find an MEL. The time spent on the call to *factor* in the *while* loop depends strictly on how many MELs are already enumerated. In worst case, *factor* enumerates all the maximal factors searching for one which tests *true*. This cost is bounded by the number of maximal factors. Lemma 24 shows that there are at most $|\mathcal{S}|^{k-1}$ maximal factors after $k-1$ MELs have been found. In worst case, each is tested to find the next MEL. □

The algorithm does not, however, satisfy the remaining two criteria outlined in Section 5.1. The algorithm *enumerate_mels* is suboptimal in four respects.

1. Each call to *factor* is with the original set, and with *all* the MELs found so far. Each call adds an MEL to the *mel* array. (Decomposability is not being exploited.) Therefore, the calls to *factor* will tend towards intractability quickly, since each subsequent call is over a larger input set.
2. Much redundant computation is performed across the repeated calls to *factor*. It would be better to find subsequent factors by *backtracking* through the *factor* algorithm; to continue exploring the search space without needing to reconstruct the search space each time.
3. The algorithm *factor* does not use the *Core* set that *a_mel_true* (Alg. 3) does. Therefore, much of the search space is explored repeatedly which does not need to be.
4. Redundant calls to *test* are made.

The final algorithm presented for enumerating MELs will eliminate these suboptimalities. Call it ISHMAEL (Iterative SearcH for MinimAl Elements in a Lattice).

5.4 ISHMAEL

The procedure *ishmael_factor* (Alg. 9) is a rewrite of the procedure *factor*. The use of a core set is added. The further restriction is added that any factor returned must be a superset of the core.

The procedure *ishmael* (Alg. 10) is the top level procedure. As did *all_mels*, it initially checks that the input set passes the test. Otherwise, there are no MELs to find. The procedure *ishmael_true* is similar to *a_mel_true* (Alg. 3). It employs the same core set strategy, and finds an MEL in $\mathcal{O}(N)$ time whenever invoked. However, it is also similar to *all_mels_true*(Alg. 2) in that it proceeds to find *all* the MELs of its

```
array mel(set), failure(set)
integer Mlast, Flast

ishmael(Top) {
      Mlast := 0
      Flast := 0
      if test(Top) then
            ishmael_true(Top, [])
}

ishmael_true(Top, Core) {
      if Top = [] then {
            Mlast := Mlast + 1
            mel(Mlast) := Core
      } else {
            choose Ele ∈ Top
            if cache_test((Top − [Ele]) ∪ Core) then {
                  Current := Mlast
                  ishmael_true(Top − [Ele], Core)
                  if mel(Last) ≠ Core then
                        ishmael_factor(Top − [Ele], Core ∪ [Ele], Current)
            } else
                  ishmael_true(Top − [Ele], Core ∪ [Ele])
      }
}
```

Algorithm 10: ISHMAEL—Enumerating MELs.

input set *Set* (where $Set = Top \cup Core$). This is accomplished by calling *ishmael_factor*, which will factor *Set* with respect to the MEL just found.

The two procedures *ishmael_true* and *ishmael_factor* are interleaved. The initial call to *ishmael_factor* iterates through the possible factors, and calls *ishmael_true* on each to find all the MELs for each factor. Thus, the search space of factors is traversed once, rather than repeatedly.

A call to *ishmael_true* proceeds as follows. An invocation *ishmael_true*(*Top*, *Core*) will find all MELs of *Set* which contain *Core*. The routine chooses an element, say *Ele*, from *Top* to remove. If the resulting subset tests *false*, then all MELs of *Set* must include *Ele*. In this case, the element is added to *Core* (and removed from *Top*), and *ishmael_true* recursively called. (This works the same way that *a_mel_true* in Alg. 3 does.) Otherwise, the procedure *ishmael_true* splits the search space of the lattice over *Set* into two parts, based on *Ele*:

- all subsets of *Set* which do not contain *Ele*;
- and all subsets of *Set* which do contain *Ele*.

For the former, *ishmael_true* is called to find all the MELs of $Set - \{Ele\}$ (which contain *Core*). For the latter, *ishmael_factor* is called, with the element *Ele* added to *Core*. This assures that the invocation of *ishmael_factor* finds only (and all of) those MELs of *Set* which contain *Ele*. (Actually, again just the ones which contain *Core*, which now contains *Ele* too.) The ones which do not contain *Ele* were already found. This invocation of *ishmael_factor* needs to consider only the MELs found by the preceding invocation of *ishmael_true* to factor the lattice; all other MELs are incomparable with *Set*, and so do not need to be considered.

Notice that *ishmael_factor* is not called if *Core* was found to be an MEL (and, hence, the only MEL) by the previous call to *ishmael_true*. This is because there cannot be any factors which contain $Core \cup \{Ele\}$. (So it would be futile to look for any in this case.)

The algorithm ISHMAEL satisfies all the criteria outlined in Section 5.1: it enumerates MELs as quickly as is possible, as does *enumerate_mels*; it properly exploits the decomposability of the enumeration process; and it *minimizes* the number of calls made to *test*. The next section explains what is meant by minimizing calls to *test*, and a proof that ISHMAEL does this is given.

5.5 Minimizing the Number of Calls to Test

During the search in MEL enumeration, a test procedure is called repeatedly to determine whether the current set should be explored, or to verify that a given set is minimal (by checking that each of its immediate subsets *fail* the test). Executing the test procedure may be expensive. Therefore, it is beneficial to limit the number of calls to the test procedure.

Most importantly, there should be no *redundant* calls to test ($\mathcal{T}$). The following three properties should hold:

1. no set $\mathcal{A}$ should ever be tested twice;
2. for $\mathcal{A} \in \mathcal{T}$, after $\mathcal{A}$ is tested, no superset of $\mathcal{A}$ is tested; and
3. for $\mathcal{A} \notin \mathcal{T}$, after $\mathcal{A}$ is tested, no subset of $\mathcal{A}$ is tested.

It should at least be guaranteed that the test is never called on the same set twice during the search. Recall that the algorithm *all_mels* calls *test* repeatedly over the same set. The second and the third properties are relevant because the test is monotonic with respect to subset. If $\mathcal{A} \in \mathcal{T}$, then it is known that $\forall \mathcal{B} \supseteq \mathcal{A}.\, \mathcal{B} \in \mathcal{T}$; and, likewise, if $\mathcal{A} \notin \mathcal{T}$, then it is known that $\forall \mathcal{B} \subseteq \mathcal{A}.\, \mathcal{B} \notin \mathcal{T}$.

The algorithm *ishmael* exhibits these properties inherently for any $\mathcal{A} \in \mathcal{T}$. It does not exhibit these inherently for all $\mathcal{A} \notin \mathcal{T}$. (This is why *ishmael* calls *cache_test* instead of *test*, to be explained below.)

Lemma 26. For any set $\mathcal{A} \in \mathcal{T}$, $\mathcal{A}$ is tested at most once during execution of *ishmael*. Furthermore, once $\mathcal{A}$ has been tested, no superset of $\mathcal{A}$ is ever tested.

Proof. For any $\mathcal{A} \in \mathcal{T}$, if $\mathcal{A}$ is tested, a call to *ishmael_true*(*Top, Core*), for some *Top* and *Core* such that $\mathcal{A} = Top \cup Core$, is made immediately after. Consider the first time $\mathcal{A}$ is tested.

During the activation of *ishmael_true*(*Top, Core*), $\mathcal{A}$ is never tested, nor are any of its supersets. Only subsets of $\mathcal{A}$ are potentially tested. If *ishmael_factor* is invoked, it can only be after *ishmael_true* was invoked. Thus some MELs for $\mathcal{A}$ have been recorded. The call to *ishmael_factor* then cannot test $\mathcal{A}$, nor any of its supersets, as guaranteed by factoring.

The call to *ishmael_true*(*Top, Core*) may have been a recursive call. Consider a return to a previous activation of *ishmael_true*. A call to *ishmael_factor* may proceed. Any further calls that *ishmael_factor* makes to *ishmael_true* must be incomparable with any MELs of $\mathcal{A}$, since $\mathcal{A}$'s MELs have already been recorded. Thus, $\mathcal{A}$ will not be tested there, nor will any of its supersets be. Otherwise, this activation of *ishmael_true* completes, invoking no more tests.

Consider a return to an activation of *ishmael_factor*. Any further calls it makes to *ishmael_true* must be incomparable with any MELs of $\mathcal{A}$, since $\mathcal{A}$'s MELs have already been recorded. Thus, $\mathcal{A}$ will not be tested there, nor will any of its supersets be. □

Note that the algorithm *ishmael* calls *cache_test* instead of *test*. The routine *cache_test* guarantees that the actual test is never called on a set $\mathcal{A}$ twice, nor on any subsets of $\mathcal{A}$ once $\mathcal{A}$ has been tested, for any $\mathcal{A} \notin \mathcal{T}$. This is accomplished by caching the failures. The routine *cache_test*(Alg. 11) shows an implementation. A more efficient routine could be written, using hashing or the like.

Eventually, if there are many MELs, there will be many elements in the failure cache and the cache will become expensive to check. At some point, checking the cache may be more expensive than executing

```
boolean cache_test(Set) {
    Fails := false
    I := 1
    while I ≤ Flast and not Failed {
        if Set ⊆ failure(I) then
            Fails := true
        I := I + 1
    }
    if Fails then
        return false
    else if test(Set) then
        return true
    else {
        add_failure(Set)
        return false
    }
}

add_failure(Set) {
    Index := 1
    Offset := 0
    while (Index + Offset) ≤ Flast {
        if Set ⊃ failure(Index + Offset) then
            Offset := Offset + 1
        else {
            if Offset ≠ 0 then
                failure(Index) := failure(Index + Offset)
            Index := Index + 1
        }
    }
    Flast := Flast − Offset + 1
    failure(Flast) := Set
}
```

Algorithm 11: Test with cacheing.

test. However, by the time this point is reached, ISHMAEL will have become highly intractable due to the factoring of the lattice over so many MELs. So any question of when to stop using the cache is really a moot issue.

5.6 Heuristics

ISHMAEL offers a good search strategy for enumerating MELs, a much better search strategy than *all_mels*'s. Next, it should be considered whether there are any good heuristics for directing or pruning the search in ISHMAEL that would lead to improved performance, in average case. Much of the work on MFSs in the past has focussed on such heuristics. These heuristics should be re-evaluated in light of the complexity results of MEL enumeration presented in this paper, and with respect to ISHMAEL's search strategy. In particular, heuristics for MFS enumeration are considered here.

Janas [24] raises the point that, for most queries, many of the potential subqueries need never be considered while searching for MFSs.

Definition 27. Two atoms in a query are *joined iff* they share a variable in common.

> Two atoms in a query are *connected iff* the two are joined, or there exists a third atom in the query to which the first atom is *joined*, and to which the second atom is *connected*. (The relation *connected* is the transitive closure of *joined*.)
>
> A *query* is said to be *connected iff* every two atoms in the query are connected.[17] It is said to be *disjoint* otherwise.

It is reasonable to insist that queries be connected. (A user could always ask any disconnected parts as separate queries.) In searching for failed subqueries, it is only necessary to consider connected subqueries. For instance, consider the subquery

$$\leftarrow \textit{ward}(\textit{patient: } P),\ \textit{contagious}(\textit{name: } I),\ \textit{staph}(\textit{name: } I).$$

This query is not connected. It can only fail if both

$$\leftarrow \textit{ward}(\textit{patient: } P).$$

and

$$\leftarrow \textit{contagious}(\textit{name: } I),\ \textit{staph}(\textit{name: } I).$$

fail independently. An ISHMAEL algorithm for MFSs could be modified to not test disjoint subqueries, but to break them into their connected subqueries and test those instead.

At the one extreme, there is a single variable which every literal in the query shares.

$$\leftarrow a_1(X),\ \ldots,\ a_n(X).$$

In this case, *all* the subqueries are connected. At the other extreme, consider a query which is a chain.

$$\leftarrow a_1(X_1, X_2),\ a_2(X_2, X_3),\ \ldots,\ a_n(X_n, X_{n+1}).$$

That is, any two adjacent literals share a unique variable, but non-adjacent literals share no variable. Any less, and the query would be disjoint. In this case, only subqueries which represent contiguous subsequences are connected. There are $(n(n+1)/2)+1$ of these (counting $\emptyset$ as connected). A great majority of the subqueries are disjoint. In the limit, the probability that any given subquery is disjoint is one.

[17] Janas does not define *connected* in [24] in quite this way, but to the same effect.

It is assumed that, in general, a significant number of the subqueries of a query are disjoint. Hence, handling of disjoint queries by decomposing them should speed up the search for MFSs significantly, in average case.

Motro [31] suggests that certain subqueries be *materialized*; that is, the query be evaluated and the answer set stored as a table for later use. If a subquery is part of many queries that will be tested, that subquery will need to be recomputed many times. If it is materialized initially though, it would only have to be computed once.

ISHMAEL is shown to be optimal with respect to *test* in certain ways. It never tests the same query twice. Once a query tests *true* (evaluates to the empty answer set), no superquery of it will be tested; and once a query tests *false*, no subquery of it is tested. (See Section 5.5.) Of course subqueries of the queries which test *true* may test *false* themselves (have non-empty answer sets). These are candidates for materializing.

It is not clear what a good materialization heuristic would be. Not all failing subqueries should be materialized. There are too many of them, and not all will be used repeatedly as components of other queries. Criteria are needed to decide which to choose. The idea warrants further consideration. Motro further suggests that techniques for optimizing multiple queries [22, 34] could be exploited in MFS enumeration as well.

In the *test* used in enumerating MFSs, note that it is not necessary to compute *answers* for each subquery considered; it is only necessary to compute whether it has *an* answer. This is an easier computation. Some relational database systems have a facility to ask such *existence* queries. The use of such a facility can greatly speed up the test.

ISHMAEL allows for a natural *halt* heuristic, to curtail how much time is spent searching for MELs. Time can be measured by a clock, by the number of steps the algorithm has taken, or by the number of tests made. Given that ISHMAEL guarantees an optimal enumeration of MELs, it will always have some results (some MELs found) after any reasonable cut-off. It would be worthwhile to estimate bounds on ISHMAEL that would guarantee that this many MELs have been found after the algorithm has proceeded this many steps. Given such bounds, one could limit the algorithm to a given number of steps or tests, and be guaranteed that so many MELs are found (or that the algorithm will have halted in the case that there are not, *a priori*, that many MELs).

In light of the complexity of MEL enumeration, certain heuristics can now be judged as detrimental. In [31], Motro imposes a bound on the depth of the recursion in searching for maximally generalized failing queries (for which MFSs are a special case). This was to limit the size of the search space, to render it more manageable. However, as discussed in Section 3.2, it is not the depth of the lattice which is problematic for search, it is the breadth. This bound heuristic may easily exclude all the MFSs.[18] Any answer the algorithm does find can no longer be guaranteed to be minimal. Fortunately, a depth limit is not necessary with an ISHMAEL-style algorithm.

6 Issues

The MEL problem and the enumeration algorithms raise a number of interesting issues. We address two such issues here. In Sec. 6.1, we consider how the set of MSSs can be derived if given the set of MFSs. In Sec. 6.2, we consider the probability of finding one given MEL over another. Finally, in Sec. 6.3, we outline issues for future study.

[18] If the depth limit is reasonably large, the new search space is effectively just as hard to search as the original. Otherwise, if the depth limit is small, it will eliminate most all of the MFSs with high probability.

6.1 Converting from MFSs to MSSs

An algorithm such as ISHMAEL can be used to enumerate all the MFSs of a query. The algorithm can be run again with a different test to find the MSSs of the query. In Section 4, it was shown that both of these problems are of equal complexity.

It would seem, however, that once the MFSs of a query are known, the MSSs of the query should be derivable. It should not be necessary to run ISHMAEL again, nor to ask further questions to the database (to invoke *test* more times). The set of MSSs *is* deducible from the set of MFSs, and vice-versa, but the conversion is **NP**-*hard*.

As MEL stands for a Minimal Element in the Lattice which *passes* the test, let XEL stand for a maXimal Element in the Lattice which *fails* the test. Thus XEL is the generalization of MSS, as MEL is the generalization of MFS.[19]

Theorem 28. Let $\mathcal{M}$ be the collection of all MELs of $\mathcal{L}$, and $\mathcal{X}$ be the collection of all XELs of $\mathcal{L}$.

Let $\mathcal{N}$ be the collection of all immediate subsets of the MELs:

$$\mathcal{N} = \{\mathcal{B} \mid \mathcal{B} = (\mathcal{A} - \{e\}), \mathcal{A} \in \mathcal{M} \text{ and } e \in \mathcal{A}\}.$$

Then

$$\mathcal{X} = \mathit{max\_factors}(\mathcal{L}, \mathcal{M}) \cup \{\mathcal{A} \in \mathcal{N} \mid \neg\exists\mathcal{B} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M}).\ \mathcal{A} \subset \mathcal{B}\}.$$

Proof. $\mathcal{X} \supseteq \mathit{max\_factors}(\mathcal{L}, \mathcal{M}) \cup \{\mathcal{A} \in \mathcal{N} \mid \neg\exists\mathcal{B} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M}).\ \mathcal{A} \subset \mathcal{B}\}$

Consider $\mathcal{A} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M})$. Assume $\mathcal{A} \in \mathcal{T}$. Then it has some MEL (which is a subset of, or is equivalent to, $\mathcal{A}$). Any MEL of $\mathcal{A}$ is an MEL of $\mathcal{L}$. This MEL is in $\mathcal{M}$. However, $\mathcal{A}$ is pairwise incomparable with all the sets of $\mathcal{M}$. This is a contradiction. Thus, $\mathcal{A} \notin \mathcal{T}$.

Given $\mathcal{A} \notin \mathcal{T}$, $\mathcal{A}$ has some XEL. Any superset of $\mathcal{A}$ is a superset of some MEL in $\mathcal{M}$, since $\mathcal{A}$ is a maximal factor with respect to $\mathcal{M}$. So no superset of $\mathcal{A}$ is a XEL. Therefore, $\mathcal{A}$ is a XEL.

Consider $\mathcal{A} \in \{\mathcal{A} \in \mathcal{N} \mid \neg\exists\mathcal{B} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M}).\ \mathcal{A} \subset \mathcal{B}\}$. Clearly, $\mathcal{A} \notin \mathcal{T}$. It is a subset of some MEL $\mathcal{B}$. If it were in $\mathcal{T}$, then $\mathcal{B}$ would not be minimal. Any superset of $\mathcal{A}$ is an MEL, or a superset of an MEL. Otherwise, some superset of $\mathcal{A}$ would appear in $\mathit{max\_factors}(\mathcal{L}, \mathcal{M})$. Thus, $\mathcal{A}$ is maximal, and is a XEL.

$\mathcal{X} \subseteq \mathit{max\_factors}(\mathcal{L}, \mathcal{M}) \cup \{\mathcal{A} \in \mathcal{N} \mid \neg\exists\mathcal{B} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M}).\ \mathcal{A} \subset \mathcal{B}\}$

Consider $\mathcal{A}$, an XEL of $\mathcal{L}$. Thus, $\mathcal{A} \notin \mathcal{T}$, and $\forall\mathcal{B} \subset \mathcal{A}.\ \mathcal{B} \notin \mathcal{T}$. Therefore, no MEL is a subset of $\mathcal{A}$. Cases:

- $\exists\mathcal{B} \in \mathcal{M}.\ \mathcal{A} \subset \mathcal{B}$.
  Assume $\exists\mathcal{S} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M}).\ \mathcal{A} \subset \mathcal{S}$. Then $\mathcal{S} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M})$ is a XEL, by the argument above. This means $\mathcal{A}$ is not maximal, which contradicts the assumption that it is.
  Assume $\mathcal{B} \in \mathcal{N}.\ \mathcal{A} \subset \mathcal{B}$. Clearly $\mathcal{B} \notin \mathcal{T}$ since $\mathcal{B}$ is the subset of an MEL. This means $\mathcal{A}$ is not maximal, which contradicts the assumption that it is.
  $\mathcal{A} \in \{\mathcal{S} \in \mathcal{N} \mid \neg\exists\mathcal{B} \in \mathit{max\_factors}(\mathcal{L}, \mathcal{M}).\ \mathcal{S} \subset \mathcal{B}\}$.
- $\forall\mathcal{B} \in \mathcal{M}.\mathcal{A} \not\subset \mathcal{B}$.
  Clearly $\mathcal{A}$ is not equal to any of the MELs. Also no MEL is a subset of $\mathcal{A}$. Therefore, $\mathcal{A}$ is pairwise incomparable with each set in $\mathcal{M}$. $\mathcal{A}$ is maximal: $\mathcal{A} \cup \{e\} \in \mathcal{T}$ for any $e \in \mathcal{L}$ and $e \notin \mathcal{A}$. This means any $\mathcal{A} \cup \{e\}$ is a superset of some MEL in $\mathcal{M}$.

[19]Again. please avoid confusion. An XEL *fails* the test. In the case of an MSS, this means the subquery has a *non-empty* answer set, and so *succeeds* as a query.

Thus, $\mathcal{A} \in max\_factors(\mathcal{L}, \mathcal{M})$. □

Theorem 29. Deducing the set of XELs, $\mathcal{X}$, of a set $\mathcal{L}$ from the set of MELs, $\mathcal{M}$, of the set $\mathcal{L}$ is **NP**-*hard*.

Proof. By theorem 28, it is seen that $\mathcal{X}$ is the union of the maximal factors of $\mathcal{L}$ with respect to $\mathcal{M}$ and the immediate subsets of the sets in $\mathcal{M}$ which are not contained by the maximal factors. The latter set, the immediate subsets, is polynomial to determine. Therefore, the problem of constructing the set $max\_factors(\mathcal{L}, \mathcal{M})$ is reducible to deducing $\mathcal{X}$ from $\mathcal{M}$. Theorem 23 shows that constructing $max\_factors(\mathcal{L}, \mathcal{M})$ is **NP**-*hard*. Thus constructing $\mathcal{X}$ is **NP**-*hard*. □

The above theorem also indicates that there is no correlation between the number of MELs a set has and the number of XELs it has. If there were, then theorem 23 could not hold.

Even though determining the XELs from the MELs is hard, this is still a preferable approach if the MELs are already enumerated, rather than running ISHMAEL again. The algorithm *factor* (Alg. 4) can be modified to find all factors. (The call to *test* would be removed in this version.) By this approach, no further calls to *test* are needed to determine the XELs.

6.2 The Probability of Finding a Given MEL

The MEL problem raises a seeming paradox: it seems to offer a probabilistic attack on **NP**-*completeness*. We have shown that an enumeration of a linear number of MELs is **NP**-*complete*, but that to find an MEL is polynomial. The algorithm *a_mel* (Alg. 1) demonstrates an algorithm which finds an MEL for a set in polynomial time.

Note that the algorithm *a_mel* is non-deterministic. The *choose* step can choose any one of the remaining elements in the set to eliminate. There is a sequence of such choices which leads to any given MEL. Therefore, there is a chance that for any given MEL of, say $\mathcal{L}$, that *a_mel* will return that MEL.

If it were equally likely for *a_mel* to return any MEL of $\mathcal{L}$, then this would offer a probabilistic attack on **NP**-*complete* problems. Recall the proof of theorem 11. A **SAT** problem can be encoded as a **MEL** problem. If $n+1$ MELs could be found, then one must represent a model of the **CNF** theory input to **SAT**. Assume that the MELs are equally likely to find. Then *a_mel* could be run non-deterministically a number of times searching for an MEL that is a model. To determine that the **CNF** theory has no model, to within any degree of confidence, would require that *a_mel* be run (without finding an MEL that is a model) a linear number of times with respect to the degree of confidence.

Otherwise, it must be that not all MELs are equally likely to be found. This is the actual case, so *a_mel* offers no probabilistic attack on **NP**-*completeness*. It is worthwhile to characterize which MELs are more likely to find than others.

Consider $\mathcal{L} = \{a, b, c\}$, and say it has two MELs, $\{a\}$ and $\{b, c\}$. Fig. 5 (a) shows the lattice for $\mathcal{L}$ with the MELs boxed. It is tempting to count the number of paths to an MEL from the top of the lattice. There are two paths to $\{a\}$ from $\{a, b, c\}$: first remove b and then c; and vice-versa. There is one path to $\{b, c\}$: remove a. There is no other path possible to any MEL. Unfortunately, not all paths are equally likely, in general, so counting paths will not be sufficient. In the above example though, the paths considered are equally likely, which leads to a probability of $\frac{2}{3}$ for $\{a\}$, and $\frac{1}{3}$ for $\{b, c\}$.

It is assumed that the probability of *a_mel* choosing any given element from the set for elimination is equal to that of choosing any other. So the way to measure the probability of *a_mel* returning MEL $\mathcal{A}$ will be to measure the likelihood that the appropriate eliminations are made, which result in $\mathcal{A}$ being the only MEL remaining for discovery. This can be formalized via a recurrence equation.

The probability of finding a given MEL can only be determined if the set of all MELs is known *a priori*.

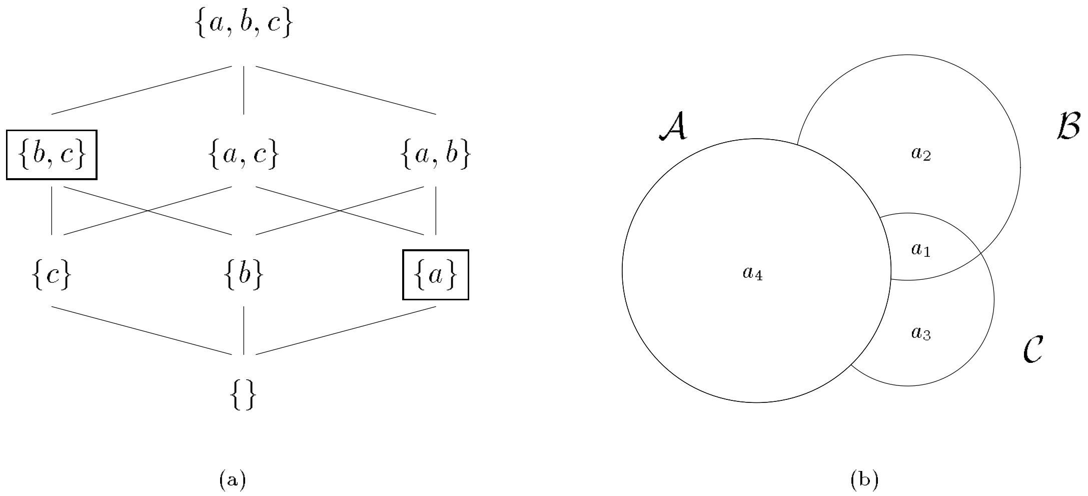


Figure 5: Probability of finding answer $\mathcal{A}$.

Definition 30. Let $\mathbf{P}_{\mathcal{S}}(\mathcal{A})$ be the probability of *a_mel* returning the MEL $\mathcal{A}$, given that $\mathcal{S}$ is the collection of all MELs.

Consider $\mathcal{L}$ in the abstract, and assume that it has three and only three MELs: $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. These sets are represented in the Venn diagram in Fig. 5 (b). The following formula calculates the probability to find $\mathcal{A}$.

$$\begin{aligned}&\mathbf{P}_{\{\mathcal{A},\mathcal{B},\mathcal{C}\}}(\mathcal{A}) = \tfrac{|a_1|}{n} + \tfrac{|a_2|}{n}\mathbf{P}_{\{\mathcal{A},\mathcal{C}\}}(\mathcal{A}) + \tfrac{|a_3|}{n}\mathbf{P}_{\{\mathcal{A},\mathcal{B}\}}(\mathcal{A}) \quad \text{where}\\ &\quad a_1 = (\mathcal{B}\cap\mathcal{C}) - \mathcal{A},\\ &\quad a_2 = \mathcal{B} - (\mathcal{A}\cup\mathcal{C}),\\ &\quad a_3 = \mathcal{C} - (\mathcal{A}\cup\mathcal{B}), \text{ and}\\ &\quad n = |(\mathcal{A}\cup\mathcal{B}\cup\mathcal{C}) - \mathcal{A}\cap\mathcal{B}\cap\mathcal{C}|\end{aligned}$$

If an element of a_4 is chosen by *a_mel* for elimination, then there is zero probability that $\mathcal{A}$ will be found. So a term for this is absent in the equation. Choosing an element from a_1 excludes both $\mathcal{B}$ and $\mathcal{C}$ from further consideration, leaving only $\mathcal{A}$. The conditional probability that $\mathcal{A}$ is found given that an element from a_1 was eliminated is 1. Choosing from a_2 eliminates only $\mathcal{B}$. Then the conditional probability $\mathbf{P}_{\{\mathcal{A},\mathcal{C}\}}(\mathcal{A})$ needs to be determined. Choosing from a_3 is a symmetric case.

This can be generalized to the following recurrence relation.

$$\begin{aligned}&\mathbf{P}_{\mathcal{S}}(\mathcal{A}) = \sum_{\mathcal{R}\subseteq(\mathcal{S}-\{\mathcal{A}\})} \frac{|(\cap\mathcal{S}) - (\cup(\mathcal{S}-\mathcal{R}))|}{|(\cup\mathcal{S}) - (\cap\mathcal{S})|}\cdot\mathbf{P}_{\mathcal{S}-\mathcal{R}}(\mathcal{A})\\ &\mathbf{P}_{\{\mathcal{A}\}}(\mathcal{A}) = 1\end{aligned}$$

Unfortunately, this recurrence relation is hard to solve, in general. Furthermore, it requires the set of all MELs *a priori* as input; these are not known in advance.

Let us solve a specific case though. Consider the set $\mathcal{L} = \{p_1, \ldots, p_{2n}\}$.

- $\mathcal{M} = \{\{e_1, \ldots, e_n\} \mid \forall i \in \{1, \ldots, n\}.\ e_i \in \{p_i, p_{n+i}\}\}$
- $\mathcal{F} = \{\{p_i, p_{n+i}\} \mid i \in \{1, \ldots, n\}\}$
- $\mathcal{S} = \mathcal{M} \cup \mathcal{F}$

Any **CNF** theory **C** can be converted into a positive **CNF** theory, $pos(\mathbf{C})$. Assume the set of propositional variables of **C** is $\{p_1, \ldots, p_n\}$. Collection $\mathcal{M}$ represents all the possible complete models of **C**, in which p_{n+i} represents $\neg p_i$. Collection $\mathcal{F}$ represents all the contradictions.

The probability that *a_mel*, run on $\mathcal{S}$, will return a set from $\mathcal{M}$ (and not from $\mathcal{F}$) is calculated. It is not necessary to use the recurrence relation to solve this. Each set in $\mathcal{M}$ is of size n, and $|\mathcal{M}| = 2^n$. While $|Set| \geq n$ in *a_mel*, any element is eligible to be chosen for elimination, since *Set* necessarily contains some set in $\mathcal{F}$. Consider that *a_mel* has arrived at a set $\mathcal{A}$ of size n. Any set of size n is equally likely to be that set. There are $\binom{2n}{n}$ such sets. If $\mathcal{A} \in \mathcal{M}$, then the algorithm halts, and $\mathcal{A}$ is returned. Otherwise, a set from $\mathcal{F}$ will eventually be returned.

The probability that a set from $\mathcal{M}$ is returned is $2^n/\binom{2n}{n}$. By Sterling's approximation, this is $(e/2n)^n$. Clearly, $(1/2^n) \gg (e/2n)^n$ in the limit. Therefore $\mathbf{P}_{\mathcal{S}}(\mathcal{A}) < (1/2^n)$, for *any* $\mathcal{A} \in \mathcal{M}$, and $|\mathcal{L}| = 2n$. The same argument holds if only some subset of $\mathcal{M}$ is assumed as MELs, along with $\mathcal{F}$, or even if only *one* set from $\mathcal{M}$ is assumed to be an MEL. This clearly shows why ISHMAEL does not help to solve **SAT**.

This leads us to the following hypothesis: *a_mel* tends to find MELs of smaller cardinality, with high probability. For ISHMAEL, this means it should tend to find MELs of smaller cardinality before finding those of larger cardinality.

This tendency may be good news. In many applications, the MELs of smaller cardinality are often preferred. In abduction, for instance, it is more likely that a small set of conditions holds, rather than a large set of conditions, in general. The fact that *a_mel* and such algorithms are more likely to find these smaller sets is beneficial.

6.3 Future Work

It will be worthwhile to adapt the algorithms presented here to work over extended lattices as proposed by Motro (section 2). Instead of removing atoms from a query, atoms are relaxed. Then *maximally generalized failing queries* (MGQs) can be found. MFSs are a special case of this. So clearly, the complexity of enumerating MGQs is as hard as enumerating MFSs. It does not appear, however, that it is any worse.

The algorithm *a_mel* can be adapted for the extended lattice. Assume that each literal in the query can be relaxed k steps. Then the number of steps to find the first maximally generalized query is $k \cdot N$. It should be possible to modify ISHMAEL's search for the extended lattice. A new factoring algorithm which does replacements rather than eliminations will be needed.

There are other issues that need to be addressed before providing MGQs can be a cooperative technique in its own right. In particular, the relaxation operators used to relax given atoms in the query must be provided The step size, or how much to relax a condition, must be determined too. This type of information is not available with databases, and somehow the information has to be manifested. Motro [31] introduces a *supposition generalizor* into the architecture of his system SEAVE, which decides how to relax atoms. In [13], we consider how to use taxonomic information represented inherently in the rules of a deductive database to relax atoms in queries. Chu et al. [3, 4] consider how to extract and employ such taxonomic information in relational database, and have implemented such in their system CoBase.

Heuristics and bounds are needed to ensure that the extended lattice to be searched is finite. In Fig. 3, the lattice is infinite; $A \leq 30 \Longrightarrow A \leq 31 \Longrightarrow \ldots$ A limit is needed on how far this condition should be relaxed.

Complexity analyses of variations on MEL enumeration would be beneficial. It would be interesting to explore what further conditions could be placed on which MELs qualify as answers, and still for the problem to stay within the same complexity bounds. Clearly, many desirable conditions push the problem into intractability. In [2], Bylander et al. look into these issues for abduction problems. For instance, if one adds the condition that the answer be an MEL of *minimum* cardinality, then the problem becomes intractable even to find one. There are, however, conditions that can be added which do not make the enumeration problem harder. Some of these may be useful in certain problem domains.

7 Conclusions

Algorithms designed in the past to search for minimal failing and maximally succeeding subqueries have been ineffective as they are highly intractable. This has been largely due to the fact that the complexity nature of these problems has been ill-understood.

This paper presents a complexity profile of such problems, and presents an algorithmic approach to them. With these algorithms, it is now possible to build practical facilities for relational database systems for finding and presenting MFSs and MSSs. Such facilities will enable database systems to give more cooperative responses to users (in this case, whenever a user's query fails), and this can make using database systems easier. Databases are growing in size and complexity, and so are database applications (and, hence, queries). This means that cooperative techniques, such as identifying MFSs and MSSs, will become indispensable in future database systems.

With many organizations now owning large, valuable data stores, there is a growing interest in *data mining*. (See [33].) Data stores are examined for general patterns, which may be characterizations of the data in general. Such a characterization may reflect some hidden semantics of the data at large.

When a query fails when evaluated against a database, it may simply be that there is no data which provides an answer, or it may be that this query *must* fail due to the semantics of the database. (This distinction was drawn in Section 1.) This offers a potential data mining tool [11, 35]. Whenever a query fails, but there is no proof that it must fail (guaranteed by the integrity constraints of the database), it is possible that the query represents a *missing* integrity constraint. Failing queries can be collected and later analyzed, either by a program or a database administrator. Some of these failing queries may then be *promoted* as integrity constraints. For such a technique to be most effective, the MFSs of failing queries should be considered, not the queries themselves. The MFSs offer logically stronger statements, and better characterize the database as integrity constraints.

The more general results on MEL enumeration may be applicable in a number of domains outside of databases. For certain semantics for logic programs, there exists the need for minimization in the proof procedure. This arises in the deduction of negative information via a non-monotonic rule for negation. This problem is encountered in a proof procedure for stable theories of Fernández and Lobo [10]. The enumeration of (minimal) answers by the proof procedure follows the same complexity as enumerating MELs. In this case, all the answers are desired. An ISHMAEL-style algorithm can be used for the enumeration.

Much work has been done in the domain of abductive reasoning. Many abductive reasoning tasks are more complex than MEL enumeration, but not all. While a fair amount is known about the complexity of abductive reasoning [2], enumeration algorithms such as ISHMAEL may be new to this domain. Algorithms as ISHMAEL may be useful when an enumeration of the possible abductive supports is needed.

Interest in minimal failing subqueries came about originally from interest in false presuppositions in natural language dialog. Kaplan, Lee, and Janas showed that the identification problem for false presuppositions to be independent of natural language itself; rather the problem of finding false presuppositions depends solely upon the logical structure of the statements made. In this paper, the false presuppositions problem has been addressed for databases. It may be possible that these techniques can be adapted for natural language systems to identify false presuppositions that occur in natural language dialog.

Acknowledgments

A number of people have contributed to the ideas in this work. I thank José Alberto Fernández for first formalizing the MEL problem as a characterization of MFS. Yuan Liu and Lev Novik contributed much in discussions on search strategies. Carolina Ruiz and Sergio Alvarez helped to characterize the recurrence relation for determining the probabilities of MELs.

Jack Minker has contributed much in his support of this research and through many discussions. I also thank him, John Guthrie, and William Gasarch for their proofing of the paper, and their many suggestions that have led to much improvement. This research was supported by the National Science Foundation under grant IRI-9300691 and by the Air Force Office of Scientific Research under grant AFOSR-91-0350.

References

[1] N. Belnap and T. Steel. *The Logic of Questions and Answers.* Yale University Press, New Haven, CT, 1976.

[2] T. Bylander, D. Allemang, M. C. Tanner, and J. R. Josephson. The computational complexity of abduction. *Artificial Intelligence*, 49:25–60, 1991.

[3] W. W. Chu. CoBase: A cooperative database system. In Demolombe and Imielinski [9], pages 41–73. To appear.

[4] W. W. Chu, Q. Chen, and R.-C. Lee. A structured approach for cooperative query answering. *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[5] P. Cole and J. Morgan, editors. *Syntax and Semantics.* Academic Press, 1975.

[6] A. Colmerauer and J. Pique. About natural logic. In Gallaire et al. [19], pages 343–365.

[7] F. Corella, S. J. Kaplan, G. Wiederhold, and L. Yesil. Cooperative responses to boolean queries. In *First International Conference on Data Engineering*, pages 77–85, Silver Spring, Maryland, 1984. IEEE Computer Society Press.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 1991.

[9] R. Demolombe and T. Imielinski, editors. *Non Standard Queries and Answers.* Oxford University Press, 1994. To appear.

[10] J. A. Fernández and J. Lobo. A proof procedure for stable theories. Technical Report UMIACS-TR-93-14 and CS-TR-3034, The University of Maryland Institute for Advanced Computer Studies and the Department of Computer Science, University of Maryland at College Park, 1993.

[11] W. J. Frawley, G. Piatetsky-Shapiro, and C. J. Matheus. Knowledge discovery in databases: An overview. In Piatetsky-Shapiro and Frawley [33], chapter 1.

[12] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992. Invited paper.

[13] T. Gaasterland, P. Godfrey, and J. Minker. Relaxation as a platform for cooperative answering. *Journal of Intelligent Information Systems*, 1:293–321, 1992.

[14] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. In Demolombe and Imielinski [9], pages 1–39. Appears orginally as [12].

[15] T. Gaasterland, P. Godfrey, J. Minker, and L. Novik. A cooperative answering system. In A. Voronkov, editor, *Proceedings of the Logic Programming and Automated Reasoning Conference*, Lecture Notes in Artificial Intelligence 624, pages 478–480. Springer-Verlag, St. Petersburg, Russia, July 1992.

[16] A. Gal. *Cooperative Responses in Deductive Databases.* PhD thesis, Department of Computer Science, University of Maryland, College Park, Maryland, 1988.

[17] A. Gal and J. Minker. A natural language database interface that provides cooperative answers. *Proceedings of the Second Conference on Artificial Intelligence Applications*, December 11–13 1985.

[18] A. Gal and J. Minker. Informative and cooperative answers in databases using integrity constraints. In V. Dahl and P. Saint-Dizier, editors, *Natural Language Understanding and Logic Programming*, pages 277–300. North Holland, 1988.

[19] H. Gallaire, J. Minker, and J.-M. Nicolas, editors. *Advances in Database Theory, Volume 1.* Plenum Press, New York, 1981.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness.* A Series of Books in the Mathematical Sciences. W. H. Freeman and Company, New York, 1979.

[21] P. Godfrey, J. Minker, and L. Novik. An architecture for a cooperative database system. In W. Litwin and T. Risch, editors, *Proceedings of the First International Conference on Applications of Databases*, Lecture Notes in Computer Science 819, pages 3–24. Springer Verlag, Vadstena, Sweden, June 1994.

[22] J. Grant and J. Minker. Optimization in deductive and conventional relational data base systems. In Gallaire et al. [19], pages 195–234.

[23] H. Grice. Logic and conversation. In Cole and Morgan [5].

[24] J. M. Janas. On the feasibility of informative answers. In Gallaire et al. [19], pages 397–414.

[25] A. Joshi, B. Webber, and I. Sag, editors. *Elements of Discourse Understanding.* Cambridge University Press, 1981.

[26] S. J. Kaplan. Appropriate responses to inappropriate questions. In Joshi et al. [25], pages 127–144.

[27] S. J. Kaplan. Cooperative responses from a portable natural language query system. *Artificial Intelligence*, 19(2):165–187, Oct. 1982.

[28] R. M. Lee. Conversational aspects of database interactions. In *Proceedings of the 4*th *International Conference on Very Large Data Bases*, Berlin, 1978.

[29] E. Mays. Correcting misconceptions about database structure. In *Proceedings of the CSCSI '80*, 1980.

[30] K. McCoy. Correcting object-related misconceptions. In *Proc. of COLING10*, Stanford University, California, July 1984.

[31] A. Motro. SEAVE: A mechanism for verifying user presuppositions in query systems. *ACM Transactions on Office Information Systems*, 4(4):312–330, October 1986.

[32] A. Motro. FLEX: A tolerant and cooperative user interface to databases. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):231–246, June 1990.

[33] G. Piatetsky-Shapiro and W. J. Frawley, editors. *Knowledge Discovery in Databases.* AAAI Press and MIT Press, Menlo Park, California, 1991.

[34] T. Sellis and S. Shosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2), June 1990.

[35] M. Siegel, E. Sciore, and S. Salveter. Rule discovery for query optimization. In Piatetsky-Shapiro and Frawley [33], chapter 24.

[36] P. F. Strawson. *Introduction to Logical Theory.* Methuen, London, 1974.

[37] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Principles of Computer Science Series. Computer Science Press, Incorporated, Rockville, Maryland, 1988.