Implementation and evaluation of a self-monitoring approximation algorithm for 3-Hitting-Set

Bachelor-Arbeit

zur Erlangung des Grades eines Bachelor of Science

an der Universität Trier

Fachbereich IV

vorgelegt von

Khoa Le

Musterstraße

Musterstadt

Matrikelnummer

Musterstadt, im [Monat] [Jahr]

ERKLÄRUNG ZUR BACHELORARBEIT

Hiermit erkläre ich, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe.

Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher nicht veröffentlicht.

_____

Datum

_____

Unterschrift

Abstract

In this thesis we implemented an approximation algorithm for the HITTING SET problem. The algorithm is based on various reduction rules and the approximation ratio is estimated with the local ratio method. We show test results on real-world networks that can be interpreted as derivative problems of HITTING SET. At last, we compare these results with two linear programming based algorithms.

Contents

Todo list

proof that expansion step of 1 is sufficient ? 16
explanation ? 19
get data for random rule application to compare 19
update with new test results using other solvers 23

1 Introduction

The HITTING SET problem is one of Karp's 21 NP-complete problems. It has been extensively researched

1.1 Preliminaries

A hypergraph $G = (V, E)$ consists of a set of vertices V and a set of edges E. Each element of E is a subset of V. The degree of a vertex v is the number of edges that are incident to v. We also denote the degree with $deg(v)$ for a vertex v. An edge e has size m, if it contains m elements.

2 Programming Language

We chose the *Go* programming language. It is a statically typed, compiled language, with a C-like syntax. It is using a garbage collector to handle memory management, which at first seems off-putting for an application like this. Since Go's GC is very efficient, we are not worried about that fact. Go also provides great developer tooling without the need for additional third party tools. The built-in profiler *pprof* can visualize captured performance profiles as flame graphs or standard graphs, which makes it easy to spot performance bottlenecks.

3 Data Structures

3.1 Vertex

```go
type Vertex struct {
    id int32
    data any
}
```

The `Vertex` datatype has two fields. The field `id` is an arbitrary identifier and `data` serves as a placeholder for actual data associated with the vertex.

3.2 Edge

```go
type Edge struct {
    v map[int32]bool
}
```

The `Edge` datatype has one field. The field `v` is a map with keys of type `int32` and values of type `bool`. When working with the endpoints of an edge, we are usually not interested in the associated values, since we never mutate the edges. This simulates a *Set* datatype while allowing faster access times than simple arrays/slices.

3.3 Hypergraph

```
type HyperGraph struct {
    Vertices      map[int32]Vertex
    Edges         map[int32]Edge
    edgeCounter   int32
    IncMap        map[int32]map[int32]bool
    AdjCount      map[int32]map[int32]int32
}
```

The `HyperGraph` datatype has five fields. Both fields `Vertices` and `Edges` are maps with keys of type `int32` and values of type `Vertex` and `Edge` respectively. We chose this set-like data structure over lists again because of faster access times, but also operations that remove edges/vertices are built-in to the map type. The field `edgeCounter` is an internal counter used to assign ids to added edges. The field `IncMap` is a map of maps, essentially storing the hypergraph as a sparse incidence matrix. We will also derive vertex degrees from this map. And at last the `AdjCount` map, which will associate every vertex $v \in V$ to all vertices adjacent to v. Additionally, this map will also store the number of times such a vertex is adjacent to v.

4 Misc. Algorithms/Utilities

4.1 Edge Hashing

Given a set S of size n, compute a unique hash of S.

Time Complexity: $n + n \cdot \log(n)$

We start by sorting S with a Quick-Sort-Algorithm. We then join the elements of S with the delimiter "|", returning a string of the form "$|i_0|i_1|\ldots|i_n|$". Whenever we refer to *the hash of an edge* we refer to the output of this function, using the endpoints of the edge as the input set. This hash value is useful when we want to check the existence of specific edges.

4.2 Compute Subsets of Size s

Given an array arr and an integer s, compute all subsets of arr of size s.

Algorithm 1: An algorithm to compute all subsets of size s

Input: An array arr, the subset size s and a list *subsets*.
Output:

1   $data \leftarrow [\,]$
2   $n \leftarrow |arr|$
3   $last \leftarrow s - 1$
4   FnRecursive$(0, 0)$
5   **func** FnRecursive(i, *next*):
6   **for** $j \leftarrow next$ **to** n **do**
7       $data[i] \leftarrow arr[j]$
8   **if** $i = last$ **then**
9         $subsets.push(data)$
10   **else**
11   FnRecursive$(i + 1, j + 1)$

Lists in Go are not very memory efficient, but since we exclusively call this function with arr representing the vertices in an edge, the value is usually fixed at 3. The raised memory problems occur at values of $n > 10000$, justifying the continued usage of lists. For the case where we have to compute a lot of subsets, we provide a slightly different version of this function. Instead of passing in the *subsets* list, we pass in a callback function that is called whenever we find a subset, using the found subset as an argument.

4.3 Two-Sum

Given an array `items` of integers and an integer target t, return indices of the two numbers such that they add up to t.

Time Complexity: n, where n denotes the size of `items`.

Algorithm 2: An algorithm for the Two-Sum problem

Input: An array of integers arr, a target value t
Output: Two indices a, b, such that $arr[a] + arr[b] = t$, a boolean indicating if a solution was found

1 $lookup \leftarrow \text{map}[\mathbb{N}]\mathbb{N}$
2 **for** $i \leftarrow 0$ **to** $len(arr)$ **do**
3   **if** $lookup[t - arr[i]]$ *exists* **then** **return** $(i, lookup[t - arr[i]]), true$
4
5   **else**
6 $lookup[arr[i]] \leftarrow i$

7 **return** $nil, false$

We start by creating a map called *lookup*. Iterating other the elements of arr, we check if the entry $lookup[t - arr[i]]$ exists. If the entry exists, we return a pair $(i, lookup[t - arr[i]])$ and the boolean value *true* since we found a solution. If the entry does not exist, we add a new entry to the *lookup* map using $arr[i]$ as key and i as value. If no solution was found, we return nil and the boolean value $false$.

This algorithm is an ingredient for the implementation of one of the reduction rules, specifically the approximative vertex domination rule. The actual implementation accepts a map instead of an array as its first parameter. We also implemented a version that finds all solutions, which accepts an additional callback function as a parameter. Instead of returning the solution, we call the callback function with the solution as the first argument. This version is an ingredient for the implementation of the approximative double vertex domination. We refer to this version as Two-SumAll inside the algorithm listings.

5 Hypergraph Models

5.1 First Testing Model

```go
func GenerateTestGraph(n int32, m int32, tinyEdges bool) *HyperGraph
```

Let us explain the arguments first:

- `n` are the number of vertices the graph will have
- `m` is the number of edges the graph will at most have
- `tinyEdges` when `false` indicates that we do not want to generate edges of size 1.

We use a very naive approach for generating (pseudo-)random graphs. We first create an empty Hypergraph struct and add `n` many vertices to that graph. We then compute a random `float32` value `r` in the half-open interval $[0.0, 1.0)$. This value will be used to determine the size of an edge e. The edges are distributed based on their size as follows.

$$size(\mathbf{r}) = \begin{cases} 1 & \mathbf{r} < 0.01 \\ 2 & 0.01 \le \mathbf{r} < 0.60 \\ 3 & \text{else} \end{cases}$$

The result of $size(\mathbf{r})$ is stored in a variable `d`. We then randomly pick vertices in the half-open interval $[0, \mathbf{n})$, until we have picked `d` many distinct vertices. If an edge with these endpoints does not exist, we add it to our graph.

That results in the graph having at most `m` edges and not exactly `m`, since we did not want to artificially saturate the graph with edges. One could also look into generating random bipartite graphs that translate back to a hypergraph with the desired vertex and edge numbers. The advantage of this model over a model like the Erdős–Rényi model is that we can compute hypergraphs with large vertex count fast. The model was primarily used during the implementation phase of the algorithm to gather profiling data.

5.2 Preferential Attachment Hypergraph Model

In the Preferential Attachment Model, one will add edges to an existing graph, with a probability proportional to the degree of the endpoints of that edge. This edge will either contain a newly added vertex or will be comprised of vertices already part of the graph. We will use an implementation by Antelmi et al. as reference [1], which is part of their work on *SimpleHypergraphs.jl* [2], a hypergraph software library written in the Julia language. The implementation is based on a preferential attachment model proposed by Avin et al. in [3].

```
func GeneratePrefAttachmentGraph(n int, p float64, maxEdgesize int32)
```

- `n` is the amount of vertices the graph will have
- `p` is the probability of adding a new vertex to the graph
- `maxEdgesize` is the maximum size of a generated edge

6 Reduction Rules

The ususal signature of an implemented reduction rule looks as follows.

```
func NameRule(g *HyperGraph, c map[int32]bool) int32
```

We take both a pointer to a `HyperGraph` struct `g` and a map `c` as arguments and mutate them. We then return the number of rule executions. We prioritize time complexity over memory complexity when implementing rules, which does not equate to ignoring memory complexity completely.

6.1 Executions

The proposed rules are meant to be applied exhaustively. A one-to-one implementation of a rule will only find one of the structures the rule is targeting. Calling such a rule implementation exhaustively will take polynomial time but will be very inefficient regarding memory writes and execution time. It is therefore advantageous to design the algorithms for the rules, with the aspect of exhaustive application in mind.

The general outline of an algorithm will look as follows.

1. (If needed) Construct auxiliary data structures that are used to find parts of the graph, for which the rule can be applied.

2. If we can apply rules do:

   Iterate over the auxiliary data structure, or the vertices/edges themselves.

   - Identify targets of the rule.
   - Mutate the graph according to the rule.
   - Mutate the auxiliary data structure according to the rule.

This way we minimize memory writes by reusing existing data structures we built in step 1, but also save on execution time, since we mutate and iterate the auxiliary data structure at the same time. We can also alter these algorithms with minimal changes, to only execute a rule once.

6.2 Algorithms

The algorithm descriptions occasionally omit implementation details. We do this to try and keep these descriptions as concise as possible. The real implementations correspond to the pseudocode listings. We do not impose the use of our proposed hypergraph model in those listings. Instead we use general language, for example $e$incident to$v$ instead of $e \in H.IncMap[v]$. Proofs are provided in the case, where the algorithm does not simply follow the definition of a rule.

We further introduce a *map* data structure in our pseudocode with syntax $\mathtt{map}[A]B$, which describes a mapping from A into B. We use square brackets to indicate access and mutation of the mapping, e.g. $\gamma[0] \leftarrow 1$ maps the value 1 to the key 0 in map γ. The map type also exposes a primitive function with the signature $delete(\gamma, x)$, which simply means that we want to delete the entry with key x from our map. When iterating over a map in a $\mathtt{for}$-loop we destructure the entries into a $(key, value)$-pair, e.g. $\mathtt{for}\ (\_, v) \in map\ \mathtt{do}$. Unused values are omitted with the underscore symbol.

6.2.1 Tiny/Small Edge Rule

- tiny edges: Delete all hyperedges of size one and place the corresponding vertices into the hitting set.
- small edges: If e is a hyperedge of size two, i.e. $e = \{x, y\}$, then put both x and y into the hitting set.

$O(|E|^2)$ **Algortihm**. We iterate over all edges of the graph. If the current edge e is of size t, put e into the partial hitting set and remove e and all edges adjacent to vertices in e from the graph.

Algorithm 3: Algorithm for exhaustive application of Tiny/Small Edge Rule

Input: A hypergraph $G = (V, E)$, a set C, an integer t denoting the size of the edges to be removed
Output: An integer denoting the number of rule applications.

1   $rem \leftarrow \emptyset$
2   $exec \leftarrow 0$
3   **for** $e \in E$ **do**
4   **if** $|e| = t$ **then**
5       $exec \leftarrow exec + 1$
6   **for** $v \in e$ **do**
7         $C \leftarrow C \cup \{v\}$
8         $V \leftarrow V \setminus \{v\}$
9   **for** f incident to v **do**
10          $E \leftarrow E \setminus \{f\}$

11 **return** $exec$

6.2.2 Edge Domination Rule

- (hyper)edge domination: A hyperedge e is *dominated* by another hyperedge f if $f \subset e$. In that case, delete e.

$O(|E|)$ **Algorithm**. We partition our set of edges into two disjoint sets *sub* and *dom*. The set *dom* will contain edges that could be dominated. The set *sub* will contain hashes of edges e that could dominate another edge. We then iterate over the set *dom* and compute every strict subset of the current edge f. For each of these subsets, we test if the hash of the subset is present in our set *sub*. If it is then f is dominated by another edge.

The exact time complexity is as follows.

Algorithm 4: Algorithm for exhaustive application of Edge Domination Rule

Input: A hypergraph $G = (V, E)$ without size one edges, a set C
Output: An integer denoting the number of rule applications.

1 $sub \leftarrow \emptyset$
2 $dom \leftarrow \emptyset$
3 $exec \leftarrow 0$
4 **for** $e \in E$ **do**
5   **if** $|e| = 2$ **then**
6 $sub \leftarrow sub \cup \{hash(e)\}$
7   **else**
8 $dom \leftarrow dom \cup \{e\}$

9 **if** $|sub| = 0$ **then**
10   **return** $exec$

11 **for** $e \in dom$ **do**
12 $subsets \leftarrow$ getSubsetsRec(e, 2)
13   **for** $f \in subsets$ **do**
14   **if** $hash(f) \in sub$ **then**
15 $E \leftarrow E \setminus \{e\}$
16 $exec \leftarrow exec + 1$
17   **break**

18 **return** $exec$

$$T = |E| \cdot d \cdot \log(d) + (|E| \cdot (d + 2^d + (2^d \cdot d \cdot \log(d))))$$

Specifically applied to $d = 3$, this results in a time complexity of:

$$T = |E| \cdot 3 \cdot \log(3) + (|E| \cdot (11 + 24 \cdot \log(3)))$$
$$= |E| \cdot (3 \cdot \log(3) + (11 + 24 \cdot \log(3)))$$

Lemma 6.1 *Algorithm 4 finds all edges of G that are dominated, iff G has no size one edges.*

Proof. Let e be a dominated edge. Since there are no size one edges, edges with size two cannot be dominated. Thus e has to be of size three. Then simply removing e will not create or eliminate an edge domination situation. It is therefore sufficient to only check size three edges for the domination condition. $\square$

This also allows us to parallelize the main part of the algorithm, where we check each edge in our *dom* set. We can achieve a speedup of approximately 2 on a six-core CPU and a pseudo-random graph with one million vertices and two million edges.

6.2.3 Vertex Domination Rule

- A vertex x is dominated by a vertex y if, whenever x belongs to some hyperedge e, then y also belongs to e. Then, we can simply delete x from the vertex set and from all edges it belongs to.

$O(|V|^2 \cdot |E|)$ **Algorithm** A vertex v is dominated, if one of the entries in $AdjCount[v]$ is equal to $deg(v)$. In that case we remove v from all edges and our vertex set.

Algorithm 5: Algorithm for exhaustive application of Vertex Domination Rule

Input: A hypergraph $G = (V, E)$

Output: An integer denoting the number of rule applications.

1 $exec \leftarrow 0$

2 $outer \leftarrow true$

3 **while** $outer$ **do**

4 $outer \leftarrow false$

5   **for** $v \in V$ **do**

6 $dom \leftarrow false$

7   **for** $(\_, val) \in AdjCount[v]$ **do**

8   **if** $val = deg(v)$ **then**

9 $dom = true$

10   **break**

11   **if** dom **then**

12 $outer = true$

13   **for** e incident to v **do**

14 $e \leftarrow e \setminus \{v\}$;

15 $V \leftarrow V \setminus \{v\}$

16 $exec \leftarrow exec + 1$

17 **return** $exec$

6.2.4 Approximative Vertex Domination Rule

- approximative vertex domination: Assume there is a hyperedge $e = \{x, y, z\}$ such that, whenever x belongs to some hyperedge h, then y or z also belong to h. Then, we put y and z together into the hitting set that we produce.

$O(|V|^2 \cdot |E|)$ **Algorithm**. The additional factor $|E|$ looks scary at first, but will only occur in the worst case, if there exists a vertex v that is incident to all edges in G. We start by iterating other the $AdjCount$ map of the graph, referring to the current value in the iteration as $AdjCount[v]$. We then use the Two-Sum algorithm to compute and return the first pair (a, b) in $AdjCount[v]$, s.t. for (a, b) holds,

$$AdjCount[v][a] + AdjCount[v][b] = deg(v) + 1$$

If such a pair exists, then we conclude that for every edge f incident to v, it holds that either $a \in f$ or $b \in f$.

Lemma 6.2 *The outlined procedure above is correct, under the assumption that the underlying graph does not contain any duplicate edges or size one edges.*

Proof. Let G be a hypergraph without size one edges. Let v be an entry in $AdjCount$ and $sol = (a, b)$ the result of calling our Two-Sum algorithm on $AdjCount[v]$ with a target sum of $n = deg(v) + 1$.

Proposition. If sol is non-empty, then the edge $\{v, a, b\}$ exists.

Let $sol = (a, b)$ be the solution obtained by calling our Two-Sum algorithm on $AdjCount[v]$ with a target sum of $n = deg(v) + 1$. For the sake of contradiction let us assume that the edge $\{v, a, b\}$ does not exist. Since our graph does not contain duplicate edges, size one edges and not the edge $\{v, a, b\}$, there exist $deg(v) + 1$ many edges that contain either $\{a, v\}$ or $\{b, v\}$. This however contradicts that there only exist $deg(v)$ many edges containing v. Therefore the assumption that $\{v, a, b\}$ does not exist, must be false.

Since $\{v, a, b\}$ exists, a and b can only occur $n - 2 = deg(v) - 1$ times in other edges containing v. Since duplicate edges of $\{v, a, b\}$ cannot exist, we know that every other edge containing v also contains a or b, but not both simultaneously. $\square$

We then add the two vertices in the solution *sol* to our partial solution C.

Algorithm 6: Algorithm for exhaustive application of Approximative Vertex Domination Rule

Input: A hypergraph $G = (V, E)$, a set C
Output: An integer denoting the number of rule applications.

1 $exec \leftarrow 0$
2 $outer \leftarrow true$
3 **while** *outer* **do**
4 $outer \leftarrow false$
5   **for** $(v, count) \in AdjCount$ **do**
6 $sol, ex \leftarrow \texttt{TwoSum}(count, deg(v) + 1)$
7   **if** *not* *ex* **then**
8   **continue**
9 $outer \leftarrow true$
10 $exec \leftarrow exec + 1$
11   **for** $w \in sol$ **do**
12 $C \leftarrow C \cup \{w\}$
13 $V \leftarrow V \setminus \{w\}$
14   **for** e incident to w **do**
15 $E \leftarrow E \setminus \{e\}$

16 **return** *exec*

Idea: The initial idea for this algorithm involved the usage of a complete incidence matrix, where edges are identified by the rows and the vertices are identified by the columns. To check the *Domination Condition* for a vertex v, the algorithm would select all edges/columns that contain v and then add up the columns. Now let n be the number of edges containing v. If there exist two entries in the resulting column that have a combined value of $n + 1$, then the rule applies for v under the assumption that there are no duplicate edges. This would result in an algorithm with a worse time complexity of $|V| + |V|^2 \cdot |E|$.

6.2.5 Approximative Double Vertex Domination Rule

- approximative double vertex domination: Assume there is a hyperedge $e = \{x, y, a\}$ and another vertex b such that, whenever x or y belong to some hyperedge h, then a or b also belong to h. Then, we put a and b together into the hitting set that we produce.

$\mathcal{O}(|V|^2 + |E|)$ **Algorithm**. We start by creating a map called $tsHashes$, which maps subsets of E to subsets of vertices. We then iterate over all vertices in V. For the current vertex x, compute all Two-Sum solutions with input array/map $AdjCount[x]$ and target $deg(x)$. If there exists a solution $sol = \{z_0, z_1\}$, such that $tsHashes[sol] \neq \emptyset$, then for all $y \in tsHashes[sol], y \neq x$ construct two edges $\{x, y, z_0\}$ and $\{x, y, z_1\}$. If one of these two edges exists in E, then we found a approximative double vertex domination situation. If $tsHashes[sol] = \emptyset$, map $tsHashes[sol]$ to $tsHashes[sol] \cup \{x\}$.

Lemma 6.3 *Algorithm 7 is correct.*

Proof. Let $G = (V, E)$ be a hypergraph. Let x be the current vertex in the algorithm's iteration over V. If $T = \text{Two-SumAll}(AdjCount[x], deg(x))$ is empty, then x will not be able to trigger a approximative double vertex domination situation. If T is not empty, then we know that there exist sets $\{a, b\}$, such that all edges incident to x either contain a or b. If $tsHashes[\{a, b\}]$ is empty, then there are currently no other vertices for which $\{a, b\}$ is a Two-Sum solution. In that case we add x to $tsHashes[\{a, b\}]$. Otherwise there exist vertices y, such that $\{a, b\}$ is a Two-Sum solution for y. All edges incident to x and y either contain a or b. If for one of these vertices y there exists a size three edge $\{x, y, a\}$, or without loss of generality $\{x, y, b\}$, then we found a approximative double vertex domination situation at $\{x, y, a\}$ or $\{x, y, b\}$ respectively. If none

of these edges exist, then x could still trigger another approximative double vertex situation with another vertex. Thus we need to add x to $tsHashes[\{a,b\}]$. $\square$

Algorithm 7: Algorithm for exhaustive application of Approximative Double Vertex Domination Rule

Input: A hypergraph $G = (V, E)$, a set C
Output: An integer denoting the number of rule applications.

1 $exec \leftarrow 0$
2 $outer \leftarrow true$
3   **for** $outer$ **do**
4 $outer \leftarrow false$
5 $tsHashes \leftarrow \texttt{map}[2^V]2^V$
6   **for** $x \in V$ **do**
7   **for** $sol \in \texttt{TwoSumAll}(AdjCount[x], deg(x))$ **do**
8 $\{z_0, z_1\} \leftarrow sol$
9   **if** $tsHashes[sol] \neq \emptyset$ **then**
10   **for** $y \in tsHashes[sol]$ **do**
11   **if** $y = x$ **then**
12   **continue**
13 $f_0 \leftarrow \{x, y, z_0\}$
14 $f_1 \leftarrow \{x, y, z_1\}$
15 $found \leftarrow false$
16   **for** e incident to y **do**
17   **if** $e = f_0$ **or** $e = f_1$ **then**
18 $found \leftarrow true$
19   **break**
20   **if** $found$ **then**
21 $exec \leftarrow exec + 1$
22 $outer \leftarrow true$
23 $C \leftarrow C \cup sol$
24   **for** $a \in sol$ **do**
25   **for** e incident to a **do**
26 $E \leftarrow E \setminus \{e\}$
27   **break**
28 $tsHashes[sol] \leftarrow tsHashes[sol] \cup \{x\}$
29   **else**
30 $tsHashes[sol] \leftarrow \{x\}$
31 **return** $exec$

6.2.6 Small Triangle Rule

- small triangle situation: Assume there are three small hyperedges $e = \{y, z\}$, $f = \{x, y\}$, $g = \{x, z\}$. This describes a triangle situation (e, f, g). Then, we put $\{x, y, z\}$ together into the hitting set, and we can even choose another hyperedge of size three to worsen the ratio.

$O(|E| + |V|^2)$ **Algorithm** We start by constructing an adjacency list $adjList$ for all edges of size two. We then iterate over the entries of the list. For the current entry $adjList[v]$ we compute all subsets of size two of the entry. If there exists a subset s such that $s \in E$, then we found a small triangle situation. If we find a triangle situation, we put the corresponding vertices in our partial solution and alter the adjacency list to reflect these changes. We do this by iterating over all vertices that are adjacent to the triangle. For every

vertex w of these vertices we delete all vertices of the triangle from the entry $adjList[w]$.

This last step will introduce the quadratic complexity, since in the worst case, for a vertex v in a triangle, there could exist $|V|$ many size two edges that contain v. This worst case occurs very rarely, which justifies using this quadratic algorithm. We could alternatively move the last step of the algorithm outside of the loop and wrap both procedures with an outer loop which breaks if we do not find any more triangles. This simulates calling the rule exhaustively, while achieving a linear time complexity.

Algorithm 8: : Algorithm for exhaustive application of Small Triangle Rule

Input: A hypergraph $G = (V, E)$, a set C
Output: An integer denoting the number of rule applications.

1 $adjList \leftarrow \texttt{map}[V]2^V$
2 $rem \leftarrow \emptyset$
3 $exec \leftarrow 0$
4 for $e \in E$ **do**
5   **if** $|e| \neq 2$ **then**
6   **continue**
7 $\{x, y\} \leftarrow e$
8 $adjList[x] \leftarrow adjList[x] \cup \{y\}$
9 $adjList[y] \leftarrow adjList[y] \cup \{x\}$
10 for $(z, val) \in adjList$ **do**
11   **if** $|val| < 2$ **then**
12   **continue**
13 $subsets \leftarrow \texttt{getSubsetsRec}(val, 2)$
14   **for** $s \in subsets$ **do**
15 $\{x, y\} \leftarrow s$
16   **if** $y \in adjList[x]$ **or** $x \in adjList[y]$ **then**
17 $exec \leftarrow exec + 1$
18 $C \leftarrow C \cup \{x, y, z\}$
19 $rem \leftarrow rem \cup \{x, y, z\}$
20   **for** $u \in \{x, y, z\}$ **do**
21   **for** $v \in adjList[u]$ **do**
22 $adjList[v] \leftarrow adjList[v] \setminus \{u\}$
23 $delete(adjList, u)$
24   **break;**
25 for $v \in rem$ **do**
26 $V \leftarrow V \setminus \{v\}$
27   **for** e incident to v **do**
28 $E \leftarrow E \setminus \{e\}$
29 return $exec$

6.2.7   Extended Triangle Rule

- Assume that the hypergraph contains a small edge $e = \{y, z\}$. Moreover, there are hyperedges f, g such that $e \cap f = \{y\}$, $e \cap g = \{z\}$, $f \cup g = \{v, x, y, z\}$ and $|f| = 3$. Then, put all of $f \cup g$ into the hitting set.

$O(|E|^3)$ **Algorithm** We start by iterating over E until we find a size two edge e. We then iterate over the vertices of e. Let a be the vertex in the current iteration. Assign a to the variable y and $e \setminus \{y\}$ to $\{z\}$. Then we iterate over the edges f that are incident to y. If f is not of size three or $z \in f$, continue with the iteration. Else, iterate over the edges g that are incident to z. If $g \setminus \{z\} \subset f$, then we found a extended

triangle situation. In that case, save f in a variable f_0 and break out of the loop that iterates over the edges incident to y. If $f_0 \neq nil$ then put $f_0 \cup \{z\}$ into the partial hitting set C and break out of the loop iterating over the endpoints of e.

Algorithm 9: Algorithm for exhaustive application of Extended Triangle Rule

---

Input: A hypergraph $G = (V, E)$, a set C
Output: An integer denoting the number of rule applications.

1 $exec \leftarrow 0$
2 $outer \leftarrow true$
3 **for** $outer$ **do**
4 $outer \leftarrow false$
5   **for** $e \in E$ **do**
6   **if** $|e| \neq 2$ **then**
7 $\lfloor$ **continue**
8   **for** $a \in e$ **do**
9 $y \leftarrow a$
10 $\{z\} \leftarrow e \setminus \{y\}$
11 $f_0 \leftarrow nil$
12   **incy:**
13   **for** f incident to y **do**
14   **if** $|f| \neq 3$ **or** $z \in f$ **then**
15 $\lfloor$ **continue**
16   **for** g incident to z **do**
17 $cond \leftarrow true$
18   **for** $b \in g$ **do**
19   **if** $b = z$ **then**
20 $\lfloor$ **continue**
21   **if** $b \notin f$ **then**
22 $cond \leftarrow false$
23   **break**
24   **if** $cond$ **then**
25 $f_0 \leftarrow f$
26   **break incy**
27   **if** $f_0 \neq nil$ **then**
28 $outer \leftarrow true$
29 $exec \leftarrow exec + 1$
30 $C \leftarrow C \cup f \cup \{z\}$
31 $V \leftarrow V \setminus f \cup \{z\}$
32   **for** h incident to $f \cup \{z\}$ **do**
33 $\lfloor$ $E \leftarrow E \setminus \{h\}$
34   **break**

35 **return** exec

6.2.8 Small Edge Degree 2 Rule

- small edge degree 2: Let v be a vertex of degree 2, and let the two hyperedges containing v be $e = \{x, v\}$ and $f = \{v, y, z\}$. Then we can select a hyperedge g that contains one of the neighbors of v in f but not x, for example $g = \{u, w, z\}$ (when $y = w$ is possible as a special case) or $g = \{u, z\}$. We put x, u and z and w (when existing) into the hitting set.

$O(|V| \cdot |E|)$ **Algorithm** We start by iterating other all vertices in V. If the vertex v in the current iteration is of degree two, check if there is a size two edge e and a size three edge f incident to v. If these two edges exist, save $e \setminus \{v\}$ in a variable x. Then iterate over the vertices w in $f \setminus v$ and check if there exists an edge h incident to w, such that $h \neq f$ and $x \notin h$. If such an edge exists, then we found a small edge degree 2 situation. We then put both x and all vertices of h into the partial hitting set.

Algorithm 10: Algorithm for exhaustive application of Small Edge Degree 2 Rule

Input: A hypergraph $G = (V, E)$, a set C
Output: An integer denoting the number of rule applications.

1   $exec \leftarrow 0$
2   $outer \leftarrow true$
3   **for** $outer$ **do**
4   **for** $v \in V$ **do**
5   **if** $deg(v) \neq 2$ **then**
6   **continue**
7       $s2, s3 \leftarrow nil$
8   **for** e incident to v **do**
9   **if** $|e| = 3$ **then**
10          $s3 \leftarrow e$
11   **else if** $|e| = 2$ **then**
12          $s2 \leftarrow e$
13   **if** $s2 = nil$ **or** $s3 = nil$ **then**
14   **continue**
15       $\{x, \_\} \leftarrow s2$
16       $found \leftarrow false$
17       $rem \leftarrow nil$
18   **for** $w \in s3 \setminus \{v\}$ **do**
19   **for** f incident to w **do**
20   **if** $x \in f$ **or** $s3 = f$ **then**
21   **continue**
22   **else**
23           $found \leftarrow true$
24           $rem \leftarrow f$
25   **break**
26   **if** $found$ **then**
27   **break**
28   **if** $found$ **then**
29         $outer \leftarrow true$
30         $exec \leftarrow exec + 1$
31   **for** $a \in \{x\} \cup rem$ **do**
32          $C \leftarrow C \cup \{a\}$
33   **for** h incident to a **do**
34           $E \leftarrow E \setminus \{h\}$

6.2.9   F3 Low Degree Rule

- F3 low degree: Let v be a vertex with degree 2 and let e be an edge that is incident to x. If there exists an edge f that is incident to a vertex in e and does not contain v, put f into the hitting set.

As seen in algorithm 11 we extended this rule in the final implementation. If there is no vertex for which the

rule applies, then we check the condition of the rule once again with the vertex v_{min}. If this attempt does not work, then we put a random size three edge into the hitting set. The effectiveness of this rule will be discussed in section 8.

Algorithm 11: Algorithm that selects a single size three edge to put into the hitting set

Input: A hypergraph $G = (V, E)$, a set C
Output: An integer denoting the number of rule applications.

1 $min \leftarrow \infty$
2 $v_{min} \leftarrow nil$
3 $f_0 \leftarrow nil$;
4 **check:**
5 **for** $x \in V$ **do**
6   **if** $deg(x) = 2$ **then**
7 $found \leftarrow false$
8   **incx:**
9   **for** e incident to x **do**
10   **for** $v \in e \setminus \{x\}$ **do**
11   **for** $f \neq e$ incident to v **do**
12   **if** $x \notin f$ and $|f| = 3$ **then**
13 $found \leftarrow true$
14 $f_0 \leftarrow f$
15   **break incx**

16   **if** $found$ **then**
17 $C \leftarrow C \cup f_0$
18   **for** $v \in f_0$ **do**
19   **for** e incident to e **do**
20 $E \leftarrow E \setminus \{e\}$

21   **return** 1
22   **else if** $deg(x) < min$ **and** $deg(x) > 1$ **then**
23 $min \leftarrow deg(x)$
24 $v_{min} \leftarrow x$
25   jump to label **incx** with $x \leftarrow v_{min}$
26   **if** $f_0 = nil$ **then**
27 $f_0 \leftarrow$ random size three edge
28   **return** $f_0 = nil$

6.3 Self-Monitoring

Each of the reduction rule functions returns a `int32` value, which indicates the number of rule executions. We store the ratios for each rule in a map of the form:

```
var ratios = map[string]pkg.IntTuple{
    "kRulename": {A:1, B:1},
}
```

Where `A` denotes the number of vertices put into the partial solution by a single rule execution. And `B` denotes the number of vertices present in an optimal solution. We can then use these values to calculate the estimated approximation factor as follows.

```
execs := ApplyRules(g, c)

var nom float64 = 0
```

```
var denom float64 = 0

for key, val := range execs {
    nom += float64(ratios[key].A * val)
    denom += float64(ratios[key].B * val)
}
```

7   Algorithms

7.1   Main Algorithm

It is crucial to apply the rules in a given order. Some rules expect that other rules cannot be applied. As for the exact rules, one can save execution time, if a specific order is enforced when executing them. We use the order proposed in the original paper [4], also referred to as precedence of rule executions. We will just use the shorter term *precedence* going forward.

1. Exact Rules: vertex domination → tiny edge → edge domination
2. approximate (double) vertex domination rules
3. small edge degree 2 rule
4. small triangle rule
5. extended triangle rule
6. small edge rule

The main algorithm will apply all rules exhaustively according to the precedence, possibly mutating the input graph. If the graph has no more edges, then we are done. If not, put a size three edge into the partial hitting set and start over with the rule application. Note that the step of "applying all rules exhaustively" leaves room for interpretation and that the produced hitting sets are highly dependant on the actions taken during this step. We will refer to the procedure in this step as *rule strategy* or just *strategy* if the context allows it. We now present three strategies to apply the rules in the precedence.

The base strategy is quite simple. Execute a rule exhaustively and then do the same for the next rule in the precedence. Do this until no more rules can be applied. This is obviously not very optimal, since we do not check, wether a "better" rule can be applied again, before possibly executing a worse rule. We can however add some heuristics that preserve the fast execution time and improve on effectiveness. Instead of executing the vertex domination → tiny edge → edge domination cascade only once, we execute it three times. By just doing this, we can recoup a lot of cascades we might have missed with the base strategy. Iteration values higher than three did not yield better results. We should also execute this cascade after rules that are applied the most. Since these rules are the ones most likely to make the extact rules applicable. That would be the approximative (double) vertex domination rules. The second strategy is also derived from the same base strategy. The only modification is that we start over with the vertex domination rule, whenever one of the rules was applied at least once. This should in theory lead to more exact rules being executed, while sacrificing a bit of execution time. The third strategy tries to maximize the exact rule executions. We start by applying the exact rules exhaustively. We then execute the other rules at most once, according to the precedence. If a rule has been applied, we execute the exact rules exhaustively and start over with the first non-exact rule in the precedence.

7.2   Incremental Frontier Algorithm

As experienced with the DBLP coauthor graph, there are some graph instances that do not work well with the prior algorithm. These instances do not admit any rule executions after applying the fallback rule. Running the algorithm on the whole graph again, knowing that only small parts of the graph have changed is not very time efficient. The algorithm should only look at the part of the graph where the fallback rule, or in fact any rule, was applied. We therefore propose a new approach, which involves the usage of a *vertex frontier*. Such frontiers are common in search algorithms such as BFS, where each vertex in the frontier has the same distance to the root vertex. But let us explain the general structure of the algorithm first.

We first apply all reduction rules exhaustively for the entire graph G. If the graph has no more edges, we are done. Else we try to find a size three edge e, which will trigger a vertex domination situation if removed. We choose a random size three edge if there is none. We then build up our initial frontier. We put all vertices into the frontier that are adjacent to a vertex in e, excluding vertices in e itself. Then remove all edges that are incident to vertices in e. Next the frontier will be expanded, by adding all edges incident to the frontier to a new graph H. All vertices in H that were not part of the frontier will form the next frontier. The amount of expansion steps can be set by the user. The fields `AdjCount` and `IncMap` of G will be reused by H. Thus, changes in H will be reflected in G and vice versa. The main loop will be explained next.

proof that expansion step of 1 is sufficient ?

We apply the rules as usual, but only on H. The rules are modified, such that they also return the vertices adjacent to vertices in a modified/removed edge. If there are any, then H will be expanded at these vertices and the loop will be continued. If not, apply the targeted fallback rule, considering the edges in the original graph G and expand accordingly. This will be repeated until G has no more edges.

Algorithm 12: Incremental 3-approximation algorithm for 3-HS

Input: Hypergraph $G = (V, E)$, and a partial hitting set C
Output: A hitting set C

1 Apply all reduction rules exhaustively mutating G and C
2 $l \leftarrow 2$
3 **if** $|E| = 0$ **then**
4   **return** C
5 $H \leftarrow$ empty hypergraph
    /* H will act as a mask over G, the rules will only be applied to vertices/edges in
      H */
6 **for** $|E| > 0$ **do**
7   Apply all reduction rules exhaustively on H mutating H,G and C
8 $exp \leftarrow$ vertices of edges adjacent to removed or modified edge
9   **if** $|exp| > 0$ **then**
10 $H \leftarrow$ ExpandFrontier(G, l, exp)
11   **continue**
12 $e \leftarrow$ F3LowDegree(G)
13   **if** $e = \emptyset$ **then**
14   **continue**
15 $H \leftarrow$ ExpandFrontier(G, l, e)

16 **return** C
17 **func** ExpandFrontier(G, l, exp):
18 $H \leftarrow$ empty hypergraph
19   **for** $i \leftarrow 0$ **to** l **do**
20 $next \leftarrow \emptyset$
21   **for** $v \in exp$ **do**
22   **for** $e \in G.E$ incident to v **do**
23   **if** $e \notin H.E$ **then**
24 $H.E \leftarrow H.E \cup \{e\}$
25   **for** $w \in e$ **do**
26   **if** $w \notin H.V$ **then**
27 $H.V \leftarrow H.V \cup \{w\}$
28 $next \leftarrow next \cup \{w\}$

29   **if** $|next| = 0$ **then**
30   **break**
31 $exp \leftarrow next$
32 $H.IncMap = G.IncMap$
33 $H.AdjCount = G.AdjCount$
34   **return** H

Algorithm 12 was designed around both rule strategy 1 and 2. It is not compatible with strategy 3, since the auxiliary graph could at some point be empty and thus $|exp| = 0$, with the original graph still admitting rule applications. In such a case, we apply the first applicable rule in the precedence and use the neighborhood of the removed vertices to rebuild the auxiliary graph.

8 Applications and Results

All tests were run on a machine with a AMD Ryzen 5 3600 3.6 GHz six-core, 12-thread CPU and 32 gigabytes of 3200 MT/s DDR4 RAM. The machine was running Ubuntu Server 22.04.04 LTS and the binaries were

compiled with Go version 1.22.1. Measured execution times do not include the time it takes to load in a graph from disk or the time needed to transform the graph to a new problem instance. We provide results for TRIANGLE VERTEX DELETION and CLUSTER VERTEX DELETION problems using real world networks, as well as randomly generated hypergraphs.

8.1 F3 Low Degree Rule

The goal of the F3 low degree rule is to reduce the degree of a vertex, which preferably has a degree of two. Applying the rule would lower the degree of the vertex to one, making the vertex domination rule applicable. We tested the effectiveness of the F3 low degree rule on random ER hypergraphs. First, we generated 100 random ER hypergraphs with 1000 vertices and approximately 3000 edges. We ran our algorithm on every graph 10 times each to account for run-to-run variance. We collected data for two datasets. For the first dataset we selected a random size three edge when applying the fallback rule. For the second dataset we used the F3 low target degree rule instead. The results can be seen in table 6.

We do in fact see a slight increase of vertex domination rule executions, with the mean number of executions increasing from about 345 to 349. More interesting observations can be made about the fallback rule and extended triangle rule. The mean number of fallback rule executions decreased from about 44 to 25 when not selecting random edges. The number of extended triangle rule executions increased from about 47 to 63. The mean values for the over rules stay mostly the same.

To investigate these changing values, we reran the algorithm with both fallback variants on some of the graphs. This time keeping a log of all the rules that are applied. Every time the F3 low degree rule is executed, the $F3 \rightarrow VDom \rightarrow ETri$ cascade is most likely triggered. For the random variant, one can see batches of fallback rule executions and the occasional $F3 \rightarrow VDom \rightarrow ETri$ cascade in the logs.

So what happens is, that the second variant substitutes multiple fallback rule executions with a single $F3 \rightarrow VDom \rightarrow ETri$ cascade. The desired effects are underwhelming, with the mean hitting set size decreasing from about 592 to 590. Since we did not regress, we chose to keep using this variant of the rule over the random variant.

8.2 Triangle Vertex Deletion

> **Problem Name:** TRIANGLE VERTEX DELETION
> **Given:** A graph $G = (V, E)$ and a non-negative integer k.
> **Output:** Is there a set $C \subseteq V$ of size at most k, such that $G \setminus C$ is a triangle-free graph.

TRIANGLE VERTEX DELETION can be reduced to the 3-HITTING SET problem. Triangles in the input graph will be represented as size 3 edges in a hypergraph, containing the vertices that make up the triangle. A hitting set of size at most k on this hypergraph can be used to transform the original standard graph into a triangle-free graph, removing at most k vertices.

8.2.1 DBLP Coauthor Graph

We evaluated the algorithms on a DBLP coauthor graph and ER graphs of varying densities. It was considered to use the complete DBLP coauthor graph, which was quite large. We ultimately decided to use a dataset of the largest connected component. The network is made available by the Stanford Network Analysis Project and part of a paper by J. Yang and J. Leskovec [5] about community detection in real-world networks. The network contains 2224385 triangles, resulting in a 3-HITTING SET instance of the same size. We now present the collected data during a run of the algorithm on this hypergraph, which can be found in table 1.

Rule strategy 1 has the fastest execution time with a mean of 29 seconds, but also the worst ratio with a mean of 1.5708. Rule strategy 3 has the best ratio with a mean of 1.4195, but also the worst execution time of about 8 minutes. We used the frontier technique in all the tests, since we could observe a speedup of about 90, just using the base rule strategy. This speedup can be attributed to the fact that the original coauthor graph exerts a community structure. These communities are preserved when converting it to a
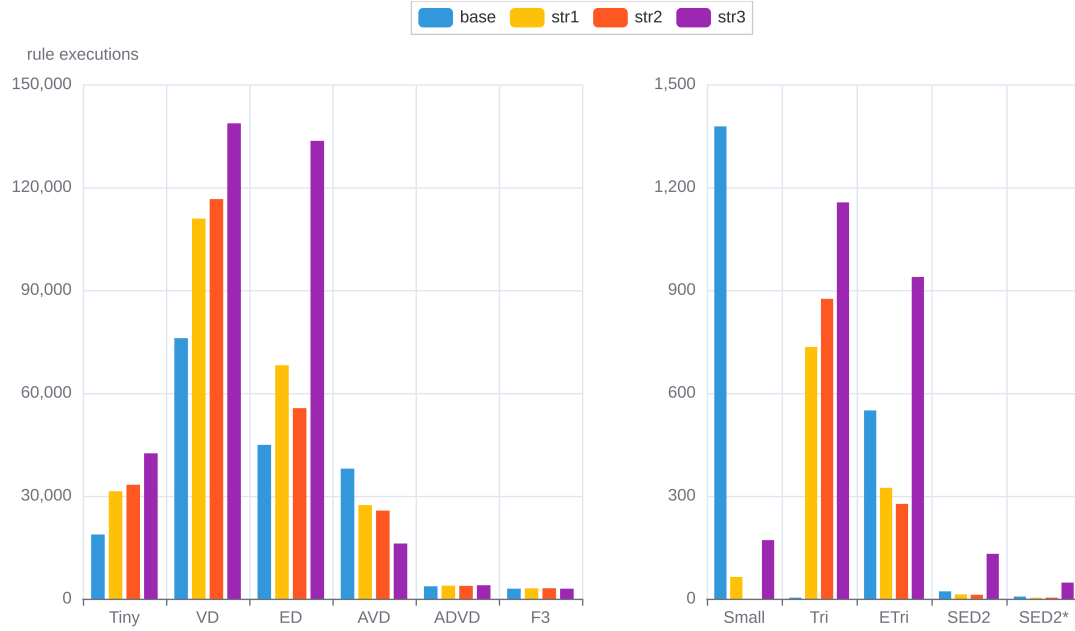
Figure 1: Mean number of rule executions for Triangle Vertex Deletion on DBLP coauthor graph per rule strategy, $n = 100$

Triangle Vertex Deletion and thus 3-Hitting Set instance. The expansion steps will only expand the auxiliary graph H inside these communities. This drastically reduces the number of edges the algorithm has to process per iteration. One can see the opposite happen in dense hypergraphs. Due to the high density, even a small expansion two neighborhoods deep will add all edges of G to H, eliminating possible gains. In fact, constant rebuilding of the auxiliary graph H will even worsen the execution time. This can be observed when using any rule strategy besides strategy 3 on a dense 3-uniform ER hypergraph. On inspection of the generated CPU profile, around 30% of the time is spent computing the auxiliary graph H. See figure 4 for a flamegraph obtained from one of those profiles.

Figure 1 displays the number of rule executions per strategy. Note that the there is a rule called SED2* in this chart. Recall that the small edge degree 2 rule can either put four or three vertices into the hitting set. Both cases were counted seperately, where SED2* refers to the version that puts three vertices into the hitting set. There are several interesting observations to be made, for example the decrease of edge domination rule executions for strategy 2 compared to strategy 1. Quite frankly we do not know why this happens. We can also see that strategy 3 does indeed increase the amount of exact rule executions. Another observation can be made about the number of small edge rule executions. Strategy 2 does not apply the small edge rule at all, which is surprising since it is conceptually similar to strategy 3. This is probably a result of only applying rules one at a time using strategy 3. The exact rules that are executed after each rule application will possibly destroy situations that are favorable for other rules. The approximative vertex domination rule, which is executed a lot more with strategy 2 compared to 3, might have removed all the size two edges as a byproduct. The number of fallback rule executions is also interesting. All strategies are executing this rule around 3000 times. This leads us to the hypothesis that as long as the order in the precedence is obeyed, a "worse" strategy will not result in significantly more fallback rule executions. It could also be true that this number stays the same for any precedence.

explanation?

get data for random rule application to compare

19

	(a) base rule strategy			
	ratio	$|C|$	est. opt	time
mean	1.7594	115672	65745	17 sec
std	0.0005	79	41	1 sec
min	1.7577	115469	65635	16 sec
median	1.7594	115677	65741	17 sec
max	1.7610	115998	65904	19 sec

	(b) rule strategy 1			
	ratio	$|C|$	est. opt	time
mean	1.5708	106339	67697	29 sec
std	0.0006	65	40	1 sec
min	1.5691	106183	67597	26 sec
median	1.5708	106342	67699	29 sec
max	1.5727	106500	67797	31 sec

	(c) rule strategy 2			
	ratio	$|C|$	est. opt	time
mean	1.5437	105084	68071	104 sec
std	0.0006	63	35	2 sec
min	1.5422	104922	67956	99 sec
median	1.5438	105084	68072	104 sec
max	1.5453	105261	68146	110 sec

	(d) rule strategy 3			
	ratio	$|C|$	est. opt	time
mean	1.4195	99456	70066	474 sec
std	0.0009	52	31	7 sec
min	1.4171	99323	69987	457 sec
median	1.4196	99450	70069	474 sec
max	1.4215	99575	70133	493 sec

Table 1: Results for Triangle Vertex Deletion on DBLP coauthor graph; $n = 100$

8.2.2 Amazon Product Co-Purchasing Graph

To validate that our previous findings were not just due to the structure of the DBLP coauthor graph, we conduct the same tests with an unrelated network. We chose an Amazon product co-purchasing graph, which had a similar number of vertices and edges compared to the DBLP coauthor graph. The resulting Triangle-Vertex-Deletion instance only contains 667129 edges and thus should be structurally distinct, compared to the instance obtained from the DBLP coauthor graph. As seen in table 2 and figure 2, the algorithm achieves a mean estimated ratio of 1.3136 and behaves the same as for the DBLP coauthor graph.

8.3 Cluster Vertex Deletion

Another derivative problem of Hitting Set is the Cluster Vertex Deletion problem. The problem can be formulated as follows,

> **Problem Name:** Cluster Vertex Deletion
> **Given:** A graph $G = (V, E)$ and a non-negative integer k.
> **Output:** Is there a set $C \subseteq V$ of size at most k, such that $G \setminus C$ is a cluster graph.

A cluster graph is a union of disjoint complete graphs which is equivalent to a graph that does not contain an induced path on three vertices, also called a P_3. These P_3 can be interpreted as edges in a hypergraph. A hitting set of size at most k on this hypergraph can be used to transform the original standard graph into a cluster graph, removing at most k vertices.

We tested our algorithm on the *Rome Graphs*, a collection of 11534 undirected graphs used and made available by Di Battista et al. in [6]. We ran the algorithm ten times for each derived problem instance and picked the run with the lowest hitting set size to include during calculation of the statistics. The results can be seen in table 3. The algorithm achieved a mean ratio of 1.1188 with a standard deviation of about 0.1. We also recorded a ratio of at most 2 for the selected problem instances. It can be observed and inferred from the low ratio that the most applied rules are exact rules.

8.4 Comparison with LP rounding based Algorithms

There exist randomized approximation algorithms for $d - $ Hitting Set that rely on LP rounding, which promise quite competitive approximation ratios. They are also fast since most of the work happens during
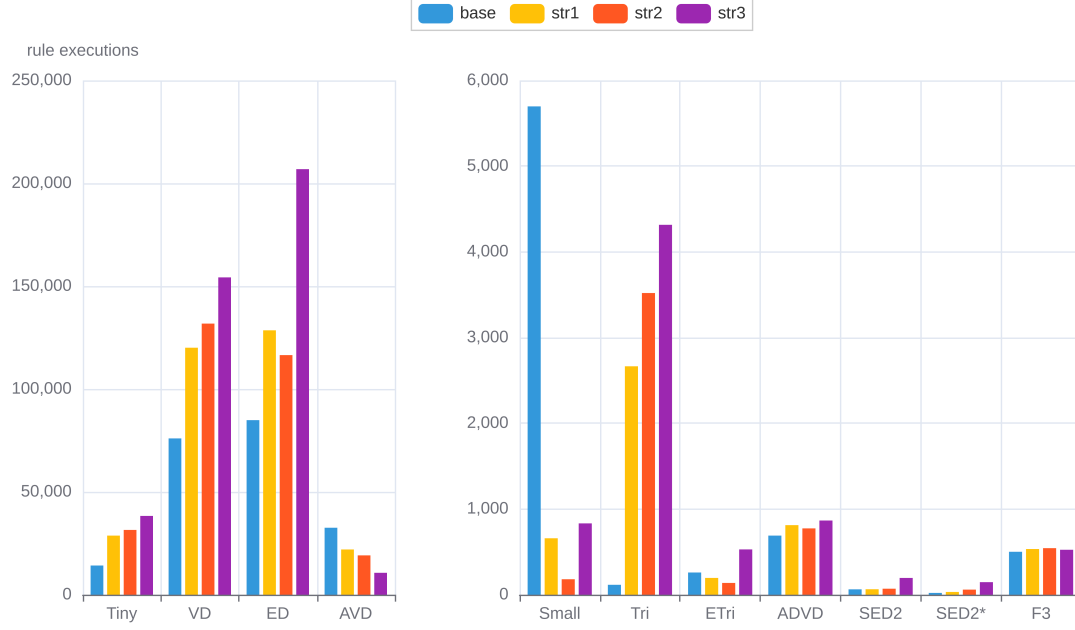
Figure 2: Mean number of rule executions for TRIANGLE VERTEX DELETION on Amazon product co-purchasing graph per rule strategy, $n = 100$

the solving of the LP. We will begin by explaining the ILP/LP formulation of HITTING SET and its dual problem SET COVER, since one of the algorithms is actually a SET COVER algorithm. After that we will briefly examine the two proposed algorithms and end with a comparison to our algorithm. Note that we will sometimes use slightly different notation and variable names than the original papers to avoid confusion (for example Δ is defined in both, in one as maximum vertex degree and the other as maximum edge size).

Let $G = (V, E)$ be a hypergraph with $n := |V|$ and $m := |E|$. Let $V = \{v_1, v_2, \dots\}$ and $E = \{e_1, e_2, \dots\}$ be the set of vertices and edges respectively. Abusing notation, we also allow referring to a vertex or edge using its index, i.e. vertex 1 and vertex v_1 shall mean the same. The HITTING SET problem can now be formulated as an integer linear program as follows,

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} x_i \\
\text{subject to} \quad & \sum_{i \in e_j} x_i \geq 1 \quad \forall e_j \in E \\
& x_i \in \{0, 1\} \quad \forall i \in [n]
\end{aligned}
$$

The LP relaxation of this ILP would allow for x_i to be any value in the interval $[0, 1]$ for all $i \in [n]$. The dual to the problem, the SET COVER problem, can be formulated similarly for the unweighted case,

| | (a) base rule strategy | | | |
	ratio	$|C|$	est. opt	time
mean	1.7466	95256	54539	3 sec
std	0.0003	82	41	0 sec
min	1.7458	95032	54433	3 sec
median	1.7466	95269	54543	3 sec
max	1.7474	95441	54632	4 sec

| | (b) rule strategy 1 | | | |
	ratio	$|C|$	est. opt	time
mean	1.4731	86359	58624	6 sec
std	0.0005	61	36	0 sec
min	1.4719	86213	58534	6 sec
median	1.4731	86359	58627	6 sec
max	1.4741	86496	58723	6 sec

| | (c) rule strategy 2 | | | |
	ratio	$|C|$	est. opt	time
mean	1.4215	84813	59666	38 sec
std	0.0005	58	32.3300	0 sec
min	1.4206	84670	59586	37 sec
median	1.4215	84814	59664	38 sec
max	1.4227	84927	59740	39 sec

| | (d) rule strategy 3 | | | |
	ratio	$|C|$	est. opt	time
mean	1.3136	80829	61532	166 sec
std	0.0006	59	34.6300	2 sec
min	1.3121	80684	61444	160 sec
median	1.3136	80836	61535	166 sec
max	1.3149	80943	61616	171 sec

Table 2: Results for Triangle Vertex Deletion on Amazon product co-purchasing graph; $n = 100$

	ratio	Tiny	VD	ED	Small	Tri	ETri	AVD	ADVD	SED2	SED2*	F3	$|C|$	est. opt
mean	1.1188	13.77	31.58	84.01	0.21	0.05	0.01	1.38	0.07	0	0.08	0	18	16
std	0.0962	7.20	15.75	56.79	0.45	0.23	0.09	1.39	0.26	0.07	0.27	0.01	9	8
min	1	0	0	0	0	0	0	0	0	0	0	0	2	2
median	1.1111	12	28	75	0	0	0	1	0	0	0	0	16	14
max	2	32	69	310	5	2	1	9	2	1	2	1	40	33

Table 3: Results for Cluster Vertex Deletion on graphs from the Rome Graphs collection. Hitting sets were computed ten times for each graph and the run with the lowest hitting set size was chosen for final calculation.

$$\begin{aligned}
\text{minimize} \quad & \sum_{j=1}^{m} x_j \\
\text{subject to} \quad & \sum_{j:i\in e_j} x_j \geq 1 \quad \forall i \in [n] \\
& x_j \in \{0,1\} \quad \forall j \in [m]
\end{aligned}$$

And again, the LP relaxation of this ILP would allow for x_j to be any value in the interval $[0,1]$ for all $j \in [m]$.

It might not obvious at first that Set Cover is the dual of Hitting Set, but formulating both problems informally gives it away in an understandable way. One problem tries to cover sets with elements, while the over aims at covering elements with sets. For a formal reduction from Hitting Set to Set Cover let $G = (V, E)$ be a hypergraph and C be a subset of V. Let $S_v = \{e \in E \mid v \in e\}$ for $v \in V$, i.e. the set of edges that are incident to a vertex v. Now C is a hitting set if and only if $S = \{e \in S_v \mid v \in C\}$ is a set cover. Let us assume that C is a hitting set for G, then it holds that,

$$\forall e \in E \mid e \cap C \neq \emptyset \quad \Leftrightarrow \quad \forall e \in E \; \exists v \in C \text{ s.t. } e \in S_v \quad \Leftrightarrow \quad \bigcup_{v \in C} S_v = E$$

The first algorithm we look at was proposed by Ouali et. al. in [7]. Let $G = (V, E)$ be a hypergraph. The algorithm starts by solving the LP relaxation of the hitting set ILP for G. Let $\mathrm{Opt}^* = \sum_{i=1}^n x_i^*$ be an optimal solution to the LP relaxation. With that solution construct the following four sets:

$$S_0 = \{i \in [n] \mid x_i^* = 0\} \qquad\qquad S_\geq = \left\{i \in [n] \mid 1 \neq x_i^* \geq \tfrac{1}{\lambda}\right\}$$

$$S_1 = \{i \in [n] \mid x_i^* = 1\} \qquad\qquad S_< = \left\{i \in [n] \mid 0 \neq x_i^* < \tfrac{1}{\lambda}\right\}$$

Remove all vertices in S_0 from V and remove the vertices in S_0 from all edges. Then put all vertices in S_1 and $S_\geq$ into the hitting set C, by removing all vertices in $S_1 \cup S_\geq$ from V and removing all edges incident to $S_1 \cup S_\geq$ from E. Next is the randomized rounding step. For all vertices $i \in S_<$ include i in the hitting set with probability $\lambda \cdot x_i^*$, independently for all i. If $|E| = 0$ return the hitting set C. Else, as long as $|E| > 0$, select a vertex from an uncovered edge and put it into the hitting set C. We will further elaborate on the definition of λ when discussing the test results of the algorithms.

Next, we look at the algorithm proposed by Saket and Sviridenko in [8]. Let V be a ground set and $E \subseteq 2^V$ a set system. We also want to define two variables, Δ the maximum number of sets any element in V is contained in and d the maximum size of any set in E. The algorithm starts by solving the LP relaxation to the SET COVER ILP for (V, E). Let $\mathrm{Opt}^* = \sum_{j=1}^m x_j^*$ be an optimal solution to the LP relaxation. For all $j \in [m]$, choose to include set e_j with probability $p_j = \min\left\{1, \alpha\Delta \cdot x_j^*\right\}$, where $\alpha = 1 - e^{-\frac{\ln d}{\Delta - 1}}$. Let I^r be the set of elements of V that are uncovered. For all $v \in I^r$, choose a set with the lowest weight in E that contains v and include it in the cover.

We used PuLP[9] to model the linear programs in Python and implemented the algorithms in Python as well. Both algorithms were tested with three open source solvers GLPK[10], CLP[11] and HiGHS[12] a comparatively newer solver that uses a parallel dual simplex method developed by Q. Huangfu and J. A. J. Hall [13]. Table 4 shows the results for both algorithms on the TRIANGLE VERTEX DELETION instance of the Amazon product co-purchasing graph, which is a 3-uniform hypergraph with maximum vertex degree of 551. Algorithm (1) promises a hitting set size of $|C| \leq d(1 - \frac{d-1}{8\Delta}) \cdot \mathrm{Opt}^*$ with probability $\frac{3}{4}$, under the assumption that the hypergraph is d-uniform and $3 \leq d \leq \frac{16}{3}\Delta$. Our hypergraph fulfills both of these requirements, which should result in a ratio of at most $3(1 - \frac{3-1}{8 \cdot 551}) \cdot \mathrm{Opt}^* = 2.9986 \cdot \mathrm{Opt}^*$. This upper bound is only valid for $\lambda = l(1 - \epsilon)$ where $\epsilon = \frac{l\mathrm{Opt}^* - |S_1|}{2m}$. Algorithm (2) promises a hitting set size of $|C| \leq ((\Delta - 1)(1 - e^{-\frac{\ln d}{\Delta - 1}}) + 1) \cdot \mathrm{Opt}^*$. Applied to our set cover instance, this yields a ratio of $((3 - 1)(1 - e^{-\frac{\ln 551}{3 - 1}}) + 1) \cdot \mathrm{Opt}^* = 2.9148 \cdot \mathrm{Opt}^*$.

Algorithm (1) slightly outperforms algorithm (2) in terms of hitting set size. The LP solution obtained with the HiGHS solver using the interior point method yields the smallest hitting set for both algorithms, with a execution time of about 1 minute. The hitting set obtained with the GLPK solver comes close but at a way higher execution time of about 1 hour. The CLP solver is not very well suited for the algorithms on this specific problem instance. Execution time as well as the resulting hitting set size are worse compared to HiGHS and GLPK. Compared to our algorithm, both algorithm (1) and (2) yield hitting sets with bigger size.

update with new test results using other solvers

We also conducted the same test from the Cluster Vertex Deletion section with both algorithms using the HiGHS solver with the interior point method. The results can be seen in table 5. Both algorithms perform very similarly, with almost identical values for Opt^* and the hitting set sizes. Both have a high maximum hitting set size, which is almost double the size compared to our algorithm. The Opt^* values for both are very similar to the estimated optimum values that our algorithm computes, but the difference between optimum and actual hitting set size is smaller with our algorithm. This is can be observed for every statistic we calculated. The most extreme case is the standard deviation for hitting set size and Opt^*. We can see that

algorithm	solver	method	ratio UB	actual ratio	Opt*	|C|	solver time
Hitting	GLPK	simplex		1.5814	64452.6941	101928	1 hour
Set LP	CLP	simplex	2.9986	1.7838	64452.6992	114972	50 sec
(1)	HiGHS	simplex		1.6709	64452.6993	107697	17 sec
	HiGHS	ipm		1.5713	64452.6993	101276	76 sec
Set	GLPK	simplex		1.5944	64452.6940	102762	1 hour
Cover	CLP	simplex	2.9148	1.8068	64452.6992	116455	47 sec
LP (2)	HiGHS	simplex		1.6826	64452.6993	108447	19 sec
	HiGHS	ipm		1.5797	64452.6993	101820	68 sec

Table 4: Results for LP based rounding algorithms for HITTING SET and SET COVER on TRIANGLE VERTEX DELETION instance of Amazon product co-purchasing graph

	ratio UB	ratio	|C|	Opt*
mean	2.9697	1.4246	25	17
std	0.0197	0.4268	16	8
min	2.7500	1	2	2
median	2.9758	1.3565	21	15
max	2.9925	2.5714	80	35

(a) Hitting Set LP (1)

	ratio UB	ratio	|C|	Opt*
mean	2.6132	1.4050	24	17
std	0.1089	0.4101	16	8
min	1.8453	1	2	2
median	2.6408	1.3220	21	15
max	2.8000	2.5385	76	35

(b) Set Cover LP (2)

Table 5: Results for CLUSTER VERTEX DELETION using LP rounding based algorithms on graphs from the Rome Graphs collection. Same method as previous CVD benchmark.

the standard deviation for the hitting set size with a value of 16, is twice as big as the standard deviation for Opt*, for both algorithms.

8.5   Preferential Attachment Hypergraphs

8.6   3-Uniform ER Hypergraphs

9   Testing

Every reduction rule is tested for their correctness with unit tests. We create small graphs in these tests, which contain structures, which the rules are targeting. We then test for the elements in the partial solution, number of edges/vertices left in the graph and degree of the vertices left.

10   Conclusion

The implmented HITTING SET algorithm can compute hitting sets for large inputs in a reasonable amount of time. The algorithm also yields smaller hitting sets compared to LP randomized rounding based algorithms.

10.1   Further Work

The only part of the algorithm that is parallelized is the edge domination rule. The main algorithm itself could also be parallelized, when using the incremental variant. The graph could at some point during the run of the algorithm consist of multiple connected components. These components could then be processed in parallel. Every component would yield its own number of rule executions and hitting sets, which are aggregated at the end. The question is, if the overhead of finding these components is greater than the possible speed up.

There already exist algorithms for vertex and edge domination, which run on a GPU, see [ref_gpu]. It would be interesting to know if such algorithms also exist for the other domination type reduction rules, namely the approximate (double) vertex domination rule. Our implementation of these rules rely on arguments about vertex degrees and adjacency between vertices, which is similar to the standard vertex domination implementation.

A   Appendix

	ratio	Tiny	VD	ED	Small	Tri	ETri	AVD	ADVD	SED2	SED2*	F3	$|C|$	est. opt
mean	1.9196	59.99	345.23	2.40	0.46	0.04	47.26	0.03	2.02	44.47	9.16	44.33	592.46	308.68
std	0.0273	5.86	9.71	1.56	0.63	0.19	5.27	0.18	1.35	3.93	2.67	6.54	8.59	3.51
min	1.8344	43	316	0	0	0	29	0	0	31	2	26	564	298
median	1.9199	60	345	2	0	0	47	0	2	44	9	45	592	309
max	2.0364	79	375	8	3	1	61	2	7	57	19	70	622	320

(a) random edge in F3 rule

	ratio	Tiny	VD	ED	Small	Tri	ETri	AVD	ADVD	SED2	SED2*	F3	$|C|$	est. opt
mean	1.8690	58.28	349.60	2.33	0.47	0.05	63.68	0.03	2.01	42.57	8.59	25.51	590.66	316.06
std	0.0225	5.93	9.64	1.57	0.62	0.21	4.10	0.16	1.39	4.06	2.70	3.73	8.98	3.3400
min	1.7987	41	321	0	0	0	50	0	0	29	1	16	562	305
median	1.8675	58	350	2	0	0	64	0	2	43	9	25	590	316
max	1.9525	75	382	10	3	1	77	1	9	56	17	40	617	328

(b) F3 low degree rule

Table 6: Results for 100 random 3-uniform ER hypergraphs with an edge to vertex ratio of 3. Data was collected 10 times per graph to account for run-to-run variance.

	ratio	$|C|$	est. opt	time
mean	1.3142	80927.77	61579.82	168 sec
std	0.0006	55.84	32.98	3 sec
min	1.3128	80786	61511	161 sec
median	1.3142	80926.50	61579.50	168 sec
max	1.3157	81071	61654	176 sec

Table 7: Results for Triangle Vertex Deletion on Amazon product co-purchasing graph using rule strategy 3; F3 rule selects a random size three edge; $n = 100$

Figure 3: Mean number of rule executions for 100 random 3-uniform ER hypergraphs with an edge to vertex ratio of 3. Data was collected 10 times per graph to account for run to run variance.

Figure 4: Flamegraph of pprof CPU performance profile. Graph instance was a dense 3-uniform ER hypergraph with 1000 vertices.

References

[1] C. Spagnuolo *et al.* SimpleHypergraphs.jl. https://github.com/pszufe/SimpleHypergraphs.jl/blob/master/src/models/random-models.jl, 2020.

[2] C. Spagnuolo *et al.* Analyzing, Exploring, and Visualizing Complex Networks via Hypergraphs Using Simplehypergraphs.jl. *Internet Math.* vol. 2020. 2020, doi: 10.24166/IM.01.2020.

[3] C. Avin, Z. Lotker, Y. Nahum, and D. Peleg. Random Preferential Attachment Hypergraph, in *ASONAM '19: International Conference on Advances in Social Networks Analysis and Mining, Vancouver, British Columbia, Canada, 27-30 August, 2019*, F. Spezzano, W. Chen, and X. Xiao, Eds., ACM, 2019, pp. 398–405. doi: 10.1145/3341161.3342867.

[4] L. Brankovic and H. Fernau. Parameterized Approximation Algorithms for Hitting Set, in *Approximation and Online Algorithms*, R. Solis-Oba and G. Persiano, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 63–76.

[5] J. Yang and J. Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowl. Inf. Syst.* vol. 42, no. 1. pp. 181–213. 2015, doi: 10.1007/S10115-013-0693-Z.

[6] G. D. Battista, A. Garg, G. Liotta, R. Tamassia, E. Tassinari, and F. Vargiu. An Experimental Comparison of Four Graph Drawing Algorithms. *Computational Geometry.* vol. 7, nos. 5-6. pp. 303–325. Apr. 1997, doi: 10.1016/s0925-7721(96)00005-3.

[7] M. E. Ouali, H. Fohlin, and A. Srivastav. A Randomised Approximation Algorithm for the Hitting Set Problem. *Theor. Comput. Sci.* vol. 555. pp. 23–34. 2014, doi: 10.1016/J.TCS.2014.03.029.

[8] R. Saket and M. Sviridenko. New and Improved Bounds for the Minimum Set Cover Problem, in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 15th International Workshop, APPROX 2012, and 16th International Workshop, RANDOM 2012, Cambridge, MA, USA, August 15-17, 2012. Proceedings*, A. Gupta, K. Jansen, J. D. P. Rolim, and R. A. Servedio, Eds., in Lecture notes in computer science, vol. 7408. Springer, 2012, pp. 288–300. doi: 10.1007/978-3-642-32512-0_25.

[9] J.-S. Roy, S. A. Mitchell, C.-M. Duquesne, F. Peschiera, and A. Phillips. PuLP. https://github.com/coin-or/pulp, 2024.

[10] GLPK. https://www.gnu.org/software/glpk/glpk.html, 2024.

[11] J. Forrest. Coin-or/Clp: Release Releases/1.17.9. https://github.com/coin-or/Clp; Zenodo, 2024. doi: 10.5281/zenodo.10041272.

[12] J. Hall, I. Galabova, Q. Huangfu, L. Gottwald, and M. Feldmeier. HiGHS. https://github.com/ERGO-Code/HiGHS, 2024.

[13] Q. Huangfu and J. A. J. Hall. Parallelizing the Dual Revised Simplex Method. *Math. Program. Comput.* vol. 10, no. 1. pp. 119–142. 2018, doi: 10.1007/S12532-017-0130-5.