アルゴリズム 75－5
（2000．11．10）

極小横断列挙のメモリ効率の良いアルゴリズム

玉木 久夫
明治大学理工学部
tamaki@cs.meiji.ac.jp

概要

ハイパーグラフの極小横断を列挙する、メモリ効率の良いアルゴリズムを与える。与えられたハイパーグラフ $\mathcal{H}$ のサイズが $n = \sum_{X\in\mathcal{H}} |X|$ であり、m 個の極小横断を持つとき、このアルゴリズムは $\mathcal{H}$ のすべての極小横断を $(n+m)^{O(\log n)}$ 時間で、$O(n \log n)$ ワードのメモリを用いて列挙する。

Space-efficient enumeration of minimal transversals of a hypergraph

Hisao Tamaki
School of Sience and Technology, Meiji University

Abstract

We give a space-efficient algorithm for enumerating the minimal transversals of a hypergraph. Given a hypergraph $\mathcal{H}$ of size $n = \sum_{X\in\mathcal{H}} |X|$ which has m minimal transversals, our algorithm enumerates all the minimal transversals of $\mathcal{H}$ in $(n+m)^{O(\log n)}$ total time using $O(n \log n)$ words of storage.

1 Introduction

Given a finite set V called the set of *vertices*, a *hypergraph* $\mathcal{H}$ on V is a set of subsets of V. Each element of $\mathcal{H}$ is called a *hyperedge* or simply an *edge* of $\mathcal{H}$. For the purpose of this paper, the size $\mathsf{size}(\mathcal{H})$ of a hypergraph $\mathcal{H}$ is defined by $\mathsf{size}(\mathcal{H}) = \sum_{X\in\mathcal{H}} |X|$. An edge E of a hypergraph $\mathcal{H}$ is said to be *minimal* in $\mathcal{H}$, if no proper subset of E is an edge of $\mathcal{H}$. We denote by $\mathsf{MIN}\mathcal{H}$ the hypergraph consisting of all the minimal edges of hypergraph $\mathcal{H}$. A hypergraph is called *simple* if all of its edges are minimal. A *transversal* of a hypergraph $\mathcal{H}$ on V is a subset X of V that intersects every edge of $\mathcal{H}$: $\forall Y \in \mathcal{H} : X \cap Y \neq \emptyset$. A transversal is also often called a *vertex cover*, and sometimes a *hitting set* in the literature. A transversal X of $\mathcal{H}$ is said to be *minimal* if no proper subset of X is a transversal of $\mathcal{H}$. We denote by $\mathsf{TR}\mathcal{H}$ the set of all the minimal transversals of $\mathcal{H}$. Clearly, $\mathsf{TR}\mathcal{H}$ is a simple hypergraph.

The problem of enumerating all the minimal transversals of a given hypergraph is known as *monotone dualization* or simply *dualization*, since it is equivalent to computing a dual of a monotone boolean function with both input and output represented in DNF (disjunctive normal form) [1, 6, 7]. The dualization problem arises in diverse areas of computer science (see [1, 6, 7, 3] for references) and its complexity has been studied by several researchers[1, 2, 6, 7, 8, 10]. Since the number of minimal transversals can be exponential in the size of the given hypergraph, the time complexity is usually measured in terms of the total size N of the input and the output. In this measure, the best known algorithm for dualization is by Fredman and Khachiyan [7] that runs in $N^{O(\log N/\log\log N)}$ time. It is open whether this problem can be solved in polynomial time. This open problem is of particular interest since there are many problems that are known to be polynomially equivalent to the dualization problem[1, 6, 3]. Several special classes of hypergraphs are known for which dualization can be done in polynomial time. These classes include graphs[14, 9], hypergraphs with their edge size bounded by a constant[6, 2], and hypergraphs corresponding to regular Boolean functions[5, 10, 11].

A problem closely related to dualization is that of *duality testing*: given two hypergraphs $\mathcal{H}$ and $\mathcal{G}$, decide whether $\mathcal{G} = \mathsf{TR}\mathcal{H}$. It is straightforward to verify that this condition is equivalent to the symmetric condition $\mathcal{H} = \mathsf{TR}\mathcal{G}$, when both $\mathcal{H}$ and $\mathcal{G}$ are simple. The above cited algorithm of Fredman and Khachiyan is in fact formulated as an algorithm for duality testing. This algorithm certifies its negative answer (that $\mathcal{G} \neq \mathsf{TR}\mathcal{H}$) by pointing out an edge of $\mathcal{G}$ that is not a minimal transversal of $\mathcal{H}$ or by generating a minimal

transversal of $\mathcal{H}$ not present in $\mathcal{G}$. Dualization is achieved through repeated applications of this testing procedure: start with an empty set as a candidate for $\mathrm{TR}\mathcal{H}$, apply the duality testing procedure to $\mathcal{H}$ and the candidate hypergraph; if the test succeeds then stop, otherwise augment the candidate hypergraph by the missing minimal transversal reported by the testing procedure and repeat. More generally, it is known that dualization can be done through a polynomial number of calls to a duality oracle, even if the oracle does not certify its negative answer[1].

Dealing with duality testing rather than dualization has a certain advantage because of the symmetric roles of the two input hypergraphs. Indeed, Fredman and Khachiyan [7] exploit this symmetry in the design of their algorithms. As far as the polynomiality question is concerned, two problems are equivalent owing to the reduction mentioned above. However, the reduction requires that all the generated minimal transversals be stored in memory and be used as input to the duality testing procedure at the next step. In some applications, the number of minimal transversals is so large that this storage requirement is prohibitive. For example, an experiment in logic minimization research reported in [4] (see also [12]) involves the exhaustive enumeration of more than 10^8 minimal transversals of a hypergraph with around 100 edges and 50 vertices; experiments of larger scale are not conducted due to the limitation on the main storage[13].

In such applications, it is desirable to have an enumeration procedure whose storage requirement does not depend on the number of transversals to be enumerated. In this paper, we achieve this goal without seriously sacrificing the time complexity: we give an algorithm that runs in $(n+m)^{O(\log n)}$ time and uses $O(n \log n)$ space, where $n = \mathrm{size}(\mathcal{H})$ is the size of the input hypergraph $\mathcal{H}$ and m is the number of minimal transversals of $\mathcal{H}$. Here, we adopt the word model for space complexity: we count the number of memory words used, each of which may contain an integer of $O(\log n)$ bits.

In the next section, we observe that the recursive approach of Fredman and Khachiyan to duality testing can be translated into a direct algorithm for dualization that does not resort to the above reduction. This algorithm runs in $(n+m)^{O(\log n)}$ time but requires space that depends on m. In section 3 we show that this algorithm can be reformulated so that the space requirement is $O(n \log n)$.

2 Recursive problem decomposition

In the following description of our algorithms, we assume that the vertices of the hypergraphs we deal with are integers and are therefore totally ordered.

We call a hypergraph *trivial* if no vertex appear in its edges, i.e., if it is either empty or contains the empty set as the only edge. For each hypergraph $\mathcal{H}$ and a vertex v that does not appear in $\mathcal{H}$, we denote by $v+\mathcal{H}$ the hypergraph $\{\{v\} \cup X \mid X \in \mathcal{H}\}$, obtained by adding v to every edge of $\mathcal{H}$. For each nontrivial hypergraph $\mathcal{H}$, let $v_{\mathcal{H}}$ denote the smallest vertex that appears in $\mathcal{H}$. Then, $\mathcal{H}$ is uniquely decomposed as

$$\mathcal{H} = (v_{\mathcal{H}} + \mathcal{H}^1) \cup \mathcal{H}^0$$

where $\mathcal{H}^1 = \{X \setminus \{v_{\mathcal{H}}\} \mid v_{\mathcal{H}} \in X \in \mathcal{H}\}$ and $\mathcal{H}^0 = \{X \in \mathcal{H} \mid v_{\mathcal{H}} \notin X\}$. We define $\mathcal{H}^* = \mathcal{H}^1 \cup \mathcal{H}^0$.

This decomposition of the input hypergraph leads to decompositions of the dualization problem into subproblems. The following two lemmas reformulate, in our dualization setting, the problem decomposition rules of Fredman and Khachiyan [7] for duality testing.

Lemma 2.1 *Let $\mathcal{H}$ be a nontrivial hypergraph. Then,*

$$\mathrm{TR}\mathcal{H} = \mathrm{TR}\mathcal{H}^* \ \cup \ (v_{\mathcal{H}} + (\mathrm{TR}H^0 \setminus \mathrm{TR}H^*))$$

Proof: Omitted. ∎

Let $\mathcal{H}$ be a nontrivial hypergraph. For each $Y \in \mathrm{TR}\mathcal{H}^0$, define $\mathcal{H}|_Y$ by $\mathcal{H}|_Y = \{X \in \mathcal{H}^1 \mid X \cap Y = \emptyset\}$.

Lemma 2.2 *Let $\mathcal{H}$ be a nontrivial hypergraph. Then,*

$$\mathrm{TR}\mathcal{H} = \mathrm{MIN}\{X \cup Y \mid Y \in \mathrm{TR}\mathcal{H}^0, X \in \mathrm{TR}(v_{\mathcal{H}} + \mathcal{H}|_Y)\} \tag{1}$$

Proof: Omitted. ∎

The above lemmas provide two ways of decomposing a given problem instance into subproblems. The decomposition based on Lemma 2.1 has an advantage when $\mathcal{H}^0$ is small. On the other hand, the

decomposition based on Lemma 2.2 is advantageous when $\mathcal{H}^1$ is small. Thus, we may expect to get some efficiency by choosing between the two decomposition rules based on the size of $\mathcal{H}^1$ relative to the size of $\mathcal{H}$. This is in essence the strategy taken by Fredman and Khachiyan in their duality testing algorithm (the second algorithm in their paper [7]). Employing carefully designed criteria for selection that exploit the symmetry between the roles of two input hypergraphs, they achieve a time upper bound of $N^{O(\log N/\log\log N)}$. In our case, where the symmetry is unavailable, we employ a simpler criterion and settle with a time bound of $(n+m)^{(O(\log n)}$, where n is the size of the input and m is the number of minimal transversals to be enumerated[1].

Algorithm A
Input: Hypergraph $\mathcal{H}$
Output: $\mathrm{TR}\mathcal{H}$

Case 0: If $\mathcal{H}$ is trivial then return the trivial answer $\mathrm{TR}\mathcal{H}$. Otherwise:

Case 1: If $\mathsf{size}(\mathcal{H}^1) \geq \mathsf{size}(\mathcal{H})/2$ then use Lemma 2.1 to reduce the problem to two subproblems:
1. Compute recursively $\mathrm{TR}\mathcal{H}^0$ and $\mathrm{TR}\mathcal{H}^*$.
2. Return $\mathrm{TR}\mathcal{H}^* \cup (v_{\mathcal{H}} + (\mathrm{TR}\mathcal{H}_1 \setminus \mathrm{TR}\mathcal{H}^*))$.

Case 2: If $\mathsf{size}(\mathcal{H}^1) < \mathsf{size}(\mathcal{H})/2$ then use Lemma 2.2 to reduce the problem to subproblems:
1. Compute recursively $\mathrm{TR}\mathcal{H}^0$.
2. For each $Y \in \mathrm{TR}\mathcal{H}^0$, let $\mathcal{H}|_Y = \{X \in \mathcal{H}^1 \mid X \cap Y = \emptyset\}$ and compute recursively $\mathrm{TR}\mathcal{H}|_Y$.
3. Let $\mathcal{G} = \{X \cup Y \mid Y \in \mathrm{TR}\mathcal{H}^0, X \in \{\{v\}\} \cup \mathrm{TR}H|_Y\}$ and return $\mathrm{MIN}\mathcal{G}$. (Here we use the fact that $\mathrm{TR}(v_{\mathcal{H}} + \mathcal{H}|_Y) = \{\{v\}\} \cup \mathrm{TR}\mathcal{H}|_Y$.)

Before analyzing the running time of the above algorithm, we note the following easy facts.

Proposition 2.3 *Let $\mathcal{H}$ be hypergraph and let $m = |\mathrm{TR}\mathcal{H}|$. Then, $|\mathrm{TR}\mathcal{H}^0| \leq m$, $|\mathrm{TR}\mathcal{H}^*| \leq m$, and moreover $|\mathrm{TR}\mathcal{H}|_Y| \leq m$ for every $Y \in \mathrm{TR}\mathcal{H}^0$.*

Theorem 2.4 *Let $\mathcal{H}$ be an arbitrary hypergraph with $n = \mathsf{size}(\mathcal{H})$ and $m = |\mathrm{TR}\mathcal{H}|$. Then, the running time of the above algorithm is $(n+m)^{O(\log n)}$.*

Proof: Omitted. ■

The space requirement of this algorithm clearly depends on m, since the result of each recursive call must be stored and the size of such an intermediate result may be almost as large as the the output size of the original problem. In the next section we show that this dependency can be removed without sacrificing the asymptotic time bound.

3 Process formulation: an $O(n \log n)$ space algorithm

In this section, we reformulate the recursive algorithm of the previous section as that of recursive *processes*. Each process spawns subprocesses to solve its subproblems. Each process reports transversals one by one, rather than returning the set of transversals as a whole.

In Algorithm A, several set operations, such as union and difference, are used which operate on the results of recursive calls. If we implement these operations naively in our process version, then each process needs to accumulate the transversals reported by its subprocesses, defeating our goal of saving space. Fortunately, as we show below, some set operations used in the algorithm can be easily removed, and others can be replaced by certain tests, which operate on an individual transversal and decide locally if the transversal must be included in the answer.

We first note that it is easy to test if a given set of vertices is a transversal (or minimal transversal) of a given hypergraph.

Proposition 3.1 *Given a hypergraph $\mathcal{H}$ and a set of vertices X, we can test if X is a transversal (minimal transversal, resp.) of $\mathcal{H}$ in $O(|X|\mathsf{size}(\mathcal{H}))$ ($O(|X|^2\mathsf{size}(\mathcal{H}))$, resp.) time.*

[1] It may appear fair to say that these two bounds are incomparable, since the exponent of our bound involves the input size only. It is straightforward, however, to modify their algorithm to achieve our bound.

Although there are many ways to improve these naive bounds depending on the representations of $\mathcal{H}$ and X, the above bounds are sufficient for our present purposes.

We now describe how the set operations in Algorithm A are handled in our process version of the algorithm. The union operation in Case 1 of Algorithm A is the most trivial to handle:

$$\mathsf{TR}\mathcal{H}^* \cup (v_{\mathcal{H}} + (\mathsf{TR}\mathcal{H}_1 \setminus \mathsf{TR}\mathcal{H}^*)).$$

Since the two sets to be unioned are disjoint, we simply report all transversals in $\mathsf{TR}\mathcal{H}^0$ and then, after that, report all transversals in $v + (\mathsf{TR}\mathcal{H}^0 \setminus \mathsf{TR}\mathcal{H}^*)$. To handle the set difference operation in the latter task, we rewrite $\mathsf{TR}\mathcal{H}^0 \setminus \mathsf{TR}\mathcal{H}^* = \{X \in \mathsf{TR}\mathcal{H}^0 \mid X \text{ is not a transversal of } \mathcal{H}^*\}$. Thus, we may handle each transversal in $\mathsf{TR}\mathcal{H}^0$ separately: if a transversal passes the test that it is not a transversal of $\mathcal{H}^*$ then we report the transversal augmenting it by the vertex $v_{\mathcal{H}}$.

Case 2 of the algorithm requires slightly more effort to handle, where we compute:

$$\mathsf{MIN}\{X \cup Y \mid Y \in \mathsf{TR}\mathcal{H}^0, X \in \{\{v_{\mathcal{H}}\}\} \cup \mathsf{TR}\mathcal{H}|_Y\}$$

In general, the MIN operator requires the reference to the entire hypergraph it operates on. Fortunately in this case, the MIN operator is selecting those transversals of $\mathcal{H}$ that are minimal. Thus, it can be replaced by a local test that works individually on a transversal Z and checks if it is a minimal transversal of $\mathcal{H}$.

Finally, we need to handle the possibility that different pairs of X and Y, $Y \in \mathsf{TR}\mathcal{H}^0, X \in \mathsf{TR}\mathcal{H}|_Y$, may yield the same minimal transversal $X \cup Y$ of $\mathcal{H}$. If a process reports the same transversal more than once, then the previous estimate on the number of recursive processes no longer holds. Therefore, we must make sure each minimal transversal is reported only once. We need some definitions.

For a set of vertices X, define $\min X$ to be the smallest vertex in X. We order sets of vertices in the lexicographic order: $X < Y$ if and only if

(1) $X = \emptyset$ and $Y \neq \emptyset$, or

(2) $X \neq \emptyset$, $Y \neq \emptyset$, and $\min X < \min Y$, or

(3) $X \neq \emptyset$, $Y \neq \emptyset$, $\min X = \min Y$, and $X \setminus \{\min X\} < Y \setminus \{\min Y\}$.

Based on this ordering, we define a function $f_{\mathcal{H}} : \mathsf{TR}\mathcal{H} \to \mathsf{TR}\mathcal{H}^0$ as follows. For each $Z \in \mathsf{TR}\mathcal{H}$ and each subset $\mathcal{G}$ of $\mathcal{H}$, let $\mathsf{TR}_Z\mathcal{G}$ denote the set of minimal transversals of $\mathcal{G}$ that are subsets of Z: $\mathsf{TR}_Z\mathcal{G} = \{X \in \mathsf{TR}\mathcal{G} \mid X \subseteq Z\}$. Clearly, $\mathsf{TR}_Z\mathcal{G}$ is nonempty for every $Z \in \mathsf{TR}\mathcal{H}$ and $\mathcal{G} \subseteq \mathcal{H}$. Then, for each $Z \in \mathsf{TR}\mathcal{H}$, we define $f_{\mathcal{H}}(Z)$ to be the lexicographically last element of $\mathsf{TR}_Z\mathcal{H}^0$, i.e., $X \in \mathsf{TR}_Z\mathcal{H}^0$ such that $X' < X$ for every $X' \in \mathsf{TR}_Z\mathcal{H}^0$ distinct from X.

Proposition 3.2 *For each $Z \in \mathsf{TR}\mathcal{H}$ such that $v_{\mathcal{H}} \notin Z$, $Z \setminus f_{\mathcal{H}}(Z) \in \mathsf{TR}\mathcal{H}|_{f_{\mathcal{H}}(Z)}$.*

Proof: Omitted. ∎

Using function $f_{\mathcal{H}}$, we extend Lemma 2.2 to give a disjoint decomposition of $\mathsf{TR}\mathcal{H}$.

Lemma 3.3 *Let $\mathcal{H}$ be a hypergraph and let $\mathcal{G} = v_{\mathcal{H}} + (\mathsf{TR}\mathcal{H}^0 \setminus \mathsf{TR}\mathcal{H})$ and $\mathcal{G}_Y = \{X \cup Y \mid X \in \mathsf{TR}\mathcal{H}|_Y, (X \cup Y) \in \mathsf{TR}\mathcal{H}, f_{\mathcal{H}}(X \cup Y) = Y\}$ for each $Y \in \mathsf{TR}\mathcal{H}^0$. Then, each minimal transversal of $\mathcal{H}$ belongs to exactly one of the hypergraphs G, $G_{Y_1}, \ldots G_{Y_k}$, where $\mathcal{H}^0 = \{Y_1, \ldots, Y_k\}$.*

Proof: Omitted. ∎

Using this decomposition in Case 2 of Algorithm A and handling set operations in a manner described above, we get our second algorithm. Before describing the algorithm, we show that function $f_{\mathcal{H}}$ can be efficiently evaluated.

Proposition 3.4 *Given $Z \in \mathsf{TR}\mathcal{H}$, $f_{\mathcal{H}}(Z)$ can be computed in $O(|Z|^2\mathsf{size}(\mathcal{H}))$ time.*

Proof: Omited. ∎

Algorithm B:
Input: hypergraph $\mathcal{H}$
Output: minimal transversals of $\mathcal{H}$, printed one by one
Main steps:
(1) Initiate a process enumerate($\mathcal{H}$).

(2) Wait for a response from this process: if a transversal is reported then print it and wait further; if termination is reported then stop.

Process enumerate($\mathcal{H}$)
Decide which of the following three cases applies and execute the associated steps. Upon termination, it is reported to the initiator of this process.

Case 0: $\mathcal{H}$ is trivial:
(0.1) if $\mathcal{H} = \emptyset$ then report the empty transversal and then report termination immediately; otherwise, i.e., if $\mathcal{H} = \{\emptyset\}$, terminate immediately.

Case 1: $\mathcal{H}$ is nontrivial and size($\mathcal{H}^1$) $\geq$ size($\mathcal{H}$)/2:
(1.1) Initiate a subprocess enumerate($\mathcal{H}^0$).
(1.2) Wait for a response from this subprocess: if termination is reported then proceed to the next step; if a transversal X is reported then report $\{v_{\mathcal{H}}\} \cup X$ unless X is a transversal of $\mathcal{H}^*$, and then repeat this step.
(1.3) Initiate a subprocess enumerate($\mathcal{H}^*$).
(1.4) Wait for a response from this subprocess: if termination is reported then terminate; if a transversal X is reported then report X and repeat this step.

Case 2 $\mathcal{H}$ is nontrivial and size($\mathcal{H}^1$) $<$ size($\mathcal{H}$)/2:
(2.1) Initiate a subprocess enumerate($\mathcal{H}^0$).
(2.2) Wait for a response from this subprocess: if termination is reported then terminate; if a transversal Y is reported then do the following.

(2.2.1) If $\mathcal{H}|_Y = \emptyset$ report Y and repeat step (2.2); otherwise, report $\{v_{\mathcal{H}}\} \cup Y$ and do the following:

(2.2.1.1) Initiate a subprocess enumerate($\mathcal{H}|_Y$).
(2.2.1.2) Wait for a response from this subprocess: if termination is reported then repeat step (2.2); if a transversal X is reported then let $Z = X \cup Y$, report Z if Z is a minimal transversal of $\mathcal{H}$ and $Y = f_{\mathcal{H}}(Z)$, and then repeat this step.

It is clear that the number of processes generated by the above algorithm is equal to the number of recursive calls in Algorithm A. Therefore, the upper bound of $(n+m)^{O(\log n)}$ on running time applies to this algorithm as well.

To analyze the storage requirement of the algorithm, we need to specify how the processes are scheduled. At any moment during the algorithm execution, We call a process *live* if it has been initiated and has not been terminated. We call a live process *waiting* if it is waiting for a response of one of its subprocesses; otherwise it is *ready*. The ready process that is being executed is called *current*. The scheduling algorithm maintains a tree T of live processes, which we call the *process tree*, and a stack S of ready processes.

(1) Initially, T consists of a single process that executes the main steps of the algorithm.
(2) When process p initiates a subprocess q, we put q into T as a child node of p. If p already has a child node (at most one is possible), q is added to the *right* of it. We then make q current. Algorithm B is such that p always becomes waiting in this event.
(3) When process q reports termination to its parent process p, we remove q from T and make p current.
(4) When process q reports a transversal to its parent process p, we make p current. If q does not become waiting in this event, then we push q into the stack S. This happens, when a process in Case 0 reports the empty transversal and is ready to report termination and when a process in Case 2 at step (2.2.1) reports Y and is ready to execute the subsequent steps.
(5) When process p becomes waiting, without reporting a transversal to its parent (this also applies to the main process when it prints a reported transversal and becomes waiting), we pop a process from S and make it current.

This scheduling algorithm maintains the following invariants.
(I1) All the ready processes except for the current are in the stack S. The entries in S are ordered according to the right-to-left ordering in the process tree T (where we regard by convention that a process p is to the right of its child q).
(I2) If a process in the process tree T is waiting, it is waiting for its rightmost child to respond.
(I3) The current process is always on the path in T from the root to the rightmost leaf.

Invariants I2 and I3 together imply that when a process q passes control to its parent p, p has actually been waiting for q to respond. Invariant I1 guarantees that I3 is maintained when a process is popped from S and becomes current.

We are ready to analyze the storage requirement of our algorithm. We need two types of storage. The *process storage* stores the data associated with each process and must be retained as long as the process is in the process tree. The *temporary storage* is used for executing the current process. Once the control is passed to another process, the same temporary storage can be reused to execute the new current process. In particular, temporary storage is used to test if a given vertex set is a (minimal) transversal of a given hypergraph and to evaluate function $f_{\mathcal{H}}$. For all of these, $O(n)$ words of storage is sufficient, even if the hypergraphs are implicitly represented, where n is the size of the hypergraph that is the top level input to the algorithm. (Most naive way to do this is to first convert the hypergraph into an explicit representation in the temporary storage).

The process storage for each process must store
(1) the pointer to the parent process
(2) control point that specifies from which step to resume the process when it becomes current
(3) input hypergraph $\mathcal{H}$, and
(4) when Case 2 applies to the process, the transversal Y of $\mathcal{H}^0$ that is most recently reported from its subprocess.

Our goal is to upperbound the size of the process storage, summed over the entire process tree, by $O(n \log n)$. To achieve this, we need to be careful in choosing representation of the input hypergraph and the transversal to be reported.

We first describe our scheme of representing hypergraphs. Let σ be a finite string of 0 and $*$. For each hypergraph $\mathcal{H}$, define $\mathcal{H}^\sigma$ inductively as follow.

1. $\mathcal{H}^\epsilon = \mathcal{H}$, where ϵ is the empty string.
2. If $\mathcal{H}^\sigma$ is defined and is a nontrivial hypergraph then $\mathcal{H}^{\sigma 0} = (\mathcal{H}^\sigma)^0$ and $\mathcal{H}^{\sigma *} = (\mathcal{H}^\sigma)^*$; otherwise $\mathcal{H}^{\sigma 0}$ and $\mathcal{H}^{\sigma *}$ are undefined.

We use two ways of representing hypergraphs. In the *explicit* representation, we represent the hypergraph $\mathcal{H}$ by an ordinary data structure, such as a list of lists, using $O(\mathsf{size}(\mathcal{H}))$ storage.

In the *implicit* representation, we represent the hypergraph $\mathcal{H}^\sigma$ by a pair $(\mathcal{H}, \sigma)$, where $\mathcal{H}$ is explicitly represented.

We use these two types of representations as follows. The hypergraph that is the top level input to the algorithm is explicitly represented. Among the hypergraphs that are created during the algorithm execution and are passed as an input to a process, only the hypergraphs created as $\mathcal{H}|_Y$, $Y \in \mathrm{TR}\mathcal{H}^0$, are explicitly represented. All other hypergraphs, i.e., those created as $\mathcal{H}^0$ and $\mathcal{H}^*$ are implicitly represented. Let us say that a representation of a hypergraph is in the process tree, if the process that has this representation as its input is in the process tree.

Proposition 3.5 *If an implicit representation $(\mathcal{H}, \sigma)$ is in the process tree, then all the representations it builds upon, i.e., the explicit representation of $\mathcal{H}$ and the implicit representations $(\mathcal{H}, \sigma_1)$, ..., $(\mathcal{H}, \sigma_k)$, where σ_1, ..., σ_k are all the nonempty prefixes of σ, are in the process tree.*

It follows that we may implement implicit representations in the process tree so that the storage cost is $O(1)$ per one instance. The cost of an explicit representation of $\mathcal{H}$ is naturally $O(\mathsf{size}(\mathcal{H}))$.

We represent transversals to be reported by linked lists. For each cell in the linked list, we define the *creator process* of the cell as follows. The transversal that a trivial process reports is an empty set. This is represented by a null pointer and thus does not consume any list cell. Each nontrivial process may create a transversal to report based on a transversal reported by its subprocess. In Case 1, the transversal reported is either directly passed to the parent or discarded, so no list cell is consumed here either. In Case 2, the transversal Y reported from the subprocess for $\mathcal{H}^0$ and the transversal X reported from the subprocess for $\mathcal{H}|_Y$ are unioned, where we know that X and Y are disjoint. In this case, we use $|X|$ new list cells to store the elements of X and put these cells in front of the list for Y to form a list for $X \cup Y$. For these new cells used, the creator process is defined to be the current process. We also report a transversal $\{v_{\mathcal{H}}\} \cup Y$ in Case 2. The list representing this transversal is similarly constructed, using one new list cell for $v_{\mathcal{H}}$, of which the current process becomes the creator.

In the following analysis of the amount of process storage, we charge the cost of storing a transversal Y to the creator processes of the list cells representing Y. This is justified by the following proposition, which can be proved based on the invariants maintained by our scheduling scheme.

Proposition 3.6 *Let p be a process in the process tree and suppose it holds in its process storage a transversal Y that is reported from its subprocess. Then, for each list cell of the list representing Y, the creator process of the cell remains in the process tree (i.e., not terminated yet).*

We now analyze the amount of process storage. For each time step t and a process p in the process tree at time t, let $S_t(p)$ denote the number of process storage words that p is responsible for: more precisely, this includes
(1) the $O(1)$ storage for the parent process pointer and the resumption point.
(2) the storage for representing hypergraphs that p has created and has passed as input to a subprocess that is still in the process tree, and
(3) the list cells for which p is the creator and which are still referred to by some process in the process tree.

Let $\mathcal{H}$ be the input hypergraph for this process p.

If Case 0 applies to p then there is no storage in category (2) and (3). If Case 1 applies, the amount of storage in category (2) is $O(1)$, since only implicitly represented hypergraphs are created in this case, and there is no storage in category (3). If Case 2 applies, the amount of storage in category (2) is $O(\mathsf{size}(\mathcal{H}^1))$ since $\mathsf{size}(\mathcal{H}|_Y) \leq \mathsf{size}(\mathcal{H}^1)$ for every $Y \in \mathrm{TR}\mathcal{H}^0$, and the amount of storage in category (3) is also $O(\mathsf{size}(\mathcal{H}^1))$, because $|X| \leq \mathsf{size}(\mathcal{H}|_Y)$ for every $Y \in \mathrm{TR}\mathcal{H}^0$ and $X \in \mathrm{TR}\mathcal{H}|_Y$. To summarize, $S_t(p) \leq O(1)$ for both Case 0 and Case 1 and $S_t(p) \leq O(\mathsf{size}(\mathcal{H}^1))$ for Case 2.

Let $B_t(p)$ denote the sum of $S_t(q)$ over all the processes q in the process tree rooted at p. Let $B(n)$ denote the worst possible value of $B_t(p)$, over all possible input $\mathcal{H}$ to the process p with $\mathsf{size}(\mathcal{H}) \leq n$ and over all possible time steps t.

Depending on which case in the algorithm applies to the root process in the worst case process tree, one of the following inequalities hold: $B(n) \leq O(1)$ in Case 0; $B(n) \leq O(1) + B(n-1)$ in Case 1; and $B(n) \leq O(k) + B(k) + B(n-k-1)$ for some $1 \leq k \leq n/2$ in Case 2. Let c be a large enough constant such that $B(1) \leq c$ and

$$B(n) \leq c + B(n-1) \tag{2}$$

or

$$B(n) \leq ck + B(k) + B(n-k-1)B(n-1) \quad \text{for some } 1 \leq k \leq n/2 \tag{3}$$

for $n \geq 2$.

We show by induction that $B(n) \leq c(n \log n + 1)$ for every $n \geq 1$. The base case $n = 1$ is trivial. Let $n > 1$ and suppose $B(k) \leq c(k \log k + 1)$ holds for every $k < n$. If inequality (2) holds then we have $B(n) \leq c + c((n-1)(\log(n-1) + 1)) \leq c(n \log n + 1)$. If inequality (3) holds, then we have

$$\begin{aligned} B(n) &\leq ck + c(k \log k + 1) + c((n-k-1)\log(n-k) + 1) \\ &\leq ck + c(k(\log n - 1) + 1) + c((n-k-1)\log n + 1) \\ &\leq c(n-1)\log n + 2c \\ &\leq c(n \log n + 1) \end{aligned}$$

Thus, we have proved the following theorem.

Theorem 3.7 *Given a hypergraph $\mathcal{H}$ with $\mathsf{size}(\mathcal{H}) = n$ and $|\mathrm{TR}\mathcal{H}| = m$, Algorithm B enumerates the minimal transversals of $\mathcal{H}$ in total time of $O((n+m)^{O(\log n)})$ using $O(n \log n)$ words of storage.*

4 Open questions

There are several interesting (but apparently hard) open questions. Can we enumerate all the m minimal transversals of a hypergraph of size n in time $mn^{O(\log n)}$, i.e., in an amortized time per a transversal that is quasi-polynomial in n? Can we even enumerate with quasi-polynomial delay, i.e., can we bound the time spent for obtaining each next transversal by $n^{O(\log n)}$? If either of this is possible, can it be done space-efficiently? We can address variants of the above questions in which "quasi-polynomial" is replaced by "polynomial". Of course, we should keep in mind that such variants can be easier to solve than the long standing open question of if duality testing is in P, only if the answer is negative.

It would also be interesting to investigate whether the techniques used in this paper can be applied to other enumeration problems that are polynomially equivalent to dualization and yield space-efficient algorithms.

On a more practical side, it appears worthwhile to implement the algorithm of this paper making full use of efficient data structures and see if it competes with algorithms that are used in practice.

参考文献

[1] J. Bioch and T. Ibaraki: Complexity of identification and dualization of positive boolean functions. Information and Computation, 123(1): 50-63, 1995.

[2] E. Boros and V. Gurvich and P.L. Hammer, Dual subimplicants of positive Boolean functions, Optimization Methods and Software, 10: 147-156, 1998 (RUTCOR Research Report 11-93).

[3] E. Boros and V. Gurvich and L. Khachiyan and K. Makino: Dual-bounded hypergraphs: generating partial and multiple transversals, DIMACS Technical Report 00-62, 1999.

[4] J. T. Butler: The worst and best sum-of-product expressions for a boolean function, seminar talk, Department of Computer Science, Meiji University, April, 2000.

[5] Y. Crama: Dualization of regular boolean functions, Discrete Applied Mathematics, 16:79-85, 1987.

[6] T. Eiter and G. Gottlob: Identifying the minimal transversals of a hypergraph and related problems, SIAM Journal on Computing, 24(6): 1278-1304, 1995.

[7] M.L. Fredman and L. Khachiyan: On the complexity of dualization of monotone disjunctive normal forms. Journal of Algorithms, 21(3):618-628, 1996.

[8] V. Gurvich and L. Khachiyan: On generating the irredundant conjunctive and disjunctive normal forms of monotone Boolean functions, Discrete Applied Mathematics, 97: 363-373, 1999.

[9] D.S. Johnson and M. Yannakakis and C.H. Papadimitriou: On generating all maximal independent sets, Information Processing Letters, 27: 119-123, 1988.

[10] K. Makino and T. Ibaraki: The maximum latency and identification of positive boolean functions, SIAM Journal on Computing, 26(5): 1363-1383, 1997.

[11] U.N. Peled and B. Simeone: An $O(nm)$-time algorithm for computing the dual of a regular Boolean function, Discrete Applied Mathematics, 49: 309-323, 1994.

[12] T. Sasao and J. T. Butler: Comparison of the worst and best sum-op-products expressions for multiple-valued functions, 27th International Symposium on Multiple-Valued Logic, 55-60, 1997.

[13] T. Sasao: personal communication, May 2000.

[14] S. Tsukiyama and M. Ide and H. Ariyoshi and I. Shirakawa: A new algorithm for generating all the maximal independent sets, SIAM Journal on Computing, 6(3):505-517, 1977