

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












Contents

1	Programming Language	2
2	Datastructures	2
2.1	Vertex	2
2.2	Edge	2
2.3	Hypergraph	3
3	Misc. Algorithms/Utilities	3
3.1	Edge Hashing	3
3.2	Compute Subsets of Size s	3
3.3	Two-Sum	4
4	Hypergraph Models	5
4.1	First Testing Model	5
4.2	Preferential Attachment Hypergraph Model	5
4.3	Preferential Attachment Hypergraph Model with high Modularity (Giroire et al.)	6
5	Reduction Rules	6
5.1	Executions	6
5.2	Algorithms	6
5.2.1	Tiny/Small Edge Rule	6
5.2.2	Edge Domination Rule	7
5.2.3	Vertex Domination Rule	8
5.2.4	Approximative Vertex Domination Rule	8
5.2.5	Approximative Double Vertex Domination Rule	9
5.2.6	Small Triangle Rule	10
5.2.7	Extended Triangle Rule	12
5.2.8	Small Edge Degree 2 Rule	12
5.3	Self-Monitoring	13
6	Algorithms	16
6.1	Main Algorithm	16
6.2	Incremental Frontier Algorithm	16
7	Applications and Results	17
7.1	Triangle Vertex Deletion	17
7.2	Cluster Vertex Deletion	20
7.3	Preferential Attachment Hypergraphs	20
7.4	3-Uniform ER Hypergraphs	22
8	Testing	23

9	Branching Algorithm	23
9.1	Potential Triangle Situation	23
9.2	Edge-Cover	23
9.3	Possible Optimizations	23

References	23
-------------------	-----------

Todo list

	add new AdjCount field	3
	add AdjCount explanation	3
	rewrite with new algorithm	3
	remove explicit AdjCount generation, use the one provided by the graph struct	8
	change to new Two-Sum algorithm description	10
	add algorithm for SmallEdgeDeg2	13
	remove Factor-3 Rule function paragraph	14
	TODO	16
	add new Frontier section	16
	update the ratio and chart with frontier algorithm	17
	needs proof?	17

1 Programming Language

We chose the Go 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. If the situation arises in which we do need to handle memory manually, we can utilize Go's `unsafe` package in conjunction with C-interop.

2 Datastructures

2.1 Vertex

```
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.

2.2 Edge

```
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.

2.3 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
}
```

add new AdjCount field

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 datastructure 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 $v \in V$ all vertices adjacent to v . Additionally this map will also store the amount of times such a vertex is adjacent to v .

add Adj-Count explanation

3 Misc. Algorithms/Utilities

3.1 Edge Hashing

```
func getHash(arr []int32) string
```

Time Complexity: $n + n \cdot \log(n)$, where n denotes the size of `arr`.

We start by sorting `arr` with a QUICK-SORT-Algorithm. We then join the elements of `arr` with the delimiter `"|"`, returning a string of the form `"|id0|id1|\dots|idn"`.

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 `arr` argument.

3.2 Compute Subsets of Size s

```
func getSubsetsRecMain(arr *[]int32, i int, n int, s int, data *[]int32, index int, subsets
```

rewrite with new algorithm

Time Complexity: 2^n

Let us explain all the arguments first.

- `arr` is a pointer to an array. This array is basically the input set whose subsets we want to compute.
- `i` is an index over `arr`
- `n` is the size of `arr`.
- `s` is the size of the computed subsets
- `data` is a temporary array
- `index` is an index over `data`
- `subsets` is a list that will store all the subsets of size `s` we find

The implementation looks as follows.

```
func getSubsetsRecMain(arr []int32, i int, n int, s int, data []int32, index int, subsets *list.List) {
    if index == s {
        subset := make([]int32, s)
        for j := 0; j < index; j++ {
            subset[j] = data[j]
        }
    }
}
```

```

    }
    subsets.PushBack(subset)
    return
}

if i >= n {
    return
}

data[index] = arr[i]

getSubsetsRecMain(arr, i+1, n, s, data, index+1, subsets)
getSubsetsRecMain(arr, i+1, n, s, data, index, subsets)
}

```

The algorithm is pretty simple. For every element x in `arr`, we either put x into `data` or we do not. We have two base conditions.

1. If `index = s`, then we copy the contents of `data` into an array and add it to `subsets`.
2. If $i \geq n$, then we have considered all elements for the current path of the search tree.

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 `n` is usually fixed at 3. The raised memory problems occur at values of $n \geq 10000$, thus 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 shall be called whenever we find a subset, using the found subset as an argument.

3.3 Two-Sum

Given an array 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 1: 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  $\leftarrow$  map[N]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 over 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 is accepting a map instead of an array as its first parameter.

4 Hypergraph Models

4.1 First Testing Model

```
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 amount 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:

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

We then store the result of $size(\mathbf{r})$ in a variable `d`. We then randomly pick vertices in the half-open interval $[0, n)$, until we have picked `d` many distinct vertices. We then check if an edge containing these vertices already exists. If it 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.

Using an established model like the Erdős–Rényi Model would be more favourable. But since we just needed something tangible to start testing, this “model” will suffice for the time being.

One could also look into generating random bipartite graphs that translate back to a hypergraph with the desired vertex and edge numbers.

4.2 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

4.3 Preferential Attachment Hypergraph Model with high Modularity (Giroire et al.)

Looking at the first preferential attachment model, one can see that the resulting graph will be one big community.

We therefore also considered a model with high modularity proposed by Giroire et al.[4].

5 Reduction Rules

The usual signature of a reduction rule looks as follows:

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

We take both a `HyperGraph` struct `g` and a `Set` `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.

5.1 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 in regards to memory writes and execution time. It is therefore necessary 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. Construct auxilliary data structures that are used to find parts of the graph, for which the rule can be applied.
2. As long as we can apply rules do:

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

- Identify targets of the rule.
- Mutate the graph according to the rule.
- Mutate the auxilliary 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 auxilliary data structure at the same time.

5.2 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 further introduce a `map` data structure in our pseudocode with syntax `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. $\gamma[0] \leftarrow 1$. The `map` type also exposes a primitive function with the signature `delete(γ, x)`, which simply means that we want to delete the entry with key `x` from our map. When iterating over a map in a `for`-loop we destructure the entries into a `(key, value)`-pair, e.g. `for (_, v) ∈ map do`. Note that unused values are omitted with the underscore symbol.

5.2.1 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.

$O(|E|^2)$ **Algorithm.** 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 2: 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  $exec$ 

```

5.2.2 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 possibly 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:

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

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

$$\begin{aligned}
T &= |E| \cdot 3 \cdot \log(3) + (|E| \cdot (11 + 24 \cdot \log(3))) \\
&= |E| \cdot (3 \cdot \log(3) + (11 + 24 \cdot \log(3)))
\end{aligned}$$

Lemma 5.1 *Algorithm 3 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 ≈ 2 on a six-core CPU on a pseudo-random graph with one million vertices and two million edges.

Algorithm 3: 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    $subsets \leftarrow \text{getSubsetsRec}(e, 2)$ 
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  $exec$ 
```

5.2.3 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.

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

5.2.4 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.

$O(|V| \cdot |E|)$ **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 over the AdjCount map of the graph, referring to the current value in the iteration as $\text{AdjCount}[v]$. We then use the TWO-SUM-Algorithm to compute and return the first pair (a, b) in $\text{AdjCount}[v]$, s.t. for (a, b) holds,

$$\text{AdjCount}[v][a] + \text{AdjCount}[v][b] = v\text{Deg}[v] + 1$$

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

Lemma 5.2 *The outlined procedure above is correct, under the assumption that the underlying graph does not contain any duplicate edges.*

remove explicit Adj-Count generation, use the one provided by the graph struct

Algorithm 4: 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  $val = vDeg[v]$  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  $exec$ 
```

Proof. Let G be a hypergraph. We first remove all edges of size one with the *Tiny Edge Rule*. We then construct our map $AdjCount$. Let v be an entry in $AdjCount$ and $sol = (a, b)$ the result of calling our TWO-SUM implementation 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 and does not contain $\{v, a, b\}$, 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 it must be, that the assumption that $\{v, a, b\}$ does not exist, is 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 $\{v, a, b\}$ can not 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 .

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 amount 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 $|V| + |V|^2 \cdot |E|$.

5.2.5 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

Algorithm 5: 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      $sol, ex \leftarrow TWO-SUM(count, deg(v) + 1)$ 
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  $exec$ 
```

if 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 an approximative double vertex domination situation. If $tsHashes[sol] = \emptyset$, map $tsHashes[sol]$ to $tsHashes[sol] \cup \{x\}$.

Lemma 5.3 *Algorithm 6 is correct.*

Proof. Let $G = (V, E)$ be a hypergraph. Let x be the current vertex in the algorithm's iteration over V . If $T = TWO-SUMALL(AdjCount[x], deg(x))$ is empty, then x will not be able to trigger an 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 an 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

change to
new Two-
Sum algo-
rithm de-
scription

5.2.6 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,

Algorithm 6: 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    $tsHashes \leftarrow \text{map}[2^E]2^V$ 
6   for  $x \in V$  do
7     for  $sol \in \text{Two-SUMALL}(AdjCount[x], deg(x))$  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            $tsHashes[sol] = tsHashes[sol] \cup \{x\}$ 
29         else
30            $tsHashes[sol] = tsHashes[sol] \cup \{x\}$ 
31 return  $exec$ 

```

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 don't find any more triangles. This simulates calling the rule exhaustively, while achieving a linear time complexity.

Algorithm 7: : 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   $adjList \leftarrow \text{map}[V]2^V$ 
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      $subsets \leftarrow \text{getSubsetsRec}(val, 2)$ 
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  $exec$ 

```

5.2.7 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.

5.2.8 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.

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

```

5.3 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 amount of vertices put into the partial solution by a single rule execution. And B denotes the amount 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)
}
```

We conducted some preliminary testing on graphs with 10, 100, 1000 and 10000 vertices. We calculated the estimated approximation factor for these graphs. We did this for ratios $r = \frac{|E|}{|V|}$ of 1 to 20.

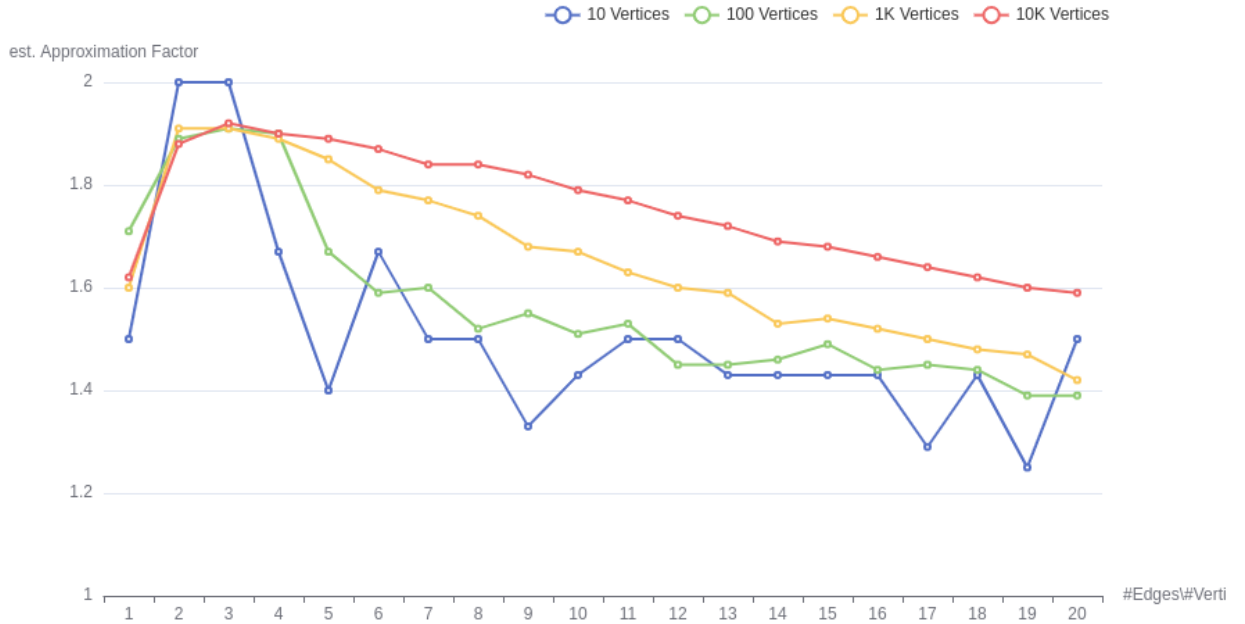


Figure 1: Estimated Approximation Factor for Graphs with 10, 100, 1K, 10K Vertices for $r \in [1, 20]$

We also looked at the number of rule executions for $r = 1$ and $r = 10$.

Note that the reduction rules alone are sufficient to compute a Hitting-Set for our graphs. The est. ratio is quite low, since we do not need to put whole size 3 edges into our hitting set. This is due to the way our random-graph model works. This will not work for all graph classes, namely 3-uniform graphs.

We can observe a large performance hit, once our graph is near 3-uniform. We present some ideas that could potentially speedup the execution for these graphs.

Factor-3 Rule Targeting. The Factor-3 rule will select the to be removed edge at random. We could alternatively choose an edge e , s.t. the removal of e will allow the execution of the Vertex Domination Rule. This could potentially lower the number of Factor-3 Rule executions quite significantly.

remove
Factor-3
Rule func-
tion para-
graph

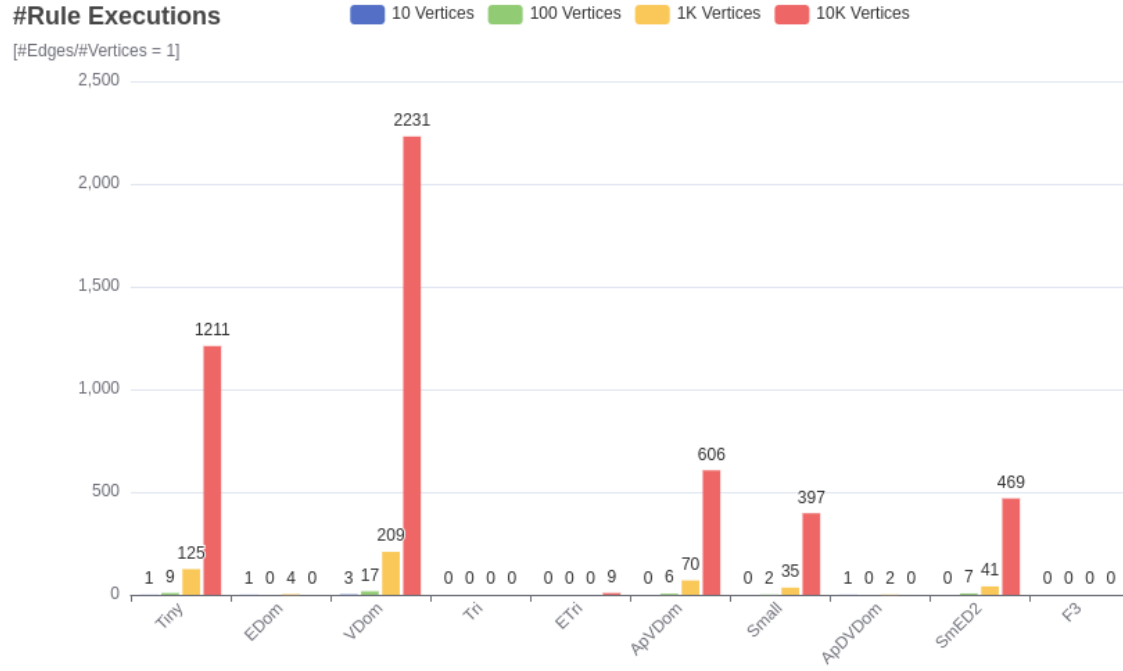


Figure 2: Number of Rule Executions, $r = 1$

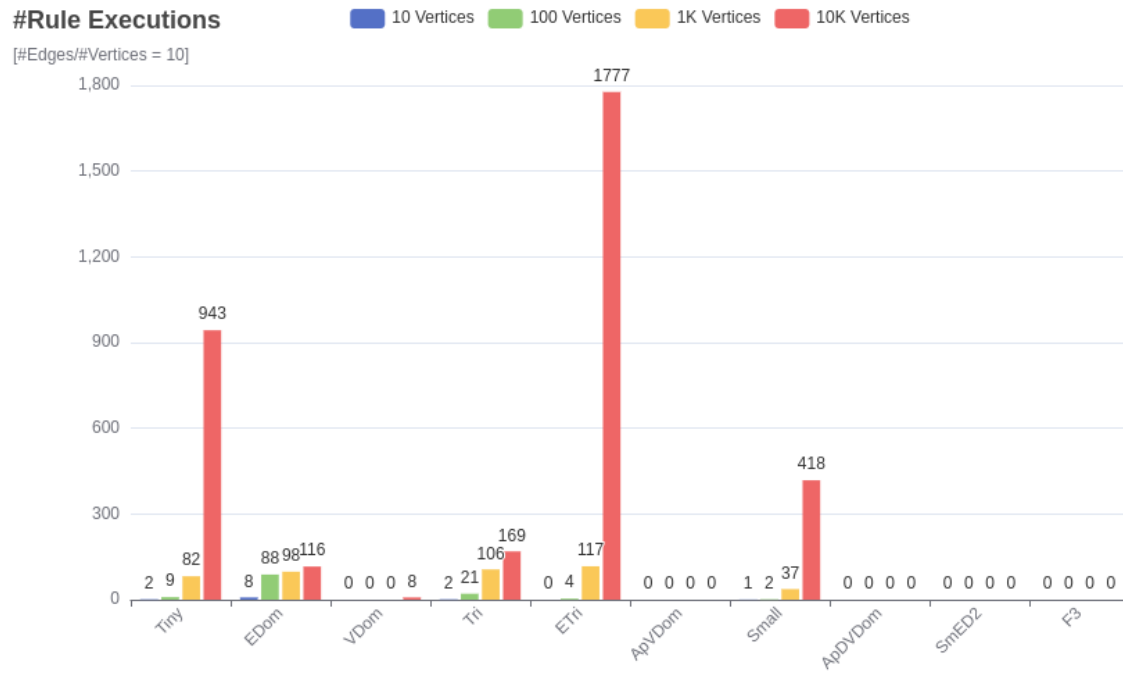


Figure 3: Number of Rule Executions, $r = 10$

6 Algorithms

6.1 Main Algorithm

TODO

6.2 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 a lot of 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 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 3 edge e , which will trigger a vertex domination situation if removed. 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.

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.

add new
Frontier section

Algorithm 9: 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  $e \leftarrow \text{F3TargetLowDegree}(G)$ 
6  $H \leftarrow \text{GetFrontierGraph}(G, l, e)$ 
   /*  $H$  will act as a mask over  $G$ , the rules will only be applied to vertices/edges in
    $H$  */
7 for  $|E| > 0$  do
8   Apply all reduction rules exhaustively on  $H$  mutating  $H, G$  and  $C$ 
9    $exp \leftarrow$  vertices of edges adjacent to removed or modified edge
10  if  $|exp| > 0$  then
11     $H \leftarrow \text{ExpandFrontier}(G, l, exp)$ 
12    continue
13   $e \leftarrow \text{F3TargetLowDegree}(G)$ 
14  if  $e = \emptyset$  then
15    continue
16   $H \leftarrow \text{ExpandFrontier}(G, l, e)$ 
17 return  $C$ 
```

7 Applications and Results

7.1 Triangle Vertex Deletion

[PROBLEM DEFINITION]

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.

We tested the algorithm 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, due to RAM limitations of the test rig. 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.

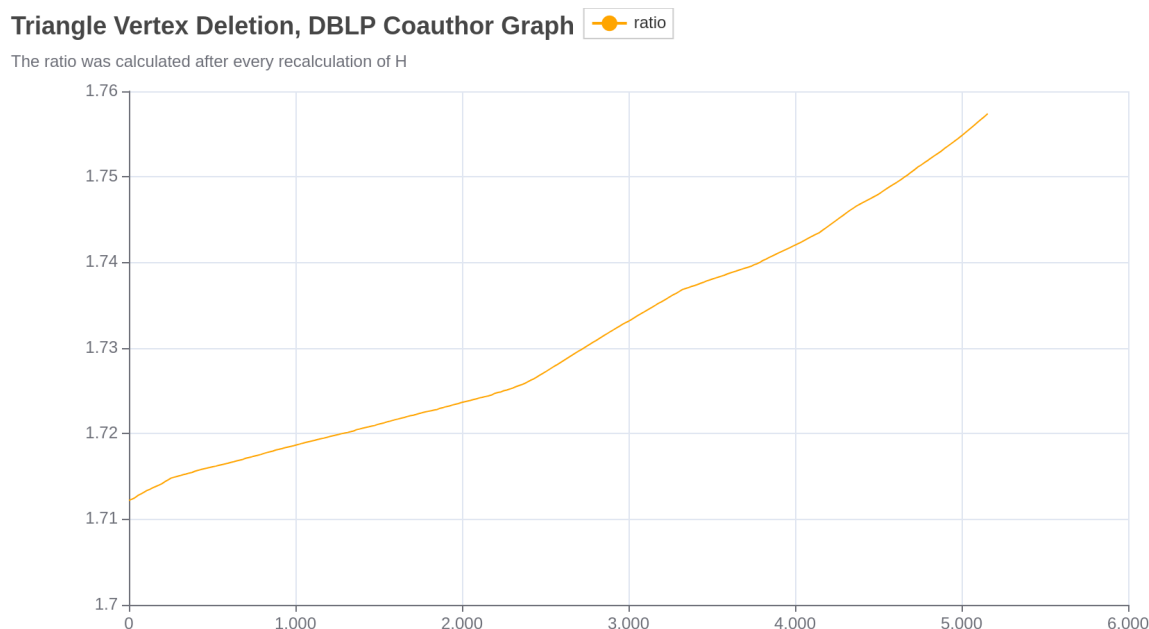


Figure 4: Triangle Vertex Deletion on a DBLP coauthor graph, the ratio was calculated after every recalculation of H

The algorithm achieved a estimated approximation ratio of ≈ 1.758 . Using the Frontier Algorithm, the previous execution time of around 25 minutes could be reduced to about 16 seconds.

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 Triangle Vertex Deletion and thus 3-Hitting-Set instance. The expansion steps will therefore only expand the auxilliary graph H inside these community. This drastically reduces the amount of edges the algorithm has to process per iteration. One can see the opposite happen in really dense hypergraphs without community structure. Due to the high density, even a small expansion two layers deep will add all edges of G to H .

update the
ratio and
chart with
frontier algo-
rithm

needs proof?

Next we looked at random ER graphs. We conducted the tests on graphs with 1000 vertices and an edge to vertex ratio of 10 to 25. Data was collected for 100 graphs per EVR.

#Rule Executions

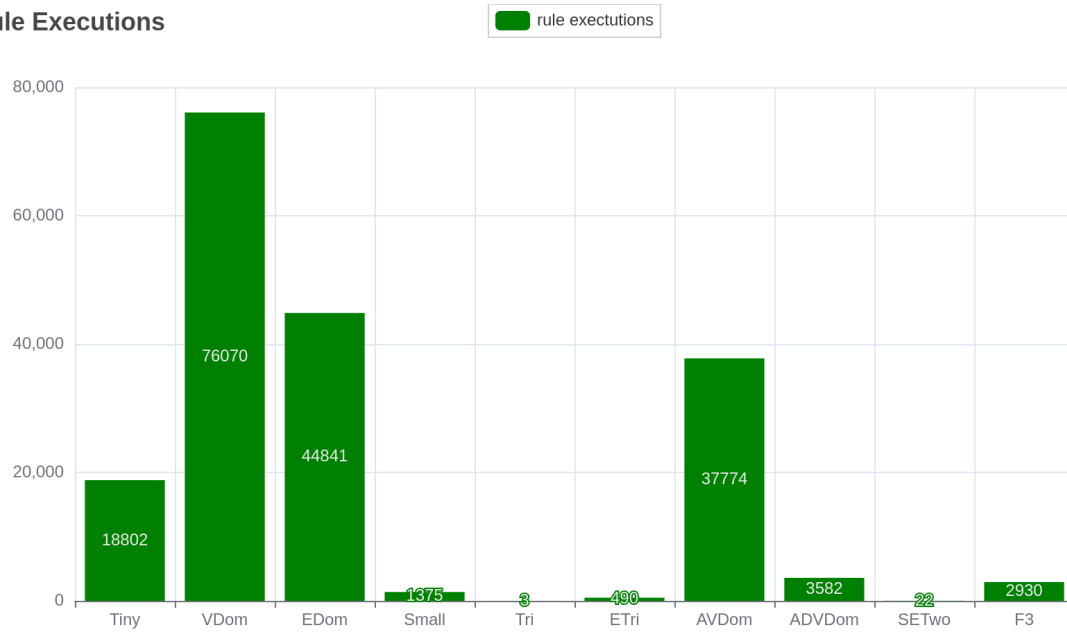


Figure 5: Triangle Vertex Deletion on a DBLP coauthor graph, amount of rule executions

Triangle Vertex Deletion for ER Graphs

1000 vertices, Edge\Vertex ratio of 10-25, 100 graphs per EVR

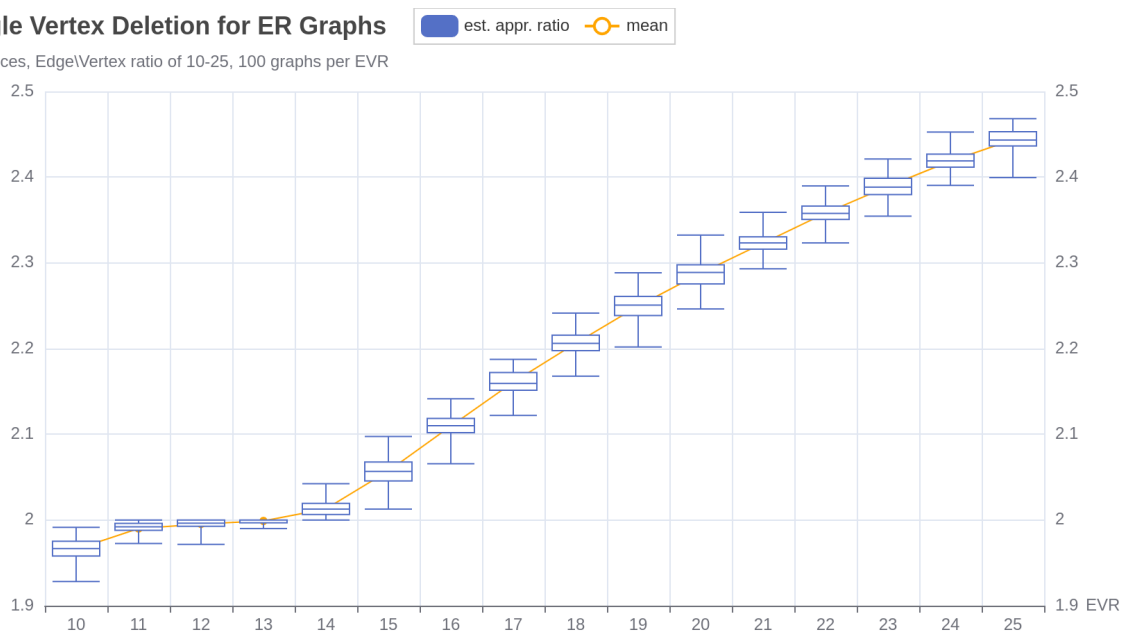


Figure 6: Triangle Vertex Deletion on ER graphs, est. appr. ratios

#Rule Executions

EVR=10 for input ER graph

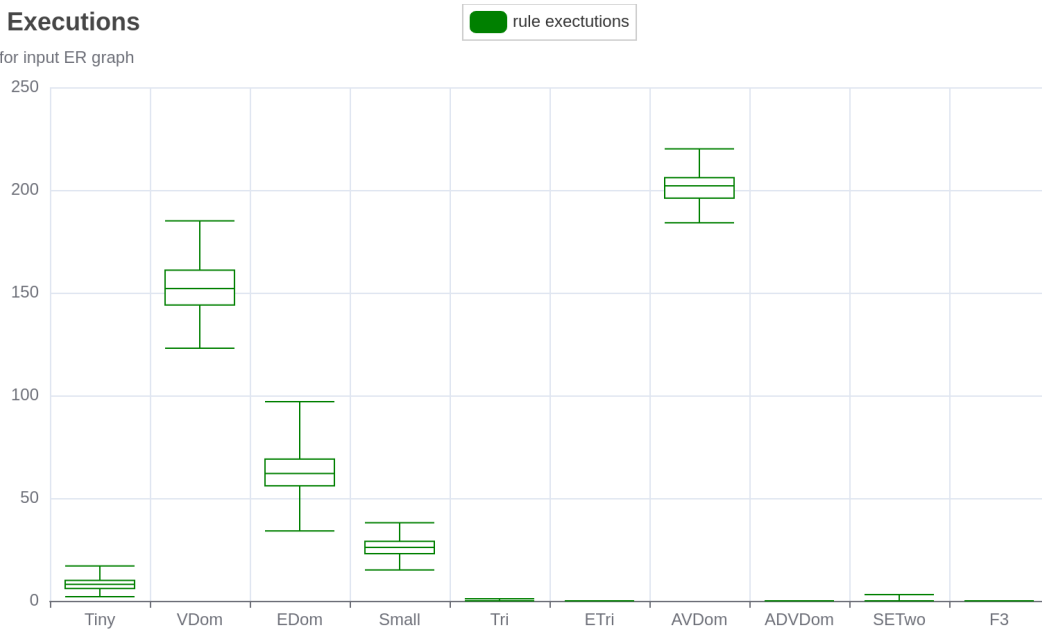


Figure 7: Triangle Vertex Deletion on ER graphs, amount of rule executions for EVR=10

#Rule Executions

EVR=25 for input ER graph



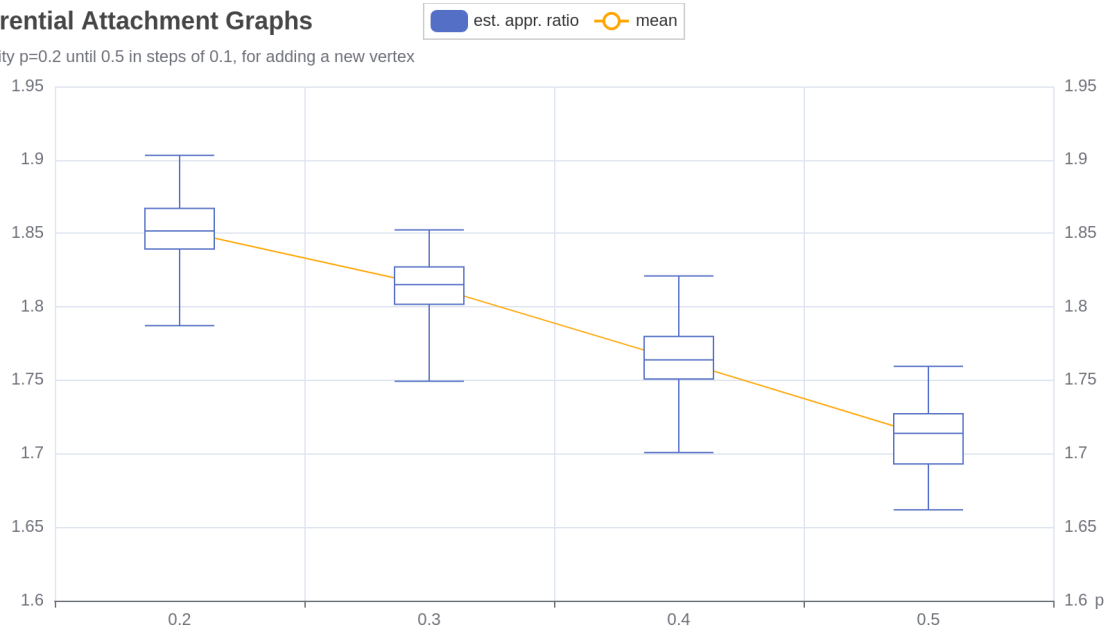
Figure 8: Triangle Vertex Deletion on ER graphs, amount of rule executions for EVR=25

7.2 Cluster Vertex Deletion

7.3 Preferential Attachment Hypergraphs

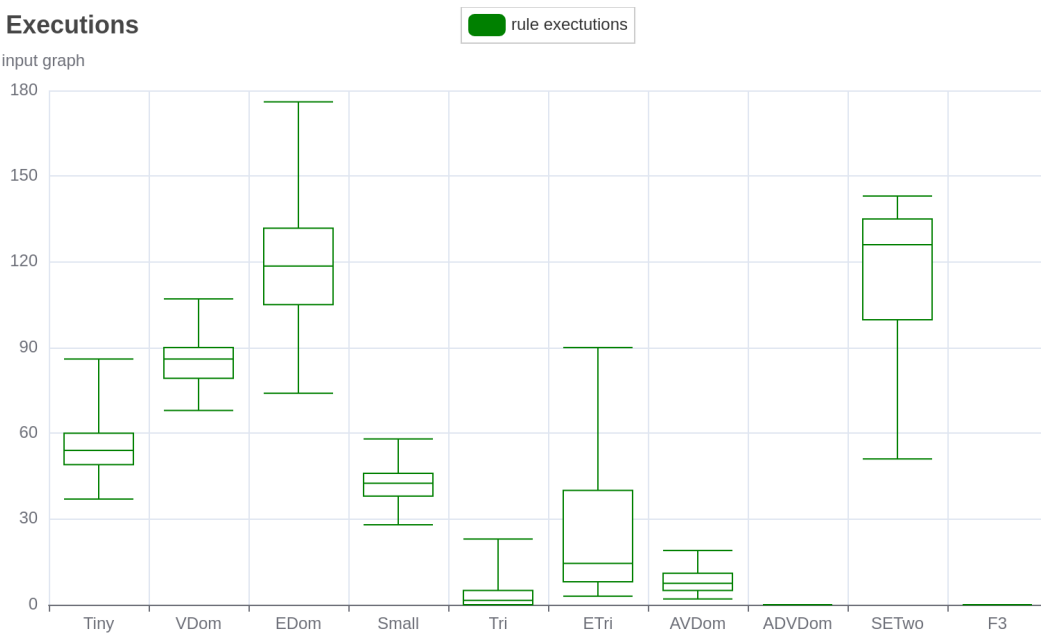
Preferential Attachment Graphs

Probability $p=0.2$ until 0.5 in steps of 0.1, for adding a new vertex



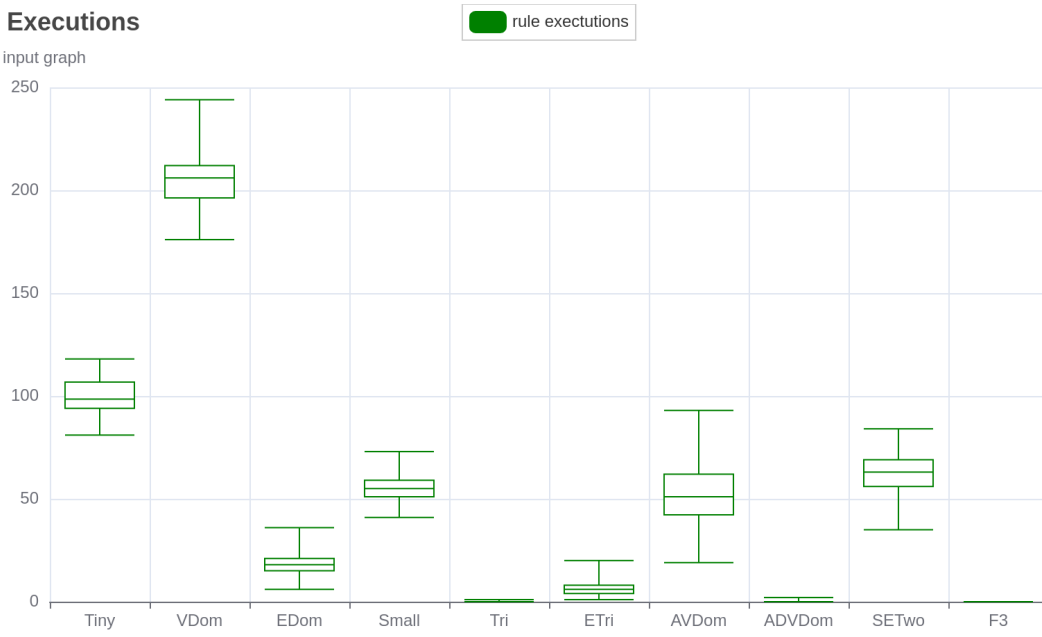
#Rule Executions

$p=0.2$ for input graph



#Rule Executions

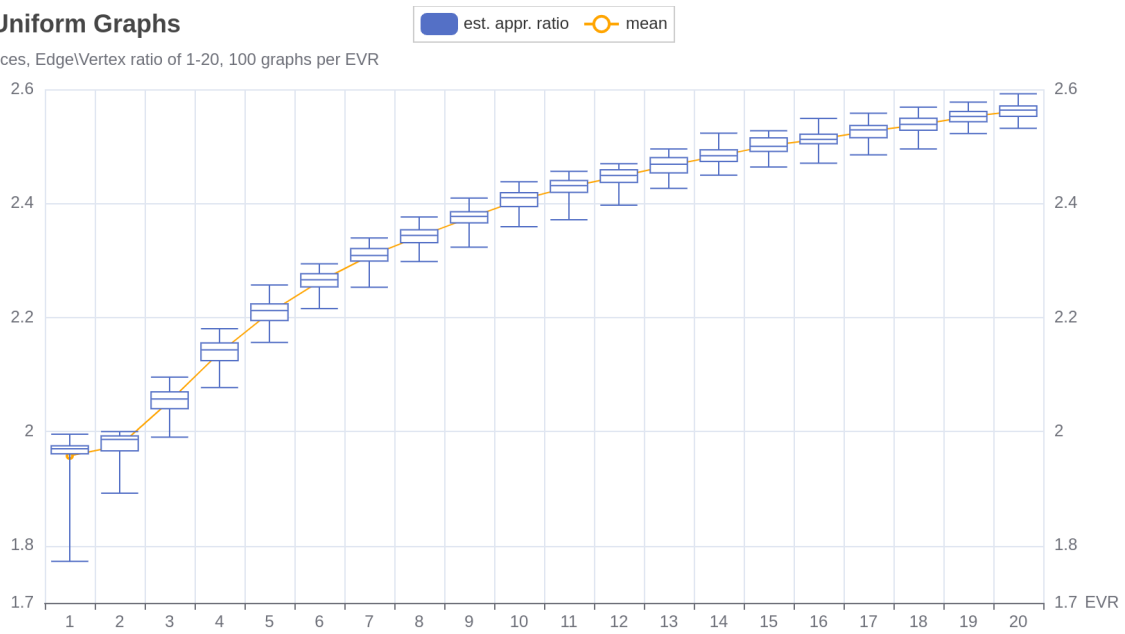
p=0.5 for input graph



7.4 3-Uniform ER Hypergraphs

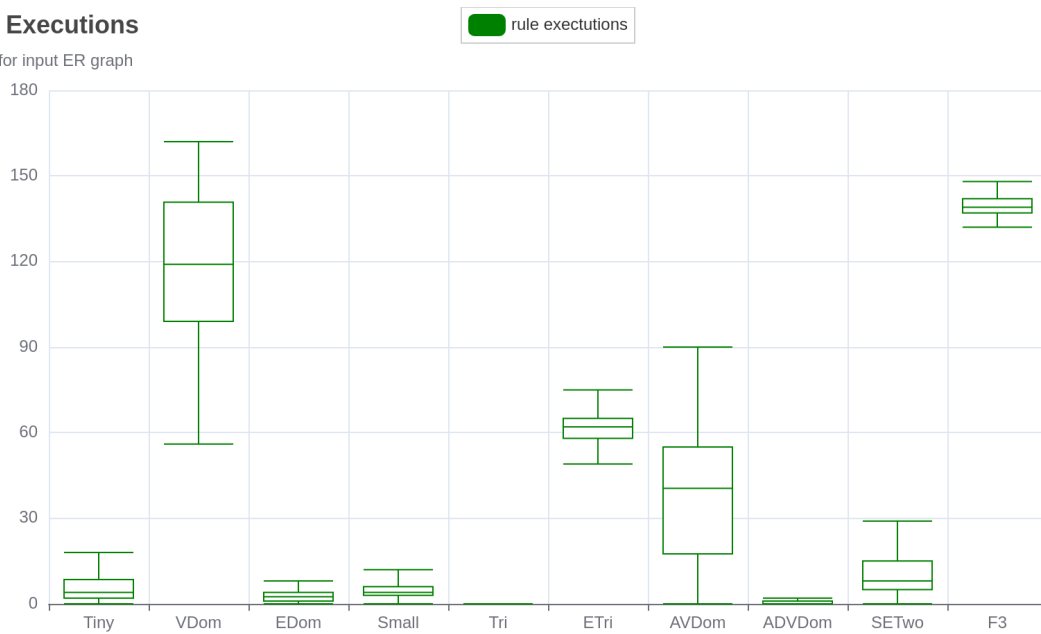
ER 3-Uniform Graphs

1000 vertices, Edge\Vertex ratio of 1-20, 100 graphs per EVR



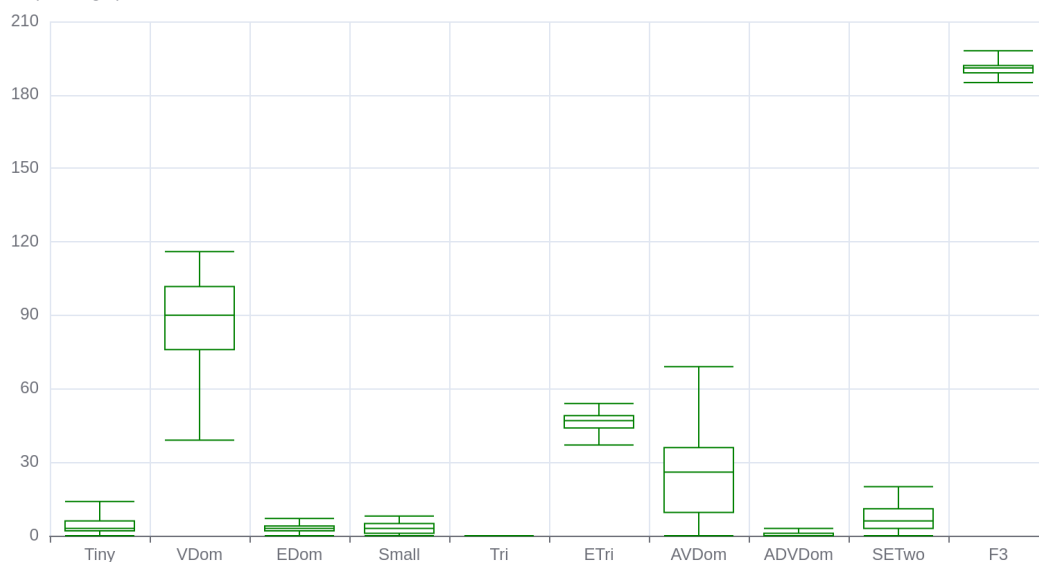
#Rule Executions

EVR=10 for input ER graph



#Rule Executions

EVR=20 for input ER graph



8 Testing

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

9 Branching Algorithm

9.1 Potential Triangle Situation

9.2 Edge-Cover

If our problem instance is simple, we can solve it in polynomial time using an edge cover algorithm for standard graphs. We transform the hypergraph of our instance to a standard graph as follows:

- Let v be a hypergraph vertex with $\delta(v) = 2$ and e_0, e_1 be the two edges that v is incident to.
- We then add two vertices to the auxilliary graph identified by e_0 and e_1 . At last we add an edge $\{e_0, e_1\}$ identified by v .

9.3 Possible Optimizations

Right now we sometimes use the Go `append` function, to add elements to a slice. Calls to `append` are more expensive than an ordinary assignment to a fixed size slice.

Using fixed size slices could improve performance slightly in cases where we know the maximum amount of elements we will add to a slice.

References

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

- [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, and D. Peleg, “Random preferential attachment hypergraphs,” *CoRR*, vol. abs/1502.02401, 2015, Available: <http://arxiv.org/abs/1502.02401>
- [4] F. Giroire, N. Nisse, T. Trollet, and M. Sulkowska, “Preferential attachment hypergraph with high modularity,” *Network Science*, vol. 10, no. 4, pp. 400–429, 2022, doi: 10.1017/nws.2022.35.
- [5] J. Yang and J. Leskovec, “Defining and evaluating network communities based on ground-truth,” *CoRR*, vol. abs/1205.6233, 2012, Available: <http://arxiv.org/abs/1205.6233>