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	Chair
Boyana Norris	Advisor
Chris Wilson	

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

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 vernacular of Odometers and their relation to hypergraphs. An odometer is simply a list of numbers. Of particular interest is that lists are used instead of sets for definitions. The definitions of hyperedge, hypergraph, are introduced in terms of odometers and lists. The *unrestricted* hypergraph is derived as a result of using a list instead of a set. Restrictions are put in place to enforce *normal* and *simple* hypergraphs.

The hypergraph traversal problem along with the following sample solutions are presented using the defintions introduced: Naive, Branch and bound, polynomial-space iteratrive. Future research directions including distributed variations, preemptive optimization, and potentially condensing transversals themselves.

Lastly odometers are revisited again in the appendix to establish clearly the operations such as Union, Intersetion, etc. All of these familiar operations on sets are directly applicable to lists of nonrepeated sorted numbers.

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

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

Overview

"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 complexity of hypergraphs should not be underestimated. The number of edges in a *normal* hypergraph is potentially 2^N. Abstracting from a *normal* to a *unrestricted* hypergraph allows the edges to grow unbounded N^∞. Even in this form an *unrestricted* hypergraph can still be reasoned about. Potentially *unrestricted* hypergraphs can be used to model all computable models. The vernacular "odometers" Fuchs [2016] are introduced as a construct that can be used to efficiently reason over all hypergraphs.

The following definitions are used extensively throughout the paper: list, odometer, hyperedge, *unrestricted* hypergraph, *normal* hypergraph, *simple* hypergraph, generalized variable, minimal hypergraph transversal, and partial hypergraph transversals.

Odometers are a structure that is independent of hypergraphs that can be reasoned about in their own context. The paper introduces equivalences between an odometer and hyperedge given a hypergraph, additionally an odometer is used as a generalized variable, and lastly a lists of odometers as a partial hypergraph

transversals. Fundamental operations on odometers are included in the appendix for completeness.

Using only these definitions, an algorithm that enumerates the minimal transversals is introduced. 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 Reiter [1987], De Kleer and Williams [1987].

The problem of enumerating all minimal traversals of a hypergraph is a superset of the problem enumerating all minimal hitting sets of a graph. All graphs are hypergraphs. All minimal hitting sets of a graph are minimal transversals of a hypergraph. The problem has already been shown to be computationally equivalent to the NP-Complete problem space Eiter [1991]. This paper defines the minimal hypergraph transversals problem and demonstrates an iterative polynomial space solution. The previous recursive solution Kavvadias and Stavropoulos [2005] was used as inspiration for the design of this solution.

There are objectively better minimal transversals, a simple example is minimizing the total count of vertices in the traversal Bailey et al. [2003]. The first step to the enumeration of objectively better minimal transversals, is first traversing ONLY the minimal traversals in an efficient way, Boros et al. [2003]. This paper introduces a new iterative polynomial space algorithm to numerate all minimal hypergraph transversals efficiently. The output of the core algorithm is a polynomial space encoding that take exponential time to enumerate.

CHAPTER II

ODOMETERS

Introduction

- Odometer, noun. an instrument for measuring distance traveled, as by an automobile.
- Odometer, portmanteau. "Ordered Meters", a list of measurements.
- Odometer, computer science. a list of numbers.

As a list of numbers, the odometer is an instance of a Turing machine *tape*.

Currently there is only one source for the vernacular term "Odometer" Fuchs [2016]. After defining the odometer succinctly, applying some restrictions, traditional set behavior can be induced easily.

Definition 2.1.1. Let an odometer o be an list of integers n and indexes $i = \{n, i\}$. 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 $o.size()$ which is the count of $\{n, i\}$ items.

Every integer can take on all values from $[-\infty, \infty]$. There are N indexable integers in every odometer. Thus for every different N there are are N^{∞} distinctly different odometers instances that can be created.

The odometer is a structure that can be reasoned about independently of a hypergraph, but this paper does not examine that research direction.

Inducing Structure

An odometer has roots in sets. An odometer is a ordered-multiset. The ordering that is applied is the indexing. Items in the set are allowed to repeat in an odometer.

For this paper odometers need to have similar behavior to mathematical sets. The following restrictions are applied to ensure functions are correct: Odometers may not have repeated integers. Odometer integer values n are sorted from least to greatest over the indexes i from least to greatest. These simple restrictions induce set behavior from odometers.

$$\forall\{n,i\} \in o, \forall\{n',i'\} \in o | n \neq n'$$

$$\forall\{n,i\} \in o, \forall\{n',i'\} \in o | (n < n' \wedge i < i') \vee (n = n' \wedge i = i') \vee (n > n' \wedge i > i')$$

Enforcing these restrictions on odometers induces set behavior for them. Thus the paper now defines the following common set functions: $Union$, $Intersection$, $Minus$, $StrictEqual$, $SetEqual$. They are defined completely in the code appendex.

These additional functions are implemented in terms of odometers and/or lists of odometers: $DoesACoverB$, $DoesAHitB$,$DoesACoverBorBCoverA$, $DoesAHitAll$, $DoesAnyHitA$, $GenerateNMinusOne(o)$. Note that all of these functions are polynomial in both space and time.

CHAPTER III

HYPERGRAPHS

Unrestricted Hypergraphs

Traditionally a hypergraph is defined as a collection H of sets $H = (V, E)$ where V is a set of vertices and E is a set of hypereges. There is no ordering and no repeated hyperedges or vertices. The following definitions are used instead.

Definition 3.0.1. Let a hyperedge e be a list of vertices: $e = \{v, i\}$. The i^{th} indexable vertex of e can be written $v_i = e[i]$. Vertices 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 $\{v, i\}$.

Definition 3.0.2. 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 odometerand an odometer can be constructed from a hyperedge. 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. Both are dependent upon the instance of the hypergraph to maintain consistency. For the rest of the paper hyperedges will now be interchangeable with odometers given a hypergraph and the functions defined.

Algorithm 1 OdometerToHyperedge

```
1: function OtoE(U, o)
2:     e ← ∅
3:     size ← U.nodes.size()
4:     for all {n, i} ∈ o do
5:         // where -2 mod 7 = 5 .
6:         e[i] ← U.nodes[n%size] // convert an integer number to index.
7:     return e
```

Algorithm 2 HyperedgeToOdometer

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

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

Definition 3.0.3. Let a *normal* hypergraph Berge [1984] 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 sublist 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

Definition 3.0.4. Let a *simple* hypergraph be $H = (V, E)$ as *normal* hypergraph with the additional restriction that no hyperedge fully contains any other hyperedge.

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

CHAPTER IV

HYPERGRAPH TRANSVERSAL

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 every odometer in a list of odometers.

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} ∈ 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 V

NAIVE SOLUTION

Given the definitions of an odometer, hypergraph, and $IsMinimalTransversal$ the following naive function enumerates all potential transversals of a hypergraph. Using an odometer and manipulating the counts inside of it makes this a trivial process. Instead of piecewise adding each vertex in a hypergraph to implement this function, the odometer is manipulated in its place then transformed to the hyperedge when needed for the callback to process this transversal.

Here the stack continues to add a new index that comes after the previous index in all cases. This eliminates the regeneration of duplicate potential transversals and reduces the complexity from N^N to only 2^N. Notice that the stack operations performed on the odometer depend on the value in the odometer and their relation to the size of the odometer.

Algorithm 4 NaiveAllPotentialTransversals

```
1:  function NaiveAllPotentialTransversals(H, CallbackFunc)
2:      count ← H.E.size()
3:      o ← ∅ // odometer
4:      o.push(0)
5:      while o.size() > 0 do
6:          if IsMinimalTransversal(o, H.E) then
7:              CallbackFunc(o, OtoE(H, o))
8:          next ← o[o.size() − 1] + 1
9:          if next < count then
10:             o.push(next)
11:         else
12:             o.pop()
13:             if o.size() > 0 then
14:                 o[o.size() − 1] ← o[o.size() − 1] + 1
```

CHAPTER VI

BRANCH AND BOUND

The improved Naive solution is the branch and bound that reduces the number of generated transversals. Using the solution from before: If the odometer is a minimal transversal: increment the current index if possible, else remove the current index and increment the next index if possible. If the odometer is not a minimal transversal: add the next index if possible, else remove current index and increment the next index if possible. Generating excessive transversals can be improved upon with return values $\{hit, nohit, subhit\}$ from IsMinimalTransversal.

Algorithm 5 BranchAndBoundTransversals

```
1:  function BranchAndBoundTransversals(H, CallbackFunc)
2:      count ← H.E.size()
3:      o ← ∅
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] ← o[o.size() − 1] + 1
10:             else
11:                 o.pop()
12:                 if o.size() > 0 then
13:                     o[o.size() − 1] ← o[o.size() − 1] + 1
14:         else
15:             next ← o[o.size() − 1] + 1
16:             if next < count then
17:                 o.push(next)
18:             else
19:                 o.pop()
20:                 if o.size() > 0 then
21:                     o[o.size() − 1] ← o[o.size() − 1] + 1
```

CHAPTER VII

RECURSIVE POLYNOMIAL SPACE

The recursive polynomial space solution that inspired the iterative version can be found in these previous works Kavvadias and Stavropoulos [1999] and Kavvadias and Stavropoulos [2005]. The recursive algorithm uses the similar definition of a transversal hypergraph but using sets instead of lists.

Definition 7.0.1. Let a generalized variable X be an odometer of vertex indexes in the original hypergraph. There are an exponential number of *possible* generalized variable. Let all hyperedges E (odometer) be converted to a generalized variable X directly without translation.

Definition 7.0.2. $Tr(H) = (V, E)$ where V is a list of generalized variables and E is a list of hyperedges such that each item $e \in E$ is a minimal transversals of a hypergraph.

Definition 7.0.3.

Let a partial hypergraph transversal $PHT = (Transversals)$ where $Transversals$ is a list of generalized variables (odometers). The entire that encode an exponential number of minimal transversals. Each generalized variable (odometer) represent a choice selection operator that satisfies all of the hyperedges. If an generalized variable (odometer) in a partial hypergraph transversal has a *size()* that is greater then 1, then this represents another alternative minimal transversal that is only different by one vertex.

The number of minimal transversals contained in a partial hypergraph transversal is the product of all of the sizes of the generalized variables

(odometers). Thus each additional generalized variable in the minimal transversal expands the number of traversals by a factor of the size of the generalized variable.

Notice the above word transversal*s* instead of transversal. The output of this algorithm at each stage is a single hyperedge e of generalized variables. The list of generalized variable (odometer) can contain multiple items itself.

and each hyperedge partial transversal of the original hypergraph. This definition is sound, but must be extended in this paper to capture the mechanics of eliminating transversals that are not *appropriate*.

Now all definitions that should be usable by both algorithms are introduced. Additionally the iterative function that determines the types of generalized variables and also breaks them into their constituent parts is defined succinctly.

This is the benefit of using an odometer instead of a list of vertexes. A generalized variable is similar to the idea of 'pick one', if you pick any value from the generalized variable you will now 'hit' the hyperedge in the original hypergraph.

Processing a partial hypergraph transversal and a new hyperedge

Define: Alpha

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

Define: Beta

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

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 ← ∅ // IHGResult.
3:      new_alpha ← edge // copy incoming edge.
4:      for all {g_t, i_t} ∈ list_of_transversals do
5:          intersect = Intersection(g_t, edge)
6:          new_alpha ← Minus(new_alpha, interset)
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 ← ∅ // Gamma type.
14:                 Gamma.XMinusY = Minus(g_t, edge)
15:                 Gamma.XIntersectY = interset
16:                 Gamma.YMinusX = Minus(edge, g_t)
17:                 return_value.Gammas.push(gamma)
18:     return_value.new_alpha = new_alpha
19:     return return_value
```

CHAPTER VIII

ITERATIVE POLYNOMIAL SPACE SOLUTION

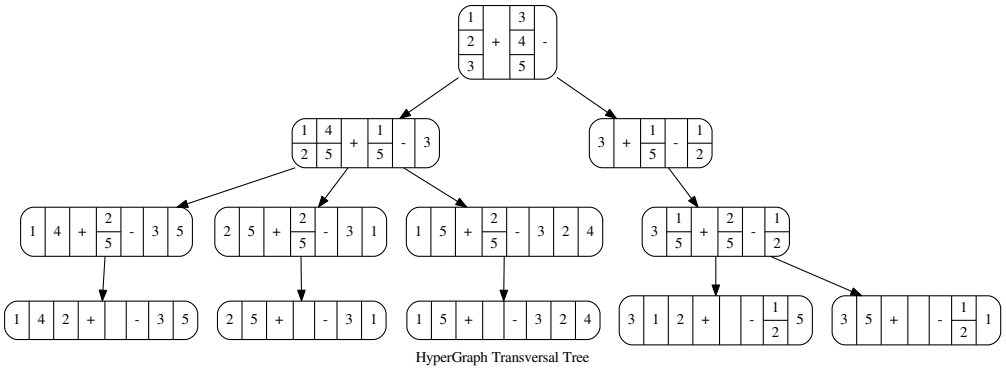
This paper now introduces the iterative polynomial space solution to enumerating all minimal hypergraph traversals. First the depth control is used to expand the tree to the leaf (all hyperedges) and stores the next nodes to be processed later. Each node is then removed and processed, if the node is a leaf then the minimal transversal is visited, if the node is not a leaf then generate new children to process. Before adding the new work items to the queue, each of the new transversals must be appropriate. The implementation of an appropriate work item is introduced later. If no children are generated then this minimal transversal does not have any children after the next edge.

Define: Partial hypergraph transversal stack frame

The previous definition of a partial hypergraph transversal does not take into account the additional list of generalized variables (odometers) needed for the function $IsAppropriate$. A partial transversal stack frame $PTSF = (Transversals, Negations)$ is a collection where $Transversals$ is a list of generalized variables (odometers) as per the previous defintion with the additional $Negations$ as a list of generalized variables (odometers) for the function $IsAppropriate$.

Generate Next Depth

This is a figure of the partial transversals enumerated while traversing the hypergraph $\{\{1,2,3\},\{3,4,5\},\{1,5\},\{2,5\}\}$. Each vertical column represents a generalized variable. Variables left of the plus + sign are partial hypergraph transversals, variables in the center are the new hyperedge being encounter. Variables to the left of the minus − sign are part of the negation list used for determining appropriate nodes to add. The leaf nodes are terminal results, they do not encounter a new hyperedge.



Algorithm 7 GenerateNextDepth

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

Depth First N-Way Tree Control

Algorithm 8 HypergraphTransversals

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

IsAppropriate

The following algorithm is used to determine if the new transversals

Algorithm 9 IsAppropriate

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

CHAPTER IX

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 implementer. 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 = Nodes // new hyperedge for a hypergraph of all the nodes.
3:     E = 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 ← WorkQueue.pop()
9:         H ← 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 ← ∅
18:    for all edge ∈ H.E do
19:        for all {edges, i} ∈ GenerateNMinusOne(edge) do
20:            new_edges ← H.E \ edge) // every edge but the new ones
21:            for all {e_n, i_n} ∈ edges do
22:                new_edges.push(e_n) // add new broken down edges.
23:            Children.push(Hypergraph(H.V, new_edges))
24:        return Children
```

CHAPTER X

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.

Currently hyperedges are always encountered in the same order at each depth of the tree. A pipeline of compute nodes with hyperedges can be built to distribute the work. A compute node could be paired with a specific hyperedge and a work queue. As each work item is created by the previous compute node it would be enqueued in the next compute node work queue.

Controlling work item generation to keep the pipeline full becomes the new problem. A cascading exponential number of work items generated by the early compute nodes will flood the later parts. The expansion of work items is currently exponential in the maximal worst case. Fundamentally another odometer state can

be added to the partial transversal frame and used to iteratively generate the next set of work items. As the pipeline of work items becomes empty the work items would be iterated, generated, and added to the pipeline to keep it full until the current work item is processed.

Hyperedge Visitation Manipulation

An interesting and notable effect of encountering a new hyperedge, is 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 an 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 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 exponential transversals of a hypergraph in polynomial space and polynomial time. It may be possible to enumerate a polynomial encoding of exponential transversals of a hypergraph in both polynomial space and polynomial time.

The current algorithm *already* outputs a polynomial space encoding of an exponential transversal. The partial transversal frame is interpreted as an encoding that generates exponential transversals. A partial transversals can be merged with

another partial transversal if and only if the following holds: They contain the same number of generalized variables. There is only one generalized variable in both transversals that are not equivalent. Notice that both collapsing and comparing against all transversals at a given level is polynomial in space and time. Collapsing all partial transversals at every depth would need to be proven to work correctly.

The output of such an algorithm would be a polynomial space encoding, the same as today. Extracting all of the transversals via interpretation of the encoding is an exponential time operation. Instead of extracting all the transversals, the encodings themselves could represent such things as the clique of cliques, central clusters in clusters, or the eigenvalues that are of the same size.

APPENDIX A

APPENDIX A: BACKGROUND

Lists, Queues, Stacks, N-Trees

Definition A.1.1. Let a list l be an ordered list of things $\{t, i\}$ where each thing t is of the same type, and t is only distinguishable by its index i.

This paper assumes the reader is familiar with the basic data structures such as lists, arrays, stacks, queues, trees, graphs etc.

For the purposes of this paper, $x.push(y)$ inserts y at the last index of the list, $y \leftarrow x.pop()$ will remove from the last index of the list and assign it to the variable y. $x.enque(y)$ will insert y at the last index of the list. $y \leftarrow x.dequeue()$ will remove from the first index of the list and assign it to the variable y. In these example x has an associated type, y is a value of that same type in all cases. A list-of-lists structure is used in this paper to traverse 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.

APPENDIX B

ALGORITHMS REFERENCED IN PAPER

Algorithm 11 Union

```
1: function Union(A, B)
2:     returnValue ← ∅
3:     for all {n, i} ∈ A do
4:         if !returnValue.contains(n) then
5:             returnValue.push(n)
6:     for all {n, i} ∈ B do
7:         if !returnValue.contains(n) then
8:             returnValue.push(n)
9:     return returnValue
```

Algorithm 12 Intersection

```
1: function Intersection(A, B)
2:     returnValue ← ∅
3:     for all {n_A, i_A} ∈ A do
4:         for all {n_B, i_B} ∈ 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 ← ∅
3:     for all {n_A, i_A} ∈ A do
4:         add ← true
5:         for all {n_B, i_B} ∈ B do
6:             if n_A = n_B then
7:                 add ← 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} ∈ A do
3:         for all {n_B, i_B} ∈ B do
4:             if n_A! = n_B then
5:                 return false
6:     return true
```

Algorithm 15 SetEqual

```
1: function SetEqual(A, B)
2:     A ← Sort(A);
3:     B ← Sort(B);
4:     return StrictEqual(A, B)
```

Algorithm 16 DoesACoverB

```
1: function DoesACoverB(A, B)
2:     for all {n_A, i_A} ∈ A do
3:         found ← false
4:         for all {n_B, i_B} ∈ B do
5:             if n_A = n_B then
6:                 found ← 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} ∈ A do
3:         for all {n_B, i_B} ∈ 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} ∈ 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} ∈ 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 ← ∅ // list of odometers
3:     for all {n, i} ∈ o do
4:         add ← 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 ← ∅ // list of (list of true—false)
3:     max = 1 << n // max is 2n̂ bit shifted.
4:     for all i ∈ 0..max do
5:         add ← ∅ // list of true—false
6:         counter ← 1
7:         while counter < max do
8:             if counter&i = counter then
9:                 add.push(true)
10:            else
11:                add.push(false)
12:            counter ← counter << 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, 2016.

Thomas Eiter and Georg Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6): 1278–1304, 1995.

Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

Johan De Kleer and Brian C Williams. Diagnosing multiple faults. *Artificial intelligence*, 32(1):97–130, 1987.

Thomas Eiter. *On transveral hypergraph computation and deciding hypergraph saturation*. na, 1991.

Dimitris J Kavvadias and Elias C Stavropoulos. An efficient algorithm for the transversal hypergraph generation. *J. Graph Algorithms Appl.*, 9(2):239–264, 2005.

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.

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.