

# ALGORITHM FOR ENUMERATING HYPERGRAPH TRANSVERSALS

by

ROSCOE CASITA

A THESIS

Presented to the College of Arts and Sciences: Computer and Information Science  
and the Graduate School of the University of Oregon  
in partial fulfillment of the requirements  
for the degree of  
Masters of Computer Information Science

June 2017

## THESIS APPROVAL PAGE

Student: Roscoe Casita

Title: Algorithm for Enumerating Hypergraph Transversals

This thesis has been accepted and approved in partial fulfillment of the requirements for the Masters of Computer Information Science degree in the College of Arts and Sciences: Computer and Information Science by:

Boyana Norris  
Boyana Norris  
Chris Wilson

Chair  
Advisor

and

Scott L. Pratt

Vice President for Research & Innovation/  
Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2017

© 2017 Roscoe Casita

## THESIS ABSTRACT

Roscoe Casita

Masters of Computer Information Science

Computer and Information Science

June 2017

Title: Algorithm for Enumerating Hypergraph Transversals

This paper introduces the hypergraph traversal problem along with the known solutions: Naive, Branch and bound, and polynomial-space iterative. version of the algorithm. Lists instead of sets are used to define Odometers and Hypergraphs. All familiar set operations on Odometers are redefined as list operations for succinctness. Lastly future research directions are examined.

## CURRICULUM VITAE

NAME OF AUTHOR: Roscoe Casita

### GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR

Oregon Institute of Technology, Klamath Falls, OR

### DEGREES AWARDED:

Master of Computer Information Science, 2017, UO, 2017, 3.37 GPA

B.S. Software Engineering Technology, O.I.T., 2007, 2.51 GPA

A.S. Computer Engineering Technology, O.I.T., 2007, 2.51 GPA

Minor, Applied Mathematics, O.I.T., 2007, 2.51 GPA

### AREAS OF SPECIAL INTEREST:

Machine Learning and Hypergraphs

### PROFESSIONAL EXPERIENCE:

Software Engineer III, Datalogic Scanning INC., 9 years

### GRANTS, AWARDS AND HONORS:

Eagle Scout, Boy Scout Troop 888, Volcano, CA, 2000

### PUBLICATIONS:

## ACKNOWLEDGEMENTS

For my parents, my wife, my kids, and all future generations.

## TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION . . . . .	1
II.	DEFINITIONS . . . . .	2
	Lists, Queues, Stacks, N-Trees, etc... . . . .	2
	Odometers . . . . .	2
	Hypergraphs . . . . .	3
	Minimal Transversal of a Hypergraph . . . . .	7
	All minimal transversals of a hypergraph . . . . .	7
III.	NAIVE SOLUTION . . . . .	8
IV.	BRANCH AND BOUND . . . . .	9
V.	RECURSIVE POLYNOMIAL SPACE . . . . .	10
VI.	ITERATIVE POLYNOMIAL SPACE SOLUTION . . . . .	11
	IS Appropriate . . . . .	12
	Generate Next Depth . . . . .	12



Chapter	Page
VII. TESTING . . . . .	16
Generating Hypergraphs . . . . .	16
VIII. FUTURE DIRECTIONS . . . . .	18
Distributed Computation via Iterative Containers . . . . .	18
Hyperedge Visitation Manipulation . . . . .	19
Advanced Algorithmic Modifications . . . . .	19
Experimental Research . . . . .	20
APPENDIX: ALGORITHMS REFERENCED IN PAPER . . . . .	21
REFERENCES CITED . . . . .	25

## LIST OF FIGURES

Figure

Page

## LIST OF TABLES

Table	Page
-------	------

## CHAPTER I

### INTRODUCTION

“The theory of hypergraphs is seen to be a very useful tool for the solution of integer optimization problems when the matrix has certain special properties” - Berge [1984]

Hypergraphs are a recent mathematical and computational discovery. The number of edges in a *normal* hypergraph is potentially  $2^N$ . Abstracting from a *normal* to a *unrestricted* hypergraph allows the edges to unbounded  $N^\infty$ . An *unrestricted* hypergraph can still be reasoned about. Odometers Fuchs [2016] are defined and the functions that operate on them. Then hyperedges are shown to be mutually interchangeable with odometers given a hypergraph and transformation functions.

The NP-Complete problem enumeration of all minimal traversals of a hypergraph is defined Eiter [1991]. There are multiple corresponding NP-Complete problems that are shown to be equivalent Eiter and Gottlob [1995]. Finding all minimal transversals is a significant and worthy problem in computation and especially in AI Eiter and Gottlob [2002].

There are objectively better minimal transversals such as the total count of vertexes in the traversal Bailey et al. [2003]. Traversing a hypergraph is the first step to efficient traversal Boros et al. [2003]. Optimal traversals of a hypergraph are fundamentally similar to optimization of a property and NP-Hard. It is possible to at least generate efficient traversals that generate .

## CHAPTER II

### DEFINITIONS

#### **Lists, Queues, Stacks, N-Trees, etc...**

This paper assumes the reader is familiar with the basic data structures such as lists, arrays, stacks, queues, trees, graphs etc. Let a list  $l$  be an ordered list of things  $\{t, i\}$  where each thing  $t$  must be the same type, and  $t$  is only distinguishable by its index  $i$ . For the purposes of this paper,  $x.push(y)$  will insert  $y$  at the last index of the list,  $x.pop()$  will remove from the last index of the list.  $x.enqueue(y)$  will insert  $y$  at the last index of the list.  $x.dequeue()$  will remove from the first index of the list. A list-of-lists structure is used in this paper to represent the traversal of an N way branching tree. The internal C++ implementation is an array representation with  $O(1)$  index lookup time,  $O(N)$  seek, insert, and remove times.

#### **Odometers**

An odometer is an ordered multiset of integer numbers. Let an odometer  $o$  be an list of integers  $n$  and indexes  $\{n, i\}$ . The  $i^{th}$  indexable integer of an odometer can be written  $n_i = o[i]$ . Integers  $n$  can be repeated, they are distinguished via their index. Indexes  $i$  are unique non-repeating whole numbers from  $[0, \infty]$ . The size of the odometer is written as  $o.size()$ , is the count of  $\{n, i\}$ .

Instead of reasons about hyperedges, odometers and used in their place. In the next section is it shown they are mutually exchangeable. Bit vectors are commonly used instead for set operations, but in this case the number of vertexes is usually far larger then system bit sizes (32,64 etc). When using full integers

for items in the list there are  $N^M$  values are available, where  $N$  is the count of integers, and  $M$  is the number of values that integers can take on. Thus  $O(N^{(2^{bits})})$  is the general complexity without restrictions. Fuchs [2016]

An odometer is a construct used extensively throughout this paper as it can be treated as an ordered set of numbers, an unordered bag of numbers, as an instance container to store state. As an unrestricted list of numbers the odometer is similar to an instance of a turing machine tape.

The following common functions are defined in Code Appendix C: *Union*, *Intersection*, *Minus*, *StrictEqual*, *SetEqual*. The following functions are implemented, additionally they are short circuit versions when possible: *DoesACoverB*, *DoesAHitB*, *DoesACoverBorBCoverA*, *DoesAHitAll*, *DoesAnyHitA*. Please note that all functions are polynomial in both space and time. Also note that *DoesAHitAll* implements the hitting set test. *GenerateNMinusOne(o)* is used for both minimal hitting set and other functions.

## Hypergraphs

### *Unrestricted Hypergraphs*

The traditional hypergraph definition  $H = (V, E)$  is terse for implementers. Traditionally a hypergraph is defined as a collection of sets where there is no ordering and repeated elements are not allowed. The following definitions were used to implement the hypergraph interface. The odometer is of particular interest as it can be used independently from hypergraphs for linear integer optimization techniques.

Let a hyperedge  $e$  be a list of vertexes:  $e = \{v, i\}$ . The  $i^{th}$  indexable vertex of  $e$  can be written  $v_i = e[i]$ . Vertexes  $v$  can be repeated, they are distinguished

via their index. Indexes  $i$  are unique non-repeating whole numbers from  $[0, \infty]$ . The size of the hyperedge written as  $e.size()$  is the count of  $\{v, i\}$ .

Let an unrestricted hypergraph  $U$  be a single hyperedge  $nodes$  and the two functions  $OtoE$  and  $EtoO$ .  $OtoE$  is the surjective function to map a given odometer to a hyperedge.  $EtoO$  is the injective function to map a given hyperedge to an odometer. The hyperedge  $U.nodes$  cannot repeat any vertexes  $v$  for the function  $EtoO$  to behave correctly.

Given these definitions, the following is now possible given a hypergraph: A hyperedge can be constructed from an odometer. An odometer can be constructed from a hypergraph. While the functions in the paper use hyperedges the code uses odometers in place of hyperedges. Thus every instance of a hyperedges can be converted to an instance of an odometer, and every instance of an odometer can be converted to an instance of a hyperedge.

Specifically the odometer is an instance of a set of integer numbers that can be reasoned about independently of a hypergraph. The code implements some common set functions that allow the constraints to be expressed, such as union, minus, include short circuit versions of functions for faster performance.

---

**Algorithm 1** OdometerToHyperedge

---

```

1: function OTOE( $U, o$ )
2:    $e \leftarrow \emptyset$ 
3:    $size \leftarrow U.nodes.size()$ 
4:   for all  $\{n, i\} \in o$  do
5:     // where  $-2 \bmod 7 = 5$  .
6:      $e[i] \leftarrow U.nodes[n \% size]$  // convert an integer number to index.
7:   return  $e$ 

```

---

Notice that these functions provide polynomial time access to all permutations, combinations, repeats, patterns etc. Thus reasoning about a

---

**Algorithm 2** HyperedgeToOdometer

---

```
1: function ETOO( $U, e$ )
2:    $o \leftarrow \emptyset$ 
3:   for all  $\{v_e, i_e\} \in e$  do // use hashmap of v to i
4:     for all  $\{v_n, i_n\} \in U.nodes$  do // to reduce  $O(n^2)$  to  $O(n)$ 
5:       if  $v_e = v_n$  then
6:          $o[i_e] \leftarrow i_n$  // lookup index and save as integer.
7:   return  $o$ 
```

---

hyperedge is equivalent to reasoning about its corresponding odometer, and vice versa. Vertex data can be complex and large, thus reasoning about the odometer in place of the hyperedge is for performance and interesting reasons noted later.



### *Normal Hypergraphs ?*

Let a *normal* hypergraph be  $H = (V, E)$  where  $V$  is a list of vertexes  $v, i$ ,  $E$  is a list of hyperedges  $e, i$  where each hyperedge  $e$  is a subset of  $V$ . The following trivial restrictions must be imposed to get the expected behavior out of a *normal* hypergraph given the unrestricted list definitions. No hyperedge contains a duplicated vertex. Every vertex in all hyperedges is contained in the hypergraph list of vertexes. There are no duplicate hyperedges. The maximal size of a hyperedge is the size of all hypergraph vertexes. Every vertex exists in at least one hyperedge. There are no duplicate vertexes in the hypergraph.

$$\forall e \in E, \forall v, v' \in e | v \neq v'$$

$$\forall e \in E, \forall v \in e | v \in V$$

$$\forall e \in E, \nexists e' \in E | e = e'$$

$$\forall e \in E | |e| \leq |V|$$

$$\forall v \in V, \exists e \in E | v \in e$$

$$\forall v \in V, \nexists v' \in V | v = v'$$

### *Simple Hypergraphs*

Let a *simple* hypergraph be  $H = (V, E)$  as *normal* hypergraph with the additional restriction that no hyperedge fully contains any other hyperedge.

$$\forall e, e' \in E | e \not\subseteq e' \wedge e' \not\subseteq e$$

## Minimal Transversal of a Hypergraph

Let the transversal of a hypergraph  $T \subseteq H.V$  be a hitting set of all the hyperedges of a hypergraph such that  $DoesAHitAll(T, H.E) = true$ . Using the definitions of *GenerateNMinusOne* the following implementation determines if an odometer hits ever odometer in a list.

---

### Algorithm 3 IsMinimalTransversal

---

```

1: function ISMINIMALTRANSVERSAL( $o, list\_of\_o$ )
2:   if DoesAHitAll( $o, list\_of\_o$ ) = false then
3:     return false
4:   for all  $\{o_n, i_n\} \in GenerateNMinusOne(o)$  do
5:     if DoesAHitAll( $o_n, list\_of\_o$ ) then
6:       return false
7:   return true

```

---

## All minimal transversals of a hypergraph

There are  $2^{|V|}$  possible combination sets that can be derived from the hypergraph  $H = (V, E)$  and therefore  $2^{|V|}$  transversals that need to be enumerated in the worst case. Tractable scalable algorithms fundamentally need to use polynomial space storage and exponential time to enumerate the traversals efficiently.

## CHAPTER III

### NAIVE SOLUTION

---

**Algorithm 4** NaiveAllPotentialTransversals

---

```
1: function NaiveAllPotentialTransversals(H, CallbackFunc)
2:   count  $\leftarrow H.E.size()$ 
3:   o  $\leftarrow \emptyset$ 
4:   o.push(0)
5:   while o.size() > 0 do
6:     if IsMinimalTransversal(o, H.E) then
7:       CallbackFunc(o, OtoE(H, o))
8:     cur  $\leftarrow o[o.size() - 1]$ ;
9:     next  $\leftarrow cur + 1$ ;
10:    if next < count then
11:      o.push(next)
12:    else
13:      o.pop()
14:      if o.size() > 0 then
15:        o[o.size() - 1]  $\leftarrow o[o.size() - 1] + 1$ 
```

---

## CHAPTER IV

### BRANCH AND BOUND

---

**Algorithm 5** BranchAndBoundTransversals

---

```
1: function BranchAndBoundTransversals(H, CallbackFunc)
2:   count  $\leftarrow H.E.size()$ 
3:   o  $\leftarrow \emptyset$ 
4:   o.push(0)
5:   while o.size() > 0 do
6:     if IsMinimalTransversal(o, H.E) then
7:       CallbackFunc(o, OtoE(H, o))
8:       if o[o.size() - 1] < count - 1 then
9:         o[o.size() - 1]  $\leftarrow$  o[o.size() - 1] + 1
10:      else
11:        o.pop()
12:        if o.size() > 0 then
13:          o[o.size() - 1]  $\leftarrow$  o[o.size() - 1] + 1
14:      else
15:        cur  $\leftarrow$  o[o.size() - 1];
16:        next  $\leftarrow$  cur + 1;
17:        if next < count then
18:          o.push(next)
19:        else
20:          o.pop()
21:          if o.size() > 0 then
22:            o[o.size() - 1]  $\leftarrow$  o[o.size() - 1] + 1
```

---

## CHAPTER V

### RECURSIVE POLYNOMIAL SPACE

The recursive polynomial space solution can be found in the previous works:  
Kavvadias and Stavropoulos [1999], Kavvadias and Stavropoulos [2005],

## CHAPTER VI

### ITERATIVE POLYNOMIAL SPACE SOLUTION

This paper now introduces the iterative psuedo-polynomial space solution to enumerating all minimal hypergraph traversals. First the depth control is used to expand the tree to the leaf and store the next nodes to be processed. Each node is then removed and processed, if the node is a leaf then the minimal transversal is visited, if the node is not a leaf then generate a new set of children to process, if no children are generated then this minimal transversal does not have any children after the next edge.

*Define: Hypergraph stack frame*

A hypergraph stack frame  $HSF = (Transversals, Negations)$  is a collection where  $Transversals$  is a list of generalized variables (odometers),  $Negations$  is a list of generalized variables (odometers) for  $IsAppropriate$ .

*Define: Gamma*

A Gamma is the piecewise segmentation of an individual generalized variable intersecting parts with the incoming edge.  $G = (XMinusY, XIntersectY, YMinusX)$ .

*Define: IHGResult*

An IHGResult  $ihg\_result = (Alphas, Betas, Gammas, new\_alpha)$  is a collection where  $Alphas$  is a list of generalized variables (Odometers),  $Betas$  is a list of generalized variables (odometers),  $Gammas$  is a list of Gammas from the previous definition, and  $new\_alpha$  is the incoming edge minus all intersections.

### Generate IHGResult from Transversals and Edge

Using the previous definitions the function to break a transversals generalized variables down into the constituent types and pieces. The function *IntersectTransversalWithEdge* breaks apart the entire intersection of a minimal transversal with a new edge.

---

#### Algorithm 6 IntersectTransversalWithEdge

---

```

1: function IntersectTransversalWithEdge(list_of_transversals, edge)
2:   return_value  $\leftarrow \emptyset$  // IHGResult.
3:   new_alpha  $\leftarrow$  edge // copy incoming edge.
4:   for all  $\{g_t, i_t\} \in \text{list\_of\_transversals}$  do
5:     intersect = Intersection(g_t, edge)
6:     new_alpha  $\leftarrow$  Minus(new_alpha, intersect)
7:     if intersect.size() = 0 then
8:       return_value.Alphas.push(g_t)
9:     else
10:      if intersect.size() = g_t.size() then
11:        return_value.Betas.push(g_t)
12:      else
13:        Gamma  $\leftarrow \emptyset$  // Gamma type.
14:        Gamma.XMinusY = Minus(g_t, edge)
15:        Gamma.XIntersectY = intersect
16:        Gamma.YMinusX = Minus(edge, g_t)
17:        return_value.Gammas.push(gamma)
18:   return_value.new_alpha = new_alpha
19:   return return_value

```

---

IS Appropriate

Generate Next Depth

---

**Algorithm 7** IsAppropriate

---

```
1: function IsAppropriate(HSF, edge)
2:   list_of_new_traversals  $\leftarrow \emptyset$ 
3:   for all  $\{o, i\} \in \textit{HSF.Transversals}$  do
4:     gv  $\leftarrow o$ 
5:     for all  $\{n, i\} \in \textit{HSF.Transversals}$  do
6:       if DoesACoverB(n, gv) = true then
7:         gv  $\leftarrow \textit{Minus}(\textit{gv}, n)$ 
8:       if gv.size() > 0 then
9:         list_of_new_traversals.push(gv)
10:  if DoesAnyHitA(list_of_new_traversals, edge) = false then
11:    return false
12:  return true
```

---

Depth First N-Way Tree Control



---

**Algorithm 8** GenerateNextDepth

---

```
1: function GenerateNextDepth(HSF, edge)
2:   new_frame  $\leftarrow \emptyset$  // hypergraph stack frame
3:   return_value  $\leftarrow \emptyset$  // list of hypergraph stack frames.
4:   result  $\leftarrow$  IntersectTransversalWithEdge(HSF.Transversals, edge)
5:   if result.Gammas.size() == 0 then
6:     new_frame.Transversals = HSF.Transversals
7:     new_frame.Negations = HSF.Negations
8:     if result.Betas.size() > 0 then
9:       for all  $\{b, i\} \in$  result.Beta do
10:        new_frame.push(b)
11:   else
12:     new_frame.push(edge)
13:     if IsAppropriate(new_frame, edge) then
14:       return_value.push(new_frame)
15:   else
16:     for all list_of_bool  $\in$  Gen2expNtruefalse(result.Gammas.size()) do
17:       new_frame.Transversals  $\leftarrow$  result.Alphas
18:       new_frame.Negations  $\leftarrow$  HSF.Negations
19:       for all  $\{tf, j\} \in$  list_of_bool do
20:         gamma  $\leftarrow$  result.Gammas[j]
21:         if tf[j] = false then
22:           new_frame.Transversals.push(gamma.XMinusY)
23:           new_frame.Negations.push(gamma.XIntersectY)
24:         else
25:           new_frame.Transversals.push(gamma.XIntersectY)
26:         if IsAllTrue(tf) = true then
27:           if result.new_alpha.size() > 0 then
28:             new_frame.Transversals.push(result.new_alpha)
29:         else
30:           for all beta  $\in$  result.Betas do
31:             new_frame.Transversals.push(beta)
32:           if IsAllFalse(tf) = true then
33:             for all gamma  $\in$  result.Gammas do
34:               new_frame.Negations.push(gamma.XMinusY)
35:           if IsAppropriate(new_frame, edge) = true then
36:             return_value.push(new_frame)
37:           new_frame  $\leftarrow \emptyset$ 
38:   return return_value
```

---

---

**Algorithm 9** HypergraphTransversals

---

```
1: function HypergraphTransversals(H, CallbackFunc)
2:   edge_count  $\leftarrow H.E.size()$ 
3:   control_stack  $\leftarrow list(edge\_count)$  // list of stacks pre-sized.
4:   HSF  $\leftarrow \emptyset$  // current hypergraph stack frame
5:   HSF.Transversals.push(edge)
6:   control  $\leftarrow 0$  // depth control variable.
7:   control_stack[control].push(HSF) // load the process.
8:   while control  $\geq 0$  do
9:     if control_stack[control].size() = 0 then
10:      control  $\leftarrow control - 1$ 
11:     else
12:       frame  $\leftarrow control\_stack[control].pop()$ 
13:       if control = edge_count - 1 then
14:         CallbackFunc(frame.Transversals) // min transversal reached.
15:       else
16:         control  $\leftarrow control + 1$ 
17:         next_edge  $\leftarrow H.E[control]$ 
18:         children  $\leftarrow GenerateNextDepth(frame, next\_edge)$ 
19:         for all  $\{c, i\} \in children$  do
20:           control_stack[control].push(c) // next to be processed
```

---

## CHAPTER VII

### TESTING

#### **Generating Hypergraphs**

Hypergraphs are a relatively new data structure; large datasets are not currently modeled as hypergraphs so ensuring correctness is the onus of the implementers. We seek to prove the algorithm is correct for a large set of small hypergraphs and a small set of large hypergraphs. The following algorithm will generate all simple hypergraphs with node count  $N$ . The complexity is exponentially exponential on the order of  $N!^{N-1!^{N-2!^{\cdots}}}$ .

---

**Algorithm 10** GenHypergraphs

---

```
1: function GenHypergraphs(Nodes, CallbackFunc)
2:    $V = \text{Nodes}$  // new hyperedge for a hypergraph of all the nodes.
3:    $E = \text{Nodes}$  // single hyperedge of all the nodes.
4:    $H = (V, E)$  // hypergraph with the above.
5:   CurrentQueue.push(H)
6:   WorkQueue.push(CurrentQueue)
7:   while !WorkQueue.empty() do
8:     CurrentQueue  $\leftarrow$  WorkQueue.pop()
9:      $H \leftarrow \text{CurrentQueue.pop}()$ 
10:    if !CurrentQueue.empty() then
11:      WorkQueue.push(CurrentQueue)
12:      CallbackFunc(H) // process generated hypergraph
13:      CurrentQueue = GenerateHypergraphChildren(H)
14:      if !CurrentQueue.empty() then
15:        WorkQueue.push(CurrentQueue)
16: function GenerateHypergraphChildren(H)
17:   Children  $\leftarrow \emptyset$ 
18:   for all edge  $\in H.E$  do
19:     for all  $\{\text{edges}, i\} \in \text{GenerateNMinusOne}(\text{edge})$  do
20:       new_edges  $\leftarrow H.E \setminus \text{edge}$  // every edge but the new ones
21:       for all  $\{e_n, i_n\} \in \text{edges}$  do
22:         new_edges.push(en) // add new broken down edges.
23:       Children.push(Hypergraph(H.V, new_edges))
24:   return Children
```

---

## CHAPTER VIII

### FUTURE DIRECTIONS

#### **Distributed Computation via Iterative Containers**

Fundamentally this paper re-introduces a polynomial space algorithm for computing hypergraph transversals as an iterative procedure. Each work item that is processed is a partial transversal with the included negative sets and a new hyperedge. The abstraction completely captures all the data needed to generate the next set of work items. The N-Way-Tree algorithm that iteratively constructs the next work items or processes completed transversals can be easily replaced.

The replacement for the N-Way-Tree would be a distributed processing dispatcher. Each work item would be put in a queue for execution on a compute node. All data to process the node is present in the work item. All the future work items can be generated on a compute node, then enqueued in the distributed dispatcher processor.

Hyperedges are always encountered in the same order for all work items. A pipeline of compute nodes with hyperedges can be built to distribute the work. A compute node is paired with a specific hyperedge and a work queue.

As each work item would be dequeued on the current compute node with the current edge. Work items would be generated and enqueued in the next compute node with the next edge. Controlling work item generation to keep the pipeline full becomes a new problem.

## Hyperedge Visitation Manipulation

Conceptually a compute node contains a queue of work items and a hyperedge, an interesting and notable effect of encountering a new hyperedge when the partial transversal negates all new transversals, causing the complete removal of all future work items. Simply a new hyperedge is encountered that causes a minimal transversal to either become invalid or redundant. Eliminating future minimal transversals before they generate children that need to be eliminated is a possible optimization.

Reordering the visitation of hyperedges would need to be provably sound such that a dispatch algorithm could reorder the visitation of hyperedges to eliminate minimal transversals as quickly as possible during computation. As each work item has a list of negation odometers, it is possible to look for a hyperedge contained in the negation odometer union with the transversal odometer to generate either 1 new minimal transversal child or not being an appropriate minimal transversal and being eliminated from the work queue.

## Advanced Algorithmic Modifications

The inter-section, outer-section, and cross product sections each generate a work item. Currently all work items for a given level are expanded to the next level (exponential in the worst case), then processing the first one in the same way expanding the tree in a depth first search. Control of the expansion can be done via additional odometer states in each work item. Replacement of the function *Gen2expNtruefalse* with an iterative generator allows for a more controlled expansion.

The generation of the next list of work items is currently a fixed ordering that generates the same traversals in the same order every time. Obviously this is desirable from a completeness aspect, but from a practical perspective iterating the traversals which satisfy optimality constraints ‘better’ is of particular interest. ‘Better’ is a term that needs to be defined in terms of the particular problem that is being solved. Khachiyan et al. [2006]

### **Experimental Research**

A possible direction of research is to modify the core algorithms to look for ways to collapse the work items being generated at each depth. The compact transversal representations generated by this algorithm store exponential traversals in polynomial space. Removal of the negation sets in conjunction with collapsing work items at each level would result in a polynomial reduction by storing traversals in an exponential encoding. This is the partial transversal frame that is enumerated to generate its transversals

It is not possible to enumerate all transversals of a hypergraph in polynomial space and polynomial time. It may be possible to enumerate all exponential encodings of transversals of a hypergraph in polynomial space and polynomial time. The current algorithm already generates a polynomial space encoding of an exponential transversal. it breaks the encoding down

a representation of transversals as a polynomial space encoding that expands into an exponential number of transversals.

## APPENDIX

### ALGORITHMS REFERENCED IN PAPER

---

**Algorithm 11** Union

---

```
1: function Union( $A, B$ )
2:   returnValue  $\leftarrow \emptyset$ 
3:   for all  $\{n, i\} \in A$  do
4:     if  $\neg \text{returnValue.contains}(n)$  then
5:       returnValue.push( $n$ )
6:   for all  $\{n, i\} \in B$  do
7:     if  $\neg \text{returnValue.contains}(n)$  then
8:       returnValue.push( $n$ )
9:   return returnValue
```

---

---

**Algorithm 12** Intersection

---

```
1: function Intersection( $A, B$ )
2:   returnValue  $\leftarrow \emptyset$ 
3:   for all  $\{n_A, i_A\} \in A$  do
4:     for all  $\{n_B, i_B\} \in B$  do
5:       if  $n_A = n_B$  then
6:         returnValue.push( $n_A$ )
7:   return returnValue
```

---



---

**Algorithm 13** Minus

---

```
1: function Minus( $A, B$ )
2:   returnValue  $\leftarrow \emptyset$ 
3:   for all  $\{n_A, i_A\} \in A$  do
4:     add  $\leftarrow true$ 
5:     for all  $\{n_B, i_B\} \in B$  do
6:       if  $n_A = n_B$  then
7:         add  $\leftarrow false$ 
8:     if add = true then
9:       returnValue.push( $n_A$ )
10:  return returnValue
```

---

---

**Algorithm 14** StrictEqual

---

```
1: function StrictEqual( $A, B$ )
2:   for all  $\{n_A, i_A\} \in A$  do
3:     for all  $\{n_B, i_B\} \in B$  do
4:       if  $n_A \neq n_B$  then
5:         return false
6:   return true
```

---

---

**Algorithm 15** SetEqual

---

```
1: function SetEqual( $A, B$ )
2:    $A \leftarrow \text{Sort}(A)$ ;
3:    $B \leftarrow \text{Sort}(B)$ ;
4:   return StrictEqual( $A, B$ )
```

---

---

**Algorithm 16** DoesACoverB

---

```
1: function DoesACoverB( $A, B$ )
2:   for all  $\{n_A, i_A\} \in A$  do
3:     found  $\leftarrow false$ 
4:     for all  $\{n_B, i_B\} \in B$  do
5:       if  $n_A = n_B$  then
6:         found  $\leftarrow true$ 
7:     if found = false then
8:       return false
9:   return true
```

---

---

**Algorithm 17** DoesACoverBorBCoverA

---

```
1: function DoesACoverBorBCoverA( $A, B$ )
2:   if DoesACoverB( $A, B$ ) = true then
3:     return true
4:   if DoesACoverB( $B, A$ ) = true then
5:     return true
6:   return false
```

---

---

**Algorithm 18** DoesAHitB

---

```
1: function DoesAHitB( $A, B$ )
2:   for all  $\{n_A, i_A\} \in A$  do
3:     for all  $\{n_B, i_B\} \in B$  do
4:       if  $n_A = n_B$  then
5:         return true
6:   return false
```

---

---

**Algorithm 19** DoesAHitAll

---

```
1: function DoesAHitAll( $A, list\_of\_o$ )
2:   for all  $\{o, i\} \in list\_of\_o$  do
3:     if DoesAHitB( $A, o$ ) = false then
4:       return false
5:   return true
```

---

---

**Algorithm 20** DoesAnyHitA

---

```
1: function DoesAnyHitA( $list\_of\_o, A$ )
2:   for all  $\{o, i\} \in list\_of\_o$  do
3:     if DoesAHitB( $o, A$ ) = true then
4:       return true
5:   return false
```

---

---

**Algorithm 21** GenerateNMinusOne

---

```
1: function GenerateNMinusOne( $o$ )
2:    $returnValue \leftarrow \emptyset$  // list of odometers
3:   for all  $\{n, i\} \in o$  do
4:      $add \leftarrow o$  // copy odometer
5:      $add.remove(n, i)$  // erase 1 value.
6:      $returnValue.push(add)$ 
7:   return  $returnValue$  // N odometers, each with one item removed.
```

---

---

**Algorithm 22** Gen2expNtruefalse

---

```
1: function Gen2expNtruefalse( $n$ )
2:    $returnValue \leftarrow \emptyset$  // list of (list of true—false)
3:    $max = 1 \ll n$  // max is  $2^n$  bit shifted.
4:   for all  $i \in 0..max$  do
5:      $add \leftarrow \emptyset$  // list of true—false
6:      $counter \leftarrow 1$ 
7:     while  $counter < max$  do
8:       if  $counter \& i = counter$  then
9:          $add.push(true)$ 
10:      else
11:         $add.push(false)$ 
12:       $counter \leftarrow counter \ll 1$  // bit shift left.
13:    $returnValue.push(add)$ 
14: return  $returnValue$  // N odometers, each with one item removed.
```

---

## REFERENCES CITED

- Claude Berge. *Hypergraphs: combinatorics of finite sets*, volume 45. Elsevier, 1984.
- Phillip P. Fuchs. *Permutation Odometers*. [www.quickperm.org/odometers.php](http://www.quickperm.org/odometers.php), 2016.
- Thomas Eiter. *On transversal hypergraph computation and deciding hypergraph saturation*. na, 1991.
- Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6): 1278–1304, 1995.
- Thomas Eiter and Georg Gottlob. Hypergraph transversal computation and related problems in logic and ai. In *European Workshop on Logics in Artificial Intelligence*, pages 549–564. Springer, 2002.
- James Bailey, Thomas Manoukian, and Kotagiri Ramamohanarao. A fast algorithm for computing hypergraph transversals and its application in mining emerging patterns. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 485–488. IEEE, 2003.
- Endre Boros, K Elbassioni, Vladimir Gurvich, and Leonid Khachiyan. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals. In *European Symposium on Algorithms*, pages 556–567. Springer, 2003.
- Dimitris J Kavvadias and Elias C Stavropoulos. Evaluation of an algorithm for the transversal hypergraph problem. In *International Workshop on Algorithm Engineering*, pages 72–84. Springer, 1999.
- Dimitris J Kavvadias and Elias C Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *J. Graph Algorithms Appl.*, 9(2):239–264, 2005.
- Leonid Khachiyan, Endre Boros, Khaled Elbassioni, and Vladimir Gurvich. An efficient implementation of a quasi-polynomial algorithm for generating hypergraph transversals and its application in joint generation. *Discrete Applied Mathematics*, 154(16):2350–2372, 2006.