Parallel Hypergraph Transversals

Roscoe S. Casita

March 14, 2019

Abstract

This paper introduces a new efficient scalable parallel algorithm that computes all minimal transversals of a hypergraph. Finding all minimal transversals of a hypergraph is an important problem in data mining [1]. Previous work has focused primarily on efficient algorithms that run to completion. This paper extends an efficient algorithm to be parallel and scalable. Parallelism is achieved by duplicating an iterative state machine and splitting the workload. State of the art hypergraph transversal algorithms are compared and analyzed. Multiple hypergraph data sets are evaluated to demonstrate the various benefits and drawbacks of the various algorithms.

1 Introduction

Hypergraphs are capable of representing a significantly larger range of problems than traditional graphs; specifically, hyperedges connect N vertices. This generalization allows hypergraphs to model problems such as the set cover problem where each set is a hyperedge, databases where each row is a hyperedge and each discrete value in each column is a vertex, and Connect-4, where winning/ losing board positions, is a collection of hyperedges.

Hypergraph transversals are a cut through every hyperedge as each transversal contains at least one vertex from all hyperedges. The generation of all transversals is a significant problem as this equates to solving all possible ways to cut a hypergraph. Generating all transversals of a large sparse hypergraph is intractable as the number of traversals grows combinatorically.

An example of using all transversals is considered a database of genes from patients with a specific type of cancer, known to be genetic. Each transversal of the database is a set of genes that all patients have one or more of: the smaller the transversal (number of vertices), the fewer the genes that could be related to the type of cancer. This is a systematic way to find the combinations of genes related to a type of cancer. [2] This procedure can be done with other databases as well such as the accident, bms-webview, and win/lose data sets which are used in testing. The transversal will have different “meaning” depending on the information contained in the database.

State of the art hypergraph transversal algorithms are examined then a modification and extension of the current best algorithm are designed for parallelization. All algorithms are tested and examined on multiple data sets, including scaling performance of the new algorithm.

2 Definitions

The following definitions are necessary to discuss the related hypergraph transversal work.

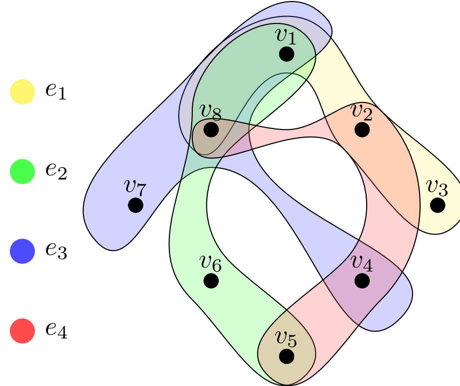


Figure 1: Example Hypergraph

Let $H = (V, E)$ be a hypergraph of vertices V and hyperedges E. Let V be a set of vertices such that $\forall v | v \in V$. Let E be a set of hyperedges such that $\forall e | e \in E$. Let each hyperedge e be a subset of vertices from V such that $\forall e \in E, \forall v \in e | v \in V$.

$$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

$$E = \{\{1, 2, 3, 8\}, \{1, 5, 6, 8\}, \{1, 4, 7, 8\}, \{2, 4, 5, 8\}\}$$

Lemma 2.1. *Let a transversal S be a subset of vertices from V such that there exists at least one vertex in every hyperedge: $\forall e \in E | \exists v \in S \wedge v \in e$.*

For every hyperedge, there is a vertex in both the hyperedge and the transversal. This is called the hitting property.

Lemma 2.2. *Let a minimal transversal S be as a transversal where the removal of any v from S will cause the hitting property to be invalidated.*

Checking to see if a transversal is a minimal transversal is a minimality check. Each vertex in S must be removed in turn and the hitting property invalidated. $t1 = \{1, 4, 6\}$ is minimal as removing any vertex invalidates the hitting property while $t2 = \{1, 2, 6\}$ is not minimal as either 6 or 1 can be removed and $t2$ still hits all hyperedges.

Previous work has shown that constructing all minimal transversals of a hypergraph is equivalent to the enumeration of all minimal covering sets, enumerating all minimal uncovering sets, constructing the dual of a hypergraph, circuit enumeration for an independent system, computing negative border from the positive border, and DNF to CNF transformation. [3] The problem is known to be NP-Hard, and there is still an open question on the existence of a polynomial time algorithm to compute the answer to these problems.

Lemma 2.3. *Let a hyperedge e be critical for a vertex v in a set S if the hyperedge contains only v from S and no other vertices from S. $\forall v' \in S \setminus v | v' \notin e$ and $v \in e$*

This is similar to the minimality check, except the check is to see if the removal of a specific v from S will invalidate the hitting property for a specific hyperedge. However, if a set s contains both $v \in e$ and $v' \in e$ for some $v \in s \wedge v' \in s$ then the hyperedge is not critical for any vertex in s.

Lemma 2.4. *Each vertex v in a minimal transversal S contains at least one critical hyperedge e.*

Proof. By contradiction: Assume S contains a v that has no critical hyperedge. If there are no hyperedges that are critical for v, then v can be removed from S. If S is a minimal transversal then $S \cap v$ cannot be a minimal transversal, as the transversal property would be invalidated by definition. Contradiction. □

Lemma 2.5. *A set S that hits all hyperedges and every vertex v in S has at least one critical hyperedge, then S is a minimal transversal.*

All hyperedges are hit by S, so the set is a hitting set. If any of the vertices $v \in S$ is removed from S, then the hyperedges critical for v are not hit. Thus the hitting property is invalidated for each v is removed from s and therefore s is a minimal transversal.

3 Related Work

The hypergraph transversal problem has been explored with several algorithms, each using different techniques. The six algorithms reviewed are BEGK, DL, BMR, KS, HBC, and SHD, none of which are parallel. Broadly they can be categorized into two categories: BEGK, DL, BMR, AND KS are constructive generation while HBC and SHD are hill climbing with pruning techniques.

The original algorithm to solve hypergraph transversals by Berge [4] starts with the set of all vertices: Check if the transversal is minimal, if it is output the transversal, otherwise remove each vertex in turn and call recursively. This suffers from over generation of minimal transversals and exponential minimality checks in the worst case. This algorithm exists on paper as a mathematical construct, and no implementations are evaluated.

The BEGK [5] algorithm constructs an initial $dual(H_0)$ by creating a set for each vertex of size 1. $dual(H)$ is iteratively solved from $dual(H_{-1})$ by adding each vertex to each set and checking if each new resulting set is a minimal transversal. Overgeneration is solved by checking against the other minimal transversals. In terms of time complexity this algorithm was proved to be the fastest, however, in practice, it is not. Parallelism is possible but not considered because the number of minimality checks, excessive checks for overgeneration matching, and maintaining all transversals in the worst case results in exponential memory usage.

The DL [2] algorithm constructs the dual by *mining* the borders of minimal transversals. An *emerging pattern* is a collection of sets used to generate a set of minimal transversals. The dual is then a collection of *emerging patterns* that generate all transversals. Overgeneration is solved by checking each *emerging pattern* against each other pattern and removing non-minimal transversals from the pattern. Memory usage is significantly reduced with the generative data structure, but the dual is still maintained in its entirety to check for overgeneration, meaning that memory usage is still exponential in the worst case.

The BMR [1] algorithm improves on DL by recursively partitioning the hypergraph before solving. First, the set of vertices is sorted by frequency, then for each vertex in the sorted set we remove each hyperedge that contains the vertex from the hypergraph and solve for the remaining set. The algorithm uses an arbitrary cut off point in recursion either choosing to partition or use

the DL algorithm to solve the remaining hypergraph. This results in improved performance but still suffers from exponential memory and time usage in the worst case. The technique of partitioning hyperedges on a vertex is used in the parallel algorithms along with updating the critical hyperedges.

The KS [6] algorithm is a recursive technique that considers "generalized variables" or subsets of V, which derive from the hyperedges and are used to generate the transversals. The KS algorithm uses a depth-first recursive search which generates a transversal from the current generalized variables then checks it against the other transversals of depth$_-$1 for invalidation. Overgeneration is solved by checking the previous $dual(H_-1)$ which may invalidate a new transversal. Memory usage does not grow exponentially: the current transversal is maintained, and the previous dual is generated on the fly. Computation time does grow with the dual, as all transversals need to check against the previous dual for validation. Parallelization was achieved with this algorithm, but it was also necessary to transform it to a breadth-first search with generative data structures to represent the "generalized variables", as a result, the memory and time became exponential and was abandoned. [7]

The HBC [8] algorithm starts with an initial dual and iteratively adds combinations of vertices to each solution, checking for minimality with the pruning criteria called the "Galois connection". The "Galois" property is defined as two operators: one for vertices, one for edges. The $f_h(E,V)$ operator is the set of vertices remaining after the removal of the unionization of E, $f_h(E,V) = V \setminus \bigcup E$. The $g_h(V,E)$ operator is the set of hyperedges remaining after removing V from each one, and then removing empty hyperedges, $g_h(E,V) = E \setminus (E \setminus V)$. This algorithm has the potential to be parallelized but suffers from exponential memory growth on dense hypergraphs with few minimal transversals. The critical property is $g_h(s,E) \setminus g_h(s \setminus v, E)$ or the set of hyperedges critical for the vertex v.

The SHD [3] hill climbing algorithm uses a recursive depth-first technique with global hypergraph updates. The algorithm defines two new properties $uncov(S,E)$ and $crit(v,S)$ as simplifications of the "Galois" properties. $uncov(S,E)$ is the set of edges remaining after removing each edge if it contains any vertex contained in S. $crit(v,S)$ is the set of hyperedges remaining after removing $S \setminus v$ from each hyperedge. Note that performing $\forall v \in S | crit(v,S)$ and checking to see if the result is the empty set is minimality checking. The algorithm maintains the partially evaluated global hypergraph state at each level of recursion, for both uncov and crit. Memory usage grows with the size of the hypergraph and time grows with the size of the dual. The concepts of *uncov* as a list of hyperedges not yet hit and *crit* as a list of partitions to store hyperedges hit by v was used. However, to parallelize the original algorithm the global hypergraph with the state had to be duplicated for each thread and was not feasible without reworking.

4 Iterative Algorithm

The author now introduces an iterative solution to enumerate all of the minimal transversals of a hypergraph. As there are several stacks of lists of composite items, each variable is name is now defined along with the associate type, see Figure 2 Each of the stacks grows and shrink together, when a new vertex is considered from the VStack then a new item is placed on the other stacks. Each depth of a stack has an associated vertex at the same depth in the VStack. A partition is a list of hyperedges hit by some vertex v

partition	A List of vertices
SM.Consider	List of vertices, candidates for VStack
SM.VStack	Stack of list of vertices, search tree
SM.CRecover	Stack of list of vertices, return to Consider
SM.Uncovered	List of hyperedges, not hit by S
SM.Critical	Stack of partition, each partition is a list of hyperedges
SM.PRecover	Stack of list of partition, return to Critical

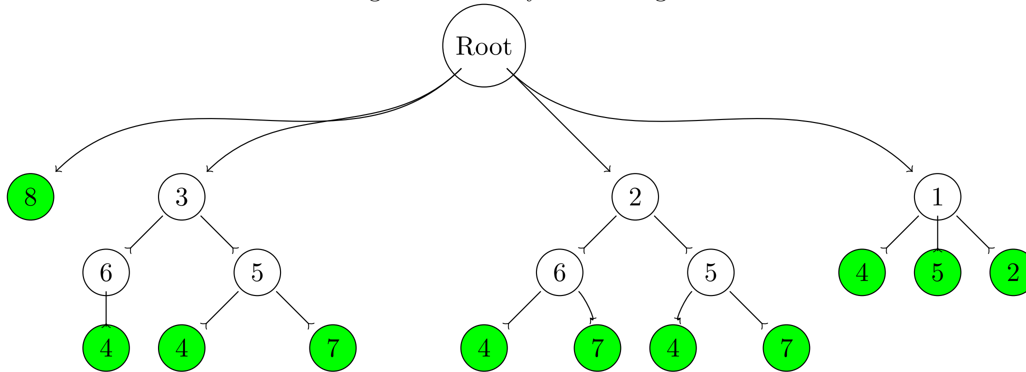
Figure 2: Stack Machine Variables

The main algorithm to step a state machine data structure a single step forward is now presented. The algorithm operates by transitioning the state machine from state to state, performing the composite operations on the state machine. Each iterative step in the function advances the state of the machine by doing one of the following: finish, output transversal, add vertex list to top of the VStack, pop a partition off Critical, or pop vertex list from VStack & pop a partition off Critical. A partitioning technique similar to BMR and SHD is used to construct smaller partitions for minimality checks. All partitions are with respect to one vertex, the critical vertex in a hyperedge.

Fundamentally each call to the function needs to advance the state machine so that the top of the VStack is either the empty list or a list of non-evaluated vertices, see Figure 3.

A depth-first stack of lists of vertices is used to explore the N-way combination search tree, Figure 4 The search tree VStack is used to explore combinations, using a list of vertices as a stack at each depth. Each of the vertices in each list/stack will become the current vertex in turn. When a vertex becomes the active vertex the combination of the top of each list/stack is used to make S.

Figure 4: N-Way Branching Search Tree



The hyperedges in Uncovered that contain the active vertex are transitioned to Critical. 4 *UpdateCriticalPartion* is called with respect to the active vertex, partitioning Uncovered into a partition for Critical and the remaining uncovered list. Additionally, each partition in Critical is examined, and if a hyperedge contains the active vertex, the hyperedge is moved from Critical to the associated PRecover, see Figure 4.

Once Uncovered and Critical have been updated for the active vertex, then validation checks that every partition still contains a hyperedge. If any partition in Critical drops to zero hyperedges then S is a invalid combination as there is at least one hyperedge that is not critical for the active

Algorithm 1 StepStateMachine

1: **function** $StepStateMachine(StateMachine\ sm, HyperGraph\ hg)$
2:     **if** $|VStack| == 0$ **then return** $Finished$
3:     **if** $|VStack.Top| == 0$ **then**
4:         $Consider \leftarrow Consider \cup CRecover.Top$
5:         $VStack.pop()$
6:         $CRecover.pop()$
7:         **if** $|VStack| > 0$ **then**
8:             $Consider \leftarrow Consider \cup VStack.Top.First$
9:             $PopVertexPartition(sm, hg)$
10:    **else**
11:         $v \leftarrow VStack.Top.First$
12:         $UpdateCriticalPartition(v, sm, hg)$
13:         **if** $Validate(sm, hg)$ **then**
14:             **if** $|Uncovered| == 0$ **then**
15:                 $S = GenSFromVStack(VStack)$
16:                 $ProcessMinimalHittingSet(S)$
17:                 $CRecover.Top \leftarrow CRecover.Top \cup v$
18:                 $PopVertexPartition(sm, hg)$
19:             **else**
20:                 $N = SelectVertices(sm, hg) \cap Consider$
21:                 $Consider \leftarrow Consider \setminus N$
22:                 $VStack.Add(N)$
23:                 $CRecover.Add(\emptyset)$
24:         **else**
25:             $CRecover.Top \leftarrow CRecover.Top \cup v$
26:             $PopVertexPartition(sm, hg)$
        return $Continue$

Figure 3: Algorithm to advance state machine

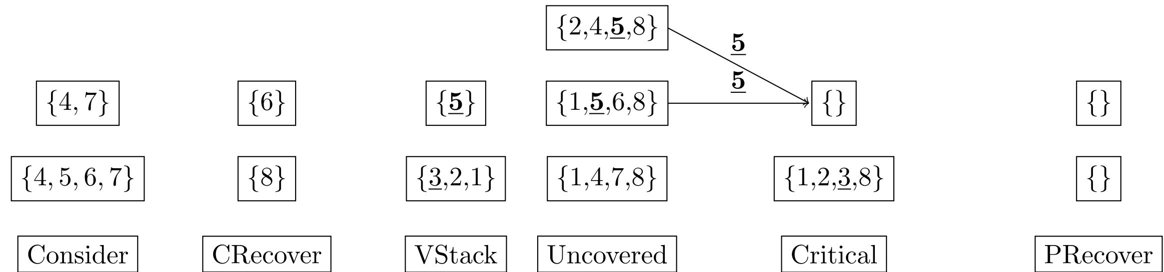


Figure 5: Partition Uncovered with vertex 5

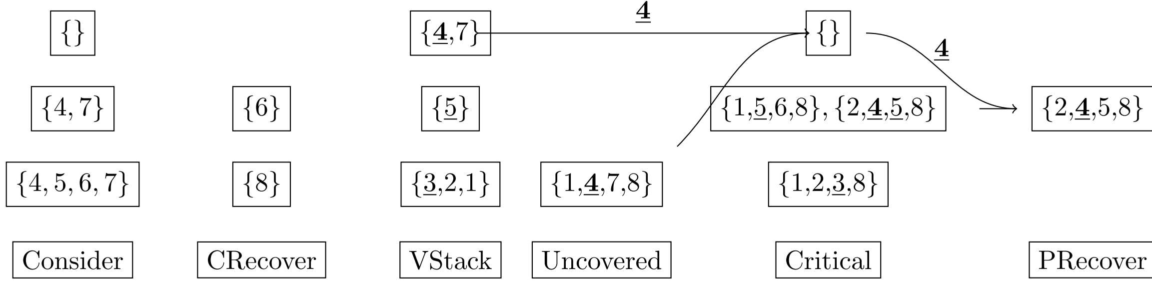


Figure 6: Partition Uncovered and Partition Critical

vertex v in S. If the active vertex is valid and the number of hyperedges in Uncovered is zero, then a minimal transversal from S has been reached and is output. If the active vertex is valid and number of hyperedges in Uncovered is greater then zero, a new list of vertices is selected and added to the search tree VStack.

When choosing a set of vertexes to add to the search tree: all of the hyperedges in Uncovered are examined and the hyperedge whose intersection with Consider is the smallest is selected, see Figure 9. By selecting a hyperedge and using its vertices, the search tree is guaranteed on the next iteration to hit at least the chosen hyperedge in Uncovered. The Consider list is used to prune the search tree when a list of vertices is selected, all of the vertices in the list are removed from Consider as they are now in the search tree. When removing the active vertex from the search tree, v is moved to either CRecover or Consider. The top partition in Critical is rejoined with uncovered. Any hyperedges in PRecover that are hit by v are rejoined with the associated Critical. When the current depth of the VStack is removed, then rejoin CRecover at that depth with Consider. All of this functionality is encompassed in PopVertexPartition, see Figure 8.

When the active vertex invalidates a partition, it should not be considered in further combinations with the remaining vertices. If S invalidates then $S \cup v$ also invalidates, so the vertex is moved to CRecover. When the active vertex makes a minimal transversal S, it should not be reconsidered as $S \cup v$ is non minimal/invalid, so the vertex is moved to CRecover. When the current depth in the search tree VStack is empty, then the associated CRecover is rejoined with Consider. Additionally, the active vertex from the previous depth has searched all of its combinations. Because the previous vertex was considered with more combinations of vertices, it should be moved to Consider so that it can be added to the search tree again.

Lemma 4.1. *Each hyperedge can only be critical for one vertex at a time.*

Proof. When a vertex is added to S and Uncovered is partitioned, the hyperedges that contain the new v are moved to Critical. Only critical hyperedges are stored in Critical, all hyperedges in Critical that are hit by v are moved to PRecover. Thus any hyperedge in Critical is critical for the associated vertex in S. □

Lemma 4.2. *Every vertex in a minimal transversal contains at least one critical hyperedge.*

Proof. Let a S be a set of vertices, where each v has a partition in Critical. For every vertex there is at least one hyperedge in Critical, if any Critical partition size drops to zero then the set S is non-minimal. □

Algorithm 2 UpdateCriticalPartition

```
1: function UpdateCriticalPartition(Vertex v, StateMachine sm, HyperGraph hg)
2:     r ← ∅
3:     for p ∈ Critical  do
4:         for he ∈ p do
5:             if v ∈ he then
6:                 r ← r ∪ he
7:                 p ← p ∩ he
8:         PRecover[p].Add({v, r})
9:     p ← ∅
10:    for he ∈ Uncovered do
11:        if v ∈ he then
12:            p ← p ∪ he
13:            Uncovered ← Uncovered ∩ he
14:    Critical.Add(p)
```

Figure 7: Update Critical and Uncovered

Algorithm 3 PopVertexPartition

```
1: function PopVertexPartition( StateMachine sm, HyperGraph hg)
2:     Uncovered ← Uncovered ∪ Critical.Top
3:     Critical.pop()
4:     for rp ∈ PRecover  do
5:         if v = rp.Top.v then
6:             Critical[rp] ← Critical[rp] ∪ rp.Top.r
7:             rp.pop()
8:     v ← VStack.Top.First
9:     VStack.Top ← VStack.Top ∩ v
```

Figure 8: Remove v from S and update state machine

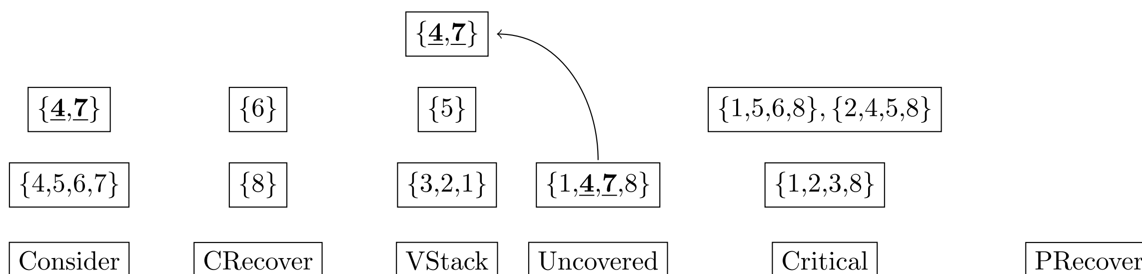


Figure 9: Select vertices to place on VStack

Lemma 4.3. *A set S that hits all hyperedges and for every vertex v in S has at least one critical hyperedge, then S is a minimal transversal.*

Proof. When the size of Uncovered drops to zero then all hyperedges have been hit by the set S, either into Critical or CRecover. Each vertex $v \in S$ has an associated Critical that is not empty, the hyperedges critical for $v \in S$. If any $v \in S$ is removed from S, then the hyperedges in Critical for that v will no longer be hit. Thus S hits all hyperedges, as they are no uncovered hyperedges, and S is minimal because if any v is removed, then some hyperedges will no longer be hit by S. $\square$

4.1 Depth First Search

The depth-first search (DFS) is a methodology of constructing the VStack, CRecover, and Consider. The DFS selects the smallest hyperedge remaining in Uncovered and takes the intersection of the edge with Consider, removing them from Consider and putting them on the VStack with an associated empty list in CRecover. 9 By selecting a hyperedge from Uncovered, it is guaranteed that the list of vertices will each hit that hyperedge in Uncovered. The list of vertices on the VStack is a subset of vertices from a specific hyperedge, thus guaranteed to hit the same hyperedge as they are considered in turn. Applied iteratively, this will keep selecting remaining hyperedges until all hyperedges are hit, and all minimal transversals are generated.

Consider shapes and prunes the search tree by only allowing vertices which are not already on the VStack to be considered. If a vertex is already on the VStack but not a current consideration, then the vertex will be considered later when it does become current. As vertices are removed from the VStack, they are placed either into the associated CRecover or Consider. When a vertex list is popped from the VStack, then the CRecover is popped and rejoined with Consider. All vertices transition from Consider to the VStack and back.

5 Parallelization

The parallelization of hypergraph transversals is a non-trivial: algorithms that are recursive need to be converted to iterative versions or have controlled task expansion. Algorithms that have exponential memory usage are not viable as the algorithm should run to completion given enough time and processing units. Recursive algorithms that use global variables that are updated upon entry and then unwound on exit require special attention as parallel execution corrupts the global data. All variables that are used in the algorithm, including the implied stack variables and global

data state, need to be explicitly added to the data structure. Each of the algorithms were considered for parallelization and multiple techniques were combined to create an stepping state machine algorithm that was then parallelized.

The state machine data structure has two list variables Uncovered and Consider, and four interrelated stacks VStack, CRecover and Critical, PRecover. Vertexes transition from Consider to the VStack then CRecover and back to Consider. Hyperedges transition from Uncovered to Critical and back additionally from Critical to PRecover and back. The state machine stepping function requires that VStack and CRecover be the same size, the same for Critical and PRecover, and that the number of partitions in Critical is one less then the number of vertex lists in VStack. The smallest representation is 1 state machine to output all of the minimal transversals.

Consider that a single call to StepStateMachine as the smallest unit of work. A full state machine representation with all of the stacks and variables is passed repeatedly to the step function until done. If a starting state machine is duplicated and both worked on simultaneously, then it would over generate, outputting each minimal transversal twice. However, if a state machine is duplicated and both machines modified so the VStack is split between them, then they produce only the set of minimal transversals.

To split a work item, the entire state machine is copied. One copy processes the active vertex and moves the rest to CRecover, the other moves the active vertex to CRecover and processes the rest of the vertices. The split operation ensures that the representations can be processed in parallel and each work item is only processed once. As a representation has an unknown depth and branches to explore in the search tree, its not possible to split the work evenly to load balance. On certain problems this degrades performance as the number of splits relative to the amount of work for each split is low. On other problems where there is a large amount of work per split, performance is improved.

The overhead for parallelization is that each time a work item is split, an entire copy of the representation must be performed. The size of the stack machine is dependent on the problem, which can be substantial. Additionally it is not possible to know how much work is left in the branch of the search tree, so the representation can be split and the new representation only takes one step before finishing.

Each thread runs the ParrallelThread algorithm, see Figure 10: retrieve and process a state machine, split the representation when necessary and queue the split work item. When a thread is waiting and a representation can be split, then a thread will split its work item and share the workload. An example of how threads operating on work items in parallel is shown in Figure 11.

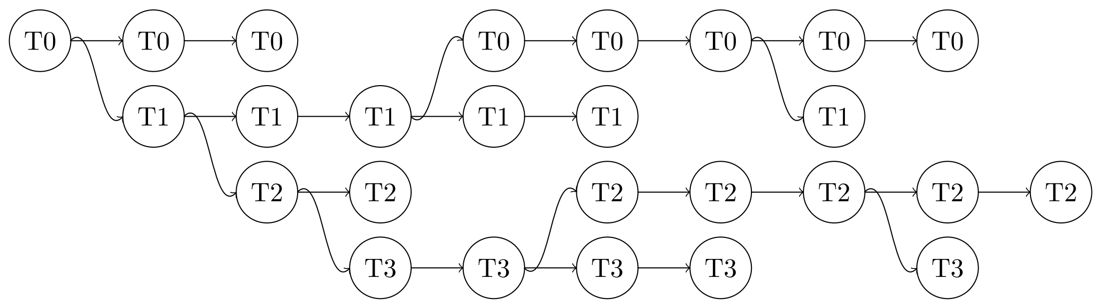


Figure 11: Parallel task splitting model

Algorithm 4 ParallelThread

```
1: function ParallelThread( WorkQueue wq, StateMachine sm, HyperGraph hg)
2:     global WorkersPaused ← ThreadCount
3:     Done ← False
4:     sm ← ∅
5:     while !Done do
6:         if sm = ∅ then
7:             if |WorkQueue| = 0 ∧ WorkersPaused = ThreadCount then
8:                 Done ← True
9:             sm ← WorkQueue.WaitItem()
10:            WorkersPaused − −
11:        else
12:            if CanSplitSM(sm) ∧ WorkersPaused > 0  then
13:                sm2 = SplitStateMachine(sm)
14:                wq.Add(sm2)
15:            else
16:            if StepStateMachine(sm, hg) = Finished then
17:                sm ← ∅
18:                WorkersPaused + +
```

Figure 10: Parallel thread to split state machine

5.1 Doubly-Linked Array-List

The doubly-linked array-list (DLAL) data structure is two arrays, a next and prev index. List-headers are a pair of head and tail indexes into the DLAL. Indexes can only appear in one list at a time, so each list-header contains a subset of indexes. Maintaining multiple list-headers allows for representing multiple subsets of the DLAL, each pointing at different head-tail indexes in the array. Removal of an item from one list must be paired with addition to another list to maintain correctness. All items must exist in a list and be moved from list to list, even if it requires another list-header to store items. The removal is $O(1)$, addition $O(1)$, and merging two lists $O(1)$.

A DLAL can be directly used as an index map, where each index in the DLAL arrays maps to an index in another object array. An additional element is added to the array-lists at the N+1 index which represents the tail of all lists. A list-header with head and tail values of N+1 represent the empty list. The use of a DLAL is a mechanism for representing multiple subsets of a set that only constructs headers to allow moving items from subset to subset.

Two DLALs are used in the algorithm data structure: one for hyperedges indexes and one for vertices indexes. Two list-header stacks are used: one for partitioned hyperedges and one for candidate vertices. Here is an example of 4 lists splitting the vertices of H into subsets: $LH1 = \{1, 2, 3, 8\}$,$LH2 = \{5, 6\}$,$LH3 = \{4, 7\}$,$LH4 = \{\emptyset\}$, see Figure 12. Indexing starts at 1 and maps directly to vertices for clarity in this image, implementation values starts at 0 and map to the index of a vertices.

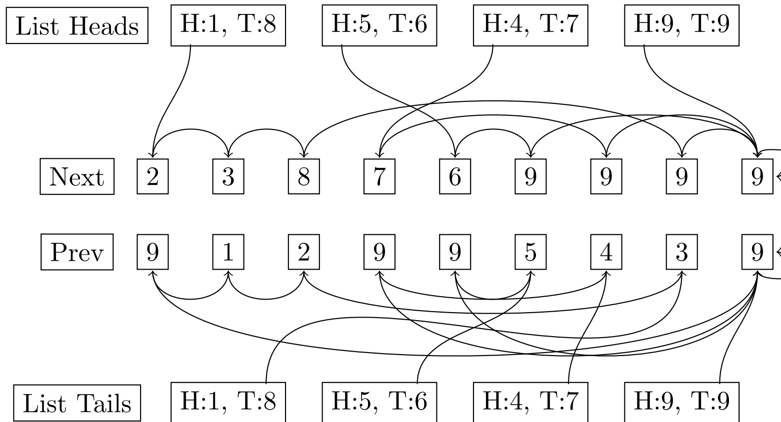


Figure 12: Doubly Linked Array List

6 Results

The various algorithms have different strengths and weaknesses that that depend upon the contents of the data set; therefore data sets that are categorically different were selected. Five mathematically generated hypergraphs and four machine learning data sets were used for testing.

There are three graphs for each dataset, time to process, memory usage, and scaling. Time to process is the amount of time taken by the algorithm to calculate all miminal transversals. Memory

usage shows the maximal memory used by each algorithm. Scaling compares SHD_D and SHD_R single threaded algorithms verse IP_Dx, where x is the number of threads from 1 to 80. Time is measured in seconds and log scaled. Hypergraph complexity is measured in either the number of hyperedges or the number of nodes. When hyperedges is used as the complexity measurement, then the axis is log scaled.

- Accidents AC: the accidents dataset is taken from the FIMI repository, where the number corresponds with the number of rows selected from the database. Frequent itemset or pattern mining is collecting the set of maximal frequent itemsets, where a maximal frequent itemset is not included in any other maximal frequent itemset, otherwise known as a transversal. The minimal infrequent itemset is the inverse or the dual of the maximal frequent itemset. A minimal infrequent itemset/transversal relates a set of conditions to every accident in the database. See Figure 13
- BMS-Webview-2 BMS2: the bms webview2 is a database of web sessions and the products viewed during the session, taken from the FIMI repository. The number corresponds to the number of rows selected from the database. Each transversal represents the minimal number of pages to visit such that every web session visited the page. See Figure 14
- Win/Lose: The win/lose data set is generated from the UCI Machine learning repository from the game Connect-4. Each hyperedge/row corresponds to a minimal winning/losing board position for the first player. The minimal transversal represents a set of moves which disturb the winning/losing plays of the first player. The first m rows correspond to the problem instance size. See Figures 15 and 16
- Matching Hypergraph $M(n)$: a hypergraph with n vertices, where n is even, and $n/2$ hyperedges forming perfect matching hyperedges, F_i is $2i-1, 2i$. There are very few hyperedges and an exponential number of minimal transversals $2^{n/2}$ See Figure 17
- Dual Matching hypergraph $DM(n)$: the dual of a matching hypergraph, $2^{n/2}$ hyperedges and $n/2$ minimal transversals.
- Threshold Hypergraph $TH(n)$: a hypergraph with n vertices, where n is even, and the hyperedge set $\{\{i,j\} : 1 \leq i \leq j \leq n, j \text{ is even }\}$. There are few hyperedges $n^2/4$ and a small number of minimal transversals $n/2+1$. See Figure 19
- Self-Dual Threshold Hypergraph $SDTH(n)$: a hypergraph with n vertices, where the hyperedge set is given as $\{\{n-1,n\}\} \cup \{\{n-1\} \cup E | E \in TH(n-2)\} \cup \{\{n\} \cup E | E \in dual(TH(n-2))\}$. $SDTH(N)$ is the dual of $SDTH(N)$; thus the number of transversals and hyperedges are the same, $(n-2)^2/4 + n/2 + 1$. See Figure 20
- Self-Dual Fano-Plane Hypergraph $SDFP(N)$: a hypergraph with n vertices an $(k-2)^2/4 + k/2 + 1$ hyperedges where $k = (n-2)/7$. Starting with the set of lines in Fano plane $H_0 = \{\{1,2,3\}, \{1,5,6\}, \{1,7,4\}, \{2,4,5\}, \{2,6,7\}, \{3,4,6\}, \{3,5,7\}\}$. Then set $H = H_1 \cup H_2 \cup \cdot \cup H_k$, where $H_1, H_2, ..., H_k$ are k disjoint copies of H_0. The dual of H is hypergraph of a 7_k unions obtained by taking one hyperedge from each of k copies of $H + 0(H_1, H_2, ..., H_k)$, this is $SDFP(N)$. The total number of hyperedges is $1 + 7k + 7k$. See Figure 21

All programs were either compiled with g++ 7.3 or provided native binaries. Tests were run on Ubuntu 18.04 with a dual Xeon Gold 6148 @ 2.40 GHz (80 logical cores total) and 384GB of memory. The only program able to take advantage of the multiple cores is the new Iterative-Parallel algorithm IP_Dx, the rest are single threaded applications. No single application outperforms on all problems as the dataset changes the structure of the problem; therefore each algorithms strengths and weaknesses are examined.

BEGK generally underperforms against all other algorithms tested. The time complexity of BEGK is the fastest from a strictly mathematical aspect, but the memory usage to store the dual and the time to access it leads to massive slowdown and program crashes on larger datasets. Notably, BERK only runs to completion in the allotted time for the Threadhold Hypergraph, see Figure 19. BEGK serves as a base reference point to evaluate the performance of other algorithms.

DL only outperforms the other algorithms when the search tree is narrow and deep, this corresponds to a few large transversals, see Figure 18. Mining the borders of the dual means that a representation of the dual is being maintained. As the complexity of the dual grows, the mined borders become numerous, resulting in many representations that need to be maintained. The performance drops off dramatically on all other data sets where the number of transversals is larger.

BMR, which is based on the DL algorithm, chooses to either partition the hypergraph on a vertex or use the DL algorithm. The extra cost to partition hinders on the mathematical data sets and improves performance on the real world data sets. The BMS data set in Figure 14 shows BMR outperforming DL when the extra cost of partitioning is counteracted by the complexity of the hypergraph.

KS is a recursive depth-first search that leverages the concept of generalized variables (subsets of hyperedges/vertices) to generate transversals which are then validated. KS re-generates the previous set of transversals to validate, so as the number of transversals grows, KS processing time grows exponentially. KS only runs to completion on the mathematical data sets, see Figures 17, 19, 20, 21. KS fails from recursive stack overflow on real world data sets when the complexity grows.

HBC/MTMiner stores the dual and iterates solutions using pruning and minimality checks. MTMiner only outperforms the other algorithms when there are a large number of small transversals, corresponding to a wide and shallow search tree, see Figure 14. MTMiner stores the dual at each level of the search tree, so as the depth of the search tree grows, MTM uses exponential memory and crashes. MTM only runs to completion on Figures 14 and 13.

SHD_D/SHD_R are variants of the same algorithm SHD. SHD_D constructs the search tree using the smallest hyperedge in uncovered while SHD_R constructs the search tree using the next unhit hyperedge in uncovered. SHD_D/R outperforms the previous algorithms on all datasets except BMS and dual matching, see Figures 14 and 18, where MTM/DL target the specific search tree shape. Notably, SHD also runs to completion on the win and lose data sets, see Figures 15 and 16, when all previous algorithms fail.

IP_D1-80 is the iterative parallel algorithm that extends SHD_D by using the critical vertex check to partition edges. IP_D1-2 is slower then SHD_D as IP_D requires more computational time in favor of smaller memory, in order to outperform SHD there must be more threads to counter the additional computation. As the number of threads goes up, real world data sets show excellent scaling such as AC, BMS, win, and lose data sets. See Figures 13, 14, 15, and 16, respectively. In some cases adding more threads degrades performance as the cost of copying the representation exceeds the benefit of multiple threads working on the problem: Matching, Figure 17. The extra computation cost is never overcome with more threads on the Threshold, Figure 19, and Self-Dual

Threshold, Figure 20 data sets, however the extra threads outperform on the Self-Dual Fano-Plane Hypergraph, see Figure 21.

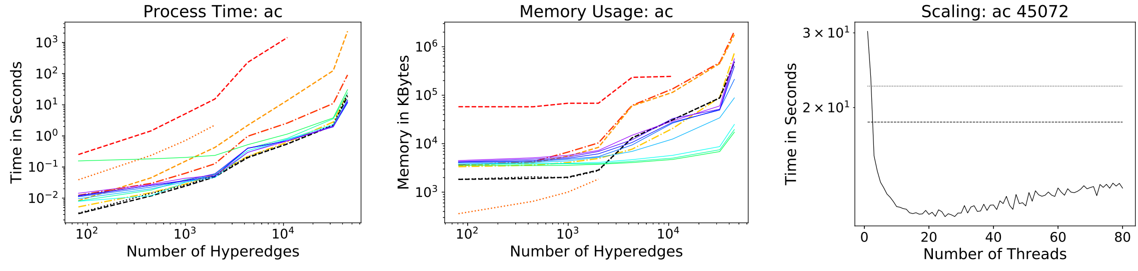


Figure 13: AC: Accidents

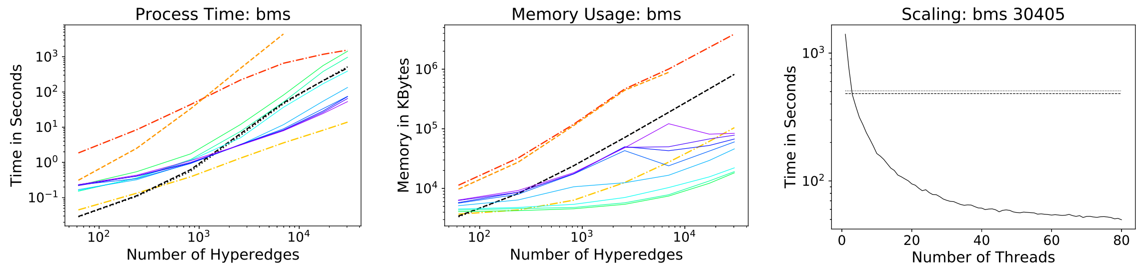


Figure 14: BMS2: BMS-Webview-2

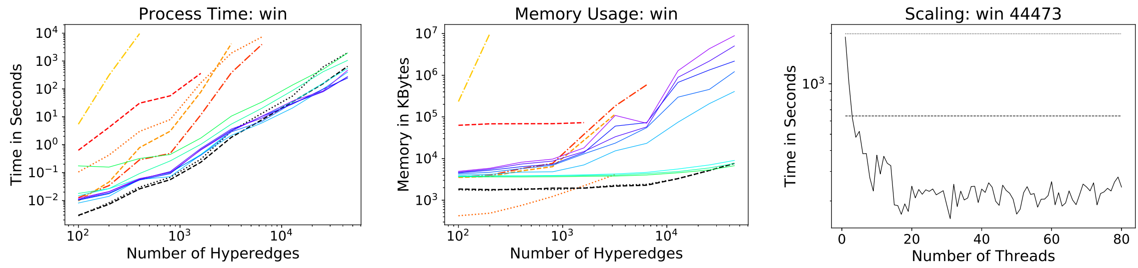
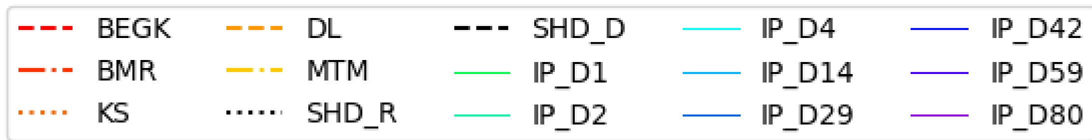


Figure 15: Win: Connect-4 Winning Positions



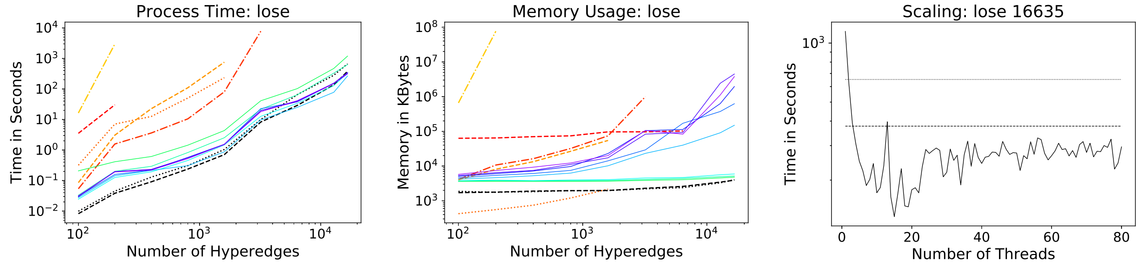


Figure 16: Lose: Connect-4 Losing Positions

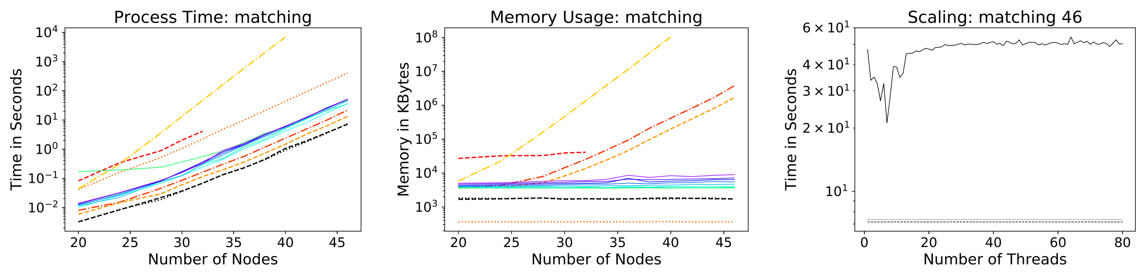


Figure 17: Matching Hypergraph

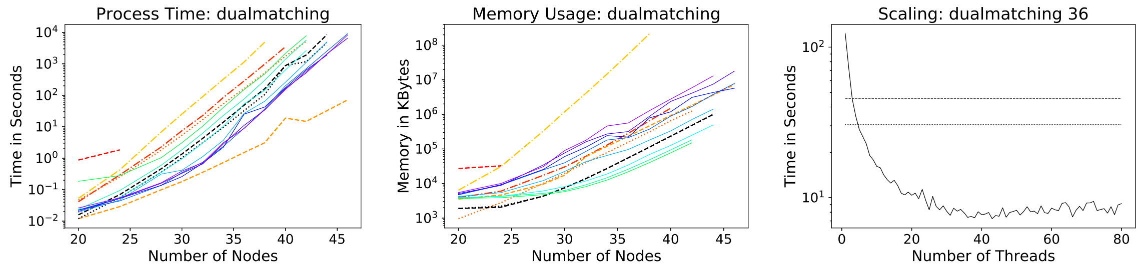
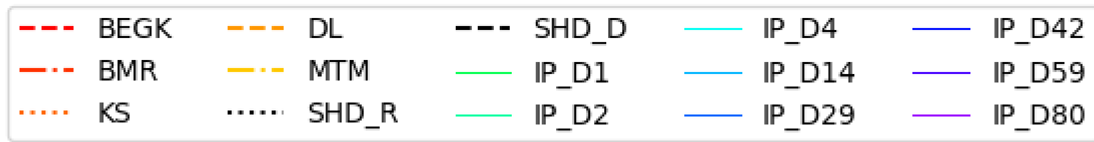


Figure 18: Dual-Matching Hypergraph



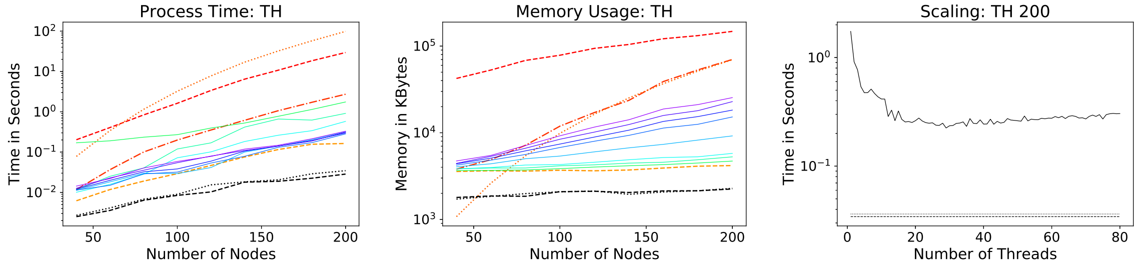


Figure 19: Threshold Hypergraph

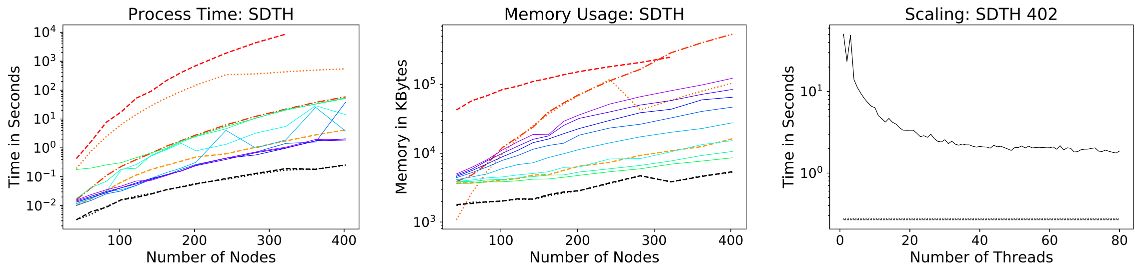


Figure 20: Self-Dual Threshold Hypergraph

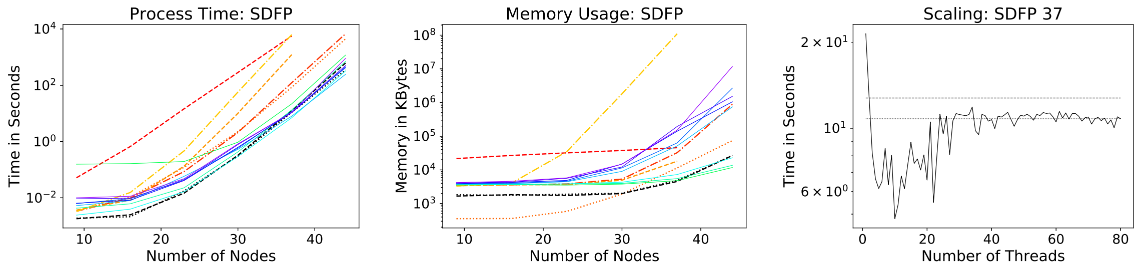
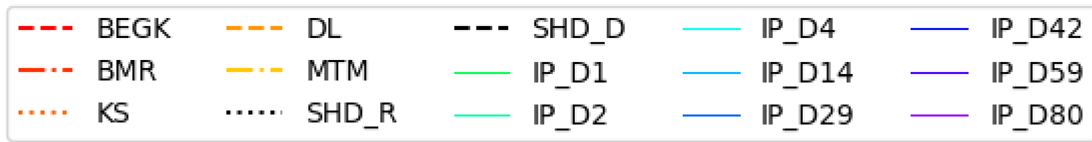


Figure 21: Self-Dual Fano-Plane Hypergraph



7 Conclusion

After examining multiple algorithms and datasets, the IP_D is shown to be the only algorithm capable of scaling to multiple cores. There are a few problems where scaling degrades performance, the cost to split the representation and distribute the work outweighs directly solving the problem. The algorithm still runs to completion just not as fast as SHD in some cases. There are BEGK, MTM, BMR algorithms that completely fail in catastrophic ways for some datasets. KS and DL run into explosive computation time for several data sets as well. IP_D and SHD are the only algorithms to complete computation on all datasets.

IP_D outperforms SHD on larger real-world datasets AC, BMS, win, lose. As IP_D was designed to scale for large problems, this is expected. The poor performance on large mathematical hypergraphs by IP_D even with scaling is unexpected. The vertex search tree for these hypergraphs does not work well with the splitting of representations. There are a large number of representations (task items) created that do little work. When the task item contains lots of work relative to the creation of it, IP_D outperforms.

IP_D is intended for use on real-world large-scale hypergraphs. Additionally, IP_D is designed so that problematic mathematical datasets process to completion instead of failing catastrophically or explode in time usage. IP_D generally outperforms other algorithms overall and shows great scaling properties on larger real-world data sets.

References

[1] 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.

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

[3] Keisuke Murakami and Takeaki Uno. Efficient algorithms for dualizing large-scale hypergraphs. In *2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–13. SIAM, 2013.

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

[5] 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.

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

[7] Roscoe Casita. *Algorithm for enumerating hypergraph transversals*. University of Oregon, 2017.

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