Efficient Algorithms for Dualizing Large-Scale Hypergraphs

Keisuke Murakami[*] Takeaki Uno[†]

Abstract

A hypergraph $\mathcal{F}$ is a set family defined on vertex set V. The dual of $\mathcal{F}$ is the set of minimal subsets H of V such that $F \cap H \neq \emptyset$ for any $F \in \mathcal{F}$. The computation of the dual is equivalent to many problems, such as minimal hitting set enumeration of a subset family, minimal set cover enumeration, and the enumeration of hypergraph transversals. In this paper, we introduce a new set system induced by the minimality condition of the hitting sets, that enables us to use efficient pruning methods. We further propose an efficient algorithm for checking the minimality, that enables us to construct time efficient and polynomial space dualization algorithms. The computational experiments show that our algorithms are quite fast even for large-scale input for which existing algorithms do not terminate in practical time.

1 Introduction

A *hypergraph* $\mathcal{F}$ is a subset family defined on a vertex set V, that is, each element (called *hyperedge*) F of $\mathcal{F}$ is a subset of V. A subset H of V is called a *vertex subset*. A *hitting set* is a subset H of V such that $H \cap F \neq \emptyset$ for any hyperedge $F \in \mathcal{F}$. A hitting set is called *minimal* if it includes no other hitting set. The *dual* of a hypergraph is the set of all minimal hitting sets. The *dualization* of a hypergraph is to construct the dual of a given hypergraph. For $V = \{1, 2, 3, 4\}, \mathcal{F} = \{\{1, 2\}, \{1, 3\}, \{2, 3, 4\}\}$, $\{1, 3, 4\}$ is a hitting set but not minimal, and $\{2, 3\}$ is a minimal hitting set. $dual(\mathcal{F})$ is $\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}\}$. It is known that $\mathcal{F} = dual(dual(\mathcal{F}))$ if no $F, F' \in \mathcal{F}$ satisfies $F \subset F'$.

Dualization is a fundamental problem in computer science, especially in machine learning, data mining, and optimization, etc. It is equivalent to (1) the minimal hitting set enumeration of a given subset family, (2) minimal set cover enumeration of a given set family, (3) enumeration of minimal subsets that are not included in any of the given set family, (4) generating CNF equivalent to a given monotone DNF, etc. The size of dual can be exponential in the input hypergraph, thus a polynomial time algorithm for dualization usually means an algorithm running in time polynomial to the input size and the output size. Although Fredman and Khachiyan[6] developed a quasi-polynomial time algorithm which runs in $O(N^{o(\log N)})$ time, where N is the input size plus output size, the existence of a polynomial time algorithm is still open.

From the importance of dualization in its application areas, a lot of researches have aimed at algorithms that terminate in short time on real world data. Reduction of the search space was studied as a way to cope with this problem [5, 6, 9, 10, 13, 16]. Finding a minimal hitting set is easy; one removes vertices one by one unless each has an empty intersection with some hyperedges. However, finding exactly all minimal hitting sets is not easy; we have to check a great many vertex subsets. The past studies have succeeded in reducing the search space, but the computational cost was substantial, hence the current algorithms may take long time when the size of the dual is large. Thus, we could solve when the input size is small, or the output size is small. Solving the dualization problem in short time when both the input and the output sizes are large is an open issue.

Usually, the size of dual increases when the average size of minimal hitting sets increases. Thus, tractable real-world problems have minimal hitting sets of small sizes on average. This motivates us to perform a hill-climbing search to find all minimal hitting sets, but we cannot use existing efficient pruning methods for maximal element enumerations. On the contrary, if we consider the complements of hitting sets, the minimal hitting sets will be maximal sets. It enables us to use the existing pruning methods, but search space drastically increases. This is one of the difficulties of the problem.

On the other hand, some incremental algorithms have been proposed[5, 10] that add hyperedges one by one and update the set of minimal hitting sets. These algorithms have smaller search spaces, but may operate one solution many times. Some existing algorithms are based on a breadth-first search to perform minimality check efficiently by using the smaller minimal hitting sets already found. This works, but consumes memory. The algorithm by [10] uses a depth-first search, but the minimality check takes long time for large scale problems.

[*]Aoyama Gakuin University, Japan.
[†]National Institute of Informatics, Japan.

In this paper, we propose a new approach to use a set system given by the minimality condition of the hitting sets. We propose a new concept called *critical hyperedge*. The sets system composed of vertex sets such that each its vertex has a critical hyperedge satisfies the monotone property, and any minimal hitting set is a maximal element of the set system. Hence, we can use hill climbing search with existing efficient pruning methods. The critical hyperedge can be computed in short time without using the minimal hitting sets already found, thus we can reduce the time complexity of the minimality check. This realizes a time efficient computation with small memory space. Moreover, we developed several methods for adapting the sparsity of the input, so that we can skip unnecessary operations. This drastically reduces the number of iterations, compared to the existing algorithms. The computational experiments show the efficiency of these methods so that in many problems our algorithms terminate in short time even when the existing algorithms do not terminate in several hours.

In the following, the proofs and some results of the computational experiments are omitted due to the limited space. They are in the appendix, and the long version[12].

2 Preliminaries and Preparations

For a hypergraph $\mathcal{F} = \{F_1, \ldots, F_m\}$ defined on vertex set V, $|\mathcal{F}|$ denotes the number of hyperedges in $\mathcal{F}$, that is m, and $||\mathcal{F}||$ denotes the sum of the sizes of hyperedges in $\mathcal{F}$, respectively. $\mathcal{F}_i$ denotes the hypergraph composed of hyperedges $\{F_1, \ldots, F_i\}$. For $v \in V$, let $\mathcal{F}(v)$ be the set of hyperedges in $\mathcal{F}$ that includes v, i.e., $\mathcal{F}(v) = \{F | F \in \mathcal{F}, v \in F\}$. For vertex subset S and vertex v, we respectively denote $S \cup \{v\}$ and $S \setminus \{v\}$ by $S \cup v$ and $S \setminus v$.

For a vertex subset $S \subseteq V$, $uncov(S)$ denotes the set of hyperedges that do not intersect with S. S is a hitting set if and only if $uncov(S) = \emptyset$. For a vertex $v \in S$, a hyperedge $F \in \mathcal{F}$ is said to be *critical* for v if $S \cap F = \{v\}$. We denote the set of all critical hyperedges for v by $crit(v, S)$, i.e., $crit(v, S) = \{F | F \in \mathcal{F}, S \cap F = \{v\}\}$. Suppose that S is a hitting set. If v has no critical hyperedge, every $F \in \mathcal{F}$ includes a vertex in S other than v, thus $S \setminus v$ is also a hitting set. Therefore, we have the following property.

PROPERTY 2.1. *$S \subseteq V$ is a minimal hitting set iff $uncov(S) = \emptyset$, and $crit(v, S) \neq \emptyset$ holds for any $v \in S$.*

If $crit(v, S) \neq \emptyset$ for any $v \in S$, we say that S satisfies the *minimality condition.* Let us consider an example of *crit*. Suppose that $\mathcal{F} = \{\{1,2\}, \{1,3\}, \{2,3,4\}\}$, and the hitting set S is $\{1,3,4\}$. We can see that $crit(1, S) = \{\{1,2\}\}, crit(3, S) = \emptyset, crit(4, S) = \emptyset$, thus S is not minimal, and we can remove either 3 or 4. For $S' = \{1,3\}$, $crit(1, S') = \{\{1,2\}\}, crit(3, S') = \{\{2,3,4\}\}$, thus S' is a minimal hitting set. Our algorithm updates *crit* to check the minimality condition quickly, by utilizing the following lemmas.

LEMMA 2.1. *For vertex subset S, $v \in S$ and $v' \notin S$, $crit(v, S \cup v') = crit(v, S) \setminus \mathcal{F}(v')$. Particularly, $crit(v, S \cup v') \subseteq crit(v, S)$ holds.* ∎

LEMMA 2.2. *For any vertex subset S and $v' \notin S$, $crit(v', S \cup v') = uncov(S) \cap \mathcal{F}(v')$.* ∎

The next two lemmas follow directly from the above.

LEMMA 2.3. *[9, 13] If a vertex subset S satisfies the minimality condition, any of its subsets also satisfy the minimality condition, i.e., the minimality condition satisfies the monotone property.* ∎

LEMMA 2.4. *[9, 13] If a vertex subset S does not satisfy the minimality condition, S is not included in any minimal hitting set. In particular, any minimal hitting set S is maximal in the set system composed of vertex subsets satisfying the minimality condition.* ∎

LEMMA 2.5. *For any vertex subset S, $\sum_{v \in S} |crit(v, S)| \leq |\mathcal{F}|$.* ∎

3 Existing Algorithms

This section is devoted to explaining the framework of the existing algorithms related to our algorithms: DL algorithm, KS algorithm, and MTminer. DL algorithm proposed by Dong and Li[5] iteratively computes $dual(\mathcal{F}_i)$ from $dual(\mathcal{F}_{i-1})$. For any $S \in dual(\mathcal{F}_i)$, either $S \in dual(\mathcal{F}_{i-1})$ holds, or $S \setminus v \in dual(\mathcal{F}_{i-1})$ holds for $\{v\} = S \cap F_i$. Note that when $S \in dual(\mathcal{F}_i)$ is not in $dual(\mathcal{F}_{i-1})$, $S \cap F_i$ is composed of exactly one vertex, since $crit(v, S)$ must be $\{F_i\}$. However, for any $S \in dual(\mathcal{F}_{i-1})$, $S \in dual(\mathcal{F}_i)$ if $S \cap F_i \neq \emptyset$. When $S \cap F_i = \emptyset$, $S \cup v$ with $v \in F_i$ may be in $dual(\mathcal{F}_i)$. BMR algorithm proposed by Bailey, Manoukian and Ramamohanarao[2] is an improved version of DL algorithm. The algorithm re-orders the hyperedges and reduces the vertex sets in each $\mathcal{F}_i$, to avoid generating unnecessary hitting sets. DL algorithm is written as follows.

```
ALGORITHM DL ($\mathcal{F} = \{F_1, \ldots, F_m\}$)
1. $\mathcal{D}_0 := \{\emptyset\}$
2. for $i := 1$ to $m$
3.   $\mathcal{D}_i := \emptyset$
4.   for each $S \in \mathcal{D}_{i-1}$ do
```

```
5.     if $S \cap F_i \neq \emptyset$ then insert $S$ to $\mathcal{D}_i$
6.     else for each $v \in F_i$ do
7.       if no $S' \in \mathcal{D}_{i-1}$ satisfies $S' \neq S$ and $S' \subseteq S \cup v$
           then insert $S \cup v$ to $\mathcal{D}_i$
8.     end for
9.   end for
10. end for
```

DL algorithm checks whether $S \cup v$ is in $dual(\mathcal{F}_i)$ or not by looking for a hitting set in $dual(\mathcal{F}_{i-1})$ included in $S \cup v$. Thus, we need to perform a breadth-first search. This needs basically $O(\sum ||dual(\mathcal{F}_i)||)$ time and is a bottleneck computation of the algorithm. Kavvadias and Stavropoulos[10] proposed a depth-first version of this algorithm. According to the hitting set generation rule, each hitting set in $\mathcal{F}_i$ is uniquely generated from a hitting set of $\mathcal{F}_{i-1}$. This gives a depth-first search starting from each hitting set in $\mathcal{F}_1$. The algorithm checks the minimality of $S \cup v$ by checking whether $S \cup v \setminus f$ is a hitting set or not for each $f \in S$. This check needs basically $O(|S||\mathcal{F}|)$ time. The algorithm is written as follows.

```
ALGORITHM KS ($S$, $i$)
1. if $i = m$ then output $S$; return
2. if $S \cap F_i \neq \emptyset$ then call KS($S$, $i+1$)
3. else for each $v \in F_i$ do
4.   for each $u \in S$ do
5.     if $S \cup v \setminus u$ is a hitting set then go to 8.
6.   end for
7.   call KS($S \cup v$, $i+1$)
8. end for
```

The cost for the minimality check increases with $|dual(\mathcal{F}_{i-1})|$, for DL algorithm, and with S and $||\mathcal{F}||$ for KS algorithm. Thus, DL algorithm will be faster when the $dual(\mathcal{F})$ is small, whereas KS algorithm will be faster when $||\mathcal{F}||$ is small and S is small on average.

Mtminer is a kind of branch and bound algorithm. It starts from the emptyset, and chooses elements one by one. For each element v, it generates two recursive calls concerned with a choice; add v to the current vertex subset, and do not add it. When the current vertex subset becomes a hitting set, it checks the minimality, and outputs it if minimal. To speed up the computation, the algorithm prunes branches through the use of the so called Galois condition. The Galois condition for S and $v \notin S$ is $|uncov(S)| = |uncov(S \cup v)|$, and when it holds, $S \cup v$ is never included in a minimal hitting set, thus we can terminate the recursive call with respect to $S \cup v$. The Galois condition is equivalent to our minimality condition, since it is equivalent to $crit(v, S \cup v) = \emptyset$[1].

```
ALGORITHM Mtminer ($\mathcal{F} = \{F_1, \ldots, F_m\}$)
1. $\mathcal{D}_0 := \{\emptyset\}$ ; $i = 0$
2. while $\mathcal{D}_i \neq \emptyset$
3.   for each $S \in \mathcal{D}_i$ do
4.     if $uncov(S) = \emptyset$ then output $S$
5.     for each $v$ larger than maximum vertex in $S$ do
6.       if $S \cup v$ satisfies the Galois condition then
             insert $S$ to $\mathcal{D}_i$
7.     end for
8.   end for
9. end while
```

4 New Algorithms and Minimality Check

Lemmas 2.1,2.3,2.4 imply that the set system composed of vertex subsets satisfying the minimality condition is anti-monotone. Furthermore, any non-minimal hitting set does not satisfy the condition and any minimal hitting set satisfies the condition, hence any minimal hitting set is a maximal element in the set system. This enables us to use many efficient existing techniques and algorithms for maximal element enumeration. Especially, it is a big advantage that we can use many pruning methods designed for maximal element enumeration. Our Minimal-to-Maximal Conversion Search (MMCS for short) performs a a depth-first search on this set system with several pruning methods. MTminer also uses the minimality condition, but performs a breadth-first search, thus it cannot use our pruning methods.

In the existing algorithms, this check was done by looking the smaller minimal hitting sets already found, or scanning all hyperedges, thus took long time. A key to an efficient search is a minimality check algorithm with small computational cost. We propose to use *crit* for checking the minimality condition. Based on the Lemmas 2.1 and 2.5, we can update *crit* in short time, thus the minimality check can be done efficiently. Our algorithm keeps $crit[u]$ and *uncov* as lists, that represent $crit(u, S)$ and $uncov(S)$, with a mark for each F_i that the mark of F_i is u if it is included in $crit[u]$, is $|V|+1$ if it is included in $uncov(S)$, and is $|V|+2$ if F_i is included in none of these. It enables us to find the list that includes F_i in constant time. The update of the mark can be also done in constant time, for each insertion and deletion of the list. When the algorithm adds a vertex v to S and generates a recursive call, it updates $crit[]$ and *uncov* to $crit[] \setminus \mathcal{F}(v)$ and $uncov \setminus \mathcal{F}(v)$, by removing each $F \in \mathcal{F}(v)$ from $crit[u]$ if F has mark u. This update operation takes $O(\min\{|\mathcal{F}(v)|, (|uncov(S)| + \sum_{v \in S} |crit(v, S)|)\})$ time. Even though this operation is simple, we can reduce the

[1]the Galois condition is proposed in 2007[9], while *crit* is proposed in 2003[13, 16]. The term "minimality condition" first appears in [9].

time complexity of an iteration of KS algorithm from $O(|V| \times ||\mathcal{F}||)$ to $O(||\mathcal{F}||)$.

```
Update_crit_uncov ($v, crit[](= crit(*, S))$,
          $uncov(= uncov(S))$)
1. **for each** $F \in \mathcal{F}(v)$ **do**
2.   **if** $F \in crit[u]$ for a vertex $u \in S$ **then**
          remove $F$ from $crit[u]$
3.   **if** $F \in uncov$ **then**
          $uncov := uncov \setminus F$; $crit[v] := crit[v] \cup \{F\}$
4. **end for**
```

4.1 Minimal-to-Maximal Conversion Search with Pruning

Our MMCS algorithm starts from $S = \emptyset$, and adds vertices to S recursively unless the minimality condition is violated. To avoid duplication, we use a list of vertices $CAND$ that represents the vertices that can be added in the iteration. The vertices not included in $CAND$ will not be added, even if the addition satisfies the minimality condition, i.e., the iteration given S and $CAND$ enumerates all minimal hitting sets including S and included in $S \cup CAND$ by recursively generating calls.

Suppose that an iteration is given S and $CAND$, and without loss of generality $CAND = \{v_1, \ldots, v_k\}$. For the first vertex v_1, we make a recursive call with respect to $S \cup v_1$, with $CAND = CAND \setminus v_1$, to enumerate all minimal hitting sets including $S \cup v_1$. After the termination of the recursive call, we generate a recursive call for $S \cup v_2$. To avoid finding the minimal hitting sets including v_1, we give $CAND \setminus \{v_1, v_2\}$ to the recursive call. In this way, for each vertex v_i, we generate a recursive call with $S \cup v_i$ and $CAND = \{v_{i+1}, \ldots, v_k\}$. This search strategy is common to many algorithms for enumerating members in a monotone set system, for example clique enumeration [14]. It means that the correctness is already proved.

```
Global variable: $crit[]$, $uncov$, $\mathcal{F}$
**Basic_MMCS** ($S$, $CAND$)
1. **for each** $v \in CAND$ **do**
2.   **if** $uncov \setminus \mathcal{F}(v) = \emptyset$ **then**
       output $S \cup v$ ; $CAND := CAND \setminus v$
3.   **else if** $S \cup v$ does not satisfy minimality condition
       **then** $CAND := CAND \setminus v$
4. **end for**
5. **for each** $v \in CAND$ **do**
6.   **call Update_crit_uncov** $(v, crit[], uncov)$;
7.   **call Basic_MMCS** $(S \cup v, CAND \setminus v)$;
8.   recover $crit$
9. **end for**
```

Next, let us describe a pruning method coming from the necessary condition to be a hitting set. Suppose that an iteration is given S and $CAND$, and let F be a hyperedge in $uncov(S)$. We can see that any minimal hitting set including S has to include at least one vertex in F. Thus, we have to generate recursive calls with respect to vertices in $CAND \cap F$, but do not have to do so for vertices in $CAND \setminus F$. This condition holds for any $F \in uncov(S)$, thus we choose one minimizing $|F \cap CAND|$. In our pre-experiment, the time for the choice is usually shorter than the time saved by the choice, thus we can reduce the total computation time.

pruning method: Suppose that $S \cup v$ does not satisfy the minimality condition. From Lemma 2.3, we observe that $S' \cup v$ does not satisfy the minimality condition if $S \subseteq S'$. This means that v is never added in the recursive call with respect to S. This condition also holds when $S \cup v$ is a minimal hitting set, since no superset of a minimal hitting set satisfies the minimality condition. We call the vertex v satisfying one of these conditions *violating*. We remove all violating vertices from $CAND$ in every iteration, in the following way.

Suppose that an iteration is given S and $CAND$, and is going to generate recursive calls with respect to vertices $v_1, \ldots, v_k \in F \cap CAND$. Basically, the iteration makes recursive calls with respect to (1) $S \cup v_1$ and $CAND \setminus \{v_1\}$, (2) $S \cup v_2$ and $CAND \setminus \{v_1, v_2\}$, ..., and (k) $S \cup v_k$ and $\emptyset$. For the efficient computation of violating vertices, we do these recursive calls in the reverse order, from (k) to (1). Then, we first set $CAND$ to $CAND \setminus \{v_1, \ldots, v_k\}$. We check whether v_k is a violating vertex or not, and generate a recursive call for $S \cup v_k$ if v_k is not violating. After the recursive call, we add v_k to $CAND$ for the process of v_{k-1}. If v_k is a violating vertex, we do not add v_k to $CAND$. Then, we do the same for v_{k-1}, In this way, when we generate a recursive call with respect to $S \cup v_h$, $CAND$ is composed of all non-violating vertices $v_j, j > h$.

```
global variable: $crit[u]$ (initially $\emptyset$ for each $u$),
   $uncov$ (initially $\mathcal{F}$), $CAND$ (initially $V$)
ALGORITHM **MMCS** ($S$)
1. **if** $uncov = \emptyset$ **then output** $S$ ; **return**
2. choose a hyperedge $F$ from $uncov$;
3. $C := CAND \cap F$; $CAND := CAND \setminus C$
4. **for each** $v \in C$ **do**
5.   call Update_crit_uncov $(v, crit[], uncov)$
6.   **if** $crit(f, S \cup v) \neq \emptyset$ for each $f \in S$ **then**
        **call MMCS**$(S \cup v)$; $CAND := CAND \cup v$
7.   recover the change to $crit[]$ and $uncov$ done in 5
8. **end for**
```

Theorem 4.1. *Algorithm MMCS enumerates all minimal hitting sets within $O(||\mathcal{F}||)$ time for each iteration and $O(||\mathcal{F}||)$ memory.*

Note that the time complexity is per iteration, thus the algorithm is not output polynomial.

4.2 Reverse Search Algorithm Our second algorithm is based on the reverse search [1] that reduces the search space much more, and can be considered as an irredundant version of KS algorithm. Let $\mathcal{S} = \bigcup_{i=1}^{m} dual(\mathcal{F}_i)$, that is the set of vertex subsets that are operated by KS algorithm. Let us denote the minimum i such that $F_i \in crit(v, S)$ by $min\_crit(v, S)$, and the minimum i such that $F_i \in uncov(S)$ by $min\_uncov(S)$. $min\_crit(v, S)$ (resp., $min\_uncov(S)$) is defined as $m+1$ if $crit(v, S)$ (resp., $uncov(S)$) is empty. Using these terms, we give a characterization of $\mathcal{S}$.

LEMMA 4.1. *$S \neq \emptyset$ belongs to $\mathcal{S}$ iff $min\_crit(v, S) < min\_uncov(S)$ holds for any $v \in S$.* ∎

For $S \in \mathcal{S}$, we define $max\_min\_crit(S)$ by the minimum index i such that S is a minimal hitting set of $\mathcal{F}_i$, i.e., $max\_min\_crit(S) = \max_{v\in S}\{min\_crit(v, S)\}$. We define the *parent* $P(S)$ of S by $S\backslash v$, where v is the vertex such that $min\_crit(v, S) = max\_min\_crit(S)$. Since any F_i is critical for at most one vertex, $max\_min\_crit(S)$ and the parent are uniquely defined. The parent-child relation given by this definition is acyclic, thus forms a tree spanning all the vertex subsets in $\mathcal{S}$ and rooted at the emptyset. Our algorithm performs a depth-first search on this tree starting from the emptyset. This kind of search strategy is called reverse search[1].

This search strategy is essentially equivalent to KS algorithm modified so that we skip all redundant iterations in which we add no vertex to the current vertex subset. In a straightforward implementation of KS algorithm, we have to iteratively compute the intersection of F_i and the current vertex subset S until we meet the F_i that does not intersect with S. When $uncov(S)$ is not so large, it takes long time. Particularly, when $uncov(S) = \emptyset$, we may spend $\Theta(||\mathcal{F}||)$ time. On contrary, our algorithm has only to maintain $uncov(S)$, that is much lighter.

The depth-first search starts from the emptyset. When it visits a vertex subset S, it finds all children of S iteratively and generates a recursive call for each child. In this way, we can perform a depth-first search only by finding children of the current vertex subset. The way to find the children is shown in the following lemma.

LEMMA 4.2. *Let $S \in \mathcal{S}$ and $i = min\_uncov(S)$. A vertex subset S' is a child of S iff (1) $i < m+1$, (2) $S' = S \cup v$ for some $v \in F_i$, and (3) $max\_min\_crit(v', S') < i$ holds for any $v' \in S$.* ∎

RS algorithm can also use violating vertices for pruning. We find all violating vertices before step 3 and put marks on them. We then make recursive calls only for non-violating vertices, and never add the violating vertices, that are marked, during the execution of the recursive call. Those marks put in the iteration is deleted when the iteration ends. In our implementation, we apply this pruning only for the vertices in F_i, so that the computation time for pruning will never be the bottleneck.

```
global variable: $crit[u]$ (initially $\emptyset$ for each $u$),
   $uncov$ (initially $\mathcal{F}$)
ALGORITHM **RS** ($S$)
1. **if** $uncov = \emptyset$ **then output** $S$; **return**
2. $i := min\{j|F_j \in uncov\}$
3. **for each** $v \in F_i$, put a mark if $v$ is violating (similar to 4-8)
4. **for each** $v \in F_i$ **do**
5.   call Update_crit_uncov ($v, crit[], uncov$)
6.   **if** $\min\{t|F_t \in crit[f]\} < i$ for each $f \in S$ **then**
         **call RS**($S \cup v$)
7.   recover the change to $crit[]$ and $uncov$ done in 5
8. **end for** 9. delete marks put in step 3.
```

THEOREM 4.2. *Algorithm RS enumerates all minimal hitting sets in $O(||\mathcal{F}|| \times |\mathcal{S}|)$ time for each iteration and $O(||\mathcal{F}||)$ space.*

Proof. Since the parent-child relationship induces a rooted tree spanning all vertex subsets in $\mathcal{S}$, the algorithm certainly enumerates all vertex subsets in $\mathcal{S}$. Since any minimal hitting set is included in $\mathcal{S}$, all minimal hitting sets are found by the algorithm. The update of $crit[]$ and $uncov$ is done in $O(|F(v)|)$ time, thus an iteration of the algorithm takes $O(||\mathcal{F}||)$ time. In total, the algorithm takes $O(||\mathcal{F}|| \times ||\mathcal{S}||)$ time.

The algorithm requires extra memory for storing $crit[]$ and $uncov$ and for memorizing the hyperedges removed in step 4. Since $crit[]$ and $uncov$ are pairwise disjoint, the total memory for $crit$ and $uncov$ is $O(m)$. If a hyperedge is removed from a list, it will not be removed again in the deeper levels of the recursion, from the monotonicity of $crit$. Thus, it also needs $O(m)$ memory. The most memory is for $\mathcal{F}(v)$ of each v, and takes $O(||\mathcal{F}||)$ space. ∎

THEOREM 4.3. *Algorithm RS enumerates all minimal hitting sets in $O(||\mathcal{F}|| \times |\mathcal{S}|)$ time for each iteration and $O(||\mathcal{F}||)$ space.* ∎

5 Computational Experiments

In this section, the practical efficiency of our algorithms is evaluated by the computational experiments on various instances. Our algorithms ware implemented in C, without any sophisticated library such as binary tree. Existing algorithms were implemented in C++ by using the vector class in STL.

Table 1: Computation time on randomly generated instances

probability	0.9	0.8	0.7	0.6	0.5
BEGK	64	510	-	-	-
DL	20	210	-	-	-
BMR	1.8	20	320	-	-
MTminer	0.043	0.74	9.0	130	-
KS	3.1	37	290	-	-
RS	0.12	0.87	6.4	52	390
MMCS	0.093	0.84	6.1	52	410
cRS	0.13	2.6	29	300	-
cMMCS	0.087	1.8	21	250	-
#vertices	50	50	50	50	50
$\|\mathcal{F}\|$	1000	1000	1000	1000	1000
$\|F\|^*$	45.06	39.90	35.02	29.95	25.36
$\|dual(\mathcal{F})\|$	30429	364902	2509943	16809231	115198131
$\|S\|^*$	3.75	4.88	5.94	7.31	9.01

Table 2: Computation time on self-dual threshold graphs

SDTH	122	162	202	242	282
BEGK	27	110	310	400	870
DL	0.13	0.37	0.83	1.1	2.0
BMR	0.87	2.7	7.2	11	20
MTminer	-	-	-	-	-
KS	6.3	25	74	180	390
RS	0.017	0.049	0.065	0.10	0.17
MMCS	0.025	0.041	0.068	0.11	0.18
#vertices	122	162	202	242	282
$\|\mathcal{F}\|$	3662	6482	10102	14522	19742
$\|F\|^*$	4.45	4.46	4.47	4.47	4.48
$\|dual(\mathcal{F})\|$	3662	6482	10102	14522	19742
$\|S\|^*$	4.45	4.46	4.47	4.47	4.48

Table 3: Computation time on Self-dual Fano-plane graphs

SDFP	16	23	30	37	44
BEGK	1.3	27	590	-	-
DL	0.004	0.22	22	-	-
BMR	0.003	0.11	3.4	260	-
MTminer	0.011	0.62	70	-	-
KS	0.002	0.032	0.64	26	-
RS	0.001	0.022	0.39	16	530
MMCS	0	0.014	0.42	20	-
#vertices	16	23	30	37	44
$\|\mathcal{F}\|$	64	365	2430	16843	117692
$\|F\|^*$	6.27	9.63	12.89	15.97	18.99
$\|dual(\mathcal{F})\|$	64	365	2430	16843	117692
$\|S\|^*$	6.27	9.63	12.89	15.97	18.99

KS algorithm and Fredman Khachiyan algorithm (BEGK[4, 11]) were given by the authors of the papers. MTminer is given by François Rioult, and is available at https://forge.greyc.fr/projects/kdariane/wiki/Mtminer. All experiments were done on a 3.2 GHz Core i7-960 with a Linux operating system with 24GB of RAM memory. Note that none of the implementations used multi-cores. The codes and the instances are available at our Dualization Repository (http://research.nii.ac.jp/ũno/dualization.html).

5.1 Problem Instances We used almost all instances examined in the existing studies we referred, and prepared several new instances. The first category consists of randomly generated instances. Each hyperedge includes a vertex i with probability p. Note that we prepared one instance for each parameter, thus the results are not an average. The instances in the second category were generated by the dataset "connect-4" taken from the UCI Machine Learning Repository [15]. Connect-4 is a board game, and each row of the dataset corresponds to a minimal winning/losing stage of the first player, and a minimal hitting set of a set of winning stages is a minimal way to disturb wining/losing plays of the first player. We took the first m rows to make problem instances of different sizes.

The third instances are generated from the frequent itemset (pattern) mining problem. These are new instances. An itemset is a hyperedge in our terminology. For a set family $\mathcal{F}$ and a support threshold σ, an

Table 4: Computation time on losing stage in Connect-4

lose	100	200	400	800	1600	3200	6400	12800
BEGK	4.7	51	110	340	-	-	-	-
DL	0.11	6.4	44	210	-	-	-	-
BMR	0.047	2.2	5.1	16	130	-	-	-
MTminer	21	-	-	-	-	-	-	-
KS	0.057	2.6	4.6	20	97	-	-	-
RS	0.009	0.052	0.14	0.41	1.6	15	98	420
MMCS	0.006	0.044	0.09	0.28	0.94	12	40	180
#vertices	80	80	80	80	80	80	80	80
$\|\mathcal{F}\|$	100	200	400	800	1600	3200	6400	12800
$\|F\|^*$	8	8	8	8	8	8	8	8
$\|dual(\mathcal{F})\|$	2341	22760	33087	79632	212761	2396735	4707877	16405082
$\|S\|^*$	11.19	12.43	13.59	14.62	15.73	17.06	17.41	19.09

itemset is called *frequent* if it is included in at least σ hyperedges, and *infrequent* otherwise. A frequent itemset included in no other frequent itemset is called a *maximal frequent itemset*, and an infrequent itemset including no other infrequent itemset is called a *minimal infrequent itemset.* The problem instances are generated by taking the complement of each maximal itemsets of the datasets "BMS-WebView-2" and "accidents", taken from the FIMI repository [7]. The minimal hitting sets correspond to minimal infrequent itemsets.

The fourth instances are used in previous studies [10, 4].

- Matching graph (M(n)) is a graph with n vertices (n is even) and $n/2$ hyperedges forming a perfect matching, that is, hyperedge F_i is $\{2i-1, 2i\}$. This instance has few hyperedges but $2^{n/2}$ minimal hitting sets.
- Dual Matching graph (DM(n)) is dual(M(n)).
- Threshold graph (TH(n)): a hypergraph with n vertices (n is even) and hyperedge set $\{\{i,j\} : 1 \leq i < j \leq n, j \text{ is even}\}$. This instance has few hyperedges $n^2/4$ and few minimal hitting sets $n/2+1$.
- Self-Dual Threshold graph (SDTH(n)): The hyperedge set of SDTH(n) is given as $\{\{n-1,n\}\} \cup \{\{n-1\} \cup E \mid E \in \text{TH}(n-2)\} \cup \{\{n\} \cup E \mid E \in dual(\text{TH}(n-2))\}$. SDTH($n$) has the same number of minimal hitting sets as its hyperedges, $(n-2)^2/4 + n/2 + 1$.
- Self-Dual Fano-Plane graph (SDFP(n)) has the hyperedge set $\{\{x_{n-1}, x_n\}\} \cup \{\{x_{n-1}\} \cup f | f \in H\} \cup \{\{x_n\} \cup f | f \in dual(H)\}\}$, where H is the union of disjoint copies of $\{\{1,2,3\}, \{1,5,6\}, \{1,7,4\}, \{2,4,5\}, \{2,6,7\}, \{3,4,6\}, \{3,5,7\}\}$.

5.2 Results Table 1 - 6 compare the computation times in seconds. The other results are shown in the Appendix. $|\mathcal{F}|$ and $|dual(\mathcal{F})|$ are the numbers of hyperedges and minimal hitting sets, respectively. $|F|^*$ and $|S|^*$ are the average sizes of hyperedges and minimal hitting sets, respectively. "-" means that the computation time was more than 1,000 seconds, and "fail" implies that the computation did not terminate normally because of a shortage of memory or some errors. Our algorithm has a function to input the hyperedges by their complements, for the case that the hyperedges are large on average. The update time of *crit* and *uncov* depends on the size of the complement of $\mathcal{F}(v)$, since $crit(f, S \cup v) = crit(f, S) \setminus \mathcal{F}(v) = crit(f, S) \cap \overline{\mathcal{F}(v)}$. Thus, when the average hyperedge size is close to $|V|$, the computation time is short. The computational time for this is written in the rows with "cRS" and "cMMCS".

We observe the results for random instances, connect-4, accidents, and bms2, that are of large sizes, and have many solutions, i.e., our targets. The results of the other instances are shown in Appendix and the long version[12]. In almost all cases, our algorithms RS and MMCS are the best. Especially, when the problem size is large, the difference of the computation time becomes larger. Among the existing algorithms, MTminer is competitive for accidents, and faster than our implementations for bms2. According to the results, MT-miner seems to be efficient for dense instances and/or instances in that the size of each solution is small. The computation time of DL and BMR depends on $|dual(\mathcal{F})|$ and $|S|^*$, while that of KS, RS and MMCS depends on $|\mathcal{F}|$ and $|F|^*$. In particular, RS and MMCS are quite faster than any other algorithm in almost all instances. The exceptions are matching graphs and dual matching graphs (see Appendix); both are extreme cases of only few small minimal hitting sets that can be easily found, and of few small hyperedges. BEGK is the

Table 5: Computation time on all maximal frequent sets from "accidents"

accidents	200k	150k	130k	110k	90k	70k	50k	30k
BEGK	0.54	3.2	8.7	22	87	430	-	-
DL	0.004	0.042	0.28	0.98	4.8	31	270	-
BMR	0.008	0.041	0.074	0.17	2.3	5.7	21	140
MTminer	0.001	0.007	0.019	0.056	0.28	0.88	4.1	37
KS	fail	fail	fail	fail	fail	fail	fail	fail
RS	0.001	0.011	0.02	0.052	0.26	0.78	3.3	32
MMCS	0.002	0.013	0.034	0.05	0.23	0.76	3.2	28
cRS	0	0.007	0.027	0.05	0.23	1.4	12	230
cMMCS	0.001	0.005	0.019	0.051	0.18	0.95	8.4	170
#vertices	64	64	81	81	336	336	442	442
$\lvert\mathcal{F}\rvert$	81	447	990	2000	4322	10968	32207	135439
$\lvert F\rvert^*$	57.48	56.34	72.85	72.23	326.66	326.08	325.31	430.39
$\lvert dual(\mathcal{F})\rvert$	253	1039	1916	3547	7617	17486	47137	185218
$\lvert S\rvert^*$	2.57	3.77	4.25	4.73	5.09	5.70	6.46	7.32

slowest in most instances; algorithms with smaller complexity are not always faster. KS algorithm embodies an idea to unify the isomorphic vertices into one to reduce the number of iterations, but it seems that this is not so much efficient in our experiments. In our extra experiments, such isomorphic vertices exist in only a few iterations, thus the improvement brought about by unifying them would be limited.

Table 7 lists the average ratios of computation times relative to the case without pruning. The value is the average over all instances in the categories, and smaller values mean more improvement. In some cases the ratio is slightly larger than 1.0, however basically the pruning works well especially for RS. The reason that the pruning is not so efficient for MMCS is that MMCS already has a pruning method, thus the improvement is limited.

We evaluated the total memory usage of each algorithm. Some of them are in Table 8 and 9, and remainings are in Appendix. The memory usage of the existing methods seems to mainly depend on the number of minimal hitting sets, and the input size. On the other hand, the memory usage of our algorithms and KS depends only on the input size. There is a big difference between the breadth-first group and the depth-first group. The difference inside the group is constant, and mainly comes from the memory used by the standard template library.

6 Conclusion

We proposed efficient algorithms for solving the dualization problem. The new depth-first search type algorithms are based on reverse search and branch and bound with a reduced search space. We also proposed an efficient minimality condition check method that exploits a new concept called "critical hyperedges". Computational experiments showed that our algorithms outperform the existing ones in almost all cases, while using less memory even for large-scale problems with up to millions of hyperedges. Sometimes, our algorithms take long time for the minimality check. Shortening this time will be one of the future tasks. More efficient pruning methods are also an interesting topic of future work.

Acknowledgments: Part of this research is supported by the Funding Program for World-Leading Innovative R&D on Science and Technology, Japan. We thank Khaled Elbassioni and Elias C. Stavropoulos, and François Rioult for giving their programs.

References

[1] D. Avis and K. Fukuda, "Reverse Search for Enumeration," *Disc. Appl. Math.,* **65**, 21-46 (1996).

[2] J. Bailey, T. Manoukian and K. Ramamohanarao, "A Fast Algorithm for Computing Hypergraph Transversals and its Application in Mining Emerging Patterns," *ICDM 2003,* 485-488, (2003).

[3] C. Berge, "Hypergraphs," *North-Holland Mathematical Library 45,* (1989).

[4] E. Boros, K. Elbassioni, V. Gurvich, and L. Khachiyan, "An Efficient Implementation of a Quasi-Polynomial Algorithm for Generating Hypergraph Transversals," *ESA 2003*, 556-567, (2003).

[5] G. Dong and J. Li, "Mining Border Descriptions of Emerging Patterns from Dataset Pairs ," *Knowledge and Information Systems,* **8**, 178-202 (2005).

Table 6: Computation time on all maximal frequent sets from "BMS-WebView2"

bms2	800	500	400	200	100	50	30	20	10
BEGK	-	-	-	-	-	-	-	-	-
DL	0.87	3.9	5.4	94	-	-	-	-	-
BMR	4.7	18	20	110	380	1000	-	-	-
MTminer	0.027	0.072	0.12	0.38	1.2	3.3	8.2	15	43
KS	fail	fail	fail	fail	fail	fail	fail	fail	fail
RS	0.039	0.12	0.15	0.87	9.2	71	340	800	-
MMCS	0.048	0.089	0.15	1.1	13	92	400	950	-
cRS	0.004	0.009	0.015	0.056	0.25	1	4	10	47
cMMCS	0.003	0.007	0.012	0.053	0.25	1.1	4.4	12	62
#vertices	3400	3400	3400	3400	3400	3400	3400	3400	3400
$\|\mathcal{F}\|$	62	152	237	823	2591	6946	17315	30405	74262
$\|F\|^*$	3338.68	3261.89	3338.18	3337.39	3336.36	3335.91	3335.23	3334.97	3334.19
$\|dual(\mathcal{F})\|$	4616	16991	15993	89448	438867	1289303	2297560	3064937	4582209
$\|S\|^*$	1.29	1.88	1.82	1.99	2.01	2.02	2.04	2.07	2.15

Table 7: Reduction ratio of computation time by pruning method

name	prob	win	lose	accidents	bms2	matching	dual-matching	TH	SDTH	SDFP
RS (all)	0.33	0.37	0.44	0.34	0.56	0.73	0.16	1.00	0.18	0.30
MMCS (all)	0.94	0.98	1.09	0.73	1.01	1.08	1.03	0.86	1.03	0.46
RS (large)	0.29	0.19	0.19	0.17	0.33	0.96	0.11	0.77	0.11	0.15
MMCS (large)	0.93	0.96	0.95	0.44	1.00	1.01	1.00	0.83	1.13	0.68

[6] M. L. Fredman and L. Khachiyan, "On The Complexity of Dualization of Monotone Disjunctive Normal Forms," *Journal on Algorithms,* **21**, 618-628 (1996).

[7] Frequent Itemset Mining Dataset Repository, http://fimi.cs.helsinki.fi/data/

[8] M. Hagen, "Algorithmic and Computational Complexity Issues of MONET," *Ph.D. Thesis, Friedrich-Schiller-Universitat Jena,* (2008).

[9] C. Hébert, A. Bretto and B. Crémilleux, "A Data Mining Formalization to Improve Hypergraph Minimal Transversal Computation", *Fundamental Informaticae* **80**, 415-433 (2007).

[10] D. J. Kavvadias and E. C. Stavropoulos, "An efficient algorithm for the transversal hypergraph generation," *Journal of Graph Algorithms and Applications,* **9**, 239-264 (2005).

[11] L. Khachiyan, E. Boros, K. Elbassioni and V. Gurvich, "An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals and its application in joint generation", *Discrete Applied Mathematics* **154**, 2350-2372 (2006).

[12] K. Murakami and T. Uno, "Efficient Algorithms for Dualizing Large-Scale Hypergraphs", arXiv:1102.3813 (2011).

[13] K. Satoh and T. Uno, "Enumerating Maximal Frequent Sets Using Irredundant Dualization", *LNAI* **2843**, 256-268 (2003).

[14] E. Tomita, A. Tanaka and H. Takahashi, "The Worst-case Time Complexity for Generating all Maximal Cliques and Computational Experiments", *Theoretical Computer Science* **363**, 28-42 (2006).

[15] UCI machine learning repository, http://archive.ics.uci.edu/ml/

[16] T. Uno and K. Satoh, "Detailed Description of an Algorithm for Enumeration of Maximal Frequent Sets with Irredundant Dualization", *FIMI 2003, CEUR Workshop Proceedings 90*, (2003).

Table 8: Memory usage for accidents (MB)

accidents	130k	110k	90k	70k	50k	30k
BEGK	250	250	250	260	-	-
DL	46	49	160	300	1200	-
BMR	28	35	170	320	1200	4800
MTminer	27	27	30	43	110	770
KS	fail	fail	fail	fail	fail	fail
RS	13	13	24	41	98	490
MMCS	13	13	24	41	98	490
cRS	12	12	13	13	16	30
cMMCS	12	12	13	13	16	30

Table 9: Memory usage for Connect-4(lose) (MB)

lose	100	200	400	800	1600	3200	6400	12800
BEGK	87	88	88	89	-	-	-	-
DL	44	49	54	66	-	-	-	-
BMR	20	33	52	96	96	-	-	-
MTminer	690	-	-	-	-	-	-	-
KS	2	2.2	2.2	2.4	2.9	-	-	-
RS	12	12	12	12	12	12	72	72
MMCS	12	12	12	12	12	12	77	77

Appendix

LEMMA 6.1. *For vertex subset S, $v \in S$ and $v' \notin S$, $crit(v, S \cup v') = crit(v, S) \setminus \mathcal{F}(v')$. Particularly, $crit(v, S \cup v') \subseteq crit(v, S)$ holds.*

Proof. For any $F \in crit(v, S)$, $(S \cup v') \cap F = \{v\}$ holds if F is not in $\mathcal{F}(v')$, and thus it is included in $crit(v, S \cup v') \setminus \mathcal{F}(v)$. Conversely, $(S \cup v') \cap F = \{v\}$ holds for any $F \in crit(v, S \cup v')$. This means that $S \cap F = \{v\}$, and $F \in crit(v, S)$. ∎

LEMMA 6.2. *For any vertex subset S and $v' \notin S$, $crit(v', S \cup v') = uncov(S) \cap \mathcal{F}(v')$.*

Proof. Since any hyperedge not in $uncov(S)$ has a non-empty intersection with S, F can never be a critical hyperedge for v'. Any critical hyperedge for v' includes v' thus, we can see that $crit(v', S) \subseteq uncov(S) \cap \mathcal{F}(v')$. Conversely, for any hyperedge F included in $uncov(S) \cap \mathcal{F}(v')$, $F \cap (S \cup v') = \{v'\}$, thereby $uncov(S) \cap \mathcal{F}(v') \subseteq crit(v', S)$. Hence, the lemma holds. ∎

The next two lemmas follow directly from the above.

LEMMA 6.3. *[9, 13] If a vertex subset S satisfies the minimality condition, any of its subsets also satisfy the minimality condition, i.e., the minimality condition satisfies the monotone property.*

LEMMA 6.4. *[9, 13] If a vertex subset S does not satisfy the minimality condition, S is not included in any minimal hitting set. In particular, any minimal hitting set S is maximal in the set system composed of vertex subsets satisfying the minimality condition, but not vice varsa.*

LEMMA 6.5. *For any vertex subset S, $\sum_{v \in S} |crit(v, S)| \leq |\mathcal{F}|$.*

Proof. From the definition of the critical hyperedge, any hyperedge $F \in \mathcal{F}$ can be critical for at most one vertex. Thus, the lemma holds. ∎

LEMMA 6.6. *$S \neq \emptyset$ belongs to $\mathcal{S}$ iff $min\_crit(v, S) < min\_uncov(S)$ holds for any $v \in S$.*

Proof. Suppose that $S \in \mathcal{S}$, thus $S \in dual(\mathcal{F}_i)$ for some i. We can see that $min\_uncov(S) > i$, $crit(v, S)$ includes a hyperedge $F_j \in \mathcal{F}$ with $j < i$, and thus $min\_crit(v, S) < i$ for any v. Thus, $min\_crit(v, S) < min\_uncov(S)$ holds for any $v \in S$.

Conversely, suppose that $min\_crit(v, S) < min\_uncov(S)$ holds for any $v \in S$. Then, we can see that $crit(v, S) \neq \emptyset$ for any $v \in S$ because $min\_crit(v, S) < m + 1$. Let $i = min\_uncov(S) - 1$. Note that $i \leq m$. We can then see that S is a hitting set of $\mathcal{F}_i$ and $min\_crit(v, S) \leq i$. This in turn implies that S is a minimal hitting set in $\mathcal{F}_i$, and thus, it belongs to $\mathcal{S}$. ∎

LEMMA 6.7. *Let $S \in \mathcal{S}$ and $i = min\_uncov(S)$. A vertex subset S' is a child of S iff (1) $i < m+1$, (2) $S' = S \cup v$ for some $v \in F_i$, and (3) $max\_min\_crit(v', S') < i$ holds for any $v' \in S$.*

Proof. Suppose that S' is a child of S. We can see that $uncov(S)$ is not empty, and thus (1) holds. From the definition of the parent, S is obtained from S' by removing a vertex v from S'. From $max\_min\_crit(S') < min\_uncov(S')$ and $uncov(S) = uncov(S') \cup crit(v, S')$, we obtain $min\_crit(v, S') = max\_min\_crit(S') = min\_uncov(S)$. This means that $F_i \in crit(v, S')$, and thus (2) holds. This equation also implies that (3) holds.

Suppose that S' is a vertex subset satisfying (1), (2) and (3). From (2), we see that $min\_crit(v, S') = i$. Since $uncov(S') > i$, this together with (3) implies that S' satisfies the conditions in Lemma 4.1 and thereby is included in $\mathcal{S}$. $min\_crit(v, S') = i$ and (3) leads to $max\_min\_crit(S') = i$ and $P(S') = S' \setminus v = S$. Note that condition (1) guarantees the existence of F_i given condition (2), thus it is implicitly used in the proof. ∎

Table 10: Computation time on winning stage in Connect-4

win	100	200	400	800	1600	3200	6400	12800
BEGK	1.2	5.2	46	55	430	-	-	-
DL	0.005	0.061	1.6	6.2	180	-	-	-
BMR	0.006	0.044	0.52	0.67	17	710	-	-
MTminer	7.0	390	-	-	-	-	-	-
KS	0.021	0.14	1.1	3.2	73	860	-	-
RS	0.001	0.005	0.032	0.078	0.41	4.7	20	83
MMCS	0.001	0.006	0.021	0.056	0.27	2.6	11	48
#vertices	80	80	80	80	80	80	80	80
$\lvert\mathcal{F}\rvert$	100	200	400	800	1600	3200	6400	12800
$\lvert F\rvert^*$	8	8	8	8	8	8	8	8
$\lvert dual(\mathcal{F})\rvert$	287	1145	6069	11675	71840	459502	1277933	11614885
$\lvert S\rvert^*$	10.70	11.95	14.15	14.84	16.46	17.69	18.67	20.54

Table 11: Computation time on matching graphs

name matching	20	24	28	32	36	40
BEGK	0.045	0.72	1.1	4.4	36	fail
DL	0.003	0.012	0.04	0.21	0.89	3.9
BMR	0.003	0.016	0.045	0.19	1.2	5.3
MTminer	0.033	0.39	4.9	59	670	-
KS	0	0.003	0.01	0.044	0.2	0.87
RS	0	0.004	0.013	0.059	0.25	1.1
MMCS	0.002	0.006	0.023	0.06	0.26	1.1
#vertices	20	24	28	32	36	40
$\lvert\mathcal{F}\rvert$	10	12	14	16	18	20
$\lvert F\rvert^*$	2	2	2	2	2	2
$\lvert dual(\mathcal{F})\rvert$	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
$\lvert S\rvert^*$	10	12	14	16	18	20

Table 12: Computation time on dual matching graphs

name dual-matching	20	24	28	32	36	40
BEGK	1.4	3.1	8.9	67	fail	fail
DL	0.01	0.054	0.25	1.2	7.1	70
BMR	0.038	0.4	4.2	49	540	-
MTminer	0.044	0.67	9.0	120	-	-
KS	0.012	0.071	0.56	5.6	60	780
RS	0.007	0.054	0.5	4.8	50	-
MMCS	0.014	0.075	0.64	6.8	73	-
#vertices	20	24	28	32	36	40
$\lvert\mathcal{F}\rvert$	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
$\lvert F\rvert^*$	10	12	14	16	18	20
$\lvert dual(\mathcal{F})\rvert$	10	12	14	16	18	20
$\lvert S\rvert^*$	2	2	2	2	2	2

Table 13: Computation time on threshold graphs

name TH	40	80	120	160	200
BEGK	0.28	0.84	2.7	7.5	19
DL	0.004	0.027	0.091	0.24	0.52
BMR	0.009	0.15	0.6	2.6	6.6
MTminer	-	-	-	-	-
KS	0.021	0.34	2.5	11	35
RS	0.001	0.003	0.016	0.019	0.048
MMCS	0	0.003	0.01	0.026	0.037
#vertices	40	80	120	160	200
$\lvert\mathcal{F}\rvert$	400	1600	3600	6400	10000
$\lvert F\rvert^*$	2	2	2	2	2
$\lvert dual(\mathcal{F})\rvert$	21	41	61	81	101
$\lvert S\rvert^*$	29.05	59.02	89.02	119.01	149.01

Table 14: Total memory requirement for randomly generated instances (megabytes)

probability prob	0.9	0.8	0.7	0.6	0.5
BEGK	51	130	-	-	-
DL	49	120	-	-	-
BMR	30	100	660	-	-
MTminer	27	48	340	5400	-
KS	2.3	2.3	2.3	-	-
RS	12	12	12	12	12
MMCS	12	12	12	12	12
cRS	12	12	12	12	-
cMMCS	12	12	12	12	-

Table 15: Total memory requirement for on self-dual threshold graphs (megabytes)

name SDTH	122	162	202	242	282
BEGK	110	140	170	200	230
DL	44	45	47	47	50
BMR	34	56	89	130	190
MTminer	-	-	-	-	-
KS	10	20	37	61	95
RS	12	13	13	14	15
MMCS	12	13	13	14	15

Table 16: Total memory requirement for Self-dual Fano-plane graphs (megabytes)

name SDFP	16	23	30	37	44
BEGK	46	51	56	-	-
DL	43	43	45	-	-
BMR	20	23	63	72	-
MTminer	27	57	1800	-	-
KS	2	2.1	2.9	8.4	-
RS	12	12	12	15	37
MMCS	12	12	12	15	-

Table 17: Total memory requirement for the dataset of winning stage in Connect-4 (megabytes)

name win	100	200	400	800	1600	3200	6400	12800
BEGK	82	87	87	87	88	-	-	-
DL	43	44	45	46	67	-	-	-
BMR	20	20	22	24	55	200	-	-
MTminer	260	10000	-	-	-	-	-	-
KS	2	2	2.1	2.4	2.9	4	-	-
RS	12	12	12	12	12	12	13	20
MMCS	12	12	12	12	12	12	13	77

Table 18: Total memory requirement for all maximal frequent set from "BMS-WebView2" (megabytes)

name bms2	800	500	400	200	100	50	30	20	10
BEGK	-	-	-	-	-	-	-	-	-
DL	52	65	74	300	1100	-	-	-	-
BMR	38	57	84	320	1200	2400	-	-	-
MTminer	27	27	27	29	35	50	84	130	270
KS	fail	fail	fail	fail	fail	fail	fail	fail	fail
RS	13	16	19	34	82	200	480	830	-
MMCS	14	16	19	34	82	200	480	830	-
cRS	12	12	12	12	12	13	14	15	18
cMMCS	12	12	12	12	12	13	14	15	18

Table 19: Total memory requirement for matching graphs (megabytes)

name matching	20	24	28	32	36	40
BEGK	48	51	57	59	120	fail
DL	20	43	45	53	86	240
BMR	20	20	24	40	110	430
MTminer	28	45	190	1400	12000	-
KS	2	2	2	2	2	2
RS	12	12	12	12	12	12
MMCS	12	12	12	12	12	12

Table 20: Total memory requirement for dual matching graphs (megabytes)

name dual-matching	20	24	28	32	36	40
BEGK	51	51	58	65	fail	fail
DL	43	45	51	160	580	2300
BMR	21	24	41	110	610	-
MTminer	28	53	340	4000	-	-
KS	1.9	3	8	25	94	300
RS	13	13	15	24	66	-
MMCS	13	13	15	24	66	-

Table 21: Total memory requirement for threshold graphs (megabytes)

name TH	40	80	120	160	200
BEGK	61	87	110	140	170
DL	43	43	72	72	44
BMR	20	23	34	56	87
MTminer	-	-	-	-	-
KS	1.9	4.4	9.9	20	36
RS	12	12	12	13	13
MMCS	12	12	12	13	13