ALGORITHM FOR ENUMERATING HYPERGRAPH TRANSVERSALS

by

ROSCOE CASITA

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

September 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 Master of Science degree in the Department of Computer and Information Science by:

Boyana Norris	Chair
Chris Wilson	Core Member

and

Sara D. Hodges	Interim Vice Provost and Dean of the Graduate School

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

Degree awarded September 2017

THESIS ABSTRACT

Roscoe Casita

Master of Science

Department of Computer and Information Science

September 2017

Title: Algorithm for Enumerating Hypergraph Transversals

This paper introduces the hypergraph transversal problem along with the following iterative solutions: naive, branch and bound, and dynamic exponential time (NC-D). Odometers are introduced along with the functions that manipulate them. The traditional definitions of hyperedge, hypergraph, etc., are redefined in terms of odometers and lists. All algorithms and functions necessary to implement the solution are presented along with techniques to validate and test the results. Lastly, parallelization advanced applications, and 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, University of Oregon, 2017
B.S. Software Engineering Technology, Oregon Institute of Technology, 2007
A.S. Computer Engineering Technology, Oregon Institute of Technology, 2007

AREAS OF SPECIAL INTEREST:

Hypergraphs and Machine Learning

PROFESSIONAL EXPERIENCE:

Software Engineer I-II-III, Datalogic Scanning INC., 2007-2016

ACKNOWLEDGEMENTS

The list of people to thank for the help in completing this publication fills my life with joy. My mother for inspiring a life long passion of learning, my father for demonstrating self discipline and logistics, my spouse for her love and strength, my advisor for teaching the art of investigation, my bosses for demanding tenacity and caution, and finally our children for actualizing patient adulthood upon us.

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

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

This paper assumes the reader is familiar with sets and lists and the differences between them, specifically that lists can have repeated values and every value has an index number. Additionally, a list can be treated as a stack and queue via the push/pop and insert/remove operations. The appendix defines the additional functions needed to induce set behavior from odometers.

This paper introduces all terms first: odometer, hyperedge, hypergraph, transversal, generalized variable and compact minimal transversal. This paper extends and explains the following definitions and the relationships in later sections: An *odometer* is a list of numbers. A *hyperedge* is a list of nodes. A *hypergraph* is a list of *hyperedge*s and a list of nodes. A *transversal* is a list of nodes that hit every *hyperedge* in a *hypergraph.* A *generalized variable* (GV) is an *odometer.* Lastly, a *compact minimal transversal* (CMT) is a list of *odometer*s.

Hypergraphs are a generalization of graphs that allow edges to contain more than two nodes. All graphs are hypergraphs; thus, all graph problems should have a related hypergraph problem. Restating a problem formally when the edge count is greater than two can be difficult and sometimes there is no obvious translation.

Finding all minimal traversals of a hypergraph is comparable to the finding of all vertex cover sets of a graph, hypergraphs specifically. This problem has been shown to be equivalent to the NP-Complete problem space and is a significant and worthy problem in computation, especially in AI. Eiter [1991], Eiter and Gottlob [1995], Reiter [1987], De Kleer and Williams [1987].

The complete minimal solution space to both winning and losing games of Connect-4 is used as a test dataset in this paper. A theoretical AI player with access to these results could use them to guide perfect play.

A simple real world example of the problem is derived from Facebook; find a set of people who know every other user. Finding a single "hitting set" is known to be polynomial while finding "all hitting sets" is exponential. A previous recursive solution by Kavvadias and Stavropoulos [2005] inspired the work in this paper. The focus of this paper is a new algorithm that outputs a polynomial-polynomial space encoded representation of the entire exponential space solution.

The rest of this paper is organized as follows: **Odometers** (Algorithm II) and **Hypergraphs** (Algorithm III) are introduced followed by defining the problem in terms these new terms. Three iterative solutions that solve the problem are examined: naive, branch and bound, and dynamic. This paper then extends the dynamic iterative solution further with a series of optimizations. The authors ensure correctness with generative testing and validation. Future research directions focus on parallel possibilities, additional optimizations, and solution representation as output.

CHAPTER II

ODOMETERS

The term “odometer” currently has only one source that is not a published paper: Fuchs [2016]. An odometer is defined as a “list of numbers” in this paper and the implementation uses the C++ `std::vector<int>` type. After defining the odometer and applying some restrictions, traditional set behavior can be induced. Additional functions are introduced that operate on unrestricted odometers which are used for enumeration and state machine control.

– Odometer, noun. an instrument for measuring distance traveled.

– Odometer, portmanteau. “Ordered Meters”, a list of measurements.

– Odometer, computer science. a list of numbers.

Introduction

Functions can reason about an odometer structure independently of a hypergraph. As a list of numbers, an odometer instance is equivalent to a full Turing machine *tape* where every number represents one symbol on a tape. The *meaning* of an odometer is dependant upon the interpreting Turing machine (function).

Definition 2.1.1. Let an odometer o be a list of integers n and indices $i = \{n, i\}$. The i^{th} integer of an odometer can be written $n_i = o[i]$. Integers n can be repeated, they are distinguished via their index. Indices 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 randomly accessible integers in every odometer. Thus for every N there are are N^{∞} distinct odometer instances that can be created.

Inducing Structure

For this paper, odometers need to have similar behavior to mathematical sets. The following restrictions must be satisfied in order to ensure that the set functions behave correctly: a set may not have repeated value and the values are sorted from least to greatest by index. These simple restrictions induce set behavior from odometers without further modification. These restrictions are not placed upon Odometers, but rather only need to be enforced when set behavior is required.

$$\forall\{n, i\} \in o, \forall\{n', i'\} \in o | n \neq n'$$
$$\forall\{n, i\} \in o, \forall\{n', i'\} \in o | (n < n' \land i < i') \lor (n = n' \land i = i') \lor (n > n' \land i > i')$$

The following common set functions are defined in terms of odometers in appendix C: **Union** (Algorithm 25), **Intersection** (Algorithm 27), **Minus** (Algorithm 28), **StrictEqual** (Algorithm 29), **SetEqual** (Algorithm 30).

Hitting Odometers

The traditional hitting set calculations are now defined in terms of odometers. The **DoesAHitB** (Algorithm 1) answers the question: “does odometer A *hit* odometer B”. If A contains a number that is equal to a number in B, then A *hits* B. The **DoesAHitAll** (Algorithm 2) answers the question “does odometer A *hit* all of the odometers in a list” by leveraging **DoesAHitB** (Algorithm 1).

Algorithm 1 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 2 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 $true$
```

Minimal Hitting Odometers

The previous functions will return true for odometers which include vertices above and beyond the minimal number. Consider the trivial transversal: all vertices in the hypergraph. A *minimal* hitting odometer is defined as one where the removal of any vertex of an odometer causes **DoesAHitAll** (Algorithm 2) to return false.

Consider that each odometer must be constructed with the correct vertex removed and then tested. Calling **GenerateOdometerMinusIndex** (Algorithm 3) repeatedly will enumerate all odometers with the appropriate vertex removed.

Algorithm 3 GenerateOdometerMinusIndex

```
1: function $GenerateOdometerMinusIndex(O, index)$
2:     $returnValue \leftarrow \emptyset$
3:     for all $\{o, i\} \in O$ do
4:         if $i \neq index$ then
5:             $returnValue.push(o)$
6:     return $returnValue$
```

Leveraging all of the previous definitions **IsMinimalHittingOdometer** (Algorithm 4) determines if an odometer is a *minimal* hitting odometer for a given list of odometers. **TransversalsByNaive** (Algorithm 10) and **TransversalsByBranchAndBound** (Algorithm 11) leverage **IsMinimalHittingOdometer** (Algorithm 4).

Algorithm 4 IsMinimalHittingOdometer

```
function $IsMinimalHittingOdometer(A, list\_of\_o)$
    if $DoesAHitAll(A, list\_of\_o)$ then
        for all $\{a, i\} \in A$ do
            $test \leftarrow GenerateOdometerMinusIndex(A, i)$
            if $DoesAHitAll(test, list\_of\_o)$ then
                return $false$
    else
        return $false$
    return $true$
```

N-Dimensional Combination Counter

The **GenerateCombinationCounters** (Algorithm 5) along with **IncrementCombinationCounter** (Algorithm 6) are used for enumeration of all combinations of odometer indices. Notice this is not all combination of the values in an odometer, but rather the combinations of the indices of the values. The number of combinations for a list of odometer indexes is exponential. The number of combinations in a list of odometers is the product of each odometer size. Thus, a list of 10 odometers each with two values will result in 2^{10} unique iterated combination indexes. Counter and dimension odometers are NOT treated like sets in these routines.

Algorithm 5 GenerateCombinationCounters

```
1: function $GenerateCombinationCounters(enumerate, counter, dimensions)$
2:     for all $\{o, i\} \in enumerate$ do          ▷ for each odometer in the list
3:         $counter.push(0)$                        ▷ Initialize dimension index to 0
4:         $dimensions.push(o.size())$              ▷ Size of this dimension
```

GenerateCombinationCounters (Algorithm 5) must initialize the counter and dimensions variables to the first combination before **IncrementCombinationCounter** (Algorithm 6) is called. The function will increment the counter correctly to the next combination if there is one. Additionally the algorithm returns true if there are more combinations to enumerate and false otherwise.

Algorithm 6 IncrementCombinationCounter

```
1: function $IncrementCombinationCounter(counter, dimensions)$
2:     $control \leftarrow 0$ // stack index
3:     while $control < counter.size()$ do
4:         if $counter[control] + 1 < dimensions[control]$ then
5:             $counter[control] \leftarrow counter[control] + 1$
6:             return $true$
7:         else
8:             $counter[control] \leftarrow 0$
9:             $control \leftarrow control + 1$
10:    return $false$
```

The **IncrementCombinationCounter** (Algorithm 6) walks through the entire N-dimensional combination space in a single linear sweep. The memory use is proportional to N and enumerating every item requires exponential (in N) time. Combination counters can now be created, stored, retrieved, and iterated by using an explicit stack in memory.

CHAPTER III

HYPERGRAPHS

Hypergraphs are a recent mathematical discovery which model complex combination and permutation structures. The number of potential edges in a *normal* hypergraph is 2^N. 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 hyperedges. There is no ordering and no repeated hyperedges or vertices. Abstracting from a *normal* to an *unrestricted* hypergraph allows the edges to grow unbounded N^∞. There are only two algorithms presented in this paper that reason about *unrestricted* hypergraphs, and then restrictions are put in place for the rest of the algorithms.

Unrestricted Hypergraphs

Definition 3.1.1. Let a hyperedge e be a list of vertices: $e = \{v, i\}$. The i^{th} vertex of e can be written $v_i = e[i]$. Vertices v can be repeated, as they are distinguished via their index. Indices 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 3.1.2. 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 possible given a hypergraph: a hyperedge can be constructed from an odometer, and an odometer can be constructed from a hyperedge. Thus, every instance of a hyperedge 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 this paper hyperedges will be interchangeable with odometers given a hypergraph and the functions defined.

Algorithm 7 OdometerToHyperedge

```
1: function OtoE($U, o$)
2:     $e \leftarrow \emptyset$                                  ▷ e is a hyperedge
3:     $size \leftarrow U.nodes.size()$                          ▷ size is an integer
4:     for all $\{n, i\} \in o$ do
5:         $e[i] \leftarrow U.nodes[n \mod size]$                ▷ convert an integer number to index
6:     return $e$
```

Algorithm 8 HyperedgeToOdometer

```
1: function EtoO($U, e$)
2:     $o \leftarrow \emptyset$                                  ▷ o is an odometer
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 $o$
```

Notice that these functions provide liner time access to all permutations, combinations, and even repeated patterns. Hence, reasoning about a hyperedge is equivalent to reasoning about its corresponding odometer, and vice versa.

These routines allow access to every possible finite list composed of the vertexes. Converting from a list of vertices to a list of indices and back again is linear in the size of the list. Now odometers can be used in place of hyperedges and odometers can be constructed from a hyperedge given a specific hypergraph.

Normal Hypergraphs

The following restrictions are imposed to get the expected behavior out of a *normal* hypergraph given the unrestricted definitions.

Definition 3.2.1. Let a *normal* hypergraph be $H = (V, E)$ where V is a list of vertexes $\{v, i\}$, and E is a list of hyperedges $\{e, i\}$ where each hyperedge e is a sublist of V.

No hyperedge contains a duplicated vertex. Every vertex in all hyperedges is contained in the hypergraph vertices. There are no duplicate hyperedges. The maximal size of a hyperedge is the size of all hypergraph vertices. Every vertex exists in at least one hyperedge. There are no duplicate vertexes. These English statements are reiterated in formal logic notation to ensure exactness.

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

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

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

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

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

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

Simple Hypergraphs

Definition 3.3.1. Let a *simple* hypergraph be $H = (V, E)$ as with a *normal* hypergraph with the additional restriction that no hyperedge fully contains any other hyperedge.

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

CHAPTER IV

HYPERGRAPH TRANSVERSAL PROBLEM

The hypergraph transversal problem is the restating of NP-Complete problem in terms of hypergraphs, hyperedges, and their transversals. A simple graph version of this problem is: “Find all possible ways to split Facebook into two groups, such that the first group is friends with everyone in the second group.”. Consider a complex question: “Given a set of chemical reactions, find all sets of chemicals, such that at least one chemical from all the reactions is present”.

Transversal of a Hypergraph

Let the transversal of a hypergraph $Tr(H)$ be an odometer that contains at least one vertex from every hyperedge in the hypergraph. Notice that this has the property **DoesAHitAll** (Algorithm 2) where A is the transversal and $list\_of\_o$ is the list of hyperedges from the hypergraph.

Minimal Transversal of a Hypergraph

Let the minimal transversal of a hypergraph be a transversal such that the removal of any vertex from the transversal would invalidate the transversal property. **DoesAHitAll** (Algorithm 2) only validates the transversal property. **IsMinimalHittingOdometer** (Algorithm 4) determines the minimal transversal property, where A is the minimal transversal, and $list\_of\_o$ is the list of hyperedges from the hypergraph. The result is extended to $\{-1, 0, +1\}$ where -1 means that the transversal needs additional vertices, 0 means the transversal is minimal, and +1 means that there is a minimal transversal that is contained inside of this transversal.

Algorithm 9 IsMinimalTransversal

```
1: function IsMinimalTransversal($O, list\_of\_o$)
2:     if $DoesAHitAll(O, list\_of\_o) = false$ then
3:         return $-1$
4:     for all $\{o, i\} \in O$ do
5:         $test \leftarrow GenerateOdometerMinusIndex(O, i)$
6:         if $DoesAHitAll(test, list\_of\_o)$ then
7:             return 1
8:     return 0
```

CHAPTER V

NAIVE SOLUTION

The **TransversalsByNaive** (Algorithm 10) function enumerates all potential transversals of a hypergraph using a single odometer. The control routine increments the index values in the odometer properly to generate all combinations of indexes. Each odometer combination is tested to see if it is a minimal transversal. Notice that this is an exhaustive search that does not consider the results of **IsMinimalTransversal** (Algorithm 9). The results of this function were used to validate the other routines. As the number of nodes in the hypergraph grows, this function grows exponentially in time.

Algorithm 10 TransversalsByNaive

```
function $NaiveAllPotentialTransversals(H, CallbackFunc)$
    $count \leftarrow H.E.size()$
    $o \leftarrow \emptyset$                                        ▷ odometer
    $o.push(0)$
    while $o.size() > 0$ do
        if $IsMinimalTransversal(o, H.E) = 0$ then
            $CallbackFunc(o, OtoE(H, o))$
        $next \leftarrow o[o.size() - 1] + 1$
        if $next < count$ then
            $o.push(next)$
        else
            $o.pop()$
            if $o.size() > 0$ then
                $o[o.size() - 1] \leftarrow o[o.size() - 1] + 1$
```

CHAPTER VI

BRANCH AND BOUND

The “branch and bound” solution reduces the number of iterated transversals by considering the results of **IsMinimalTransversal** (Algorithm 9). If the odometer is not yet a minimal transversal, add another index. Otherwise, the odometer is or contains a minimal transversal. If possible, increment the top index. Otherwise, the top index needs to be removed and the next index needs to be incremented.

Algorithm 11 TransversalsByBranchAndBound

```
function $TransversalsByBranchAndBound(H, CallbackFunc)$
    $count \leftarrow H.E.size()$
    $o \leftarrow \emptyset$
    $o.push(0)$
    while $o.size() > 0$ do
        switch $IsMinimalTransversal(o, H.E)$ do
            case $0$
                $CallbackFunc(o, OtoE(H, o))$            ▷ Fall through to next case
            case $1$
                if $o[o.size() - 1] < count - 1$ then
                    $o[o.size() - 1] \leftarrow o[o.size() - 1] + 1$
                else
                    $o.pop()$
                    if $o.size() > 0$ then
                        $o[o.size() - 1] \leftarrow o[o.size() - 1] + 1$
                $break$
            case $-1$
                $next \leftarrow o[o.size() - 1] + 1$
                if $next < count$ then
                    $o.push(next)$
                else
                    $o.pop()$
                    if $o.size() > 0$ then
                        $o[o.size() - 1] \leftarrow o[o.size() - 1] + 1$
                $break$
```

The **TransversalsByBranchAndBound** (Algorithm 11) is an augmented form of **GenerateCombinationCounters** (Algorithm 5) combined with **IncrementCombinationCounter** (Algorithm 6). The control stack is manipulated based on the results of **IsMinimalTransversal** (Algorithm 9). If the currently generated combination odometer is not a hitting odometer, then another node needs to be added, and the control counter is incremented. If the odometer is a minimal hitting set, then invoke the callback function and deliver the minimal transversal. If the odometer hits at all, then remove the current node and the add next node if possible.

The performance of this algorithm on broad shallow exploration trees is exceptional for certain cases, outperforming $\leq O(N)$. On hard data sets with deep search trees the performance breaks as expected $O(2^N)$. Theoretically the compact routine sits entirely in the processor cache as noted by the performance on the random hypergraph test. The **TransversalsByBranchAndBound** (Algorithm 11) is a simplification of the MTminor algorithm, which traverses both negative and positive search spaces simultaneously. Hébert et al. [2007] Dong and Li [2005]

CHAPTER VII

DYNAMIC SOLUTION

The dynamic solution to the NP-Complete problem uses a list of polynomial encodings to represent the entire exponential encoded space. The algorithm will update the individual representations and generate all new potential transversals. The new potential transversals are validated against the previous transversals and only appropriates ones are used to update the new representation.

Definition 7.0.1. Let a generalized variable GV be an odometer of indices in the original hypergraph. All hyperedges E (odometer) are generalized variables GV (odometer) and can be interchanged directly.

Each generalized variable GV represents a choice or selection that must be made from the current hyperedge. Selecting *any* value in a generalized variable will satisfy a minimal hitting set property.

Definition 7.0.2. Let a compact minimal transversal CMT be a list of generalized variables GV. A CMT represents a group of transversals. Selecting one value from each GV will hit all of the associated hyperedges. A list of CMTs is used to store the representation of all transversals for a hypergraph.

Generalized Variable Type

The main procedure **TransversalsByCMT** (Algorithm 13) depends on **IntersectCMTWithEdge** (Algorithm 12) to derive the next solution representation given the new hyperedge. Given a CMT and a new hyperedge e', the *type* of each GV in a CMT can be determined in relation to the hyperedge

e'. The type is used to generate all of the child minimal transversals in the main procedure.

Definition 7.1.1. Let the type of GV be α if $Intersection(GV, e') = \emptyset$. Let $Alphas$ be a list of GV of type α.

Definition 7.1.2. Let the type of GV be β if $Intersection(GV, e') = GV$. Let $Betas$ be a list of GV of type β.

Definition 7.1.3. Let the type of GV be γ if GV partially covers e'. Let $Gamma$ be $(XMinusY, XIntersectY, YMinusX)$, where $XMinusY$ is $Minus(GV, e')$, $XIntersectY$ is $Intersection(GV, e')$, and $YMinusX$ is $Minus(e', GV)$. Let $Gammas$ be a list of $Gamma$.

Definition 7.1.4. Let $IHGResult$ be $(Alphas, Betas, Gammas, new\_alpha, CMTResult)$ where $Alphas$, $Betas$, and $Gammas$ were previously defined, new_alpha is e' minus the union of all $Betas$ and $XIntersectY$. $CMTResult$ can be one of the following values $\{ContainsAtLeastOneBeta, ContainsOnlyAlphas, ContainsGammas\}$.

The function **IntersectCMTWithEdge** (Algorithm 12) determines the type of all GVs in a CMT in relation to a new hyperedge e. The collections of different types will be used to generate the minimal transversals that can be derived from the current CMT and the new hyperedge e.

Algorithm 12 IntersectCMTWithEdge

```
function $IntersectCMTWithEdge(CMT, e)$
    $return\_value \leftarrow \emptyset$                                         ▷ IHGResult
    $return\_value.new\_alpha \leftarrow e$                                 ▷ copy incoming edge
    for all $\{GV, i\} \in CMT$ do
        $intersect = Intersection(GV, e)$
        $return\_value.new\_alpha \leftarrow Minus(return\_value.new\_alpha, intersect)$
        if $intersect.size() = 0$ then                                   ▷ Alpha type
            $return\_value.Alphas.push(g)$
        else
            if $intersect.size() = g.size()$ then                        ▷ Beta type
                $return\_value.Betas.push(g)$
            else
                $gamma \leftarrow \emptyset$                                      ▷ Gamma type
                $gamma.XMinusY = Minus(g, edge)$
                $gamma.XIntersectY = interset$
                $gamma.YMinusX = Minus(edge, g)$
                $return\_value.Gammas.push(gamma)$
    if $return\_value.Betas.size() > 0$ then
        $return\_value.CMTResult \leftarrow ContainsAtLeastOneBeta$
    else
        if $return\_value.Gammas.size() > 0$ then
            $return\_value.CMTResult \leftarrow ContainsOnlyAlphas$
        else
            $return\_value.CMTResult \leftarrow ContainsGammas$
    return $return\_value$
```

Generating New Potential Transversals

The main procedure **TransversalsByCMT** (Algorithm 13) is a dynamic update routine. The local variables *old-stack* and *new-stack* contain all previous minimal transversals encodings and all the next, respectively. Each update will consider each of the previous CMT with regard to each new edge e'. The results of **IntersectCMTWithEdge** (Algorithm 12) are used to derive the generated child CMT. Each appropriate CMT is then added to the next representation. After all the edges are considered the final representation contains all of the minimal transversals of the hypergraph. Lastly, calling **ExtractTransversals** (Algorithm 18) for each CMT enumerates all of the transversals in the representation. The original proof with complete mathematical details is available in the previous works Kavvadias and Stavropoulos [2005]. Unfortunately, restructuring these proofs for this paper is beyond ability of the author at this time.

If there are any GV of type $Beta$ then the current CMT will always hit the new edge. All the transversals in the CMT will hit the new edge. Hence, we add the CMT to the next list of transversals.

If there are no $Gamma$, then all GV are of type $Alpha$. Each piecewise item in the new edge e' needs to be added to each of the transversals in the CMT and considered as a child CMT. Each new child CMT contains a single GV containing a single node from the edge. The newly generated child transversal now hits the new edge but it may contain a non-minimal transversal that needs to be removed. **ProcessNewCMT** (Algorithm 17) handles the removal of inappropriate transversals and the updates to the representation.

If there are any $Gamma$, then there are $2^{|Gamma|}$ possible combinations of intersecting and non-intersecting GV that need to be considered. There is the special outer section case that needs to be considered first when there are both intersecting parts and also new nodes to pick from. When picking all of the outer sections of the $Gamma$, for each new $Alpha$, we add a new child CMT. Next, initialize the counters and iterate all of the exponential combinations that hit the new edge e'. For each of the newly generated combinations, they can contain transversals that are non-minimal transversals. **ProcessNewCMT** (Algorithm 17) handles the removal of inappropriate transversals and updates the next representation.

ExtractTransversals (Algorithm 18) is leveraged to enumerate the entire CMT representation, passing each transversal to the visitor function. The mundane helper functions **GenNextAlpha** (Algorithm 14), **GenFirstGamma** (Algorithm 15), and **GenNextGamma** (Algorithm 16) are variations on **TransversalsByNaive** (Algorithm 10) as they update the case specific odometer.

Algorithm 13 TransversalsByCMT

```
 1: function $TransversalsByCMT(H, CallbackFunc)$
 2:     $frame \leftarrow \emptyset$                                   ▷ CMT
 3:     $old\_stack \leftarrow \emptyset$                              ▷ List of CMT
 4:     $new\_stack \leftarrow \emptyset$                              ▷ List of CMT
 5:     $frame.push(H.Edges[0])$             ▷ Extract the 0th hyperedge as a GV
 6:     $old\_stack.push(frame)$
 7:     for all $\{e, i\} \in H.Edges$ do                          ▷ Starts at 1
 8:         for all $\{frame, index\} \in old\_stack$ do
 9:             $result \leftarrow IntersectCMTWithEdge(frame, e)$
10:             $new\_cmt \leftarrow \emptyset, alpha \leftarrow \emptyset, gamma \leftarrow \emptyset$
11:             switch $result.CMTResult$ do
12:                 case $ContainsAtLeastOneBeta$
13:                     $CondenseMinimalTransversals(new\_stack, frame)$
14:                 case $ContainsOnlyAlphas$
15:                     $alpha.push(0)$
16:                     while $GenNextAlpha(result, alpha, new\_cmt, frame)$ do
17:                         $ProcessNewCMT(new\_cmt, old\_stack, new\_stack, index)$
18:                 case $ContainsGammas$
19:                     if $result.new\_alpha.size() > 0$ then
20:                         $alpha.push(0)$
21:                         while $GenFirstGamma(result, alpha, new\_cmt)$ do
22:                             $ProcessNewCMT(new\_cmt, old\_stack, new\_stack, index)$
23:                     for all $\{g, i\} \in result.Gammas$ do
24:                         $gamma.push(0)$
25:                     while $GenNextGamma(result, gamma, new\_cmt)$ do
26:                         $ProcessNewCMT(new\_cmt, old\_stack, new\_stack, index)$
27:         $old\_stack \leftarrow new\_stack$
28:         $new\_stack \leftarrow \emptyset$
29:     for all $\{cmt, i\} \in old\_stack$ do          ▷ Enumerate the transversals now.
30:         $ExtractTransversals(cmt, CallbackFunc, H)$
```

Iterative Generation

GenNextAlpha (Algorithm 14), **GenFirstGamma** (Algorithm 15), and **GenNextGamma** (Algorithm 16) are iterative helper algorithms which update odometers to enumerate and generate each child CMT. These routines rely upon initialization of the counter odometer correctly.

GenNextAlpha (Algorithm 14) will enumerate all the child CMT that must be generated when the previous CMT does not intersect in any way. A single odometer is used to select the correct vertex to add. If the current index value in the odometer is greater then the size of the odometer, then enumeration has finished. Otherwise, the current index value in the odometer is the index of the new vertex (index). Extract the vertex-index and add it to a new odometer. Copy the previous CMT and add the new odometer. The newly generated CMT will now contain an odometer that “hits” the new edge being considered.

Algorithm 14 GenNextAlpha

```
1: function $GenNextAlpha(result, gen, new\_cmt, old\_cmt)$
2:     if $gen[0] >= result.new\_alpha.size()$ then
3:         return $false$
4:     $new\_cmt \leftarrow old\_cmt$
5:     $o \leftarrow \emptyset$
6:     $o.push(result.new\_alpha[gen[0]])$
7:     $gen[0] \leftarrow gen[0] + 1$
8:     $new\_cmt.push(o)$
9:     return $true$
```

GenFirstGamma (Algorithm 15) is similar to **GenNextAlpha** (Algorithm 14) as the algorithm needs to perform the same iterative piecewise addition for the outer section. There are 2^N combinations of gamma odometers that must be generated. The outer section is a special case that must be handled separately. All non-intersecting parts of the odometers in a CMT are selected, each non-intersecting part of the new edge being considered is added during each iteration.

Algorithm 15 GenFirstGamma
```
function $GenFirstGamma(result, gen, new\_cmt)$
    if $gen[0] >= result.new\_alpha.size()$ then
        return $false$
    $new\_cmt \leftarrow result.Alphas$
    for all $\{g, i\} \in result.Gammas$ do
        $new\_cmt.push(g.XMinusY)$
    $o \leftarrow \emptyset$
    $o.push(result.new\_alpha[gen[0]])$
    $gen[0] \leftarrow gen[0] + 1$
    $new\_cmt.push(o)$
    return $true$
```

GenNextGamma (Algorithm 16) advances through the $2^N - 1$ combinations, generating each possible minimal intersection of GV that hits the new edge.

Algorithm 16 GenNextGamma
```
function $GenNextAlpha(result, gen, new\_cmt)$
    $sizes \leftarrow \emptyset$
    for all $\{g, i\} \in gen$ do
        $sizes.push(2)$
    if $false = IncrementCombinationCounter(gen, sizes)$ then
        return $false$
    $new\_cmt \leftarrow result.Alphas$
    for all $\{g, i\} \in gen$ do
        if $g = 0$ then
            $new\_cmt.push(result.Gammas[g].XMinusY)$
        else
            $new\_cmt.push(result.Gammas[g].XIntersectY)$
    return $true$
```

Processing New Transversals

All new *potential* minimal transversals are now properly generated by the preceding routines. Each of the new *potential* minimal transversals needs to be considered in order to see if the child should actually be added to the next set of real minimal transversals. If any new *potential* minimal transversal *hits* any previous minimal transversal then the newly generated child is not appropriate, as noted in previous work by Kavvadias and Stavropoulos [1999, 2005].

All transversals in a child CMT must be extracted and considered in order to determine if it hits a previous transversal. An interesting result is that not all *previous* transversals need be extracted from each CMT. A transversal that *hits* every GV in a CMT is not an appropriate transversal to be added. **ProcessNewCMT** (Algorithm 17) extracts all transversals from the CMT and if any hit a previous CMT they not added to the next solution list of CMT.

Algorithm 17 ProcessNewCMT

```
1: function $ProcessNewCMT(new\_cmt, old\_stack, new\_stack, index)$
2:     $sizes \leftarrow \emptyset, indices \leftarrow \emptyset$
3:     $GenerateCombinationCounters(new\_cmt, indices, sizes);$
4:     $done \leftarrow false$
5:     while $!done$ do
6:         $transversal\emptyset$
7:         $cmt \leftarrow \emptyset$
8:         for all $\{t, i\} \in new\_cmt$ do
9:             $value \leftarrow t[indices[i]]$
10:            $o \leftarrow \emptyset$
11:            $o.push(value)$
12:            $transversal.push(value)$
13:            $cmt.push(o)$
14:        $done \leftarrow IncrementCombinationCounter(indices, sizes)$
15:        $add\_t \leftarrow true$
16:        for all $\{ot, oi\} \in old\_stack$ do
17:            if $oi = index$ then
18:                $continue$
19:            if $DoesAHitAll(transversal, ot)$ then
20:                $add\_t \leftarrow false$
21:                $break$
22:        if $add\_t$ then
23:            $CondenseMinimalTransversals(new\_stack, cmt)$
```

Transversal Extraction

The **ExtractTransversals** (Algorithm 18) extracts all transversals from a CMT in exponential time. Each transversal is constructed via index combination counters over the CMT. **TransversalsByCMT** (Algorithm 13) leverages this function to extract the final transversals.

Algorithm 18 ExtractTransversals

```
1: function $ExtractTransversals(CMT, ProcessTransversal, H)$
2:     $sizes \leftarrow \emptyset$
3:     $indices \leftarrow \emptyset$
4:     $GenerateCombinationCounters(CMT, indices, sizes)$
5:     $done \leftarrow false$
6:     while $done = false$ do
7:         $o \leftarrow \emptyset$
8:         for all $\{t, i\} \in CMT$ do
9:             $value \leftarrow t[indices[i]]$
10:            $o.push(value)$
11:        $ProcessTransversal(o, OtoE(H, o))$
12:        $done \leftarrow IncrementCombinationCounter(indices, sizes)$
```

A CMT can be merged with another CMT if and only if the following holds: they both contain the same number of generalized variables and there is only one generalized variable in both transversals that are not equivalent. Two CMTs that are different by only one GV represent the same output transversals. The following function **MergeTransversal** (Algorithm 19) tests if two CMTs can be merged and does so if they can.

Algorithm 19 MergeTransversal

```
1: function $MergeTransversal(CMT_a, CMT_b, CMT_{result})$
2:     if $CMT_a.size()! = CMT_b.size()$ then
3:         return $false$
4:     $diff\_count \leftarrow 0, s \leftarrow 0$
5:     while $s < CMT_a.size()$ do
6:         $match \leftarrow false, t \leftarrow 0$
7:         while $t < CMT_b.size()$ do
8:             if $SetEqual(CMT_a[s], CMT_b[t])$ then
9:                 $CMT_{result}.push(CMT_b[t])$
10:                $CMT_b.erase(t)$
11:                $t \leftarrow t - 1$
12:                $match \leftarrow true$
13:                $break$
14:            $t \leftarrow t + 1$
15:        if $match$ then
16:            $CMT_a.erase(s)$
17:            $s \leftarrow s - 1$
18:        else
19:            if $diff\_count > 0$ then
20:                return $false$
21:            else
22:                $diff\_count \leftarrow diff\_count + 1$
23:        $s \leftarrow s + 1$
24:    $CMT_{result}.push(Union(CMT_a[0], CMT_b[0]))$          ▷ Only diff left
```

Now that two CMTs can be merged together, a list of CMT can be collapsed into a condensed form. Given a list of CMTs adding one new CMT can cause the entire list to collapse down to one CMT, as each newly merged CMT can potentially be merged with a CMT currently in the list. The run time for **CondenseMinimalTransversals** (Algorithm 20) is exponential yet the output is log linear sized. Each time a compaction happens the size of output gets smaller and compute time goes up.

Algorithm 20 CondenseMinimalTransversals

```
function $CondenseMinimalTransversals(list\_of\_CMT, CMT)$
    $holder \leftarrow CMT$
    $done \leftarrow false$
    while $!done$ do
        $compact \leftarrow \emptyset$
        $index \leftarrow \emptyset$
        $done \leftarrow true$
        for all $\{cmt, i\} \in list\_of\_CMT$ do
            $compact \leftarrow \emptyset$
            if $MergeTransversal(cmt, holder, compact)$ then
                $index \leftarrow i$
                $holder \leftarrow compact$
                $done \leftarrow false$
                $break$
        if $!done$ then
            $list\_of\_CMT.erase(index)$
        else
            $list\_of\_CMT.push(holder)$
```

Solution

This concludes the current form of the algorithm designated Norris-Casita-Dynamic (NC-D). The iterative solution to enumerating all minimal hypergraph traversals is now complete. Each child CMT has an exponential number of transversals that are extracted in **ProcessNewCMT** (Algorithm 17) and added to the next set of CMTs. There is a shortcut in **ProcessNewCMT** (Algorithm 17) via the call to **DoesAHitAll** (Algorithm 2) which eliminates the need to extract all transversals of the old CMTs.

As each transversal is constructed as a CMT and then added to the list of CMTs to be collapsed, the enumeration of all minimal transversals at each stage is guaranteed. The representation of all transversals as a list of CMT is both the boon and curse of this algorithm. Calls to **CondenseMinimalTransversals** (Algorithm 20) are exceedingly expensive because the adjustment to the entire encoding with regard to the new transversal can take exponential time.

The algorithm differs from the KS algorithm in that all solutions are computed before the first solution is enumerated. Additionally, the storage of all transversals as a list of CMT is a dynamic encoding of the solution which is updated at each iteration.

The KS algorithm uses a similar recursive solution where the call stack grows polynomial in the edge count. Thus, the algorithm should outperform KS when the edge count is high enough to offset the penalty for operating on the list of CMT with **CondenseMinimalTransversals** (Algorithm 20). The Dual-Matching test dataset demonstrates this exact result in the results section.

CHAPTER VIII

OPTIMIZATION

Chapter VII implements the core algorithms to iteratively generate successive solution representations. Each representation is updated as it encounters each new hyperedge. There are different trade offs and potential optimizations with respect to time and space that will be discussed. However, the authors assume that there are a class of problems where the additional changes will not be possible. The additional optimizations exploit the representation that has been built and paid for with time in **CondenseMinimalTransversals** (Algorithm 20). One example is the usage of **DoesAHitAll** (Algorithm 2) in **ProcessNewCMT** (Algorithm 17) to eliminate an exponential sub-step by leveraging the representation. Further polynomial constant exchanges are made for performance enhancements in NC-DO0, NC-DO1, NC-DO3. The only true optimization is NC-DO3: if the program is able to output the *representation* of the solution, instead of the solution.

NC-DO0

ProcessNewCMT (Algorithm 17) enumerates each of the transversals trying to eliminate the new potential transversal using the previous list of CMT. Calling **CondenseMinimalTransversals** (Algorithm 20) on the complete list of CMT for all child transversals is expensive in time. Removal of any transversal in a CMT will require a list of CMT to represent the newly reduced transversals. **ProcessNewCMT0** (Algorithm 23) maintains a local list of CMT that are dynamically updated as they encounter each of the previous CMT. **MergeCMTLists** (Algorithm 21) and **RemoveHittingTransversals**

(Algorithm 22) are now added to the solution. **ProcessNewCMT0** (Algorithm 23) replaces **ProcessNewCMT** (Algorithm 17). **MergeCMTLists** (Algorithm 21) condenses two lists of CMT into one list of CMT.

Algorithm 21 MergeCMTLists

```
1: function $MergeCMTLists(list\_of\_CMT_0, list\_of\_CMT_1)$
2:     for all $\{cmt_1, i\} \in list\_of\_CMT_1$ do
3:         $CondenseMinimalTransversals(list\_of\_CMT\_0, cmt_1)$
```

RemoveHittingTransversals (Algorithm 22) simply enumerates transversals from a CMT and removes transversals that hit a previous CMT. Now the list of transversals is dynamically updated until previous CMT have been considered.

Algorithm 22 RemoveHittingTransversals

```
1: function $RemoveHittingTransversals(new\_CMT, old\_CMT)$
2:     $results \leftarrow \emptyset$
3:     $sizes \leftarrow \emptyset, indices \leftarrow \emptyset$
4:     $GenerateCombinationCounters(new\_CMT, sizes, indices)$
5:     $intersect = Unionize(new\_CMT)$
6:     $tester = IntersectCMTwithOdometer(old\_CMT, intersect)$
7:     $done = false$
8:     while $!done$ do
9:         $tr \leftarrow \emptyset, cmt \leftarrow \emptyset$
10:        for all $\{o_i, i\} \in new\_CMT$ do
11:            $val = o[indices[i]]$
12:            $t \leftarrow \emptyset$
13:            $t.push(val)$
14:            $tr.push(val)$
15:            $cmt.push(t)$
16:        $done \leftarrow IncrementCombinationCounter(indices, sizes)$
17:        $intersect = Intersection(o_i, o)$
18:        if $DoesAHitAll(tr, tester) = false$ then
19:            $CondenseMinimalTransversals(results, cmt)$
20:    return $results$
```

ProcessNewCMT0 (Algorithm 23) now maintains the dynamic update to the representation of the valid transversals as each previous CMT is considered. If any transversal gets invalidated, then it is not added to the next update list.

Algorithm 23 ProcessNewCMT0

```
1: function $ProcessNewCMT0(new\_CMT, old\_CMT\_list, new\_CMT\_list, index)$
2:     $transversals \leftarrow \emptyset$
3:     $transversals.push(new\_cmt)$
4:     for all $\{old\_cmt, i\} \in old\_CMT\_list$ do
5:         if $i == index$ then
6:             $continue$
7:         $next\_transversals \leftarrow \emptyset$
8:         for all $\{new\_cmt, j\} \in transversals$ do
9:             $temp = RemoveHittingTransversals(new\_cmt, cmt)$
10:            $MergeCMTLists(next\_transversals, temp)$
11:        $transversals \leftarrow next\_transversals$
12:    $MergeCMTLists(new\_CMT\_list, transversals)$
```

The changes in NC-DO0 were iterate a list of transversals over each old CMT and validate that none of them hit the old CMT. The previous routine **ProcessNewCMT** (Algorithm 17) iterated each transversal testing it against each CMT.

NC-DO1

ProcessNewCMT1 (Algorithm 24) now replaces **ProcessNewCMT0** (Algorithm 23). Enumerating every transversal in the list is not necessary if it is not even possible for the generated transversal to hit the previous CMT. Notice that if the intersection either way does not add up to the size of the previous transversal, then it cannot hit any transversal contained in it.

Algorithm 24 ProcessNewCMT1

```
1: function $ProcessNewCMT1(new\_CMT, old\_CMT\_list, new\_CMT\_list, index)$
2:     $transversals \leftarrow \emptyset$
3:     $transversals.push(new\_cmt)$
4:     for all $\{old\_cmt, i\} \in old\_CMT\_list$ do
5:         if $i == index$ then
6:             $continue$
7:         $next\_transversals \leftarrow \emptyset$
8:         for all $\{new\_cmt, j\} \in transversals$ do
9:             $IHGnew = IntersectCMTWithEdge(new\_cmt, Unionize(old\_cmt))$
10:            $IHGold = IntersectCMTWithEdge(old\_cmt, Unionize(new\_cmt))$
11:            $newsize \leftarrow IHGnew.Betas.size() + IHGnew.Gammas.size()$
12:            $oldsize \leftarrow IHGold.Betas.size() + IHGold.Gammas.size()$
13:            if $newsize \geq oldsize \geq old\_cmt.size()$ then
14:                $temp = RemoveHittingTransversals(new\_cmt, cmt)$
15:                $MergeCMTLists(next\_transversals, temp)$
16:            else
17:                $CondenseMinimalTransversals(next\_transversals, new\_cmt)$
18:        $transversals \leftarrow next\_transversals$
19:    $MergeCMTLists(new\_CMT\_list, transversals)$
```

Notice that the results of **IntersectCMTWithEdge** (Algorithm 12) are only used for their sizes. A second enhancement in NC-DO2 is covered in the future works section.

NC-DO3

Consider the iterative call in **TransversalsByCMT** (Algorithm 13) to **ExtractTransversals** (Algorithm 18) that extracts all of the transversals in the list of CMT and iterates over them. The Matching hypergraph test data set generates an exponentially large number of solutions. The current data file formats supported by these programs cannot be generated if the number of transversals is large. The transversal of a matching hypergraph with 100 nodes would require some 2^{100} lines in the file. The known universe is estimated to contain between

2^{80} and 2^{100} atoms. It is a safe assumption that the current file format will be incapable of representing such a solution. On the other hand, the list of CMT containing the 2^{100} solutions can be directly written to disk without enumerating the solution.

If and only if the encoded representation of the solution is acceptable, then the results of NC-DO3 are applicable. The only optimization in NC-DO3 over NC-D01 is writing the list of CMT directly to file instead of writing the interpretation of them to file. This change requires a different call-back function in **TransversalsByCMT** (Algorithm 13) that accepts CMT directly.

CHAPTER IX

VALIDATION & RESULTS

The system used to test these algorithms was an Intel i7-7700k running at 4.5 Ghz, 64GB of DD4 RAM, and a 400GB Intel NVMe SSD hard drive with 2.5GB/s throughput. All algorithms where implemented in generic standard template library C++11. As the nature of this problem relies upon complete enumeration of all transversals regardless of memory usage, compute time alone is used as the metric of performance. Inefficient memory usage is reflected in longer execution time should the virtual memory system be used.

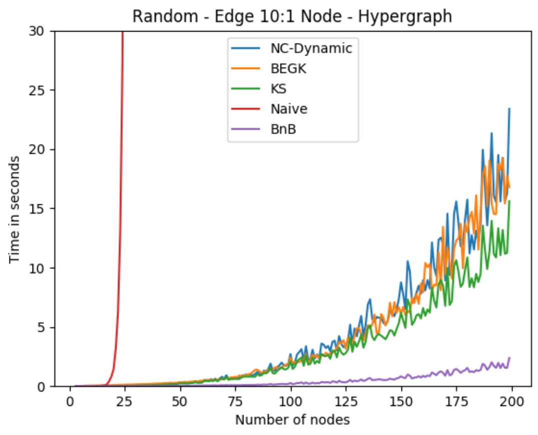
The following validation procedure was used. The naive algorithm results were collected as control, branch and bound was validated against the naive control, and last the NC-Dynamic solution was validated against naive as well.

The validation of a single hypergraph in this system consisted of iterating over the intermediate hypergraphs and testing each one. An intermediate hypergraph is constructed by taking the original hypergraph and iterating over one edge, two edges, etc, and constructing a hypergraph from each list of edges. This is an exponential slowdown in validation but also allows the isolation and repetition of core flaws during the development process. Specifically a "problem" instance could be identified by hypergraph number and intermediate number.

The iterative techniques to generate all hypergraphs where connected to the validation and testing system to prove correctness.The generative functions for hypergraphs are exponentially-exponential on the order of $N^{N-1^{N-2\cdots}}$. Hypergraphs of size 1,2,3,4,5 and 6 were validated. There were over $6 * 10^9$ hypergraphs of size 6 and took approximately three days on the test machine running as a single threaded process. Hypergraphs of size 7 - 12 were started and allowed to run which did not terminate before publication and no errors were detected.

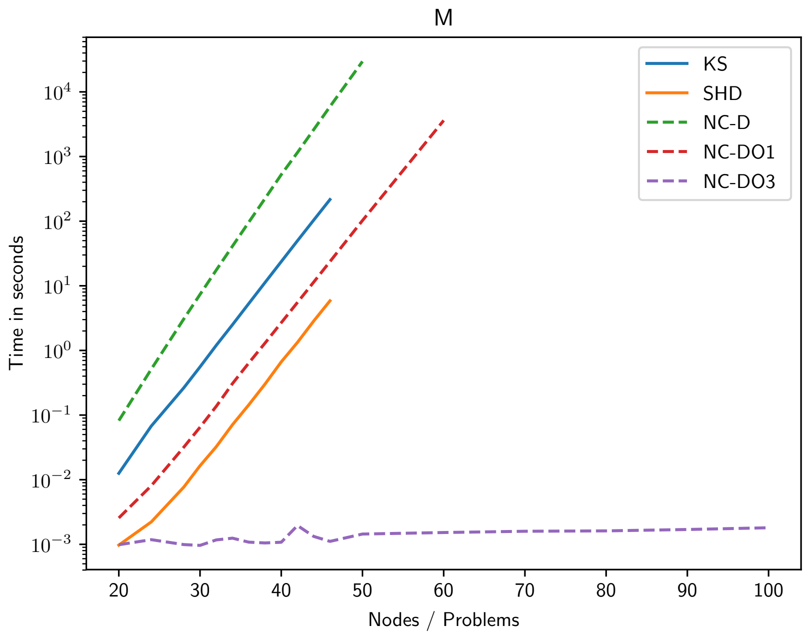
Results

Unexpected results were found when evaluating the algorithms. As noted previously Khachiyan et al. [2006], random hypergraphs with the same number of nodes and edges can vary dramatically in run time performance. The performance for similar systems varied from less than a second to hours. This is possibly indicative of how many possible hypergraphs there are. Given that there are $200^{199^{\cdots^{1}}}$ variations on a 200 node hypergraph, it is possible to select 2^{100} hypergraphs from this space and still not have any "similaraities" between them. The performance of algorithms depend directly upon the structure of the hypergraph. The first indication was the unexpected winner by magnitudes of performance: "Branch and Bound".

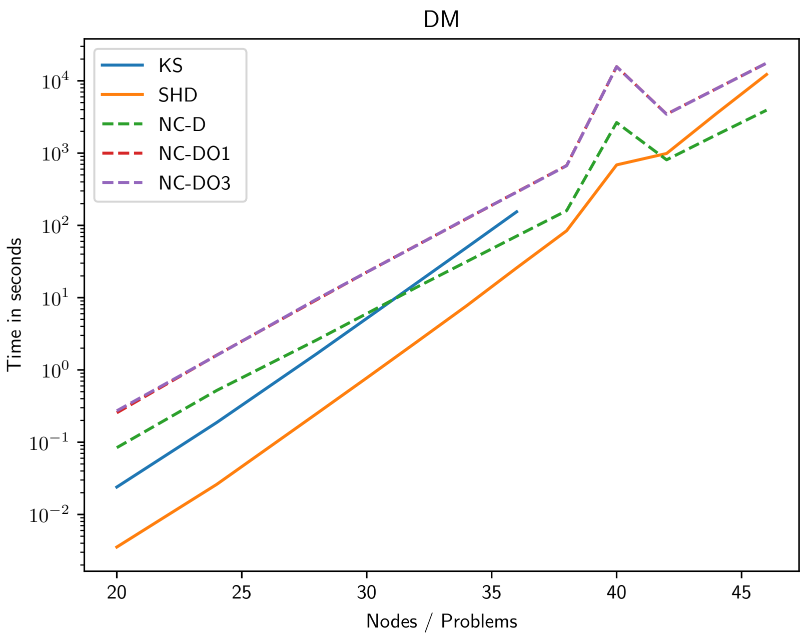


In this case, the authors believe that CPU caching with a small code implementation and an efficient list of integers representation simply outperformed for all of these "random" cases. After this was discovered, a database of hard hypergraph problems provided by Takeaki Uno was used for the remaining tests `http://research.nii.ac.jp/~uno/dualization.html`. The following categories where examined:

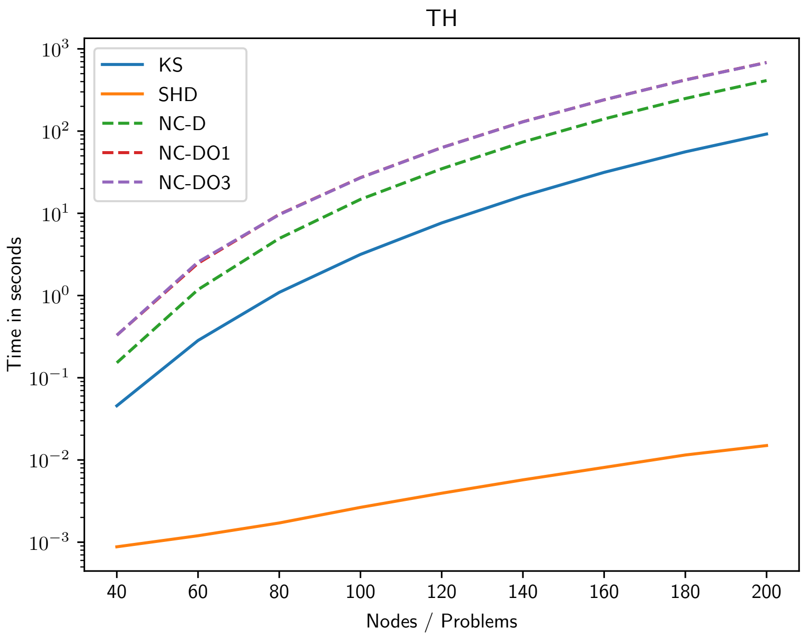
– Matching graph (M(N)): a hypergraph with n even nodes forming an induced matching with a low edges $n/2$ count but a high number of $2^{n/2}$ transversals.



– Dual Matching graph (DM(n)): a hypergraph that is the dual (solution) of the Matching graph with a high hyperedge $2^{n/2}$ count and a low number of $n/2$ transversals. $DM(n) = dual(M(n))$

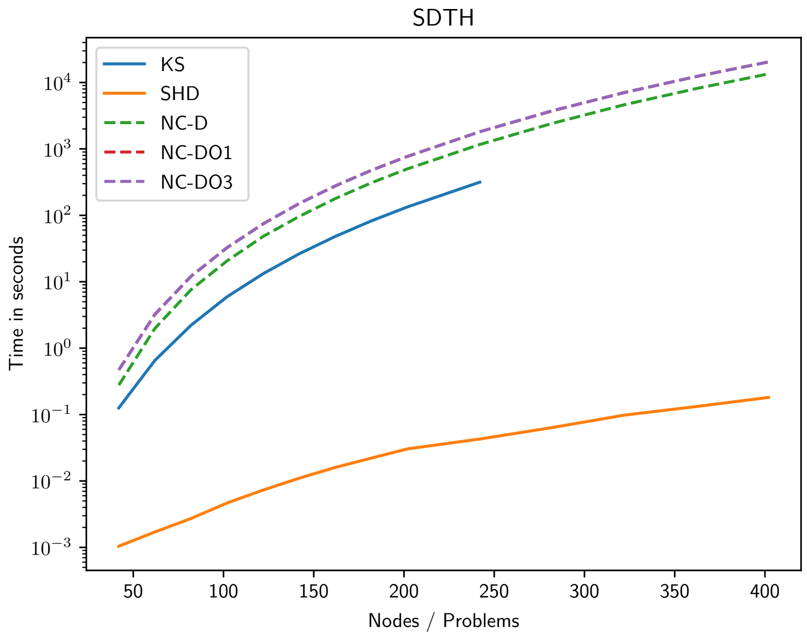


– Threshold graph (TH(N)): a hypergraph with n even nodes and hyperedge set $\{\{i, j\} : 1 \leq i \leq j \leq n, j \text{ is even}\}$ with a low hyperedges $n^2/4$ count and low number of $n/2 + 1$ transversals.

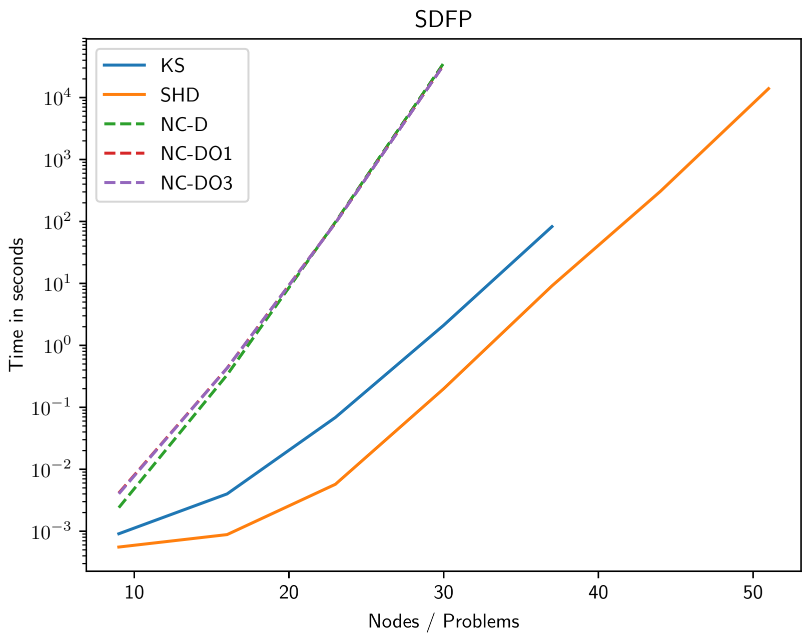


– Self-Dual Threshold graph (SDTH(n)): a self-dual hypergraph with n nodes obtained from $TH(n)$ and $dual(TH(n))$ as follows:

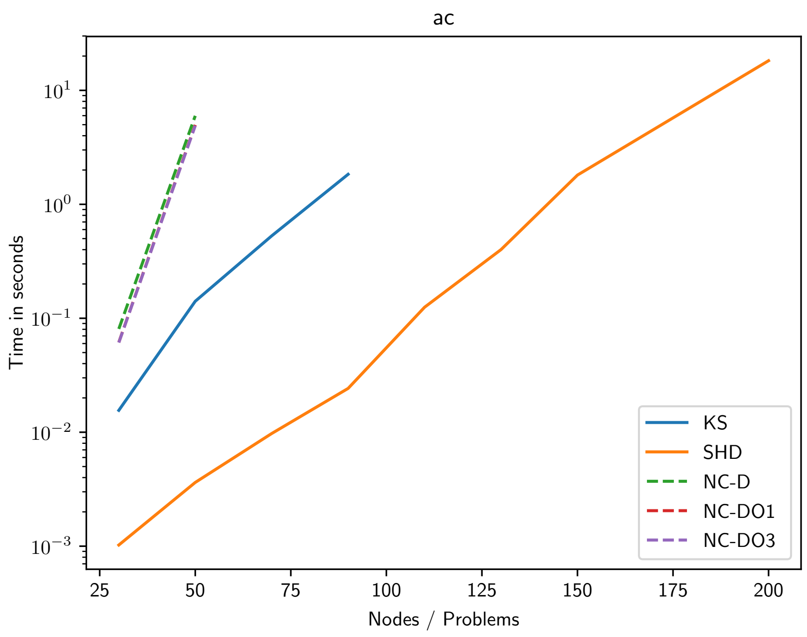
$$SDTH = \{\{n-1, n\}\} \cup \{\{n-1\} \cup E | E \in TH(n-2)\} \cup \{\{n\} \cup E | E \in dual(TH(n-2))\}$$



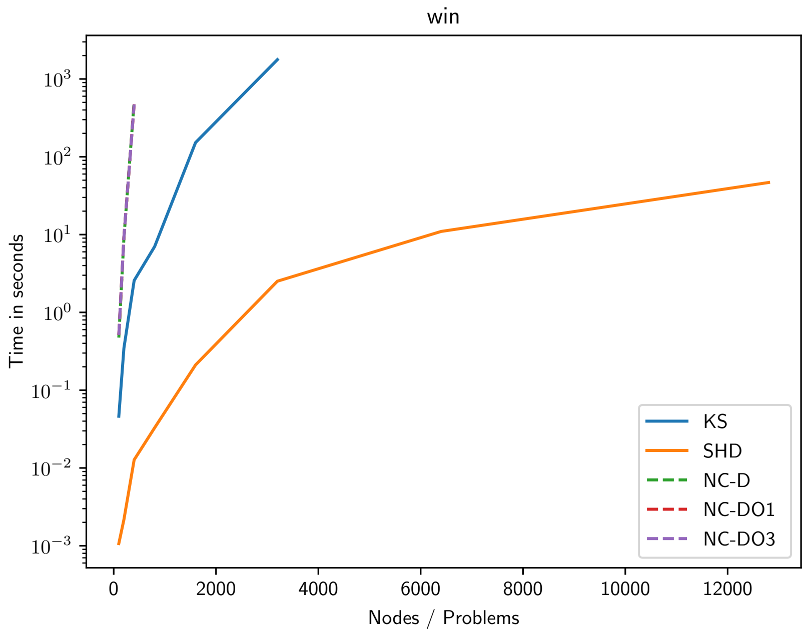
– Self-Dual Fano-Plane graph (SDFP(n)): a hypergraph with n nodes and $((k-2)^2/4 + k/2 + 1)$ hyperedges, where $k = (n-2)/7$. To construct it start with the hypergraph $H_0 = \{\{1,2,3\},\{1,5,6\},\{1,7,4\},\{2,4,5\},\{2,6,7\},\{3,4,6\},\{3,5,7\}\}$ that represent the set of lines in a Fano-plane and is self-dual. Set $H = H_1 \cup H_2 \cup ...H_k$ where $H_1, H_2, ..., H_k$ are k disjoint copies of H_0. The dual graph of H is the hypergraph of all 7^k unions obtained by taking one hyperedge from each of the k copies of H_0. Last, the SDFP hypergraph is a self-dual H as per the SDTH procedure.



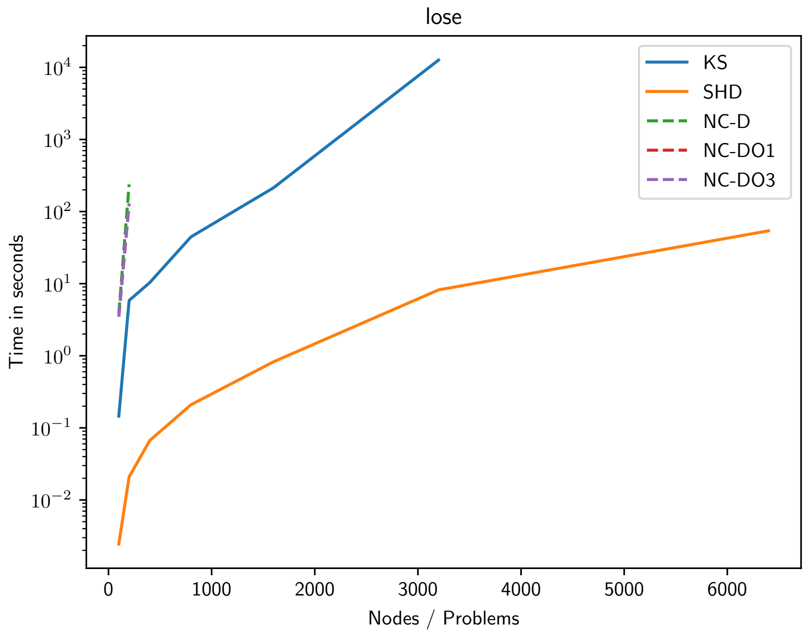
– accidents (ac): the complement of the set of maximal frequent item sets with support threshold t of the FIMI accident dataset.



– connect-4 (win): the t minimal winning board positions of the Connect Four game taken from the UCI machine learning repository.



– connect-4 (loss): the t minimal losing board positions of the Connect Four game taken from the UCI machine learning repository.



lose
KS
SHD
NC-D
NC-DO1
NC-DO3
Time in seconds
10^{4}
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
0
1000
2000
3000
4000
5000
6000
Nodes / Problems

CHAPTER X

CONCLUSIONS AND FUTURE DIRECTIONS

This paper has now demonstrated a dynamic polynomial-space, exponential-time (NC-D) iterative solution to the hypergraph transversal problem. The mechanics of **IntersectCMTWithEdge** (Algorithm 12) and **ProcessNewCMT** (Algorithm 17) were originally described in a recursive solution Khachiyan et al. [2006]. Similar to the previous recursive solution, all of the $2^{|Gamma|}$ cross-product children need to be enumerated and each child still needs to inspect the previous set of transversals to determine if it is *appropriate* to add it as a new transversal.

Improvements Over Previous Solutions

Improvements over previous solutions were in several areas. Odometers replaced both the set and bit vector representation in the KS implementation. Odometers are used to represent hyperedges, indices, generalized variables, transversals and as exponential enumeration variable counters. Lists of odometers represented compact minimal transversals and the processing of odometers in functions. Lists of lists of odometers represented the entire solution to a exponential problem where the polynomial number of polynomial encodings contains exactly all of the solutions. Each CMT takes exponential time to enumerate all of its transversals. The efficient storage is of particular interest as **ProcessNewCMT** (Algorithm 17) shortcuts an exponential number of comparisons by leveraging **DoesAHitAll** (Algorithm 2) to determine if a child is *appropriate* to add to the next generation of CMTs.

Both versions must consider the previous transversals minus its parent, but the recursive version did not have a mechanism to store the previous list of transversals. The algorithm had to generate each one by using recursion and a complex call stack to store the states. Consider the recursive solution that enumerates each of the previous transversals without storing them. This leads to exponential time to recompute previous transversals and memory use grows proportionally to the number of hyperedges.

Parallelization

There are multiple components of the NC-D algorithm that can be distributed for performance gains that scale to the number of computation units available. Consider that, like all dynamic algorithms, the next set of solutions must be computed from the current set of solutions. A three-staged pipeline could maximize the throughput of available compute nodes for this algorithm.

The first parallel stage is that each CMT has $2^{|Gamma|}$ potential children and each one can be computed given a hyperedge, in fact every CMT can generate all of its potential children CMT's independently. Consider N compute nodes each working with a specific CMT and hyperedge to fill a pipeline of children CMTs work items.

The transversal generation, testing, and compaction can happen in parallel in multiple places. The generation of each transversal from each CMT can be done independently. The testing of each new potential CMT against the previous CMT can be done independently. The condensing of the transversals into the next list of CMT can be done in parallel with memory locking mechanisms.

NC-DO2 Optimization

A further optimization on **ProcessNewCMT1** (Algorithm 24) is to replace the call to **RemoveHittingTransversals** (Algorithm 22) with a call to a new optimized version. Enumeration of all transversals of a CMT is not necessary. Enumeration of only the transversals that hit is required. This can be calculated and directly enumerated using the previous exponential extraction of all gamma types. Version NC-DO1 does the preliminary check to see if the enumeration of transversals is even required. This only eliminates the need to enumerate them if it is not possible to hit them.

The second optimization is to generate only the correct sub-transversals of a CMT that need to be considered. As previously noticed, only the intersections of the current CMT and the **Unionize** (Algorithm 26) of the previous CMT need be considered. Interestingly, the **IntersectCMTWithEdge** (Algorithm 12) function provides the exact segmented results that need to be considered. All the items in the Beta and Gamma lists can be iterated in the same exponential method. An iterator that winds through each generated CMT beta and gamma nodes constructing the sub-transversal that satisfies **DoesAHitAll** (Algorithm 2) needs to discard the generated item. As such, each non-discarded item needs to be collected in a local list of CMT to condense down the exponential non-hitting sets. The implementing of this methodology is left open for future work.

Advanced Algorithm Applications

The encoded NC-DO3 output can be analyzed without passing the encoding to an exponential decoder. Simple analysis such as deriving the number of transversals a node participates in is polynomial in the encoded output. Further analysis of these encodings can benefit clustering.

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]. A simple example is using the number of transversals a node participates in to identify clusters Bailey et al. [2003].

CondenseMinimalTransversals (Algorithm 20) is highly inefficient in both time and memory. A tree structure as an intermediate representation instead of CMT can represent the same transversals program using less memory. All new interactions of edges with tree representations will need to be derived in the same manner as the CMT.

APPENDIX

ALGORITHMS REFERENCED IN PAPER

Algorithm 25 Union

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

Algorithm 26 Unionize

```
1: function $Unionize(list\_of\_o)$
2:     $returnValue \leftarrow \emptyset$
3:     for all $\{o, i\} \in list\_of\_o$ do
4:         $returnValue \leftarrow Union(returnValue, o)$
5:     return $returnValue$
```

Algorithm 27 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 28 Minus

```
function $Minus(A, B)$
    $returnValue \leftarrow \emptyset$
    for all $\{n_A, i_A\} \in A$ do
        $add \leftarrow true$
        for all $\{n_B, i_B\} \in B$ do
            if $n_A = n_B$ then
                $add \leftarrow false$
        if $add = true$ then
            $returnValue.push(n_A)$
    return $returnValue$
```

Algorithm 29 StrictEqual

```
function $StrictEqual(A, B)$
    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 $false$
    return $true$
```

Algorithm 30 SetEqual

```
function $SetEqual(A, B)$
    $A \leftarrow Sort(A);$
    $B \leftarrow Sort(B);$
    return $StrictEqual(A, B)$
```

REFERENCES CITED

Thomas Eiter. *On transveral 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.

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.

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

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

Céline Hébert, Alain Bretto, and Bruno Crémilleux. A data mining formalization to improve hypergraph minimal transversal computation. *Fundamenta Informaticae*, 80(4):415–433, 2007.

Guozhu Dong and Jinyan Li. Mining border descriptions of emerging patterns from dataset pairs. *Knowledge and Information Systems*, 8(2):178–202, 2005.

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.

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.

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.