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 hypergraph traversal problem along with the known solutions: Naive, Branch and bound, Polynomial space (recursive). The paper introduces a polynomial space (serial) version of the algorithm. Odometers are introduced as an key type for hypergraphs and detailed for future reference. A rigorous test system is used to ensure completeness. 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

LIST OF FIGURES

Figure Page

LIST OF TABLES

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" - C. Berge Berge [1989]

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. $x.enque(y)$ 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 following statements are assumed to be familiar notation and self explanatory:

Algorithm 1 Factorial(N)

function FACTORIAL(n) // N!, WITH
    $l \leftarrow \emptyset$ // initialize an empty list, type = list of int
    $q \leftarrow \emptyset$ // list used as a queue, type = list of (list of int)
    for all $i \in 0, 1, .., n$ **do**
        $l[i] = i$ // ith index is set to the index value.
    $l.dequeue()$ // remove the zero at the front of the list.
    $q.push(l)$ // add the list of numbers to a list
    return $MultiplyList(q)$

function MULTIPLYLIST(mul)
    $returnValue \leftarrow 1$ // Multiplicative identity is 1, type = int.
    while $mul.size() > 0$ **do** // all the lists
        $sublist \leftarrow mul.pop()$ // extract the next sublist
        for all $v, i \in q$ **do** // for each element in the sub list
            $returnValue \leftarrow returnValue * v$
    return $returnValue$ // return the list of list multiply

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.

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 2 OdometerToHyperedge

function OtoE(U, o)
  $e \leftarrow \emptyset$
  $size \leftarrow U.nodes.size()$
  for all $\{n, i\} \in o$ **do**
    // where -2 % 7 = 5 as mod.
    $e[i] \leftarrow U.nodes[n\%size]$ // convert an integer number to index.
  return e

Algorithm 3 HyperedgeToOdometer

function EtoO(U, e)
  $o \leftarrow \emptyset$
  for all $\{v_e, i_e\} \in e$ **do** // use hashmap of v to i
    for all $\{v_n, i_n\} \in U.nodes$ **do** // to reduce $O(n^2)$ to $O(n)$
      if $v_e = v_n$ **then**
        $o[i_e] \leftarrow i_n$ // lookup index and save as integer.
  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 Berge [1989]

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.

$$\forall e, e' \in E | e \nsubseteq e' \wedge e' \nsubseteq e$$

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 4 IsMinimalTransversal

```
function IsMinimalTransversal(o, list_of_o)
    if DoesAHitAll(o, list_of_o) = false then
        return false
    for all {o_n, i_n} ∈ GenerateNMinusOne(o) do
        if DoesAHitAll(o_n, list_of_o) then
            return false
    return true
```

All minimal transversals

There are $2^{|V|}$ possible combination sets that can be derived from the hypergraph $H = (V, E)$. Thus there are $2^{|V|}$ transversals that need to be enumerated. Potentially Thus every piecewise combination of this hypergraphs hyperedges is a valid transversal, thus the maximal upper bound on the total number of minimal transversals of all simple hypergraphs is $O(2^{|V|})$ total transversals. Thus scalable algorithms must use polynomial space storage and exponential time to enumerate the potentially exponential number of traversals.

CHAPTER III

CHAPTER 3 HEADER

Chapter 3 starts here and goes on.

CHAPTER IV

NAIVE SOLUTION

Chapter 4 starts here and goes on.

CHAPTER V

BRANCH AND BOUND

Chapter 5 starts here and goes on.

CHAPTER VI

ITERATIVE PSEUDO-POLYNOMIAL SPACE

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 5 IntersectTransversalWithEdge

```
function IntersectTransversalWithEdge(list_of_transversals, edge)
    return_value ← ∅ // IHGResult.
    new_alpha ← edge // copy incoming edge.
    for all {g_t, i_t} ∈ list_of_transversals do
        intersect = Intersection(g_t, edge)
        new_alpha ← Minus(new_alpha, interset)
        if intersect.size() = 0 then
            return_value.Alphas.push(g_t)
        else
            if intersect.size() = g_t.size() then
                return_value.Betas.push(g_t)
            else
                Gamma ← ∅ // Gamma type.
                Gamma.XMinusY = Minus(g_t, edge)
                Gamma.XIntersectY = interset
                Gamma.YMinusX = Minus(edge, g_t)
                return_value.Gammas.push(gamma)
    return_value.new_alpha = new_alpha
    return return_value
```

IS Appropriate

Generate Next Depth

Algorithm 6 IsAppropriate

function $IsAppropriate(HSF, edge)$
    $list\_of\_new_{t}raversals \leftarrow \emptyset$
    for all $\{o, i\} \in HSF.Transversals$ **do**
        $gv \leftarrow o$
        for all $\{n, i\} \in HSF.Transversals$ **do**
            if $DoesACoverB(n, gv) = true$ **then**
                $gv \leftarrow Minus(gv, n)$
        if $gv.size() > 0$ **then**
            $list\_of\_new_{t}raversals.push(gv)$
    if $DoesAnyHitA(list\_of\_new_{t}raversals, edge) = false$ **then**
        return $false$
    return $true$

Depth First N-Way Tree Control

Algorithm 7 GenerateNextDepth

```
function GenerateNextDepth(HSF, edge)
    new_frame ← ∅ // hypergraph stack frame
    return_value ← ∅ // list of hypergraph stack frames.
    result ← IntersectTransversalWithEdge(HSF.Transversals, edge)
    if result.Gammas.size() == 0 then
        new_frame.Transversals = HSF.Transversals
        new_frame.Negations = HSF.Negations
        if result.Betas.size() > 0 then
            for all {b, i} ∈ result.Beta do
                new_frame.push(b)
        else
            new_frame.push(edge)
        if IsAppropriate(new_frame, edge) then
            return_value.push(new_frame)
    else
        for all list_of_bool ∈ Gen2expNtruefalse(result.Gammas.size()) do
            new_frame.Transversals ← result.Alphas
            new_frame.Negations ← HSF.Negations
            for all {tf, j} ∈ list_of_bool do
                gamma ← result.Gammas[j]
                if tf[j] = false then
                    new_frame.Transversals.push(gamma.XMinusY)
                    new_frame.Negations.push(gamma.XIntersectY)
                else
                    new_frame.Transversals.push(gamma.XIntersectY)
            if IsAllTrue(tf) = true then
                if result.new_alpha.size() > 0 then
                    new_frame.Transversals.push(result.new_alpha)
            else
                for all beta ∈ result.Betas do
                    new_frame.Transverals.push(beta)
                if IsAllFalse(tf) = true then
                    for all gamma ∈ result.Gammas do
                        new_frame.Negations.push(gamma.XMinusY)
            if IsAppropriate(new_frame, edge) = true then
                returnValue.push(new_frame)
                new_frame ← ∅
    return return_value
```

Algorithm 8 HypergraphTransversals

```
function HypergraphTransversals(H, CallbackFunc)
    edge_count ← H.E.size()
    control_stack ← list(edge_count) // list of stacks pre-sized.
    HSF ← ∅ // current hypergraph stack frame
    HSF.Transversals.push(edge)
    control ← 0 // depth control variable.
    control_stack[control].push(HSF) // load the process.
    while control ≥ 0 do
        if control_stack[control].size() = 0 then
            control ← control − 1
        else
            frame ← control_stack[control].pop()
            if control = edge_count − 1 then
                CallbackFunc(frame.Transversals) // min transversal reached.
            else
                control ← control + 1
                next_edge ← H.E[control]
                children ← GenerateNextDepth(frame, next_edge)
                for all {c, i} ∈ children do
                    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 9 GenHypergraphs

```
function GenHypergraphs(Nodes, CallbackFunc)
    V = Nodes // new hyperedge for a hypergraph of all the nodes.
    E = Nodes // single hyperedge of all the nodes.
    H = (V, E) // hypergraph with the above.
    CurrentQueue.push(H)
    WorkQueue.push(CurrentQueue)
    while !WorkQueue.empty() do
        CurrentQueue ← WorkQueue.pop()
        H ← CurrentQueue.pop()
        if !CurrentQueue.empty() then
            WorkQueue.push(CurrentQueue)
        CallbackFunc(H) // process generated hypergraph
        CurrentQueue = GenerateHypergraphChildren(H)
        if !CurrentQueue.empty() then
            WorkQueue.push(CurrentQueue)
function GenerateHypergraphChildren(H)
    Children ← ∅
    for all edge ∈ H.E do
        for all {edges, i} ∈ GenerateNMinusOne(edge) do
            new_edges ← H.E \ edge) // every edge but the new ones
            for all {e_n, i_n} ∈ edges do
                new_edges.push(e_n) // add new broken down edges.
            Children.push(Hypergraph(H.V, new_edges))
        return Children
```

APPENDIX A

APPENDIX A:

Appendix 1 texts go here

APPENDIX B

APPENDIX 1 TITLE

Appendix 1 texts go here

APPENDIX C

ALGORITHMS REFERENCED IN PAPER

Algorithm 10 Union

```
function Union(A, B)
    returnValue ← ∅
    for all {n, i} ∈ A do
        if !returnValue.contains(n) then
            returnValue.push(n)
    for all {n, i} ∈ B do
        if !returnValue.contains(n) then
            returnValue.push(n)
    return returnValue
```

Algorithm 11 Intersection

```
function Intersection(A, B)
    returnValue ← ∅
    for all {n_A, i_A} ∈ A do
        for all {n_B, i_B} ∈ B do
            if n_A = n_B then
                returnValue.push(n_A)
    return returnValue
```

Algorithm 12 Minus

```
function Minus(A, B)
    returnValue ← ∅
    for all {n_A, i_A} ∈ A do
        add ← true
        for all {n_B, i_B} ∈ B do
            if n_A = n_B then
                add ← false
        if add = true then
            returnValue.push(n_A)
    return returnValue
```

Algorithm 13 StrictEqual

```
function StrictEqual(A, B)
    for all {n_A, i_A} ∈ A do
        for all {n_B, i_B} ∈ B do
            if n_A! = n_B then
                return false
    return true
```

Algorithm 14 SetEqual

```
function SetEqual(A, B)
    A ← Sort(A);
    B ← Sort(B);
    return StrictEqual(A, B)
```

Algorithm 15 DoesACoverB

```
function DoesACoverB(A, B)
    for all {n_A, i_A} ∈ A do
        found ← false
        for all {n_B, i_B} ∈ B do
            if n_A = n_B then
                found ← true
        if found = false then
            return false
    return true
```

Algorithm 16 DoesACoverBorBCoverA

function $DoesACoverBorBCoverA(A, B)$
  if $DoesACoverB(A, B) = true$ **then**
    return $true$
  if $DoesACoverB(B, A) = true$ **then**
    return $true$
  return $false$

Algorithm 17 DoesAHitB

function $DoesAHitB(A, B)$
  for all $\{n_A, i_A\} \in A$ **do**
    for all $\{n_B, i_B\} \in B$ **do**
      if $n_A = n_B$ **then**
        return $true$
  return $false$

Algorithm 18 DoesAHitAll

function $DoesAHitAll(A, list\_of\_o)$
  for all $\{o, i\} \in list\_of\_o$ **do**
    if $DoesAHitB(A, o) = false$ **then**
      return $false$
  return $true$

Algorithm 19 DoesAnyHitA

function $DoesAnyHitA(list\_of\_o, A)$
  for all $\{o, i\} \in list\_of\_o$ **do**
    if $DoesAHitB(o, A) = true$ **then**
      return $true$
  return $false$

Algorithm 20 GenerateNMinusOne

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

Algorithm 21 Gen2expNtruefalse

```
function Gen2expNtruefalse(n)
    returnValue ← ∅ // list of (list of true—false)
    max = 1 << n // max is 2n̂ bit shifted.
    for all i ∈ 0..max do
        add ← ∅ // list of true—false
        counter ← 1
        while counter < max do
            if counter&i = counter then
                add.push(true)
            else
                add.push(false)
            counter ← counter << 1 // bit shift left.
        returnValue.push(add)
    return returnValue // N odometers, each with one item removed.
```

REFERENCES CITED

Claude Berge. *Hypergraphs*. North-Holland Mathematical Library, 1989.

Phillip P. Fuchs. *Permutation Odometers*. www.quickperm.org/odometers.php, 2016.