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

This document was auto-generated using pandoc.

Contents

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
    Degree int32
}
```

The `HyperGraph` datatype has four 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 `Degree` specifies the maximum degree of the graph.

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

rewrite with new algorithm

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

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` ≥ 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

$lookup \leftarrow \texttt{map}[\mathbb{N}]\mathbb{N}$
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 $nil, false$

We start by creating a map called *lookup*. We then iterate other the elements of arr, checking 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:

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

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

Reduction rules are usually meant to be applied exhaustively. We actually do not want to do that, because the algorithms for the rules rely on building up auxilliary data strucutures. Since rebuilding these structures takes a not negligeble amount of time, we want to reuse them as much as possible.

Trying to apply a rule exhaustively in a single execution will introduce some difficulties:

- Some rules will mutate the graph, especially the set of edges. This can potentially create or eliminate a structure in the graph that a rule is targeting.
- Can the auxilliary datastructure be mutated to reflect these changes in linear time?

Luckily for us, most of these problems can be solved using Go. We usually have two possible ways of implementing a rule:

1. **Outer Loop:** We wrap our main algorithm in an outer loop. We then apply the rule one time per loop iteration. The outer loop will break when the rule can't be applied anymore. This way we can reuse our auxilliary data structures.
2. **Pseudo Single Loop:** We find all structures a rule is targeting in a single loop. We do this by iterating over the data structures and simultaniously deleting the targets, mutating the structure we are iterating over. This can possibly eliminate targets which we did not reach yet, or create new targets which we already passed. This requires the use of an outer loop. Since we can not decouple the deletion of the targets from the main algorithm. We inevitably introduce a quadratic time complexity in a worst case scenario.

5.2 Algorithms

The presented algorithms are agnostic in regards to the underlying hypergraph representation. The implementations are almost identical to the pseudocode.

Further we introduce a *map* data structure in our pseudocode with syntax $\texttt{map}[A]B$, 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.

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.

$O(|E|)$ **Algortihm**. Iterate over all edges of the graph and mark all edges of size t in a set rem. We then iterate over all edges in rem. We put the endpoints of the current edge e in our partial solution. Finally we delete all edges that are edge-adjacent to e from rem and our 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.

```
rem ← ∅
inc ← map[V]2^E
exec ← 0
for e ∈ E do
    if |e| = t then
        rem ← rem ∪ {e}
    for v ∈ V do
        inc[v] ← inc[v] ∪ e
for e ∈ rem do
    exec ← exec + 1
    for v ∈ e do
        C ← C ∪ {v}
        V ← V \ {v}
        for f ∈ inc[v] do
            rem ← rem \ {f}
            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.

Algorithm 3: Algorithm for exhaustive application of Edge Domination Rule

Input: A hypergraph $G = (V, E)$, 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 ∪ {e}
    else
        dom ← dom ∪ {e}
if |sub| = 0 then
    return exec
for e ∈ dom do
    subsets ← getSubsetsRec(e, 2)
    for f ∈ subsets do
        if f ∈ sub then
            E ← E \ {e}
            exec ← exec + 1
            break
return exec
```

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. *We can either call this algorithm exhaustively until there are no more edge domination situations, or find all domination situations in a single execution of the algorithm, iff there are no size one edges.*

Instead of a Lemma, repharse into proof of exhaustiveness?

Proof. Assume that our algorithm finds an edge domination situation. Since there are no size one edges, edges with size two cannot be dominated. Thus a dominated edge has to be of size three. Then simply removing

the dominated edge will not create nor eliminate an edge domination situation. It is therefore safe to remove all dominated edges in a single execution of the rule.

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.

$O(|E| + |V|^2)$ **Algorithm**. We first construct two maps:

- $vDeg$ $\texttt{map}[V]\mathbb{N}$: this map associates a vertex v with $deg(v)$.
- inc $\texttt{map}[V]2^E$: this map associates a vertex v with all other edges that are incident to v.

We then iterate over the vertices of the graph. For the current vertex v, we compute a map $vCount$. This map keeps track of the amount of times a vertex w is adjacent to v. The vertex v is dominated, if one of the entries in $vCount$ is equal to $deg(v)$. In that case we remove v from all edges and our vertex set.

Algorithm 4: Algorithm for exhaustive application of Vertex Domination Rule

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

```
vDeg ← map[V]ℕ
inc ← map[V]2^E
for e ∈ E do
    for v ∈ e do
        vDeg[v] ← vDeg[v] + 1
        inc[v] ← inc[v] ∪ {e}
while true do
    outer ← false
    for v ∈ V do
        vCount ← map[V]ℕ
        for e ∈ inc[v] do
            for w ∈ e do
                vCount[w] ← vCount[w] + 1
        delete(vCount, v)
        dom ← false
        for (_, val) ∈ vCount do
            if val = vDeg[v] then
                dom = true
                break
        if dom then
            outer = true
            for e ∈ inc[v] do
                e ← e \ {v};
            V ← V \ {v}
            delete(inc, v)
            exec ← exec + 1
    if outer = false then
        break
return exec
```

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.

$O(|E| + |V|^2)$ **Algorithm**. The quadratic exponent in $|V|$ looks scary at first, but will only occur in the worst case if there exists a vertex v, that is adjacent to all other vertices $w \in V$.

We first construct two maps:

- *vDeg* $\texttt{map}[V]\mathbb{N}$: this map associates a vertex v with $deg(v)$.
- *adjCount* $\texttt{map}[V]\texttt{map}[V]\mathbb{N}$: this map associates a vertex v with the amount of times another vertex is adjacent to v.

Example:

Let $E = \{\{1, 2, 3\}, \{1, 2, 4\}\}$. Then *vDeg* and *adjCount* will look as follows,

$$vDeg = \{ \quad 1:2, \\ 2:2, \\ 3:1, \\ 4:1 \quad \}$$

$$adjCount = \{ \quad 1:\{2:2, \quad 3:1, \quad 4:1\}, \\ 2:\{1:2, \quad 3:1, \quad 4:1\}, \\ 3:\{1:1, \quad 2:1\}, \\ 4:\{1:1, \quad 2:1\} \quad \}$$

We then iterate other $adjCount$, 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 in $adjCount[v]$, s.t. for the pair (a, b) holds, $adjCount[v][a] + adjCount[v][b] = vDeg[v] + 1$. If such a pair exists, then we conclude that for every edge f such that $v \in f$, it holds for f that, either $a \in f$ or $b \in f$.

Lemma. *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 two maps $vDeg$ and $adjCount$. Let v be an entry in $adjCount$ and $sol = (a, b)$ be the result of calling our Two-Sum implementation on $adjCount[v]$ with a target sum of $n = vDeg[v] + 1$.

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 $n = vDeg[v] + 1$. 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 $vDeg[v] + 1$ many edges that contain either $\{a, v\}$ or $\{b, v\}$. This however contradicts that there only exist $vDeg[v]$ 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 $n - 2 = vDeg[v] - 1$ 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.

Algorithm 5:

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

Idea: The initial idea for this algorithm involved the usage of an 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 $|V| + |V|^2 \cdot |E|$, and a high memory complexity of $|V| \cdot |E|$.

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.

5.2.5.1 $\mathcal{O}(|E| + |E|^2)$ Algorithm

We start by iterating over the edges of our graph and building an incidence list.

We then iterate over the edges again. If the current edge e is not of size 3 we continue with the iteration. If $|e| = 3$ we iterate over the endpoints of e. We assign the variable `a` to v and create a map `vCount map[int32]int32`. For each vertex $w \in e \setminus v$ we look at the the edges incident to w, we then for each vertex u of these edges, increment `vCount[u]`. We ignore edges that are incident to `a` and do not care about vertices that are in e. We keep track of the amount of edges we considered with the variable `xyCount`.

If we find an entry `vCount[b]`, s.t. `vCount[b] = xyCount`, we found a double vertex domination situation in e.

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.

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

5.2.6.1 $O(|E| + |V|^2)$ Algorithm

Again the quadratic exponent in $|V|$ looks scarier than it is. This happens beacause we have to remove the triangle.

We start by constructing an edge-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 both vertices of the subset are vertex-adjacent to each other, 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 vertex-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.

5.2.7 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.

5.2.7.1 Outline

- find deg 2 vertex
- Check for at least one adjacent size two edge
- iterate over the size 3 edge neighbors w of v, or if both are size 2 try both
- select a edge adjacent to w that does not contain x

data structures: degMap incMap

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.

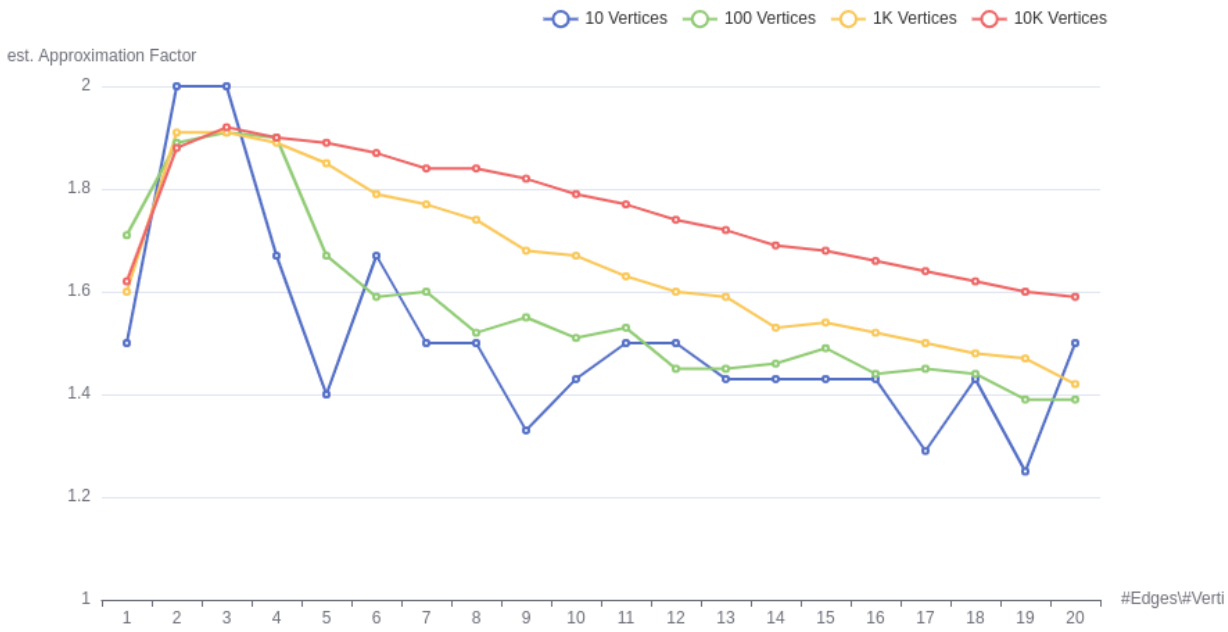

Figure 1: Estimated Approximation Factor for Graphs with 10, 100, 1K, 10K Vertices for $r \in [1, 20]$

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.

Factor-3 Rule Preprocessing. In its current implementation, the algorithm applies all rules and if there are edges left in the graph, adds an edge of size 3 to the solution if possible. In most of these cases we have to

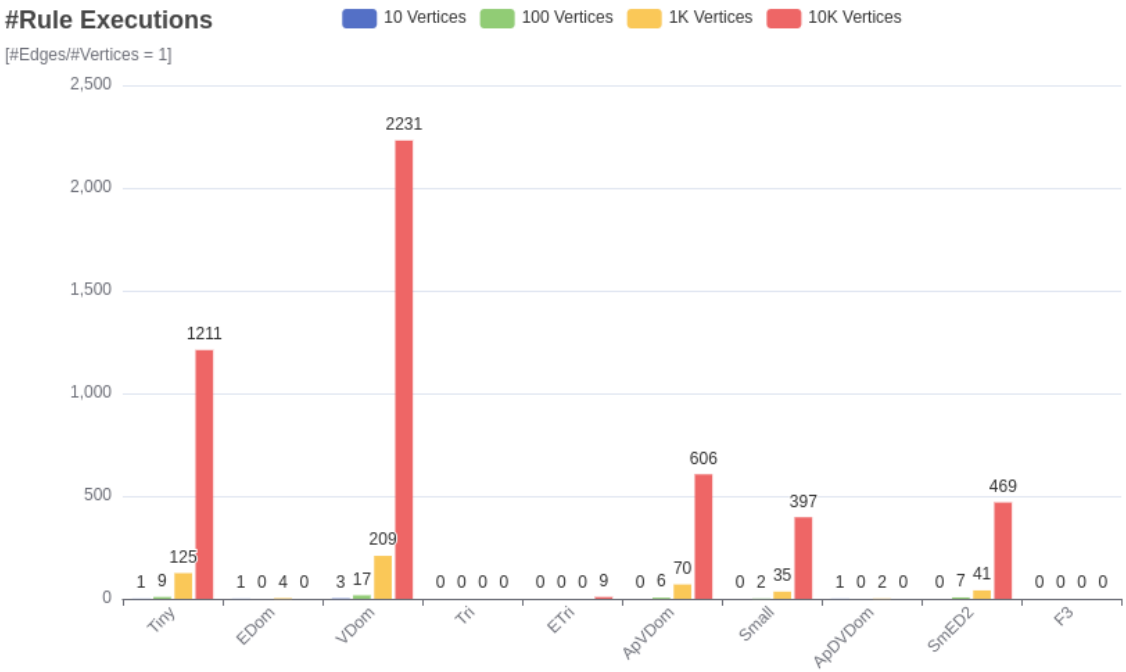

Figure 2: Number of Rule Executions, $r = 1$

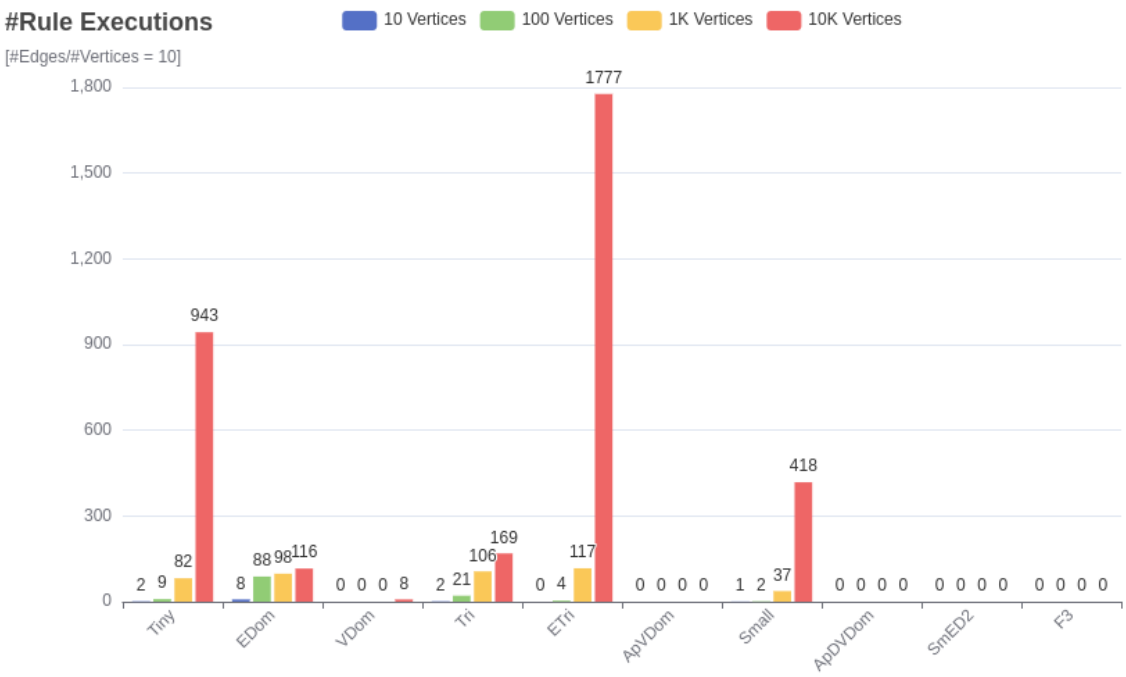

Figure 3: Number of Rule Executions, $r = 10$

apply the Fallback-Rule n times, before we can apply any other rule. We could alternatively apply this factor 3 rule at the beginning of the algorithm. We hope that this will allow for an increase of Vertex Domination Rule executions, which can cascade into Small Edge Rule executions. We kinda open another can of worms by introducing another form of preprocessing. For example one could consider:

- Amount executions of the fallback rule
  - Do we consider both the amount of edges and vertices? Or perhaps only one of those?
  - Can we predict the amount?
- Choosing the edge:
  - Can we apply heuristics?
  - Do we select them randomly?

Factor-3 Rule Function. Instead of only executing the Factor-3 Rule exactly one time, we could consider using a function instead. This function would compute a value based on the current iteration. Experiments with linear,exponential and square root functions were quite promising in regards to execution time and approximation factor. Linear and exponential functions perform faster than the square root function, while the linear and square root function offer a better estimated approximation factor than the exponential one.

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.

6 Applications

7 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 and the amount of edges left in the graph.

8 Branching Algorithm

8.1 Potential Triangle Siutation

8.2 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.

8.3 Possible Optimizations

Right now we sometimes use the Go `apppend` function, to add elements to a slice. Calls to `apppend` 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. 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.