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

Contents

Todo list

%

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
}
```

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.

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 "$|id_0|id_1|\ldots|id_n|$". 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 `arr` argument.

3.2 Compute Subsets of Size s

Given an array arr and an integer s, compute all subsets of arr of size s. rewrite with new algorithm

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

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

$data \leftarrow [\,]$
$n \leftarrow |arr|$
$last \leftarrow s - 1$
$FRecursive(0, 0)$
func `FnRecursive`(i, $next$):
  for $i \leftarrow next$ **to** n **do**
    $data[i] \leftarrow arr[j]$
    if $i = last$ **then**
      $subsets.push(data)$
    else
      $FRecursive(i + 1, j + 1)$

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 $n > 10000$, 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`.

We start by creating a map called $lookup$. Iterating other the elements of arr, we check if the entry $lookup[t - arr[i]]$ exists.

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

$lookup \leftarrow \texttt{map}[\mathbb{N}]\mathbb{N}$
for $i \leftarrow 0$ **to** $len(arr)$ **do**
    if $lookup[t - arr[i]]$ *exists* **then return** $(i, lookup[t - arr[i]]), true$
    else
        $lookup[arr[i]] \leftarrow i$
return $nil, false$

- 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:

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

The result of $size(\mathtt{r})$ is stored in a variable `d`. We then randomly pick vertices in the half-open interval $[0, \mathtt{n})$, 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.

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 ususal 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 therefor 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 auxiliary 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 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.

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 $delete(\gamma, x)$, which simply means that we want to delete the entry with key x from our map. When iterating over a map in a `for`-loop we destructure the entries into a $(key, value)$-pair, e.g. `for` $(\_, v) \in map$ `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.

$O(|E|^2)$ **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.

```
rem ← ∅
exec ← 0
for e ∈ E do
    if |e| = t then
        exec ← exec + 1
        for v ∈ e do
            C ← C ∪ {v}
            V ← V \ {v}
            for f incident to v do
                E ← E \ {f}
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:

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

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.

```
sub ← ∅
dom ← ∅
exec ← 0
for e ∈ E do
    if |e| = 2 then
        sub ← sub ∪ {hash(e)}
    else
        dom ← dom ∪ {e}
if |sub| = 0 then
    return exec
for e ∈ dom do
    subsets ← getSubsetsRec(e, 2)
    for f ∈ subsets do
        if hash(f) ∈ sub then
            E ← E \ {e}
            exec ← exec + 1
            break
return exec
```

Lemma 5.1 *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. □

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 an a pseudo-random graph with one million vertices and two million edges.

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.

$O(|V|^2 \cdot |E|)$ **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.

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.

$O(|V|^2 \cdot |E|)$ **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, refering 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,

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.

```
exec ← 0
outer ← true
while outer do
    outer ← false
    for v ∈ V do
        dom ← false
        for (_, val) ∈ AdjCount[v] do
            if val = deg(v) then
                dom = true
                break
        if dom then
            outer = true
            for e incident to v do
                e ← e \ {v};
            V ← V \ {v}
            exec ← exec + 1
return exec
```

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

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.*

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 simultaniously. □

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

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.

```
exec ← 0
outer ← true
while outer do
    outer ← false
    for (v, count) ∈ AdjCount do
        sol, ex ← Two-Sum(count, deg(v) + 1)
        if not ex then
            continue
        outer ← true
        exec ← exec + 1
        for w ∈ sol do
            C ← C ∪ {w}
            V ← V \ {w}
            for e incident to w do
                E ← E \ {e}
return exec
```

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

Lemma 5.3 *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 $tsHashes[\{a, b\}]$. □

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

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.

```
exec ← 0
outer ← true
for outer do
    outer ← false
    tsHashes ← map[2^V]2^V
    for x ∈ V do
        for sol ∈ Two-SumAll(AdjCount[x], deg(x)) do
            {z_0, z_1} ← sol
            if tsHashes[sol] ≠ ∅ then
                for y ∈ tsHashes[sol] do
                    if y = x then
                        continue
                    f_0 ← {x, y, z_0}
                    f_1 ← {x, y, z_1}
                    found ← false
                    for e incident to y do
                        if e = f_0 or e = f_1 then
                            found ← true
                            break
                    if found then
                        exec ← exec + 1
                        outer ← true
                        C ← C ∪ sol
                        for a ∈ sol do
                            for e incident to a do
                                E ← E \ {e}
                        break
                    tsHashes[sol] ← tsHashes[sol] ∪ {x}
            else
                tsHashes[sol] ← {x}
return exec
```

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 dont 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.

```
adjList ← map[V]2^V
rem ← ∅
exec ← 0
for e ∈ E do
    if |e| ≠ 2 then
        continue
    {x, y} ← e
    adjList[x] ← adjList[x] ∪ {y}
    adjList[y] ← adjList[y] ∪ {x}
for (z, val) ∈ adjList do
    if |val| < 2 then
        continue
    subsets ← getSubsetsRec(val, 2)
    for s ∈ subsets do
        {x, y} ← s
        if y ∈ adjList[x] or x ∈ adjList[y] then
            exec ← exec + 1
            C ← C ∪ {x, y, z}
            rem ← rem ∪ {x, y, z}
            for u ∈ {x, y, z} do
                for v ∈ adjList[u] do
                    adjList[v] ← adjList[v] \ {u}
                delete(adjList, u)
            break;
for v ∈ rem do
    V ← V \ {v}
    for e incident to v do
        E ← E \ {e}
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.

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.

```
exec ← 0
outer ← true
for outer do
    outer ← false
    for e ∈ E do
        if |e| ≠ 2 then
            continue
        for a ∈ e do
            y ← a
            {z} ← e \ {y}
            f_0 ← nil
            incy:
            for f incident to y do
                if |f| ≠ 3 or z ∈ f then
                    continue
                for g incident to z do
                    cond ← true
                    for b ∈ g do
                        if b = z then
                            continue
                        if b ∉ f then
                            cond ← false
                            break
                    if cond then
                        f_0 ← f
                        break incy
            if f_0 ≠ nil then
                outer ← true
                exec ← exec + 1
                C ← C ∪ f ∪ {z}
                V ← V \ f ∪ {z}
                for h incident to f ∪ {z} do
                    E ← E \ {h}
return exec
```

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.

$O(|V| \cdot |E|)$ **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.

```
exec ← 0
outer ← true
for outer do
    for v ∈ V do
        if deg(v) ≠ 2 then
            continue
        s2, s3 ← nil
        for e incident to v do
            if |e| = 3 then
                s3 ← e
            else
                |e| = 2
            s2 ← e
        if s2 = nil or s3 = nil then
            continue
        {x, _} ← s2
        found ← false
        rem ← nil
        for w ∈ s3 \ {v} do
            for f incident to w do
                if x ∈ f or s3 = f then
                    continue
                else
                    found ← true
                    rem ← f
                    break
            if found then
                break
        if found then
            outer ← true
            exec ← exec + 1
            for a ∈ {x} ∪ f do
                C ← C ∪ {a}
                for h incident to a do
                    E ← E \ {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 calculateted the estimated approximation factor for these graphs. We did this for ratios $r = \frac{|E|}{|V|}$ of 1 to 20.

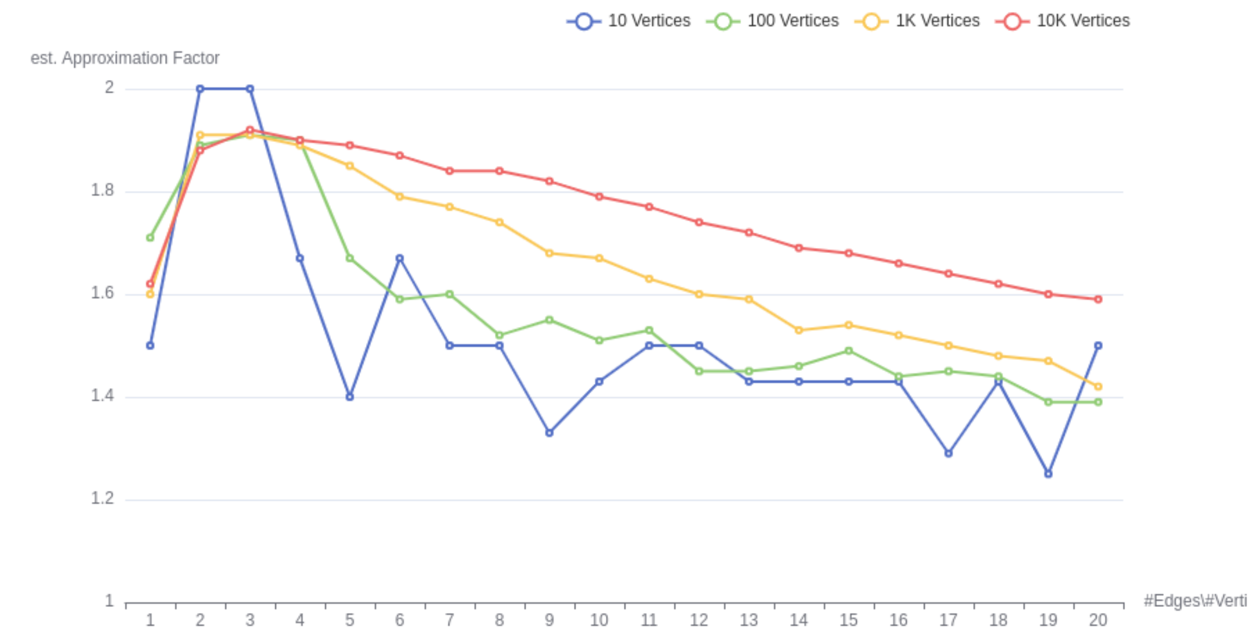


Figure 1: Estimated Approximation Factor for Graphs with 10, 100, 1K, 10K Vertices for $r \in [1, 20]$

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.

remove Factor-3 Rule function paragraph

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.

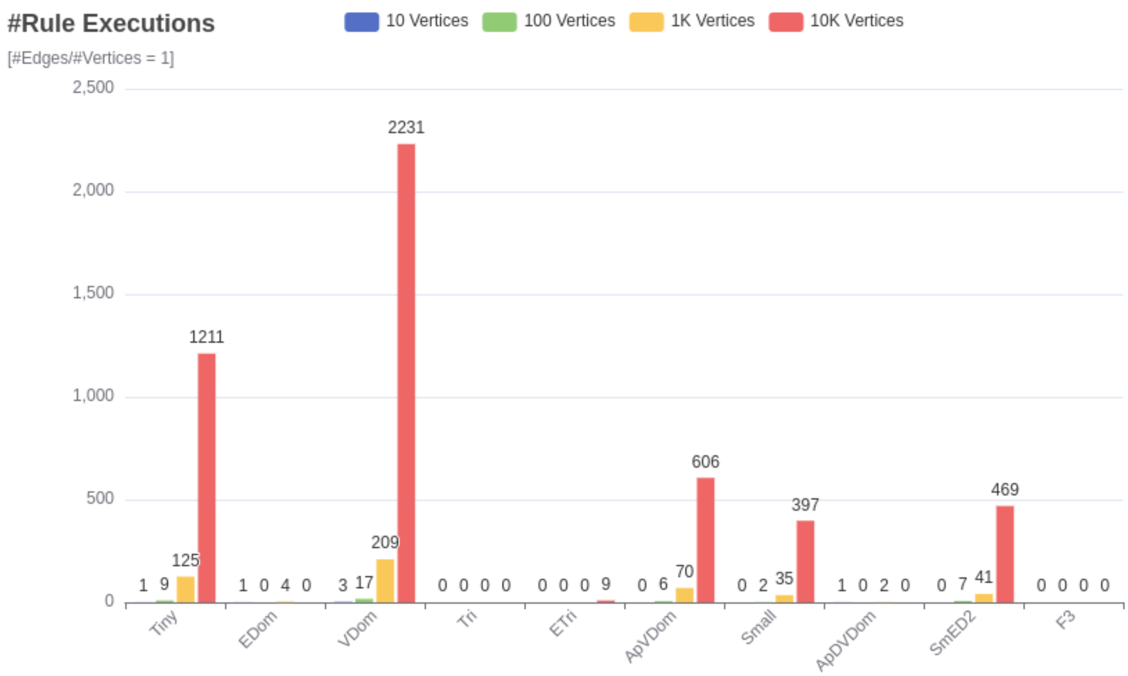


Figure 2: Number of Rule Executions, $r = 1$

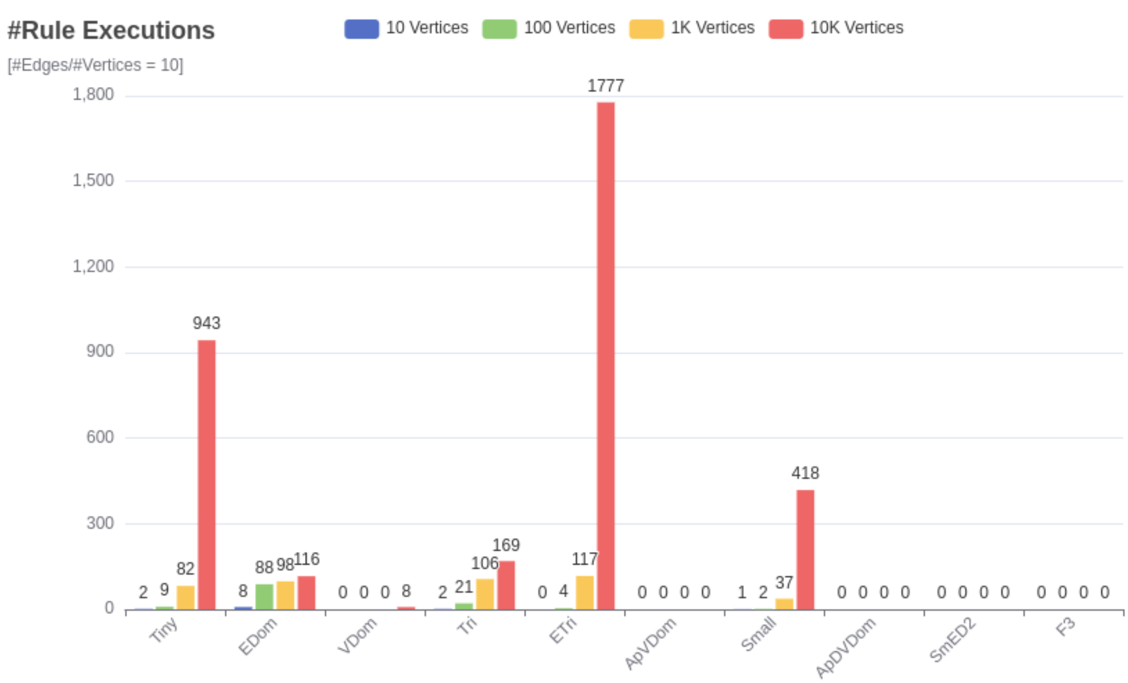


Figure 3: Number of Rule Executions, $r = 10$

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.

Algorithm 11: Incremental 3-approximation algorithm for 3-HS

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

Apply all reduction rules exhaustively mutating G and C
$l \leftarrow 2$
if $|E| = 0$ **then**
  return C
$e \leftarrow$ `F3TargetLowDegree`(G)
$H \leftarrow$ `GetFrontierGraph`(G, l, e)
`/*` H `will act as a mask over` G`, the rules will only be applied to vertices/edges in` H `*/`
for $|E| > 0$ **do**
  Apply all reduction rules exhaustively on H mutating H,G and C
  $exp \leftarrow$ vertices of edges adjacent to removed or modified edge
  if $|exp| > 0$ **then**
    $H \leftarrow$ `ExpandFrontier`(G, l, exp)
    continue
  $e \leftarrow$ `F3TargetLowDegree`(G)
  if $e = \emptyset$ **then**
    continue
  $H \leftarrow$ `ExpandFrontier`(G, l, e)
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.

7.1.1 DBLP Coauthor Graph

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. 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.

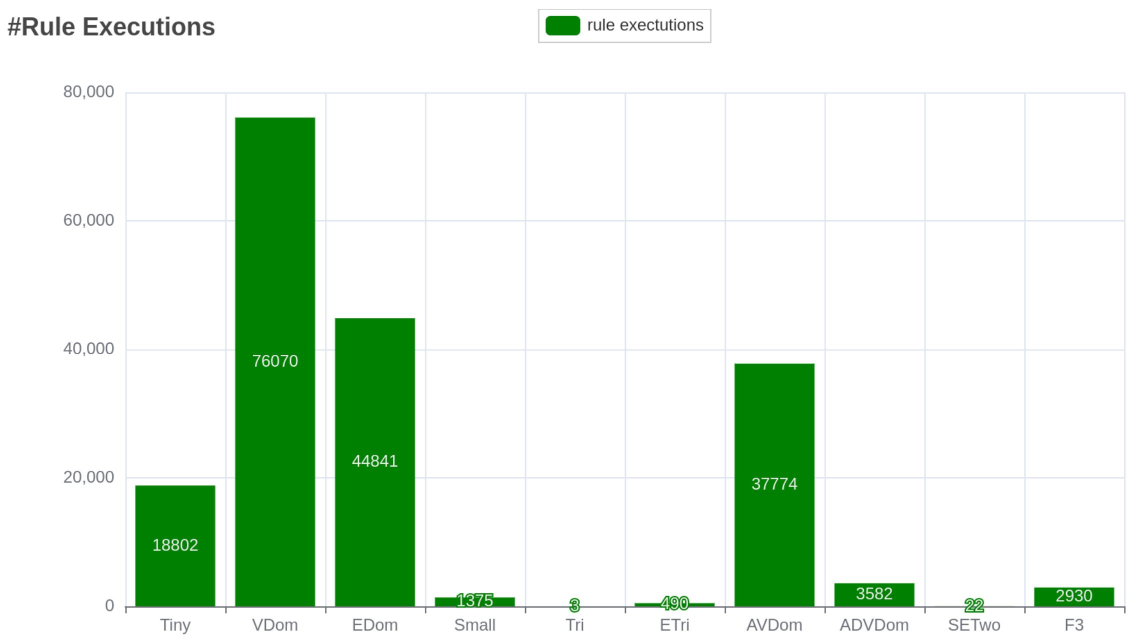


Figure 4: Triangle Vertex Deletion on a DBLP coauthor graph, amount of rule executions

The algorithm achieved a median estimated approximation ratio of ≈ 1.759. Using the incremental 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 auxiliary 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, eliminating possible gains. In fact, constant rebuilding of the auxiliary graph H will even worsen the execution time. needs proof?

Next we tested some variations of the algorithm. In our first test we disabled the use of the edge domination rule. This seems counter-intuitive at first, since the rule is reducing the problem instance "for free". But the fact that we use these data reduction rules in conjunction with approximative reduction rules makes

Table 1: Number of rule executions, percent change of mean after not using some rules

rules disabled	Tiny	VDom	EDom	Small	Tri	ETri	AVD	ADVD	SED2	F3
EDom	0.63	0.35	-100	-42.36	-100	-20.47	0.69	2.74	-43.00	1.42
EDom, Small	4.87	1.51	-100	-100	-99.32	-20.61	0.75	2.54	-41.40	1.48

Table 2: Results for TRIANGLE VERTEX DELETION on DBLP coauthor graph; $n = 100$

	Ratio	$\|C\|$
mean	1.7595	115652.28
std	0.0005	85.16
min	1.7578	115403.00
median	1.7595	115650.00
max	1.7606	115836.00

Table 3: Same as Table 2, but no edge domination rule; $n = 100$

	Ratio	$\|C\|$
mean	1.7572	114953.68
std	0.0005	78.17
min	1.7560	114724.00
median	1.7571	114948.50
max	1.7587	115157.00

Table 4: Same as Table 2, but no edge domination and small edge rule; $n = 100$

	Ratio	$\|C\|$
mean	1.7451	114197.28
std	0.0007	82.84
min	1.7435	113960.00
median	1.7452	114202.50
max	1.7472	114386.00

the situation less predictable. What if we destroy favourable structures in our graph with the rule? And if so, how often does that happen? We ran the modified algorithm 100 times and collected data about the est. approximation ratio, hitting set sizes and the rule executions. As seen in tables 2 and 3, the mean, median, minimum and maximum hitting set sizes are a little smaller while not using the edge domination rule. As seen in table 1, all of the vertex domination rules see a slight increase in number of executions. The rules with the largest difference are the small edge rule and the extended triangle rule. Both of these rules target size two edges and the usage of both are decreasing. The number of executions of the small edge rule decreased by 42.36% and the extended triangle rule by 20.47%. We suspect, that not removing dominated edges, preserves conditions, that are favourable for these vertex domination rules. Since a dominated edge contains a size two edge, it is very likely that the vertex domination rules will remove some of these edges as a byproduct. This is a possible explanation for the rather sharp decrease of small edge rule executions. Note that the sheer size of these real life networks makes it quite difficult to pinpoint the exact reason for these changes in number of rule executions.

add possible explanation for less small edge

This leads us to our next experiment. What happens if we do not use edge domination rule and the small edge rule? This obviously invalidates the correctness of the algorithm, since there exist structures made up of size two edges, that the algorithm can not target, for example a $C_n, n > 3$. One possibility to uphold correctness is, to remove a size two edge, if the fallback rule can not find a size three edge. Nonetheless, all the metrics decrease again as seen in table 4. The percent change in number of rule executions is smiliar to the previous test, when only disabling the edge domination rule. With the two exceptions being the tiny edge and vertex domination rules. We can see an increase of tiny edge by 4.87% and an increase of vertex domination by 1.51%, compared to the unmodified algorithm. The following possible interpretations of these observations are highly speculative.

- **failed cascading**: For example, let $G = (V, E)$ be a hypergraph and $e \in E$ be a size two edge. Let us assume, that a rule R_1 that gets executed after a rule R_0, will make rule R_1 applicable, such that e will be removed. Since we did not go back and recheck if R_1 can be applied again, the small edge rule will remove e.
- **preemptive removal**: What if not removing a size two edge, makes a vertex domiantion type rule possible at some point in time, during the algorithms execution? Another possibility is, that a better rule will remove the edge at some point anyway. Thus removing the edge is not necessary in the first place.

add possible explanaiton for better ratio with no small edge rule

7.1.2 Amazon Product Co-Purchasing Graph

add amazon graph section

To validate that our previous findings were not just due to the structure of the DBLP coauthor graph, we

Table 5: Number of rule executions, percent change of mean after not using some rules

rules disabled	Tiny	VD	ED	Small	Tri	ETri	AVD	ADVD	SED2	F3
EDom	0.99	0.42	-100	-52.48	-99.67	19.02	5.49	-1.84	-75.23	2.68
Edom, Small	20.20	4.86	-100	-99.89	-99.58	18.93	5.52	-2.38	-75.56	2.85

	Ratio	$\|C\|$
mean	1.7469	95252.10
std	0.0003	74.38
min	1.7460	95081.00
median	1.7469	95249.50
max	1.7477	95435.00

	Ratio	$\|C\|$
mean	1.7404	92614.51
std	0.0003	80.23
min	1.7397	92382.00
median	1.7404	92621.00
max	1.7413	92802.00

	Ratio	$\|C\|$
mean	1.6894	89951.35
std	0.0005	83.60
min	1.6876	89742.00
median	1.6894	89953.00
max	1.6905	90153.00

conduct the same tests with an unrelated network. Both base graphs have around the same number of vertices and edges, but their TVD representation differs. The Amazon graph only has 667129 many triangles.

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.

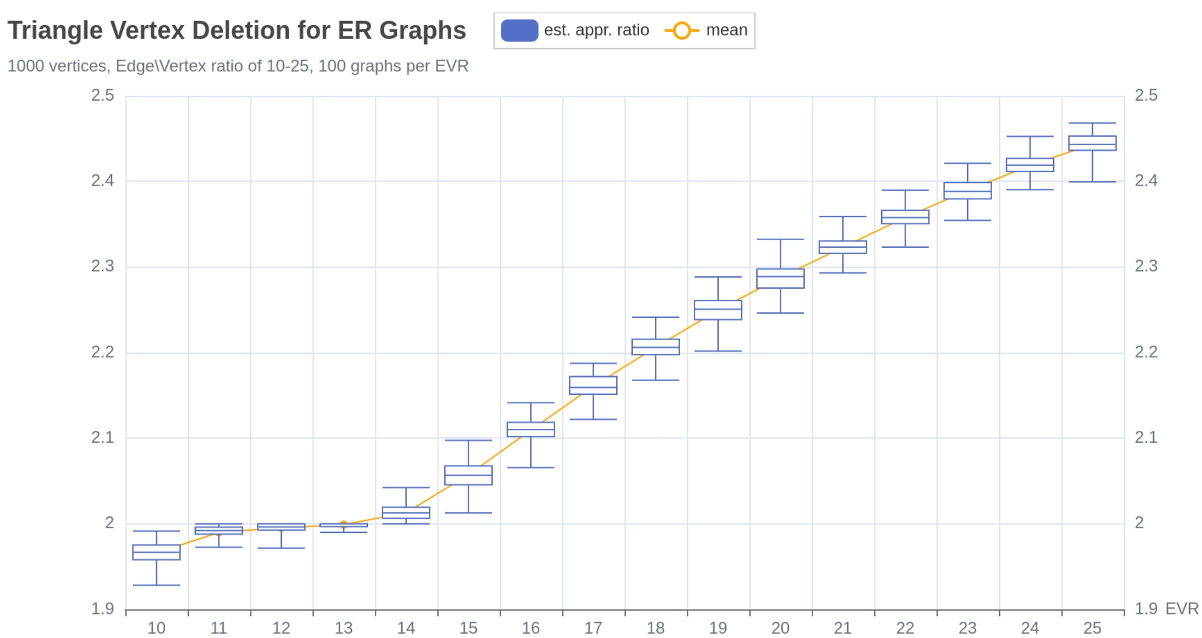


Figure 5: Triangle Vertex Deletion on ER graphs, est. appr. ratios

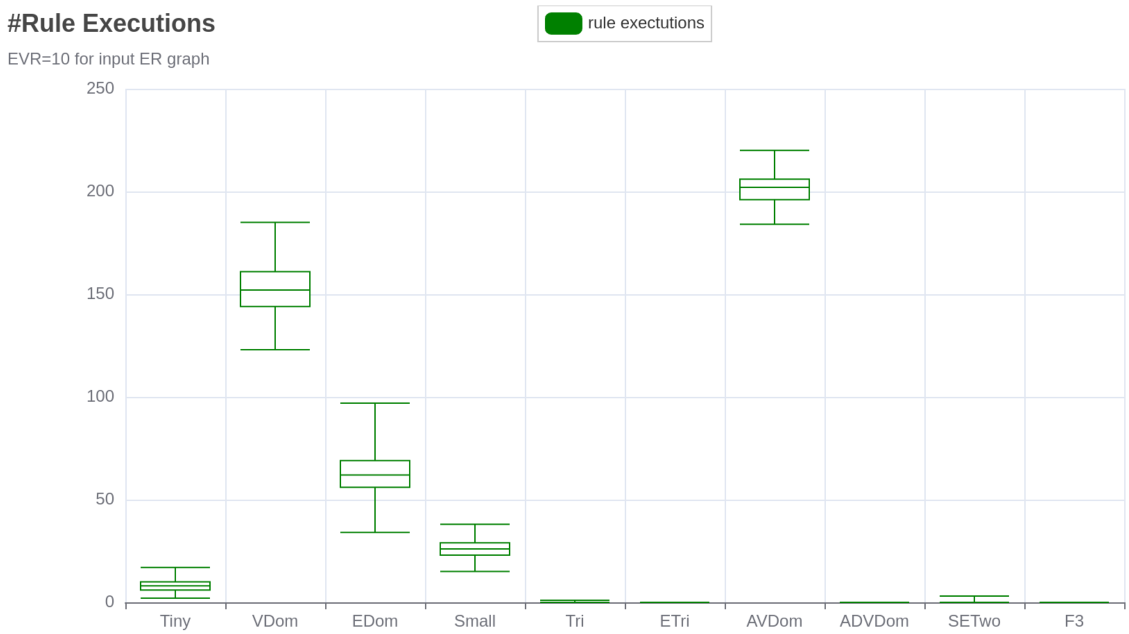


Figure 6: Triangle Vertex Deletion on ER graphs, amount of rule executions for EVR=10

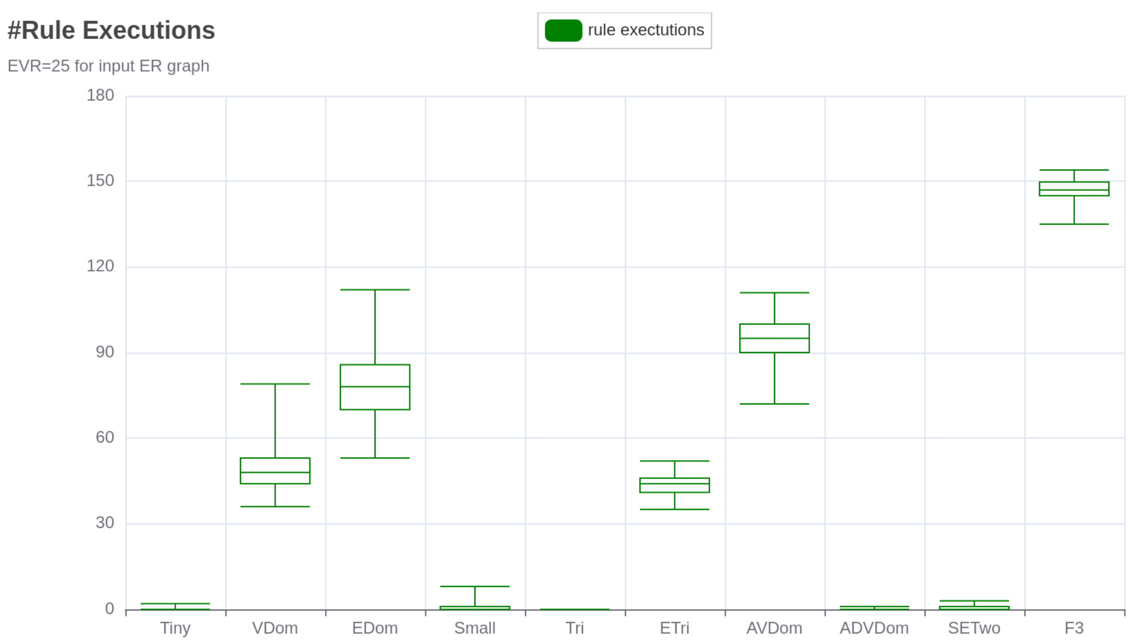
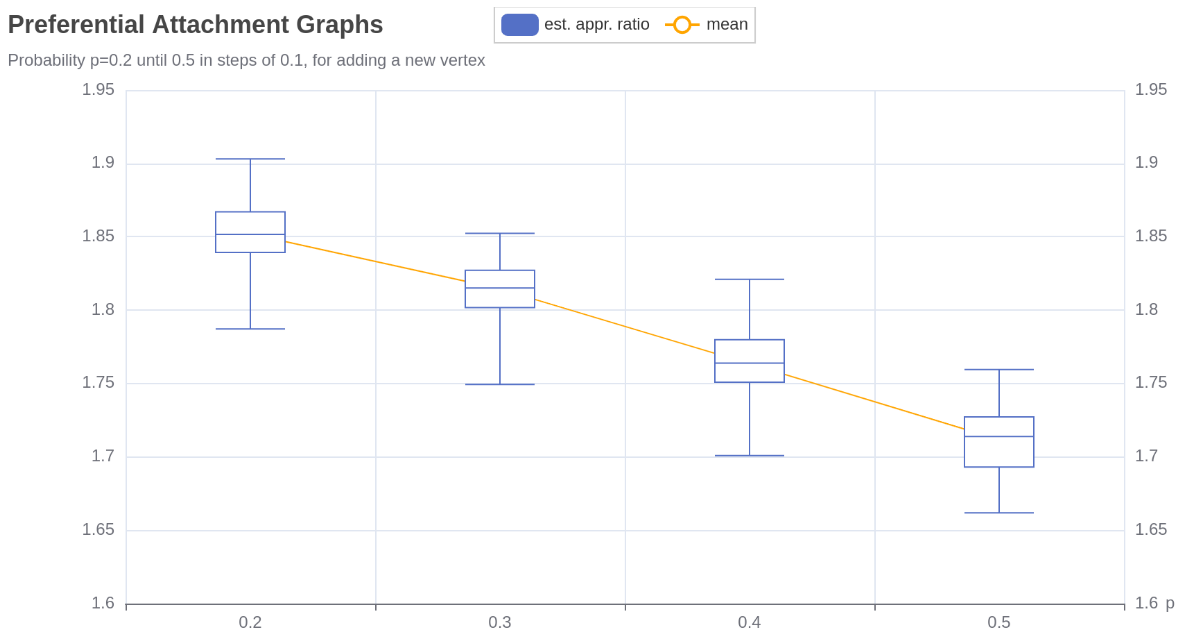
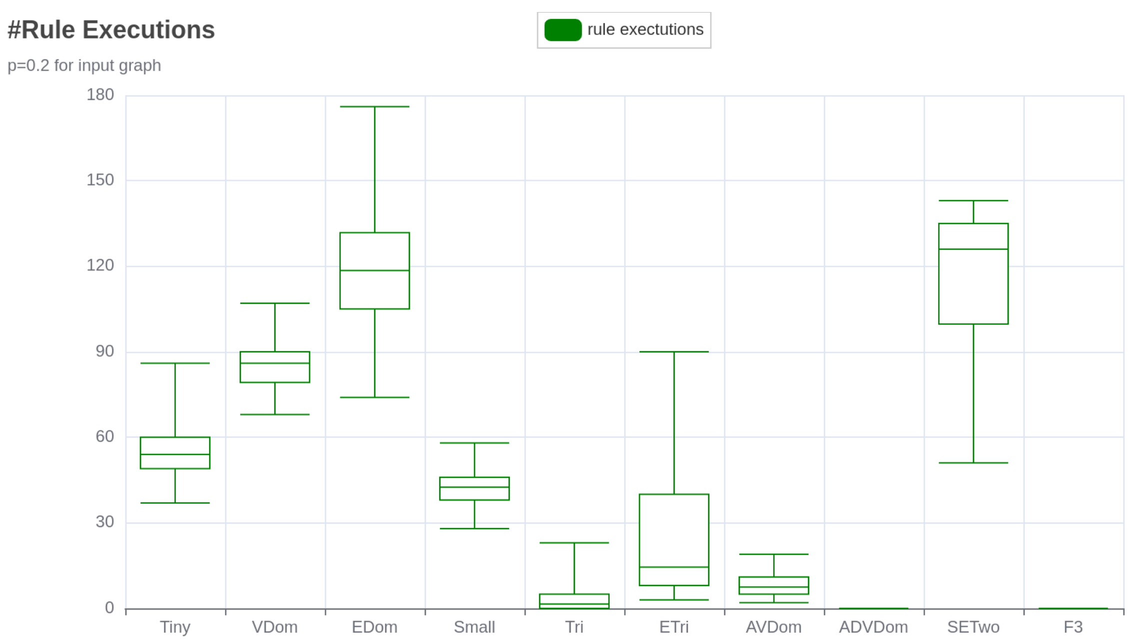


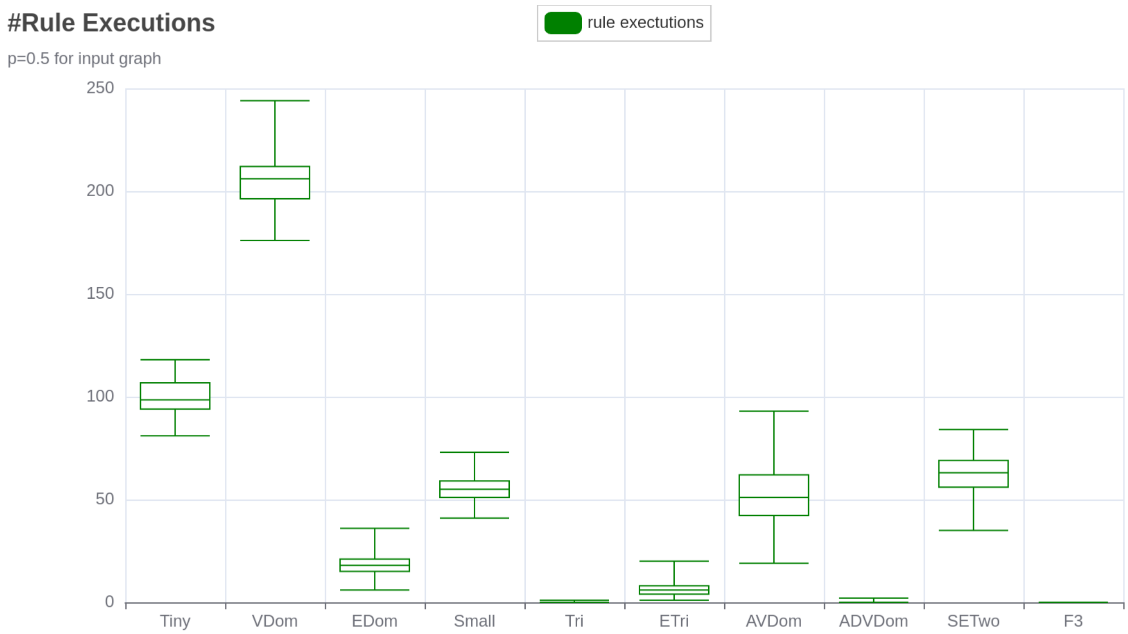
Figure 7: Triangle Vertex Deletion on ER graphs, amount of rule executions for EVR=25

7.2 Cluster Vertex Deletion

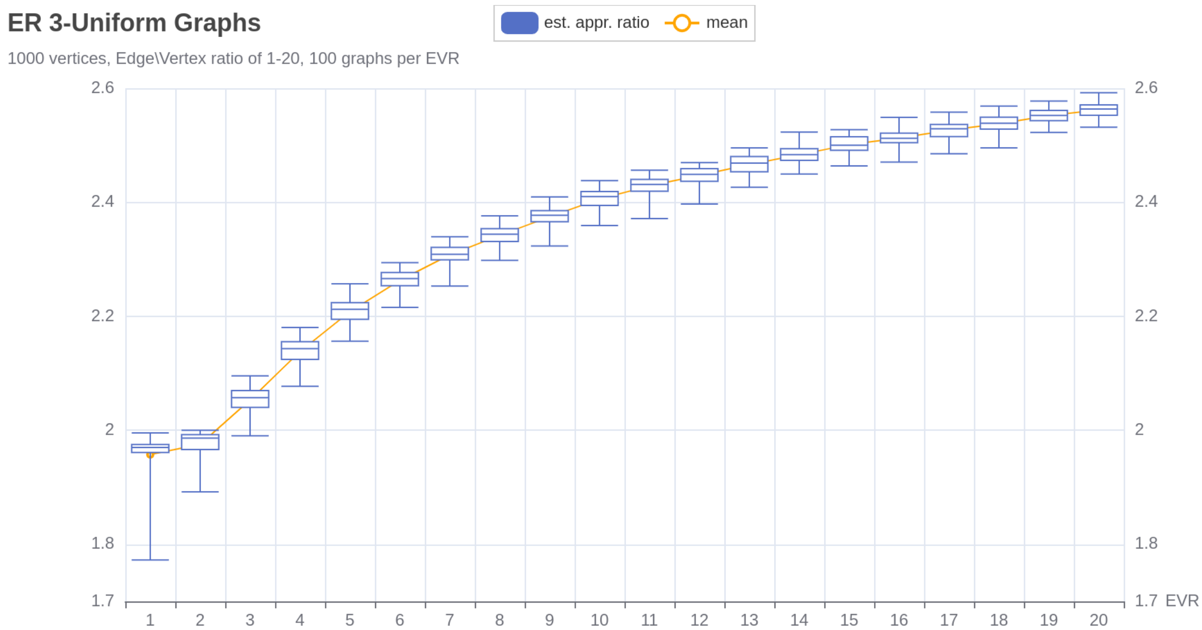
7.3 Preferential Attachment Hypergraphs

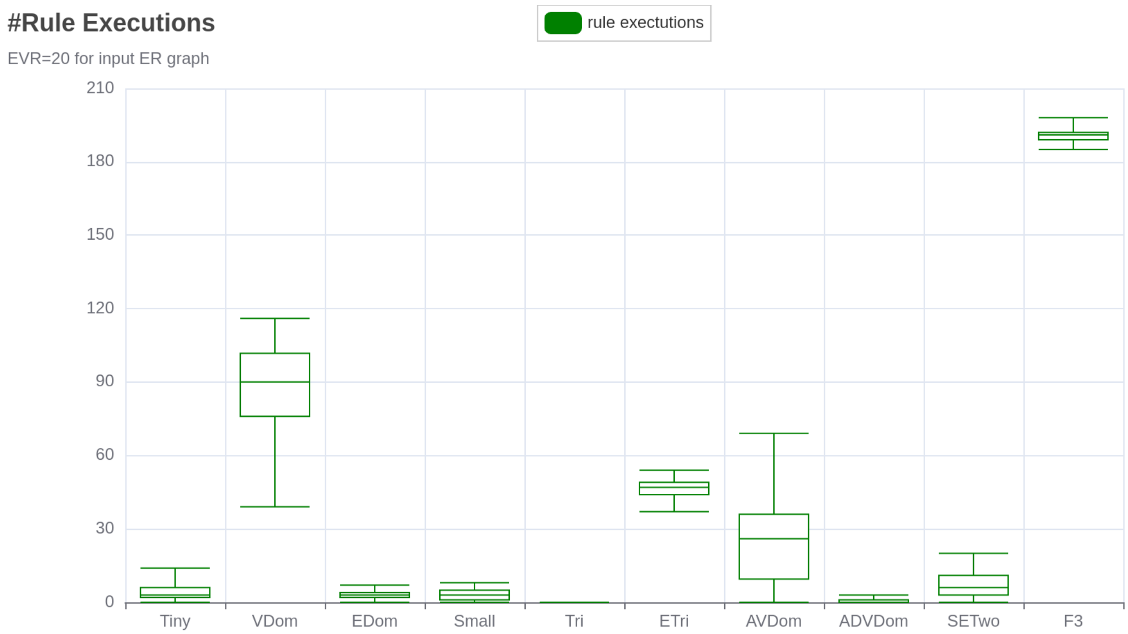





#Rule Executions
p=0.5 for input graph
rule exectutions
250
200
150
100
50
0
Tiny
VDom
EDom
Small
Tri
ETri
AVDom
ADVDom
SETwo
F3

7.4 3-Uniform ER Hypergraphs





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.

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. Trolliet, 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