

Evaluation of an Algorithm for the Transversal Hypergraph Problem

Dimitris J. Kavvadias¹ and Elias C. Stavropoulos^{2*}

¹ University of Patras, Department of Mathematics, GR-265 00 Patras, Greece
djk@math.upatras.gr

² University of Patras, Computer Engineering & Informatics Department
GR-265 00 Patras, Greece
estavrop@ceid.upatras.gr

Abstract. The Transversal Hypergraph Problem is the problem of computing, given a hypergraph, the set of its minimal transversals, i.e. the hypergraph whose hyperedges are all minimal hitting sets of the given one. This problem turns out to be central in various fields of Computer Science. We present and experimentally evaluate a heuristic algorithm for the problem, which seems able to handle large instances and also possesses some nice features especially desirable in problems with large output such as the Transversal Hypergraph Problem.

1 Introduction

Hypergraph theory [2] is one of the most important areas of discrete mathematics with significant applications in many fields of Computer Science. A hypergraph \mathcal{H} is a generalized graph defined on a finite set \mathcal{V} of nodes, with every hyperedge \mathcal{E} of \mathcal{H} being a subset of \mathcal{V} . A hypergraph is a convenient mathematical structure for modeling numerous problems in both theoretical and applied Computer Science and discrete mathematics. One of the most intriguing problems on hypergraphs is the problem of computing the *transversal hypergraph* of \mathcal{H} , denoted $\text{tr}(\mathcal{H})$. The transversal hypergraph is the family of all minimal hitting sets (*transversals*) of \mathcal{H} , that is, all sets of nodes T such that (a) T intersects all hyperedges of \mathcal{H} , and (b) no proper subset of T does. TRANSVERSAL HYPERGRAPH is the problem of generating $\text{tr}(\mathcal{H})$ given a hypergraph \mathcal{H} , and is an important common subproblem in many practical applications. Its importance arises from the fact that problems referring to notions like minimality or maximality are quite common in various areas of Computer Science.

For example, concepts of propositional circumscription [18,3] and minimal diagnosis [5] restrict interest to the set of models of an expression that are minimal. In circumscription, model checking for circumscriptive expressions reduces to determining the minimality of a model, whereas in model-based system diagnosis, finding a minimal diagnosis is equivalent to finding the prime implicants of an expression and, next, finding the minimal cover of them. Moreover, finding a

* Research supported by the University of Patras Research Committee (Project Caratheodory under contract no.1939).

maximal model is essential task in model-preference default inference [20], since, given a set of default rules, our aim is to find a maximal (most preferred) model of these rules. It can also be seen that the problem of computing the transversal hypergraph is an alternative view of the problem of generating all maximal models of a Boolean expression in CNF, having all its variables negated. Such an expression can be seen as a hypergraph whose hyperedges are the clauses of the expression and each node of a hyperedge corresponds to a negative variable of the corresponding clause. The maximal models of the expression then correspond to the transversals of the hypergraph. A symmetric situation holds for the minimal models of an expression having all its variables positive. Complexity questions related to minimal or maximal models have been discussed in [3,4,1]; more recent results can be found in [13].

An encyclopedic exposition of the applications of the TRANSVERSAL HYPERGRAPH problem can be found in [6,7]. We briefly state from there certain problems in the design of relational databases [15,16], in distributed databases [8], and in model-based diagnosis [5]. Another interesting connection was pointed out between the TRANSVERSAL HYPERGRAPH problem and the rapidly growing field of knowledge discovery in databases, or data mining [9,17].

In this paper we present and experimentally evaluate a heuristic algorithm for solving the TRANSVERSAL HYPERGRAPH problem. The algorithm was implemented and tested on a number of randomly generated problem instances. The experimental results show that the algorithm computes all transversals of a given hypergraph correctly and efficiently. This fact makes our heuristic suitable for solving problems in all areas mentioned above.

One has to be careful in defining efficiency of algorithms for problems like the TRANSVERSAL HYPERGRAPH. It is not hard to see that the transversal hypergraph $\text{tr}(\mathcal{H})$ of a hypergraph \mathcal{H} may have exponentially many hyperedges with respect to the number of nodes and the number of hyperedges of \mathcal{H} . It will therefore require exponential amount of time to compute $\text{tr}(\mathcal{H})$ in the worst case. Therefore, the usual distinction between tractable and intractable problems based on the existence or not of a polynomial-time algorithm, clearly does not apply here. Instead, more elaborate complexity measures have to be defined, that will take into account the *size of output*, too. It is natural to consider as tractable a problem with large output if it can be solved by an algorithm that is polynomial in *both* the input and the output. Such algorithms are called *output-polynomial*. A slightly stronger requirement is that the algorithm generates a new output bit in time polynomial in the input *and the output so far*. These latter algorithms are called *incrementally output-polynomial*. An even stronger requirement is that the algorithm generates two consecutive output bits in time bounded by a polynomial in the input size. These are called *polynomial delay* algorithms. There has been a recent surge of interest in such algorithms. For discussions of algorithms with output and performance criteria see [11,12,19].

The precise complexity of the TRANSVERSAL HYPERGRAPH problem is still unknown. The brute force algorithm given by Berge [2] needs time exponential in both the input and the output. However, several special cases can be solved

in polynomial time [6]. Recently, an output-subexponential algorithm was given by Fredman and Khachiyan in [7]. There, it was shown that the duality of two monotone Boolean expressions in DNF can be checked in time $O(n^{\log n})$, where n is the combined size of the input and the output. It is not hard to see that this problem is another disguised form of the TRANSVERSAL HYPERGRAPH problem (see [7,10] for further details on these issues).

Our algorithm presents in practice a remarkable uniformity in its output rate; averaging over relatively small parts of the output (e.g. 100 transversals out of a total output of tenths of thousands), we get delays deviating by at most 3 mean values. (We mention however, that at present we can prove no bound for the delay between consecutive outputs.) This happens partially because our algorithm operates in a *generate-and-forget* fashion i.e., no previous transversal is required for the generation of the next ones. In contrast, both the brute force algorithm and the Fredman–Khachiyan algorithm require all previous transversals to be stored. Moreover, the former will output the first transversal after exponentially long delay. Our approach also greatly reduces the memory requirements, since previously generated transversals need not be stored. In a different situation, the memory requirements could be devastating as the total number of transversals can be enormous. In addition, absolute time delays are very small, allowing the successful handling of large problems.

The rest of this paper is organized as follows: In Section 2 we describe our heuristic algorithm for generating all transversals of a hypergraph. In the next section we compare our algorithm with some other approaches while in Section 4 we give some implementation details. Experimental results on the performance of the algorithm are summarized in Section 5. Finally, in Section 6 some conclusions are given.

2 Description of the Algorithm

The proposed algorithm is based on the following simple algorithm of Berge (see [2,6]): Consider a hypergraph $\mathcal{H} = \{\mathcal{E}_1, \dots, \mathcal{E}_m\}$. Assume that we have already computed the transversal hypergraph $\text{tr}(\mathcal{G})$ of $\mathcal{G} = \{\mathcal{E}_1, \dots, \mathcal{E}_k\}$, for some $k < m$. It is easy to see that $\text{tr}(\mathcal{G} \cup \{\mathcal{E}_{k+1}\}) = \{\min\{t \cup \{v\}\} : t \in \text{tr}(\mathcal{G}) \text{ and } v \in \mathcal{E}_{k+1}\}$, where by $\min\{t \cup \{v\}\}$ we denote the set of minimal subsets of $t \cup \{v\}$ that are hitting sets of $\mathcal{G} \cup \{\mathcal{E}_{k+1}\}$. Based on this observation we may find all transversals of G by starting from the transversals of \mathcal{E}_1 (note that the transversals of a hypergraph with a single hyperedge are all its nodes) and adding one-by-one the rest of the hyperedges, computing at each step the set of transversals of the new hypergraph. The algorithm terminates after the addition of \mathcal{E}_m .

There are several drawbacks in the above scheme regarding its efficiency which are explained in detail in the next section. We only mention here the most severe one, in view of the complexity measures for this problem: The computation of the first *final* transversal (a transversal of the input hypergraph \mathcal{H}) is accomplished after all transversals of the graph $\{\mathcal{E}_1, \dots, \mathcal{E}_{m-1}\}$ have been com-

puted. This means that it may take an exponentially long time before the first transversal is output. No less important are the memory requirements that also emerge from the above: all intermediate transversals have to be stored and kept until used for the computation of the new transversal set.

We now explain our algorithm. Consider a hypergraph $\mathcal{H} = \{\mathcal{E}_1, \dots, \mathcal{E}_m\}$ defined on a set \mathcal{V} of n nodes. We call a set of nodes $\mathcal{X} \subseteq \mathcal{V}$ a *generalized node* if all nodes in \mathcal{X} belong in exactly the same hyperedges of \mathcal{H} . Assume that the hypergraph \mathcal{H} has a generalized node \mathcal{X} with cardinality $|\mathcal{X}| \geq 2$. Consider the hypergraph \mathcal{H}' which follows from \mathcal{H} by replacing all nodes in \mathcal{X} in all hyperedges that they appear by a new node $v_{\mathcal{X}}$. Let now $\text{tr}(\mathcal{H}')$ be the set of transversals of \mathcal{H}' . The importance of the concept of generalized node follows from the observation that $\text{tr}(\mathcal{H}) = (t \setminus v_{\mathcal{X}}) \times \mathcal{X}$, for all $t \in \text{tr}(\mathcal{H}')$ such that $v_{\mathcal{X}} \in t$. In other words, the transversals of \mathcal{H} follow by taking one by one the transversals of \mathcal{H}' that include the node $v_{\mathcal{X}}$ and replacing $v_{\mathcal{X}}$ by each node in \mathcal{X} in turn.

It is obvious that the number of transversals of \mathcal{H} produced from a single transversal of \mathcal{H}' is $|\mathcal{X}|$. The transversals of \mathcal{H}' that do not include $v_{\mathcal{X}}$ remain as they are, since they hit \mathcal{H} . Going one step further, if a hypergraph \mathcal{H} has two generalized nodes, \mathcal{X}_1 and \mathcal{X}_2 , we can compute the transversals of the hypergraph \mathcal{H}' that follows from \mathcal{H} by removing the nodes of \mathcal{X}_1 and \mathcal{X}_2 and replacing them with two new nodes, $v_{\mathcal{X}_1}$ and $v_{\mathcal{X}_2}$, respectively. We compute now the transversals of \mathcal{H} by taking the transversals of \mathcal{H}' and substituting the generalized nodes $v_{\mathcal{X}_1}$ and $v_{\mathcal{X}_2}$ (where they appear) by all possible combinations of (simple) nodes in \mathcal{X}_1 and \mathcal{X}_2 , respectively. Clearly, this procedure can be generalized to any number of generalized nodes.

We exploit the concept of the generalized vertex in such a way that, during all intermediate steps, we keep only the generalized transversals which, in turn, are split after the addition of the new hyperedge. This dramatically reduces the number of intermediate transversals, especially at the early stages (where the generalized nodes are few but large) and greatly improves the time performance and the memory requirements.

Example 1. Assume that the first two hyperedges have 100 nodes each: $\mathcal{E}_1 = \{v_1, \dots, v_{100}\}$ and $\mathcal{E}_2 = \{v_{51}, \dots, v_{150}\}$. The partial hypergraph $\{\mathcal{E}_1, \mathcal{E}_2\}$ has 2550 transversals (2500 with two nodes and 50 with one) which must be kept for the subsequent stage if we use the straightforward scheme. Using the generalized node approach, we have only 2 transversals to store, namely the set $\{\{v_{51}, \dots, v_{100}\}\}$ and the set $\{\{v_1, \dots, v_{50}\}, \{v_{101}, \dots, v_{150}\}\}$.

The second improvement to the simple scheme of Berge was especially designed having in mind the rate of output of the algorithm. Recall that the simple scheme must make a lot of computation before outputting the first transversal, after which all the rest follow almost with zero delay one from the other. This occurs because the simple scheme is based on a sort of *breadth-first* computation of the transversals (all transversals are computed after a new hyperedge is added). Instead, our implementation computes the transversals in a *depth-first* manner: At a certain level, we compute a transversal of the partial hypergraph and then

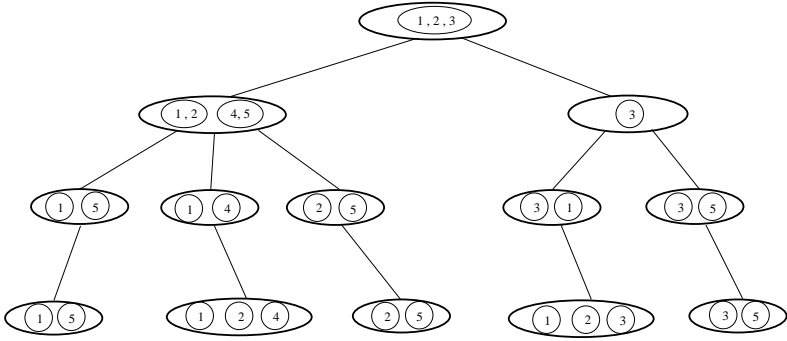


Fig. 1. Transversal tree of the hypergraph $\mathcal{H} = \{\{1, 2, 3\}, \{3, 4, 5\}, \{1, 5\}, \{2, 5\}\}$. The tree is visited in preorder.

add to it the next hyperedge. From this transversal several others follow, as we described above. However, instead of computing them all, we pick one, add the next hyperedge and continue until all hyperedges have been added; in this case we output the final transversal. We then backtrack to the previous level, pick the next transversal, etc. The whole scheme resembles a preorder visit of a tree of transversals with root the single (generalized) transversal of the first hyperedge, and internal nodes at some level, the generalized transversal of the partial hypergraph at that level. The descendants of a transversal are the transversals of the next hypergraph which include this transversal. Finally, the leaves of the tree at level m are the transversals of the original hypergraph.

Example 2. Consider the hypergraph with 5 nodes and 4 hyperedges $\mathcal{H} = \{\{1, 2, 3\}, \{3, 4, 5\}, \{1, 5\}, \{2, 5\}\}$. The tree of transversals which corresponds to the addition of the hyperedges according to the given order (top to bottom) is shown in Fig. 1. Generalized nodes are denoted by circles with thin lines. For instance, a partial transversal of the hypergraph consisting of the first two hyperedges is $\{\{1, 2\}, \{4, 5\}\}$.

The efficiency of the above is further improved by a selective way of producing new transversals at an intermediate level. This idea results in ruling out regeneration of transversals at the cost that some intermediate nodes may have no descendants. The advantage of this approach is that search in some subtrees stops at higher levels instead of exhaustively generating everything that would subsequently need to be compared to previous transversals and, possibly, discarded. The method and its correctness is described and proved in [14].

3 Comparison to Other Approaches

We already mentioned that the algorithm of Berge outputs its first transversal near the end of the total computation. Thus, it may take exponentially long time for the first transversal to output. In contrast our algorithm, as the experiments

show, delivers its output in quite a uniform way from the beginning to the end of the computation. From the implementation point of view the algorithm of Berge requires all intermediate transversals to be stored and kept until used for the computation of the new transversal set. Therefore the memory requirements are proportional to the size of the transversal tree. Since the number of transversals can be exponential, we conclude that the memory requirements can become devastating. Our algorithm instead, operates in a generate-and-forget fashion by visiting the transversal tree in preorder and so its memory requirements are proportional to the depth of the transversal tree rather than to its size. There are also other points of improvement in our algorithm compared to the algorithm of Berge; one has to do with the concept of generalized variables. In addition to the compact form of storing intermediate transversals, our algorithm also excludes the possibility of generating a transversal more than once which is a possibility in Berge's algorithm. The additional copies of a transversal must be identified and removed, otherwise they will result in a blow-up of the total number of partial transversals at the next steps. Additionally, there is an unnecessarily large number of intermediate transversals (especially in problem instances with many nodes) that have to participate in the computation of the transversal set of the new hypergraph. This is something that further adds to the blow-up of the transversals of the new hypergraph. In contrast, by using the concept of generalized variables, our algorithm greatly reduces the number of intermediate transversals, as illustrated in Example 1. Regarding the total running time of Berge's algorithm, as an example we mention that we ran Berge's algorithm for hypergraphs with 30 nodes and 30 hyperedges. The running time was more than 60 seconds while, as illustrated in Table 1, our algorithm requires only 1 second for hypergraphs with this size.

The algorithm of Fredman and Khachiyan has the best provable bound on its time performance. The algorithm actually solves the decision problem, namely the problem of deciding, given hypergraphs \mathcal{H} and \mathcal{G} , whether $\text{tr}(\mathcal{H}) = \mathcal{G}$ and, if not, it returns a minimal hitting set of one of the two hypergraphs that is not a hyperedge of the other. The algorithm runs in time $O(n^{\log n})$ where n is the combined size of \mathcal{H} and \mathcal{G} . By using repeatedly this algorithm as a subroutine, the generation problem can be solved in incremental time $O(n^{\log n})$ [10]. We are not aware of any implementation of the Fredman-Khachiyan algorithm. However, as in the previous case, both the input and the output so far have to be stored and consequently the same problem regarding the memory requirements exists here as well. Regarding its time performance, this algorithm operates in each decision step on both the input and the output so far and thus, the delay increases after each output step. In contrast, our algorithm seems to remain relatively stable in this regard. In any case, a careful implementation of the Fredman-Khachiyan algorithm would be of great interest.

Finally, for the sake of completeness, we mention the possibility of computing the transversal hypergraph by generating all possible hitting sets of the hypergraph (which are of the order of 2^n) and subsequently checking each if it is a minimal hitting set. Naturally, this simple algorithm overcomes the problem of

storing all transversals, but its time performance is unacceptable (it can be used for hypergraphs having no more than 15 nodes). Yet, we have implemented it, since it is a reliable test for verifying the correctness of our algorithm.

4 Implementation Issues

The main body of the program consists basically of two procedures. (We only mention here the more important formal parameters.)

```

procedure add_next_hyperedge(t,h) {
    Update the generalized nodes;
    while generate_next_transversal(t,t') {
        if h is last, then output t
        else {
            Let h' be the next hyperedge;
            add_next_transversal(t',h');
        }
    };
};

```

and the boolean **function** generate_next_transversal(t,t').

The first adds to the partial transversal t the next hyperedge h and repeatedly calls the second one which returns the next partial transversal t' of the new hypergraph. `generate_next_transversal` is called until no more transversals follow from t after the addition of h , in which case `generate_next_transversal` becomes false. After a new transversal t' is returned, `add_next_hyperedge` is called recursively for t' and the next hyperedge. The recursive implementation was chosen as a fast solution to the task of developing a fairly complex code. Since `add_next_hyperedge` is called once for each hyperedge, the depth of recursive calls equals the number of hyperedges. Hence, for some problem instances this can be quite large, resulting in poor memory usage and even in a code not as fast as it could be. We plan however to remove recursion from future versions of the program thus solving the above problems. This is why we have chosen to count primarily as a measure of performance the number of recursive calls between consecutive generations and not absolute time. Note however that, as shown in the next section, even the absolute time performance is quite satisfactory even for large problems. We stress that this can be improved further.

The performance of the program is very much affected by the data types used for storing generalized nodes. This is because the use of generalized nodes resulted in a code that does intensive set manipulation operations. These sets represent sets of nodes (actually integers from 1 to the number of nodes of the hypergraph). Several different methods were tried in different versions of the code. The fastest implementation represents a set as a bit vector that spans as many computer words as required depending on its size (our machine has 32-bit word). This choice may slightly limit the portability of the code but it is by far the fastest. Set operations (union, intersection, complementation, etc.) are then accomplished as low level bit operations (or, and, etc.).

5 Experimental Results

In this section we present the experimental evaluation of the performance of the algorithm on a set of test cases. Experiments were carried out on a Sun Enterprise 450, using GNU C++ 2.8.1 with compiler setting `-O3`.

Since real data were not available, the program was evaluated using a number of randomly generated hypergraphs. A random hypergraph generator was implemented and used for this task. Given the number of nodes, n , and the desired number of hyperedges, m , the random hypergraph generator uniformly and independently generates m sets of nodes, each of them corresponding to a hyperedge of the instance. The cardinality of each set lies between 1 and $n - 1$, and is randomly chosen, too. The outcoming hypergraph is simple, that is, there is no hyperedge that appears twice or is fully included in another one. Reports are averages over 50 different runs for each instance size, using a different initial seed for the random hypergraph generator in every run.

The first few experiments aimed at verifying the correctness of the code. This was done by implementing the simple algorithm that computes all transversals of a particular hypergraph using exhaustive search and then checking for minimality. Since this simple algorithm needs exponentially many steps to compute all minimal transversals, it could only be used for small instances (hypergraphs with at most 15–16 nodes). The resulting transversal hypergraph was then compared to the output of the program.

Testing the correctness of the code for larger instances was accomplished by applying the duality theorem of transversal hypergraph: $\text{tr}(\text{tr}(\mathcal{H})) = \mathcal{H}$ (see [2,6] for more theoretical issues on hypergraphs). We run our algorithm to the output for a particular instance and verified that the new output was the original input. Notice that the above theorem also offers the possibility for generating non-random problem instances with specific properties of the output that would otherwise be impossible to generate by any conventional random instance generator; we were therefore able to test our algorithm for problems with very large number of hyperedges but very few transversals.

The results of the experiments are summarized in the tables that follow. Problem sizes are identified by the number of nodes, n , and the number of hyperedges, m , of the hypergraph. In Table 1, the first three columns (after column m) give the size of the output (the minimum, the maximum, and the average), in thousands of transversals, over the 50 experiments that were conducted for each pair n and m . It is important to notice that these numbers also characterize the problem size. Larger problem instances with respect to n and m resulted in a number of transversals too large to be generated within some moderate time. We next report the performance of the algorithm in terms of its rate of output. Specifically, the next three columns give the delay time (in number of calls of procedure `add_next_hyperedge`) between consecutive outputs. The first one reports the maximum delay that was observed in the whole process of generating the transversals. This number characterizes the worst-case performance of the algorithm. The unexpectedly good behavior of this parameter is what we believe it deserves further theoretical investigation. The next column presents the av-

erage delay while the last one reports the total running time for generating all transversals. As before, every entry in these columns is the average (over the 50 runs) of the corresponding parameters. The last column of Table 1 reports the total CPU time (in seconds) required, on the average, for each run.

The experiments summarized in Table 2 aimed in better studying the rate of the output. To this end, the standard deviation σ of the delays was calculated. Each row in Table 2 corresponds to a single problem (as opposed to results from 50 runs that were reported in Table 1) since averaging standard deviations is absurd. We have chosen however, from the 50 different experiments with the same n and m , to report the one with the worst behavior with respect to σ . Even in this case, the values of σ remain relatively small (3 times the average delay at most). The size of the output (transversals) as well as the total time (number of calls of procedure `add_next_hyperedge` for generating all transversals) and the CPU time required are also reported.

In problems reported in Tables 1 and 2 all randomly generated hypergraphs have a small number of hyperedges and a very large number of transversals. Problems of this kind have relatively small delays, since the output is very large. The duality property of the transversal hypergraph mentioned above provides a way of testing the algorithm in non-random instances with very large number of hyperedges and a small (known a priori) number of transversals. As a result, the delays are now large, comparable to the total running time. The technique for doing this was by running the algorithm with input the output of a particular randomly generated instance. This also serves as a test of the correctness of the algorithm as already explained. The results are summarized in Table 3. Notice that the problem size is now defined by the number of nodes, n , and the number of transversals, while the number of edges, m , is now the size of the output.

6 Conclusion

In this paper we describe an implementation of a new algorithm for solving the TRANSVERSAL HYPERGRAPH problem. This problem has certain peculiarities regarding its efficiency, in that both the total running time and the rate of the output (delay between consecutive outputs) are of equal importance. Our experiments show a surprisingly even rate of the output, something which calls for further theoretical justification. Moreover, both the delays between consecutive outputs and the total running time suggest that quite large instances (with respect to both their input and output sizes) can be solved efficiently. Future work will include the removal of recursion, since this will further speed up the code and more in-depth study of the performance measures of the program.

Acknowledgments

The authors wish thank the referees and the program committee for their helpful comments.

Table 1. Delay time and CPU time for various problem instances. Reports are averages over 50 runs. Each problem instance is defined by the number of nodes n and the number of hyperedges m . *Transversals* corresponds to the number of generated transversals while *Delay Time* to the number of calls of procedure `add_next_hyperedge`.

	m	<i>Transversals</i> ($\times 10^3$)			<i>Delay Time</i> ($\times 10^3$)			<i>CPU Time</i> (in seconds)
		<i>min</i>	<i>max</i>	<i>average</i>	<i>max</i>	<i>average</i>	<i>total</i>	
$n = 15$	10	0.03	0.1	0.1	0.01	0.004	0.3	0
	30	0.08	0.3	0.2	0.1	0.02	3	0
	50	0.1	0.4	0.3	0.2	0.03	9	0.1
	100	0.2	0.6	0.5	0.5	0.1	30	0.2
	200	0.3	1	0.6	1	0.1	90	1
	400	0.4	1.2	0.9	2	0.3	250	2
	600	0.7	1.5	1.1	4	0.5	500	4
	800	1	1.6	1.4	5	0.6	800	7
	1000	1.4	2	1.7	6	0.7	1200	10
$n = 20$	10	0.1	0.4	0.2	0.02	0.004	0.6	0
	30	0.3	1	0.7	0.1	0.02	10	0.1
	50	0.4	2	1.2	0.3	0.03	30	0.2
	100	0.5	3	2	0.7	0.1	100	1
	200	2	6	3	2	0.1	500	5
	400	1	9	6	4	0.3	1800	20
	600	2	11	8	7	0.5	3800	50
	800	2	13	9	9	0.6	5800	70
	1000	2	14	9	12	0.9	8200	110
$n = 30$	10	0.2	2	0.6	0.01	0.003	2	0
	30	2	12	5	0.2	0.01	70	1
	50	2	20	10	0.4	0.02	200	2
	70	5	40	20	1	0.03	700	10
	100	5	80	30	1	0.05	1800	20
	200	10	200	80	3	0.1	9100	150
	400	30	350	200	9	0.2	46000	900
	600	15	500	300	12	0.4	11000	7700
$n = 40$	10	0.1	3	1.3	0.01	0.003	4	0
	30	5	60	25	0.4	0.01	300	4
	50	11	300	90	1	0.02	1600	20
	70	13	400	200	2	0.03	4400	80
	100	13	1300	400	3	0.04	1600	360
$n = 50$	10	0.3	8	3	0.02	0.003	7	0.1
	30	5	400	100	0.5	0.01	900	10
	50	40	2000	470	1	0.02	7200	140
	70	20	4100	1200	4	0.02	28000	2300
$n = 60$	10	1	15	5	0.02	0.002	12	0.3
	30	20	1500	300	1	0.01	2600	50
	50	30	3700	1300	5	0.02	21000	450

Table 2. Statistical measures (*Average Delay* and standard deviation σ) for individual problem instances. Entries at the 1st and 2nd column (*Transversals* and *Total Time*, respectively) correspond to the number of generated transversals and the number of calls of procedure `add_next_hyperedge`, respectively.

	<i>m</i>	<i>Transversals</i> ($\times 10^3$)	<i>Total Time</i> ($\times 10^3$)	<i>Average Delay</i> ($\times 10^3$)	σ ($\times 10^3$)	<i>CPU Time</i> (<i>in seconds</i>)
<i>n</i> = 15	10	0.1	0.4	0.01	0.002	0
	30	0.1	2	0.02	0.022	0
	50	0.3	10	0.04	0.040	0.1
	100	0.2	30	0.1	0.1	0.2
	200	0.4	70	0.2	0.2	1
	400	0.5	200	0.4	0.5	2
	600	0.7	450	0.6	0.8	4
	800	1	820	0.7	0.8	7
	1000	2	1200	0.8	1	10
<i>n</i> = 20	10	0.2	0.8	0.01	0.01	0
	30	0.3	6	0.02	0.03	0.1
	50	1	40	0.04	0.06	0.4
	100	2	100	0.1	0.1	2
	200	3	650	0.3	0.4	10
	400	6	250	0.4	0.6	40
	600	6	5000	0.7	0.9	70
	800	6	6000	1	1.4	100
	1000	5	8000	1.6	2	130
<i>n</i> = 30	10	0.6	3	0.004	0.01	0
	30	4	80	0.02	0.03	1
	50	10	500	0.04	0.08	10
	70	20	100	0.07	0.1	20
	100	30	3000	0.1	0.2	70
	200	80	2000	0.2	0.4	400
	400	70	33000	0.5	1	1000
	600	300	180000	0.7	1.2	6000
<i>n</i> = 40	10	2	6	0.003	0.003	0
	30	40	600	0.01	0.03	10
	50	70	2000	0.03	0.06	40
	70	300	11000	0.04	0.13	300
	100	700	40000	0.06	0.14	1100
<i>n</i> = 50	10	2	10	0.005	0.01	0.1
	30	80	1000	0.01	0.03	10
	50	600	13000	0.02	0.05	300
	70	4000	120000	0.03	0.1	6000
<i>n</i> = 60	10	2	8	0.005	0.01	0.1
	30	400	4200	0.01	0.04	100
	50	3400	63000	0.02	0.06	1600

Table 3. Total time, CPU time, and statistical measures for individual non-random (dual) problem instances. The size of the output, m , is known a priori. Each row corresponds to individual runs. *Total Time* is the number of calls of procedure `add_next_hyperedge` for the generation of all transversals.

<i>Transversals</i>	m	<i>Total Time</i> ($\times 10^3$)	<i>Average Delay</i> ($\times 10^3$)	$\sigma(\times 10^3)$	<i>CPU Time</i> (<i>in seconds</i>)	
$n = 15$	80	10	0.7	0.07	0.03	0
	200	30	10	0.4	0.3	0.1
	200	50	10	0.2	0.1	0.1
	500	100	90	1	0.8	1
	800	200	200	1	0.6	2
	1000	400	450	1	0.8	4
	1100	600	500	0.8	0.6	4
	1200	800	700	1	0.6	5
	1700	1000	1500	1.5	1.1	10
$n = 20$	250	10	4	0.4	0.2	0.1
	500	30	40	1.5	1	1
	1100	50	20	3	2	4
	1400	100	400	4	4	10
	3000	200	1900	10	8	50
	8000	400	25000	60	50	500
	8000	600	29000	5	40	500
	9000	800	20000	20	20	500
	12000	1000	44000	40	40	1000
$n = 30$	600	10	20	2	2	1
	4200	30	2700	90	130	300
	20000	50	103000	2100	2700	12000
$n = 40$	1300	10	50	5	5	20
	4300	20	3000	150	190	500
	14000	30	51000	1700	1900	12000
$n = 50$	2600	10	100	10	10	70
	32000	20	27000	1400	1600	45000

References

1. R. Ben-Eliyahu and R. Dechter. On computing minimal models. *Annals of Mathematics and Artificial Intelligence*, 18:3–27, 1996.
2. C. Berge. *Hypergraphs*, volume 45 of *North Holland Mathematical Library*. Elsevier Science Publishers B.V., Amsterdam, 1989.
3. M. Cadoli. The complexity of model checking for circumscriptive formulae. *Information Processing Letters*, 42:113–118, 1992.
4. Z. Chen and S. Toda. The complexity of selecting maximal solutions. *Information and Computation*, 119:231–239, 1995.
5. J. de Kleer, A. K. Mackworth, and R. Reiter. Characterising diagnosis and systems. *Artificial Intelligence*, 56:197–222, 1992.
6. T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM J. Computing*, 24(6):1278–1304, December, 1995.

7. M. L. Fredman and L. Khachiyan. On the complexity of dualization of monotone disjunctive normal forms. *Journal of Algorithms*, 21:618–628, 1996.
8. H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of ACM*, 32(4):841–860, 1985.
9. D. Gunopulos, R. Khardon, H. Mannila, and H. Toinonen. Data mining, hypergraph transversals, and machine learning. In *Proc. of Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 209–216, Tucson, Arizona, USA, May 12–14, 1997.
10. V. Gurvich and L. Khachiyan. Generating the irredundant conjunctive and disjunctive normal forms of monotone Boolean functions. Technical Report LCSR-TR-251, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, 1995.
11. D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, 1988.
12. D. Kavvadias and M. Sideri. The inverse satisfiability problem. *SIAM J. Computing*, 28(1):152–163, 1999.
13. D. J. Kavvadias, M. Sideri, and E. C. Stavropoulos. Generating all maximal models of a Boolean expression. Submitted.
14. D. J. Kavvadias and E. C. Stavropoulos. A new algorithm for the transversal hypergraph problem. Technical Report CTI TR 99.03.03, Computer Technology Institute, Patras, Greece, March 1999.
15. H. Mannila and K. J. Räihä. Design by example: An application of Armstrong relations. *Journal of Computer and System Sciences*, 32(2):126–141, 1986.
16. H. Mannila and K. J. Räihä. Algorithms for inferring functional dependencies. *Data & Knowledge Engineering*, 12(1):83–99, February, 1994.
17. H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. Technical Report C-1997-8, Department of Computer Science, University of Helsinki, Finland, 1997.
18. J. McCarthy. Circumscription—a form of nonmonotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
19. C. H. Papadimitriou. NP-completeness: A retrospective. In *Proc. of ICALP 98*, Bologna, Italy, 1998.
20. B. Selman and H. K. Kautz. Model preference default theories. *Artificial Intelligence*, 45:287–322, 1990.