This document was auto-generated using pandoc.

Self-monitoring Approximation Algorithms | Implementation of a 3-HS Algorithm

Building

Requirements

- golang version >= 1.18
- python3 NetworkX

Clone the repo

```
git clone https://github.com/KhoalaS/BachelorThesis.git
```

In the project directory,

build an executable

```
go build cmd/main.go
```

just run the main.go program

```
go run cmd/main.go
```

Usage

```
go run cmd/main.go [OPTIONS]
```

Options

```
-c          Make charts.
-evr int    Maximum ratio |E|/|V| to compute for random graphs.
-f string   Generate a random hypergraph with fixed ratios for the edge sizes.
-fr int     Preprocess the graph with fr many Factor-3 rule executions.
-i string   Filepath to graphml input file.
-k int      The parameter k.
-m int      Number of edges if no graph file supplied. (default 20000)
-maxv int   Maximum vertices for random graphs used in charts.
-n int      Number of vertices if no graph file supplied. (default 10000)
-o string   Export the generated graph with the given string as filename. The will create a 'graphs' fo
-p string   Use a preconfigured chart preset. For available presets run with 'list -p'.
-pa int     Generate a random preferential attachment hypergraph
-prof       Make CPU profile
-u int      Generate a u-uniform graph.
```

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.

Datastructures

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.

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.

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.

Misc. Algorithms/Utilities

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.

Compute Subsets of Size s

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

Time Complexity: $\binom{n}{s}$

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 getSubsetsRec(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]

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

The algorithm is pretty simple.

1. We either put the i^{th} element of `arr` into our temporary array `data` and increment `index` and `i`.
2. Or we only increment `i`.

We have two base conditions.

1. If `index` is equal to `s`, then we copy the contents of `data[0:s]` into an array and push it onto `subsets`.
2. If `i` is greater than or equal `n`.

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.

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.

```
func twoSum(items map[int32]int32, t int32) ([]int32, bool)
```

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

Let us again explain the arguments:

- `items` is a map that associates an integer with another integer
- `t` is the target value

- `return` values are a `int32` array containing two keys `a,b` of `items`, s.t. the sum of `items[a]+items[b]` is equal to the target `t` and a bool that indicates if such a pair was found.

The implementation looks as follows:

```
func twoSum(items map[int32]int32, t int32) ([]int32, bool) {
    lookup := make(map[int32]int32)

    for key, val := range items {
        if _, ex := lookup[t - val]; ex {
            return []int32{key, lookup[t - val]}, true
        } else {
            lookup[val] = key
        }
    }
    return nil, false
}
```

We start by creating a map called `lookup`. We then iterate other our entries `items`, each entry destructuring to a `key,val` pair. We check if the entry `lookup[t-val]` exists.

- If the entry exist, we return an array containing `val` and `lookup[t-val]`
- If the entry does not exist, we add a new entry to the lookup map with `lookup[val] = key`.

Generating Test Graphs

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:

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

We then store the result of $size(\mathtt{r})$ 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. 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.

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 A. Antelmi et al as reference, which used a model proposed by C. Avin et al.

```
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

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 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.

Notation We usually use the `teletype font` to refer to variable names or code. We will often refer to the `id` field of a vertex struct as `v` and will use the notion of a vertex v and the `id` of a vertex interchangebly. We use d to refer to the degree of the graph.

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.

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.

```
func RemoveEdgeRule(g HyperGraph, c map[int32]bool, t int)
```

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

Edge Domination Rule

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

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

$O(|E|)$ **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 compute these hashes as follows.

First we store the ids of the endpoints of e in an array. We then use the built-in `sort` package to sort the array with a *Quick-Sort* algorithm. We then compute a string `in` by joining the ids with a non-numerical delimiter like "|". We then use a hash function to obtain a hashsum of type `uint32`.

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. *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.*

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.

Remarks We technically dont have to hash the string `in` for our purposes. But we can possibly improve the lookup speed of the set *sub* if we do hash the keys first.

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.

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

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

- `vDeg map[int32]int32` : this map associates a vertex v with $deg(v)$.
- `incList map[int32]map[int32]int32` : 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 vertex adjacent to v. The vertex v is dominated, if one of the entries in `vCount` is equal to `vDeg[v]`. In that case we remove v from all edges and our vertex set.

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.

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

$O(|E| + |V|^2)$ **Algorithm** The quadratic exponent in $|V|$ looks scary at first, but will only occur in the worst case if there exists a vertex v s.t. for every other vertex w there exists an edge e with $\{v, w\} \subset e$.

We first construct two maps:

- `vDeg map[int32]int32` : this map associates a vertex v with $deg(v)$.
- `vSubCount map[int32]map[int32]int32` : this map associates a vertex v with all other vertices that are vertex-adjacent to v.

Both maps can be computed in time $|E| \cdot d^2$.

Example:

Let $E = \{\{1, 2, 3\}, \{1, 2, 4\}\}$. Then `vSub` and `vSubCount` will look as follows,

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

$$\texttt{vSubCount} = \{\ 1:\{2:2,\ 3:1,\ 4:1\},\ 2:\{1:2,\ 3:1,\ 4:1\},\ 3:\{1:1,\ 2:1\},\ 4:\{1:1,\ 2:1\}\ \}$$

We then iterate other `vSubCount`, we will refer to the current value in the iteration as `vSubCount[v]`. We then use a Two-Sum-Algorithm to compute and return the first pair in `vSubCount[v]`, s.t. for the pair `[a,b]` holds, `vSubCount[v][a]` + `vSubCount[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 `vSub` and `vSubCount`. Let `v` be an entry in `vSubCount` and n = `vSub[v]`. Now let

`sol=[a,b]` be the result of calling our Two-Sum implementation on `vSubCount[v]` with a target sum of $n = \texttt{vDeg[v]} + 1$.

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 `vSubCount[v]` with a target sum of $n = \texttt{vDeg[v]} + 1$. 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 $\texttt{vDeg[v]} + 1$ 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 $n - 2 = \texttt{vDeg[v]} - 1$ 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 to our partial solution c.

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 $|V| \cdot |E|$.

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.

$\mathcal{O}(|E| + |V| \cdot |E|^2)$ **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.

[Probably just |E|*|V|^2 being annoying]

Remarks. This Algorithm should only reach a quadratic time complexity in the worst case. Despite that, preliminary testing indicated a unexpected slow execution time. A CPU-Profile showed a large amount of Map-lookup operations. This indicated a bottleneck due to excessive map usage. We implemented the same algorithm again using an incidence matrix, hoping for a possible speedup. This was sadly not the case, in some way disproving the bottleneck theory.

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)
```

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

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.

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

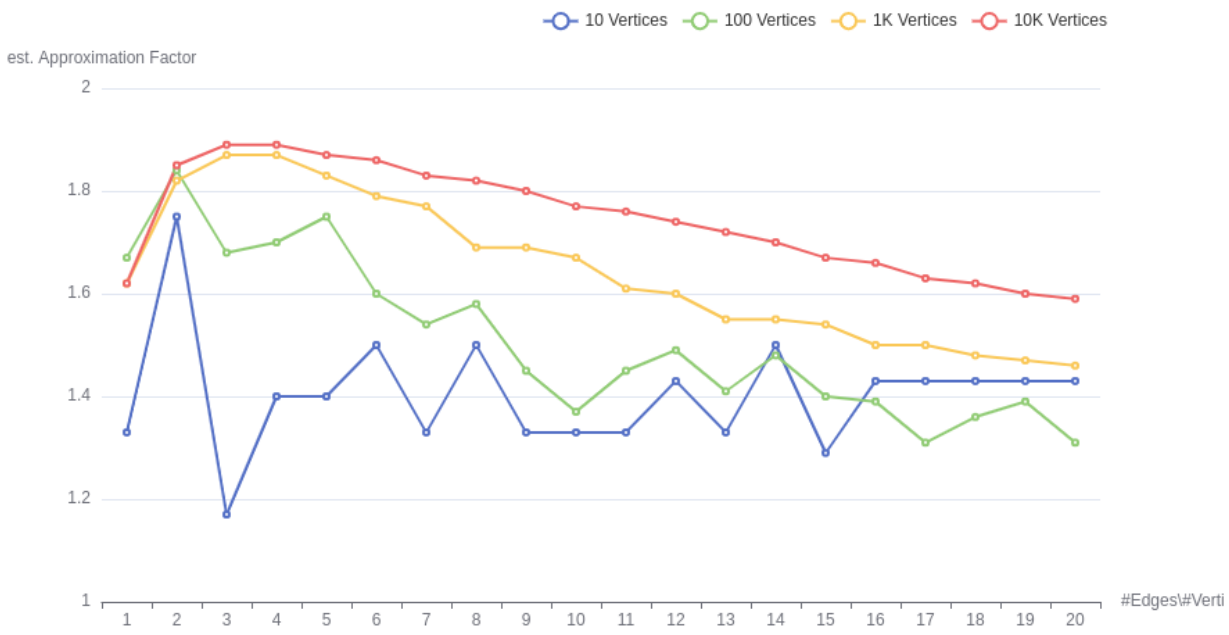
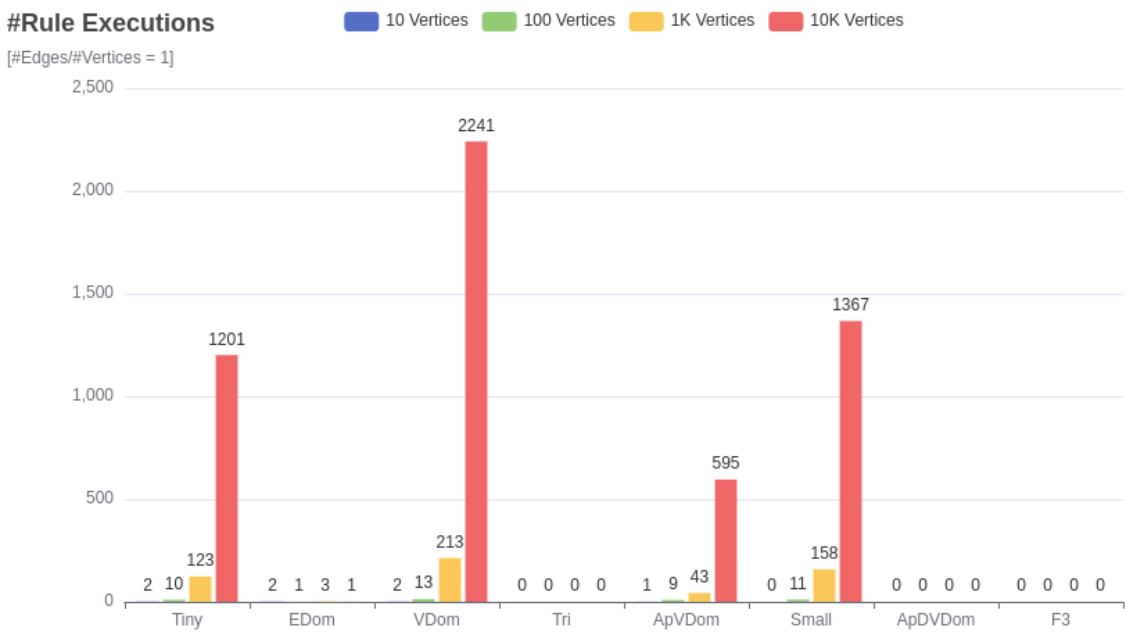
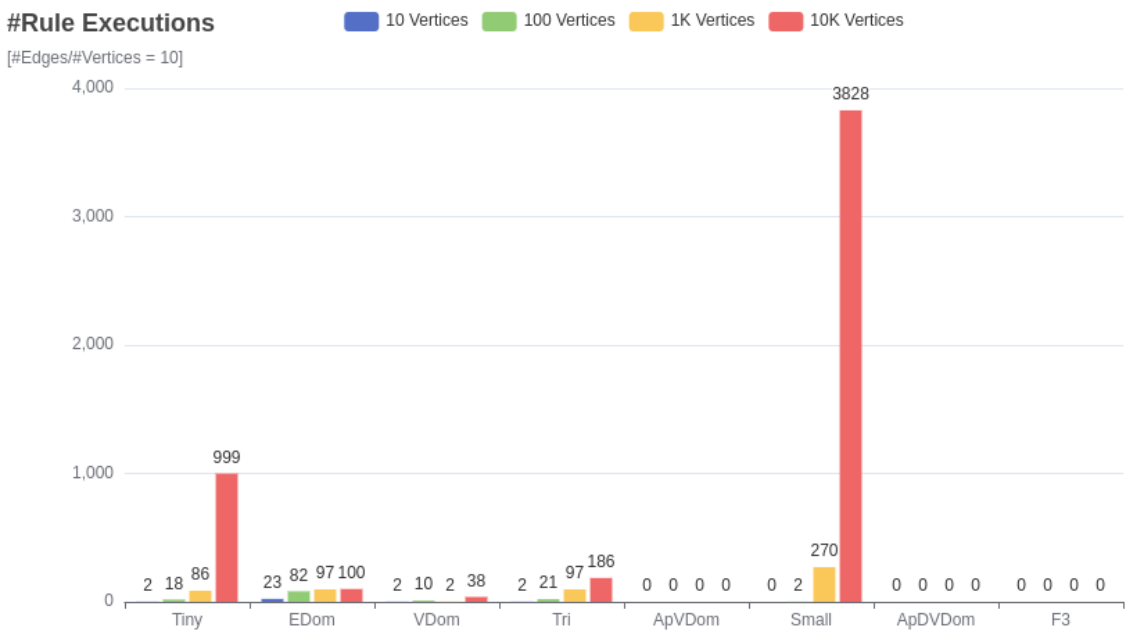


Figure 1: Estimated Approximation Factor for Graphs with 10, 100, 1K, 10K Vertices for $r \in [1, 20]$





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 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?

Lemma. Assume a run of the current algorithm executes the Fallback Rule at most n times before any other rule is executed. Then prepocessing the instance with the Fallback Rule m times, will not increase this maximum, for $m \leq ?n$.

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 part ial solution and the amount of edges left in the graph.

Branching Algorithm

Potential Triangle Siutation

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.

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.