Universiteit Utrecht

Graduate School of Natural Sciences

Algorithms for generating random graphs

Applied to Dutch company networks

Master Thesis

Niels Scholte

Mathematical sciences

Supervisors:

Dr. Ivan Kryven
University Utrecht

Prof. Dr. Frank Pijpers
Statistics Netherlands

Dr. Palina Salanevich
University Utrecht

April 14, 2023

Abstract

Uniformly generating weighted graphs that follow a degree sequences and of which the edge weights follow a target distribution has been studied in [Kryven, 2022]. However, an efficient implementation is missing. Here, we develop methods that implement their algorithm. On the one hand, we develop a method that implement the general case, where no structure of the edge weights is known and hence they have to be stored. This is also done for the binary case, where edges are either allowed or disallowed. These methods are expensive due to them requiring at least $O(n^2)$ storage and $O(n^2)$, simply, due to storing the edge weights. On the other hand, we focus on the case where the edge weights are distances and instead implement a method where the edge weights do not have to be stored. This implementation requires $O(m)$ storage. Whereas the run time is unclear, it is estimated to be $O(m^{\frac{9}{8}} d_{\max})$. In the process of developing these methods, the expected run time of the case without edge weights is improved from $O(md_{\max})$ to $O(m)$. Next, we explore an alternative way of sampling the edges, namely such that all vertices have an equal chance of being sampled. This gives more control over the edge weight distributions of individual vertices, while still outputting a graph with the desired target distribution. Lastly, we develop a method for reconstructing the Dutch company network as an internship at Statistics Netherlands.

Contents

Chapter 1

Introduction; literature review

In the Netherlands, not everyone knows each other. Some people have many colleagues and friends, and others have fewer. Some people are part of clubs where everyone knows each other, and some people are not. This structure can be captured in a graph. A graph is a collection of vertices V, in this case the vertices represent people, and edges E, the connections between pairs of vertices. In this case, they answer the question: do these two people know each other, yes or no? If the answer is yes, there is an edge between the two people, if the answer is no, there is not.

In this thesis, we focus on generating graphs given some constraints. More concretely, we take some properties of the real world, and generate graphs that also satisfy these properties. The graphs are like the real networks in terms of the selected properties.

One of these properties is the degree sequence. Continuing with the example of people in the Netherlands, this is a list of how many people each person knows. It is the degree of each vertex, how many edges each vertex has.

If we want to learn something from these generated graphs, it is also important that we produce a great variety of graphs, as opposed to producing the same ones over and over again. Because of this, uniformity is an important constraint. It means that we output every graph with the same probability. If there are 1000 graphs (where the order of the edges matters) that satisfy the constraints, and we can produce all of them, then each time we generate a graph, ideally they should all have a chance of $\frac{1}{1000}$ of being generated.

1.1 Generating random graphs given a degree sequence

1.1.1 Regular graphs

[Steger and Wormald, 1999] describes how to efficiently generate uniformly random graphs for large h-regular graphs. In other words, when every vertex has the same degree h, hence the degree sequence d is just a list containing the value h as many times as there are vertices. More concretely, let $V_n = \{1, \ldots, n\}$ be the set of n vertices, and let $d = (d_i)_{i \in V_n}$ be a given degree sequence, such that the number of edges $m = \frac{1}{2} \sum_i d_i$. Then in this particular case we have that $d_i = h$

for all $i \in V_n$ and $m = \frac{nh}{2}$. They also require that $d_{\max} = \max_i d_i < m^{\frac{1}{4}}$, i.e. that $h < m^{\frac{1}{4}}$. Although seemingly convoluted, we use $d_{\max}$ instead of h to indicate that the used logic can easily be generalized, and because we will be moving to the general case shortly.

They generate random graphs by adding one edge at a time. Let $\hat{d} = (\hat{d}_i)_{i \in V_n}$ be the sequence of remaining degrees, i.e. the degrees that have yet to be satisfied before we obtain a graph with degree sequence d. Then, by repeatedly sampling vertices $i, j \in V_n$ with probabilities proportional to

$$\hat{d}_i \hat{d}_j,$$

among all pairs i, j with $i \neq j$ and $(i, j) \notin E$, where E is the list in which we store sampled edges, we can obtain a uniformly sampled graph. This approach sometimes fails to generate a graph that satisfies the degree sequence, but this probability goes to zero as $n \to \infty$. Their efficient implementation consists of three phases.

Phase 1

In phase 1, we keep a list L in which every vertex i occurs $d_i (= h)$ times. L is partitioned into two segments, an active segment that we will sample from, and an inactive segment consisting of entries that were successfully sampled before. Phase 1 stops when the number of entries in the active segment falls below $2{d_{\max}}^2$. The inactive segment is positioned at the front of the list.

We iterate by uniformly sampling two vertices i, j from the active segment of L, and we accept them if $i \neq j$ and $(i, j) \notin E$. If (i, j) is rejected, we simply sample again, but if it is accepted the edge (i, j) is added to E, and the sampled vertices i and j are swapped to the inactive segment at the front of L where they will not be sampled again. Because of this, every time an edge is sampled, the active segment shrinks and the inactive segment grows. As a consequence, sampling a vertex i occurs with a probability proportional to $\hat{d}_i$, since i occurs exactly $\hat{d}_i$ times in the active segment of L, for all $i \in V_n$. Therefore, we sample every edge (i, j) with probability proportional to $\hat{d}_i \hat{d}_j$.

In this loop, checking if an edge (i, j) should be accepted takes at most $O(d_{\max})$ operations. This is because every vertex has at most $d_{\max}$ edges, which means it takes at most $O(d_{\max})$ operations to check all edges of a vertex i to see if there is an edge with the vertex j. In other words, the running time of phase 1 is $O(d_{\max})$ times the number of edges checked for suitability.

In the worst case, a vertex i in the active segment of L already has $d_{\max} - 1$ edges, and all of those connected vertices still have an unsatisfied degree of $d_{\max} - 1$. In other words, every entry of the active segment of L has at most $(d_{\max} - 1)^2$ entries it cannot form a suitable pair with. So for $k > 2{d_{\max}}^2$, the expected number of trials before a suitable pair is found is at most 2. Since this is always the case in phase 1 due to the stopping condition, phase 1 is expected to take at most $O(d_{\max} m)$ time.

Phase 2

Phase 2 begins when phase 1 ends, and ends when the number of vertices with remaining degree, i.e. the number of vertices k for which $\hat{d}_k > 0$, drops below $2d_{\max}$. Instead of sampling from L, we now uniformly sample vertices i and j directly from the set of vertices that still have remaining degree. If $i \neq j$ and $(i, j) \notin E$, roll a die that succeeds with probability $\frac{\hat{d}_i \hat{d}_j}{{d_{\max}}^2}$. If it does, add

(i, j) to E, otherwise sample again. Because of this we are still sampling edges with probability proportional to $\hat{d}_i \hat{d}_j$.

Note that Throughout phase 2, the number of vertices with remaining degree is at least $2d_{\max}$. Also note that, every vertex with remaining degree can have at most $d_{\max} - 1$ degrees that are already satisfied. This means that after sampling vertex i at least half of the vertices that can be sampled yield an edge (i, j) for which $i \neq j$ and $(i, j) \notin E$. Since the die roll succeeds with a probability of at least $\frac{1}{{d_{\max}}^2}$, the number of repetitions is therefore expected to be at most $O({d_{\max}}^2)$. Because at most ${d_{\max}}^2$ edges are added in phase 2, and because the check if $(i, j) \in E$ can be moved to after rolling the die, the expected runtime of phase 2 is at most $O({d_{\max}}^4)$, which is also $O(m)$.

Phase 3

Phase 3 begins when phase 2 ends, and ends when there are no more edges to be added. Because there are only $2d_{\max}$ vertices k left for which $\hat{d}_k > 0$, we only have $O({d_{\max}}^2)$ edges left that have yet to be added. Therefore we construct a list H of all possible pairs (i, j) with $\hat{d}_i, \hat{d}_j > 0$, $i \neq j$ and $(i, j) \notin \bar{E}$, which we can do in $O({d_{\max}}^3)$. Like L, this list is again partitioned into an active segment, from which we sample edges, and an inactive segment, where we store away edges that should no longer be sampled. We then sample from the active segment of H and roll the same $\frac{\hat{d}_i \hat{d}_j}{{d_{\max}}^2}$ die as in phase 2. By still checking if $\hat{d}_i, \hat{d}_j > 0$ before the die roll, we can maintain H in constant time by lazily removing edges in H that should no longer be there. Therefore, the expected runtime is, again $O({d_{\max}}^4)$. In total, the expected runtime is at most $O(d_{\max} m)$ because of phase 1.

At the end, we return E if all degrees are satisfied, i.e. if $|E| = m$, and we report failure otherwise.

1.1.2 Non-regular graphs

[Bayati et al., 2010] improves the previous algorithm by making their version also applicable to non-regular graphs. The key difference is that now, we are sampling vertices $i, j \in V_n$ with probabilities proportional to

$$\hat{d}_i \hat{d}_j \left(1 - \frac{d_i d_j}{4m}\right), \tag{1.1}$$

among all pairs i, j with $i \neq j$ and $(i, j) \notin E$, instead. Practically, this means that before accepting an edge, we now also roll a die that succeeds with probability $1 - \frac{d_i d_j}{4m}$. Recall that by assumption $d_{\max} < m^{\frac{1}{4}}$. Because $d_i d_j \leq {d_{\max}}^2 < m^{\frac{1}{2}} < 2m$ we see that $\frac{1}{2} < 1 - \frac{d_i d_j}{4m}$, hence there is no overhead in terms of algorithmic complexity, and the runtime remains $O(d_{\max} m)$.

1.2 Distributions over edge weights

[Kryven, 2022] extends [Bayati et al., 2010] to solve a slightly different problem. They study graphs with weighted edges. Instead of uniformly sampling any graph, the goal is now to sample graphs in such a way that the edges are sampled according to some given target distribution $f : \mathbb{R}_+ \to \mathbb{R}_+$

over the weights, such that $\int_{\mathbb{R}_+} f(x)\,\mathrm{d}x = 1^*$.

All weights r_{ij} of edges (i, j) are assumed to be non-negative. They are also assumed to be pairwise independent and identically distributed. In other words, we assume that in the graphs that are processed with this method, there is some underlying random process that produces these weights.

For example, if one has vertices i that randomly get assigned a location l_i from the same distribution, and the weights are the distances between locations $r_{ij} = \|l_i - l_j\|$, then the weights are non-negative and identically distributed. They are also pairwise independent. If i, j, i', j' are all different it is clear r_{ij} and $r_{i'j'}$ are independent. But even when $i' = i$, and j', j, i are all different, r_{ij} and $r_{i'j'}$ are still independent (and identically distributed) since l_j and $l_{j'}$ are still independently being sampled.

Their proposed method stays with the iterative approach of [Steger and Wormald, 1999], where one edge is added to E at a time. To achieve sampling the edges according to some target distribution f, they introduce the distribution $g : \mathbb{R}_+ \to \mathbb{R}_+$ which is the empirical distribution of edge weights, of the edges that can be sampled. In other words, g is the empirical distribution of edge weights r_{ij}, of the edges (i, j) such that $i \neq j$, $(i, j) \notin E$, and $f(r_{ij}), \hat{d}_i, \hat{d}_j, \neq 0$. As a consequence, g chances every iteration.

Their proposed method is repeatedly sampling edges with probability proportional to

$$\frac{f(r_{ij})}{g(r_{ij})} \hat{d}_i \hat{d}_j \left(1 - \frac{d_i d_j}{4m}\right), \tag{1.2}$$

still among all pairs i, j with $i \neq j$ and $(i, j) \notin E$.

Intuitively, it is not too difficult to understand what is going on; we sample edges proportional to f, and balance out the distribution we are sampling from with g. Roughly, they prove that this way we can generate random graphs with a degree sequence close to some given degree sequence d, and that the edges are sampled according to f. Although they do sketch the outline of a workable algorithm by providing the above sampling strategy, they do not provide an efficient implementation. Providing this implementation is a core focus of this thesis.

*Strictly speaking, they define f as f_n and g as g_n, and both as being continuous on their domain. However, due to the focus on an efficient algorithm and not on the exact mathematical statements, we will keep using the notation f and g, and ignore this continuity since it is not practically useful given that the available edge weights for any particular graph are inherently discrete.

Chapter 2

Designing fast implementations for existing methods

The main focus of this chapter is providing an efficient implementation for the algorithm proposed in [Kryven, 2022]. Working towards this goal, we first look to improve the algorithmic complexity of the algorithm proposed by [Bayati et al., 2010], the non-weighted non-regular case. This is discussed in section 2.1. We then move on to the case with weighted edges in sections 2.2 and 2.3. There, we first provide an algorithm for the special case where the edge weights are binary. After that, we extend this algorithm to accept any edge weights.

2.1 Without edge weights

We improved the algorithm of [Bayati et al., 2010] in two ways. The first is the use of hash tables to look up if an edge $(i, j) \in E$ in constant time. This improved the expected time complexity from $O(d_{\max} m)$ to $O(m)$.

The second is the use of an AVL tree (a self-balancing binary tree, named after Adelson-Velsky and Landis) in phases 2 and 3 to roll a die that succeeds with probability $\frac{\hat{d}_i \hat{d}_j}{\left(\max_k \hat{d}_k\right)^2}$ rather than $\frac{\hat{d}_i \hat{d}_j}{{d_{\max}}^2}$. This greatly speeds up these phases, which is useful if one wishes to violate the $d_{\max} < m^{\frac{1}{4}}$ assumption; these phases have an expected run time of at most $O({d_{\max}}^4)$.

2.1.1 Hash tables

In essence, a hash table (or hash map) is a clever way of storing and looking up data in an array using keys. These keys can be any immutable object. It works in such a way that the expected time per call to the table is $O(1)$.

This approach consists of two main parts. The first is mapping keys to array indexes. The second is collision resolution, which is dealing with multiple keys being mapped to the same array index. Ideally, when storing data we do not want to run into array slots where data is already stored,

and when retrieving data we only want to run into the data stored with the key we are looking up with.

Because beforehand we have no way of knowing exactly which keys will be used, the method used for mapping keys to indexes should spread the indexes out across the array. It should not have a bias towards some indexes over others because this increases the chance of getting collisions. In other words, this mapping function should be deterministic, since we need to end up at the same array index again when retrieving the data again with the same key, but its behaviour should also look somewhat random, since the indexes should be spread out and there should be no biases. Of course, this also depends heavily on the keys that are used. In the worst case, all keys are mapped to the same index, in which case the method loops over all stored data every time. However, this is highly unlikely, hence an expected lookup time of $O(1)$.

Common algorithms

When mapping a key to an index, it is common practise to take the bit string of the key, treat it as a binary number, and then take the modules with the length of the array to obtain the array index. As a consequence, any change in the key has the potential to change the index the key maps to.

There are many ways in which one can perform collision resolution. Here we choose to stick to the same principles mention before, namely that we should choose indices that are deterministic, but also look somewhat random. In other words, when we run into a collision, we will be recomputing the index in the array. The scheme we use for this is random probing. It recomputes the index k as $k_{\text{new}} = k_{\text{prev}} \cdot p \mod l$, where p is some prime number greater than 2, and l is the length of the array, and this length is also a power of 2. In particular, we use $p = 5$, following the implementation of the built-in hash maps of the programming language Python.

Edges and indexes

In this section, we will only be using a hash table to speed up checking if an edge $(i, j) \in E$ as a means to improve the expected run time from $O(d_{\max} m)$ to $O(m)$. Within the context of looking up data with a key in an array, this means that the key should be the edge, and the data is a boolean indicating if the edge is part of E. This boolean is redundant, and we can use a hash set instead. This way, membership of the set is implied by being able to retrieve the key.

The required hash set is special as well. It only needs to support storing and looking up entries. For example, deleting and sampling from the entries are not required. Furthermore, the keys follow a pattern as well. Namely, all keys are tuples (i, j) with i, j being integer entries such that $0 \leq i, j < n$. In other words, after sorting all individual edges such that $i < j$, we can assign a unique identifier $T_{i,j}$ to each edge as $j \cdot n + i$. We then define $T_{i,j} \mod l$, with l being the length of the array, as the array indexes.

Explicit routines

Because of the hash set, edges are now effectively stored twice. Once in an array E of shape $(m, 2)$, and once in an array $\bar{E}$ of length $2^{2+\lceil \log_2 m \rceil}$ that only stores their identifiers T_{ij}, and -1 otherwise.

More explicitly, if we accept an edge (i, j) and we wish to store it in $\bar{E}$, we sort the edge such that $i < j$, we compute $T_{ij} = j \cdot n + i$ and check if $\left(\bar{E}_k = -1 \text{ or } \bar{E}_k = T_{ij}\right)$, for $k = T_{ij} \mod 2^{2+\lceil \log_2 m \rceil}$. If it is, we set $\bar{E}_k = T_{ij}$. Otherwise, we repeat with $k = k \cdot 5 \mod 2^{2+\lceil \log_2 m \rceil}$, until we find an empty or correctly filled slot.

Checking if an edge is in E works similarly. If $\bar{E}_k = -1$ we conclude that the edge is not in $\bar{E}$ and therefore not in E, and if $\bar{E}_k = T_{ij}$ we conclude that the edge is in E.

2.1.2 AVL trees

Although we already generate m edges in $O(m)$ operations, it is important for extensions of this algorithm that all the phases only take as much time as they need to. Because of this, we take the time to improve the speed of phases 2 and 3. Both of these phases have an expected run time of at most $O({d_{\max}}^4)$. Because of this, it is also important to improve their run time if someone wishes to violate the $d_{\max} < m^{\frac{1}{4}}$ assumption.

The main idea is that when rolling a die that succeeds with probability $\frac{\hat{d}_i \hat{d}_j}{{d_{\max}}^2}$, the denominator is only there such that we can sample proportional to the numerator. Replacing the denominator with $\left(\max_k \hat{d}_k\right)^2$ should be much more efficient, given there is no overhead in computing $\max_k \hat{d}_k$. It turns out that when using an AVL tree, this overhead only costs $O({d_{\max}}^2 \log_2(d_{\max}))$ in total and is therefore negligible.

Binary search trees

If you're lucky, and you look out of the window at the nearest tree, you might notice that it has many leaves. If you were tasked with finding a very specific leaf, it would probably take quite a while. However, if you were to move from the stem up (the root), and every time the leaves branched (at a vertex) you would have instructions on which branch (or edge) to follow, you would find the correct leaf much faster. The latter is exactly what binary search trees do. A slight difference is that whereas nature grows however it pleases, binary search trees stick to only branching in at most 2 directions at every vertex, and the leaves are those vertices that do not branch any further.

In other words, leaves are also vertices. When using binary search trees, we are interested in finding vertices, not just leaves, and the data they store. Lookups again happen with a key, and it is this key that determines the location of the vertex in the tree. Since every vertex has a key, we can move across the tree - starting from the root - repeatedly asking, is this my key, and if not is the key I have greater than the key of this vertex? If the answer is yes, we go right, if the answer is no, we go left.

Because of this, the minimum and maximum values are very easy to obtain. To obtain them, simply traverse the tree all the way to the left or right, respectively.

Self-balancing binary search trees

This approach of looking for leaves generally works well, but sometimes trees like to have some leaves on the stem too. These are much easier to reach than leaves high up in the canopy. If all leaves were equally difficult to reach, we could formulate some guarantee on how long it would take to reach any vertex given how many vertices there are in total.

A solution to this problem is having the tree *balance* itself whenever it is altered. For a sufficiently balanced tree, all leaves are at almost the same number of edges away from the root, with the allowed difference being at most 1. A sufficiently balanced tree is called an AVL tree. It turns out that rebalancing a tree takes as much time as finding the position of the vertex that has to be added or deleted. In return, this ensures that the calls to the tree only require $O(\log_2 h)$ operations, with h being the total number of vertices in the tree.

Rebalancing AVL trees

Let the height of a binary search tree be the length of the longest path from the root node. Saying a binary search tree is an AVL tree is equivalent to saying that for all vertices of that tree, the heights of the right and left subtrees differ by at most 1. The main idea is that when inserting or deleting a vertex, the heights of almost all vertices are unaffected, and we only need to rebalance the tree along the path from the root to the added or deleted vertex.

Rebalancing happens from the changed vertex to the root. In other words, we traverse the path twice. First we find the correct location, and then we go back, shuffling the vertices around to correct the heights. This way, every time we rebalance a vertex, the right and left subtrees have already been rebalanced, and their heights are known.

Rotations

Rebalancing a tree, whose subtrees already have been rebalanced, comes down to re-rooting the tree. In other words, if the heights of the right and left subtrees differ by more than 1, we *rotate* the tree to the shortest side such that the new root is a vertex from the longest side.

In figure 2.1 we illustrate what it means to rotate a tree. Going from figure 2.1a to 2.1b shows a right rotation, whereas the other way around shows a left rotation. Notice that X, Y are single vertices, but A, B, C can be subtrees. Knowing this, also notice that in both 2.1a and 2.1b, for all keys k_D of a set of vertices D, we have that $k_B < k_X < k_A < k_Y < k_C$. In other words, after a rotation, all vertices are still in the correct location such that they can be efficiently found from the root.

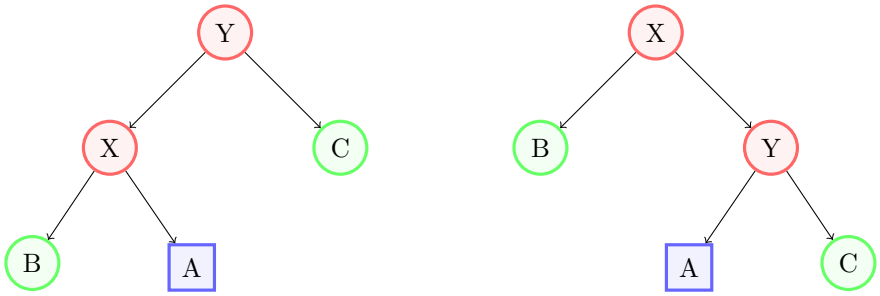
Rebalancing a vertex requires at most 2 rotations. If there is no height difference greater than 1 between the left and right subtrees, no action is required. If B (in figure 2.1a) or C (in figure 2.1a) is part of the longest path, then a single right or left rotation suffices, respectively.

However, if A is part of the longest path, 2 rotations are required. This is because A has to be rotated to become the new root. To do this, A first has to take the place to X in figure 2.1a, and Y in figure 2.1b. Focussing on figure 2.1a, the first rotation is a left rotation at X (making A the root of the left subtree), and the second is a right rotation at Y (making A the root).

Inserting and Deleting

Inserting a new vertex is straight forward. A new vertex (with the corresponding new key) always becomes a new leaf. One can then rebalance back to the root to obtain the new tree. Similarly, deleting a leaf is also straight forward.

Deleting entries is slightly trickier when they are not leaves. Then we have to distinguish 2 cases. If the vertex does not have a right subtree, replace it by its left subtree. However, if it does, replace



(a) A tree rooted at Y.

(b) A tree rooted at X.

Figure 2.1: An illustration of how to reroot a tree with rotations to the right (from figure 2.1a to 2.1b) and left (from figure 2.1b to 2.1a).

the vertex by the vertex with the lowest key from this right subtree. In either case, rebalances back to the root from the change furthest in the tree to obtain the new tree.

Alternatively, if we really want to make sure that we never move the maximum node for technical reasons, deletions can also be implemented by checking if the left subtree is empty.

Integer keys

In the case of integer keys, adjacent integers always occur in the same path from the root. Because of this, we can increment the keys by 1 faster than it is to delete and insert in sequence. This is because otherwise you traverse the same path twice times.

Because the remaining degrees are integers, we can apply this here. The tree uses the remaining degrees $\hat{d}_k$ as keys, and stores the number of occurrences as data on the vertices. I.e. if 7 nodes have a remaining degree of 5, the node with key 5 stores the value 7. This means inserting and deleting can be as simple as incrementing or decrementing the number of occurrences by 1.

After sampling an edge (i, j), we decrement $\hat{d}_i$ and $\hat{d}_j$ in the tree. Following this, we check if the vertex that used to contain the $\max_k \hat{d}_k$ still has a number of occurrences ≥ 1 (and that the vertex has not been deleting if using the delete implementation that checks the left subtree to be empty). If it does, we do not need to do anything, because remaining degrees can only go down, so its key is still $\max_k \hat{d}_k$. If it does not, we reobtain the vertex with the maximum key.

Run time

To improve the run time, we maintain an AVL tree of the entries of $\hat{d}$. The tree has to be updated ${d_{\max}}^2$ times (there are ${d_{\max}}^2$ edges left to be added in phases 2 and 3 combined), and the pointer to the node with the maximum value has to be updated $d_{\max}$ times (the tree contains $d_{\max}$ keys, with the data being the number of occurrences of this key as $\hat{d}$). Therefore, the additional cost is only $O({d_{\max}}^2 \log_2(d_{\max}))$.

This improves the run time because edges (i, j) are mostly sampled proportional to $\hat{d}_i \hat{d}_j$, the remaining degrees of the vertices. In other words, edges containing maximum remaining degree $\max_k \hat{d}_k$ vertices i or j have a high chance of being sampled.

Looking more closely, consider the case that $\max_k \hat{d}_k$ is still $d_{\max}$. When moving from ${d_{\max}}^2$ edges left to be sampled to $(d_{\max}-1)^2$ edges left to be sampled, we sample $2d_{\max}-1$ times. For every time we sample, the probability of sampling an edge containing a particular node with maximum degree $d_{\max}$, is at least $\frac{\text{maximum remaining degree}}{\text{total remaining degree}} = \frac{d_{\max}}{{d_{\max}}^2} = \frac{1}{d_{\max}}$. Combining this with sampling $2d_{\max}-1$ times, for every individual node with maximum degree, we expect to sample that node at least once. This should reduce $\max_k \hat{d}_k$ to $d_{\max}-1$ or lower when having reached $(d_{\max}-1)^2$ edges left to be sampled.

Iterating this logic, we can separate phases 2 and 3 into $d_{\max}$ sub-phases. In each sub-phase t, the expected number of repetitions per edge is then roughly $\left(\frac{\mathbb{E}_k[\hat{d}_k|\hat{d}_k>0]}{d_{\max}-t}\right)^2$. This is some constant if $\mathbb{E}_k[\hat{d}_k|\hat{d}_k>0]$ in sub-phase t is proportional to $d_{\max}-t$, potentially leading to an $O({d_{\max}}^2)$ speed-up.

2.2 With edge weights; the binary case

We now move on to finding an implementation of the algorithm proposed by [Kryven, 2022]. We first focus on the case that $f(r_{ij}) = r_{ij}$, with r_{ij} being either 0 or 1. In other words, f is implied by the edge weights. As a consequence, every edge that can be sampled, is sampled with a probability proportional to $\frac{1}{g(1)}\hat{d}_i\hat{d}_j\left(1-\frac{d_i d_j}{4m}\right)$. Since $\frac{1}{g(1)}$ is a constant, g does not need to be kept track of and can be disregarded for now. Effectively, we tackle the problem in its simplest form.

Because we do not assume any structure in the edge weights r_{ij}, we have to store the edge weights of the edges individually. Because of this, the implementation will only be fast if there are few edge weights with $r_{ij}=1$; edges for which $r_{ij}=0$ do not have to be stored.

This means we have 2 graphs. One that describes the edges that are samplable, these edges all have $r_{ij}=1$, $\hat{d}_i, \hat{d}_j > 0$ and $(i,j) \notin E$, and one that is sampled from the first graph given a degree sequence d. Both graphs change during the algorithm, but the first is the input, and the second is the output of the algorithm. If we refer to the *number of neighbours* N_i of a vertex i, we specifically refer to the number of neighbours in the first graph for which $f(r_{ij}) \neq 0$. This first graph contains the edges that are samplable. By $N_{\max}$, we refer to $\max_k N_k$ on the input, which is also the maximum throughout the algorithm.

2.2.1 Sampling vertices

The main idea of the implementation is to take phase 1 from [Bayati et al., 2010], but instead of sampling both i and j from L, we only sample i from L, and sample j from the neighbours of i. This sampling of j is done by explicitly keeping track of all neighbours of each vertex in a designated array for each vertex.

Before, we could ensure that every vertex i was sampled proportional to $\hat{d}_i$ from L, because after sampling an edge, we could remove the two sampled entries from the active segment. However, because j is no longer sampled from L, we have no way of knowing where to find an entry indicating vertex j. Therefore, we cannot update L immediately after accepting an edge, and we cannot guarantee that every vertex directly sampled from the active segment is sampled proportional to $\hat{d}_i$.

What we can do is update L retroactively, removing entries that should no longer samplable as they are sampled. To achieve this, L is expanded from being a 1-dimensional array of length that only stores the vertex numbers, to being a 2-dimensional array that also stores an index within each vertex. More concretely, if the first vertex, vertex 0, has a degree of $d_i = 3$, L contains the distinct entries $(0,0),(0,1),(0,2)$ at the start of the algorithm, all referring to vertex 0. Because of this, we can then say we only accept a sampled vertex i if the index is less than $\hat{d}_i$. For example, if $(i,0),(i,1)$ and $(i,7)$ are still in the samplable segment of L, but $\hat{d}_i = 2$, then if $(i,1)$ is sampled it is accepted because $1 < 2$, but if $(i,7)$ is sampled it is rejected and removed from the samplable segment. This way, vertices i are still accepted with probability proportional to $\hat{d}_i$, even though they are not directly sampled as such. Because every entry only needs to be removed once, this adds no significant cost.

2.2.2 Managing the neighbours

The arrays used to store the neighbours are structurally similar to L. They also have an active segment that is maintained through swapping entries, and they are also 2-dimensional, with the second dimension consisting of only two entries. The first of these two entries in the array of i is the neighbour j. The second entry stores the index of i in the array of j. This way, if we sample j from the neighbours of i, we can still delete (i,j) from both arrays.

When all degrees of vertex i are satisfied (i.e. when $\hat{d}_i = 0$), i must be deleted as a neighbour from all remaining neighbours of i. Because of this, it can happen that neighbours run out before all remaining degrees are satisfied. If this is the case for a vertex i sampled from L, the number of edges left to be placed needs to be decremented by $\hat{d}_i$ and $\hat{d}_i$ needs to be set to 0.

2.2.3 Die rolls

Combined, vertices i and j are sampled proportional to $\frac{\hat{d}_i}{N_i}$. The next step is to correct for the remaining bias with three die rolls. These are made efficient by two AVL trees, one that stores the number of neighbours N_k, and one that stored all remaining degrees $\hat{d}_k$, both for all vertices k.

The first die succeeds with probability $\frac{N_i}{\max_k N_k}$ and corrects for $\frac{1}{N_i}$. The second die succeeds with probability $\frac{\hat{d}_j}{\max_k \hat{d}_k}$, making the accepting probability proportional to $\hat{d}_i\hat{d}_j$. Lastly, we again toss a die that succeeds with probability $1 - \frac{d_i d_j}{4m}$.

Given that AVL trees need to be maintained to use $\max_k \hat{d}_k$ and $\max_k N_k$ in the dice rolls, they only provides a speed-up when these denominators change significantly throughout the iterations. As will be discussed in sections 2.2.4, $\max_k \hat{d}_k$ and $\max_k N_k$ do not change very much at the start of the algorithm. Because of this, the AVL trees are the main cost at the start of the algorithm. Therefore, this modified phase 1 is then again split into two parts. The first one without AVL trees (where we roll the dice that succeed with probabilities $\frac{N_i}{N_{\max}}$ and $\frac{\hat{d}_j}{d_{\max}}$ instead), and the second one with AVL trees. The transition happens when $\frac{m}{\log(N_{\max})^{\frac{1}{2}}}$ edges are left to be processed.

Since the AVL trees require $O(\log(N_{\max}))$ operations per edge to maintain, this results in the best case cost of $O(m\log(N_{\max})^{\frac{1}{2}})$ operations for sampling edges and updating the AVL tree. Whether

this is indeed the realized cost of the algorithm depends on the average number of repetitions per edge, over the entire algorithm. Empirically, it turns out this average number of repetition is indeed $O(\log(N_{\max})^{\frac{1}{2}})$, making the best case also the most likely candidate. Emptying the graph of neighbours takes $O(nN_{\max})$ operations, resulting in a total run time of $O(nN_{\max}+m\log(N_{\max})^{\frac{1}{2}})$.

2.2.4 Heuristics and intuition

We will now provide some insight as to why this run time also makes intuitive sense, and why the AVL trees are only needed later on. We have previously looked at $\hat{d}$, so now we will be focussing on the number of neighbours.

Given that we have to satisfy $d_{\max}$ degrees for some vertex, it is necessary that $d_{\max} \leq N_{\max}$. Going a step further, assume that $d_{\max} \ll N_{\max}$ and that, initially, $\sum_i r_{ij}$ is roughly the same for all vertices j. In other words, $\frac{N_{\max}}{N_{\min}} := \frac{\max_i \sum_{j\neq i} r_{ij}}{\min_i \sum_{j\neq i} r_{ij}}$ is initially some small constant, with $N_{\min}$ the minimum number of neighbours at the start of the algorithm, i.e. $N_{\min} := \min_i \sum_{j\neq i} r_{ij}$.

Because $d_{\max} \ll N_{\max}$, we have that $N_{\max} \approx N_{\max} - d_{\max} \approx \min_k N_k - d_{\max}$ at the start of the algorithm. In other words, satisfying the degrees of a vertex, which is at most $d_{\max}$, has hardly any impact on the die roll that succeeds with probability $\frac{N_i}{N_{\max}}$. After all, $\frac{\max_k N_k}{N_{\max}} \approx \frac{\min_k N_k - d_{\max}}{N_{\max}}$, at the start of the algorithm.

The only way to make $N_{\max}$ deviate very far from $\min_k N_k$, which in turn increases the run time, is to have vertices frequently hit a remaining degree of 0, which in turn causes the number of neighbours of many other vertices to decrease, likely also decreasing $\min_k N_k$.

Because we are more likely to sample edges between vertices i, j for which the remaining degrees $\hat{d}_i$ and $\hat{d}_j$ are high, it seems plausible that the remaining degree of the vertices should be distributed around some decreasing mean. For most iterations, this mean should be away from 0. In other words, the rate at which vertices i hit $\hat{d}_i = 0$ should increase with the number of placed edges. Therefore, it seems likely that $\min_k N_k$ also decreases most significantly at the end of the algorithm, hence increasing the need for an AVL tree of $\max_k N_k$ as more edges are placed.

Special case example

To make this more concrete, we study the special case for which $d_{\max} = 1$ (and therefore $m = \frac{n}{2}$), and where all vertices i have $N_i = N_{\max}$ neighbours at the start of the algorithm. In this example, we make no use of an AVL tree, and the die rolls are scaled with $N_{\max}$. The goal is to compute the expected total number of repetitions, and to observe that it is much less than $O(N_{\max}m)$, which is the worst case expected number of repetitions per edge, times the number of edges that have to be sampled.

Because $d_{\max} = 1$, we have that at every iteration t, we remove 2 vertices. The expected number of neighbours at iteration t is $\bar{N}_t = \mathbb{E}_t[N_j|\hat{d}_j > 0 : j \in V_n]$. It is related to the value at iteration $t-1$ as $\bar{N}_t = \bar{N}_{t-1} - \frac{2}{n-2(t-1)}\bar{N}_{t-1}$, since $n-2t$ is the number of vertices left at iteration t, and $\frac{1}{n-2(t-1)}\bar{N}_{t-1}$ is the expected number of neighbours of a single vertex at iteration $t-1$. These all have to be removed from the remaining vertices, since they are no longer potential edges.

In other words, we have that $\frac{\bar{N}_t - \bar{N}_{t-1}}{t-(t-1)} = -\frac{2}{n-2(t-1)}\bar{N}_{t-1}$. Through integration, we then find that

$\bar{N}_t = \frac{\bar{N}_0}{n}(n-2t)$.

After all, $\bar{N}_0 = \frac{\bar{N}_0}{n}(n-2\cdot 0)$. Let $\bar{N}_{t-1} = \frac{\bar{N}_0}{n}(n-2(t-1))$, then

$$\bar{N}_t = \bar{N}_{t-1} - \frac{2}{n-2(t-1)}\bar{N}_{t-1} = \bar{N}_{t-1}\frac{n-2t}{n-2(t-1)} = \frac{\bar{N}_0}{n}(n-2t).$$

Since we assumed that $\bar{N}_0 = N_{\max}$, the expected number of repetitions at iteration t is given by $\frac{N_{\max}}{\bar{N}_t} = \frac{n}{n-2t}$. Because we have to sample an edge according to some probability $\frac{n-2}{2}$ times, the total number of repetitions is expected to be $\sum_{t=0}^{\frac{n-2}{2}} \frac{n}{n-2t} \approx -\frac{n}{2}\log|n-2t||_{t=0}^{\frac{n-2}{2}} = O(n\log n)$.

In conclusion, only for very few edges, we expect a high number of repetitions. This is because $\bar{N}_t$ decreases very slowly at the start relative to its value, but very quickly towards the end. In fact, half way, at $t = \frac{m}{2} = \frac{n}{4}$, we find that $\bar{N}_{\frac{n}{4}} = \frac{\bar{N}_0}{2}$. This means that after half the edges, the expected number of repetitions per edge only doubled. In other words, by starting the AVL tree late we exploit that early on, we expect very few repetitions, while still avoiding the high costs that are expected later on.

2.2.5 An alternative implementation

Seeing this implementation, one might notice that the sampling of an edge is structured in three parts. First we sample from an array to avoid a die roll, namely that we avoid $\frac{\hat{d}_i}{\max_k \hat{d}_k}$, then we sample from another array to ensure we sample an edge for which $f(r_{ij}) \neq 0$. This gains us the die roll $\frac{N_i}{\max_k N_k}$. Finally, we fix the remaining biases with die rolls.

Because rolling $\frac{N_i \hat{d}_j}{\max_k N_k \max_v \hat{d}_v}$ turns out to be cheaper than rolling $\frac{\hat{d}_i \hat{d}_j}{(\max_k \hat{d}_k)^2}$, this turns out to be a good implementation. However, it is important to notice that we might as well have enumerated all possible edges for which $f(r_{ij}) \neq 0$, sampled uniformly from those, and then rolled the above die instead. This array of edges can be maintained in various ways, but the simplest is removing edges that have been sampled, and retroactively checking if both $\hat{d}_i, \hat{d}_j \neq 0$. Although this approach is not as efficient, it will provide insight in chapter 4.

2.3 With edge weights; the continuous case

We now extend the binary case by allowing $r_{ij} \in \mathbb{R}_+$. This means that f, the target distribution over edge weights that we want the output graph to have, and g, the distribution over edge weights that we are sampling from, must both play a role. In summary, we gain a die roll (involving f and g) before we can accept an edge, and an AVL tree that aids in this. In contrast to the binary case, the algorithm is no longer split into two parts.

2.3.1 The die roll

Recall that the probability of selecting an edge needs to be proportional to

$$\frac{f(r_{ij})}{g(r_{ij})}\hat{d}_i\hat{d}_j\left(1-\frac{d_i d_j}{4m}\right).$$

In other words, we only need to add a die roll to the binary algorithm that succeeds with a probability proportional to $\frac{f(r_{ij})}{g(r_{ij})}$, to achieve this. To perform this roll, we need to store the edge weights, make calls to f and g, and have a scaling constant.

Edge weights

Because the edge weights are part of potential edges, and potential edges are stored as neighbours, edge weights are also stored as neighbours. This means that every 2-dimensional array that stored the neighbours of a vertex, now gets a 1-dimensional counterpart that stores the edge weight. This counterpart is maintained in the same way. The second dimension of the neighbour array was not expanded to include these weights, since the data types are different. The neighbour array stores integers, whereas the edge weights are in $\mathbb{R}_+$.

Histograms

The calls to f and g are handled by storing them as histograms. This means both are represented by 1-dimensional arrays, where the indexes (i.e. the bins) represent weight ranges, and the entries (not necessarily integers) are proportional to the height of the distributions in those ranges. Although the distributions formally have an area of 1, there is no need for these histograms to sum to 1 since they are scaled by a constant before the die roll.

Since f is an input of the algorithm, the resolution (or bin size), the size of the weight ranges represented by single indexes, is chosen by the one providing f. For g, the resolution is chosen such that on average, there are as many bins as there are edge weights per bin. This means the number of bins is chosen to be $\left(\frac{1}{2}\sum_i N_i\right)^{\frac{1}{2}}$.

Scaling constant

Whereas, f remains the same throughout the algorithm, g does not, and needs to be maintained. After all, every time a neighbour is removed, a potential edge and its accompanying edge weight leaves the distribution we are sampling from.

This has consequences for the way we scale $\frac{f(r_{ij})}{g(r_{ij})}$. Namely, $\max_{r_{ij}} \frac{f(r_{ij})}{g(r_{ij})}$ can now increase as the algorithm progresses. Before, all our die rolls could be scaled by a constant because the scaled values could only decrease. Namely, for all vertices i, $\hat{d}_i$ and N_i can only decrease with the number of iterations. This enabled us to split the modified algorithm into 2 parts, where in the first part we would scale by an upper bound. Now this is no longer feasible, and we must maintain an AVL tree of $\frac{f(r_{ij})}{g(r_{ij})}$ for all edge weights r_{ij} from the start.

It also has consequences for the availability of edges. At some iteration, bins of g become empty as not enough edges are still samplable. As a consequence, we can then no longer sample edges according to f. Possible strategies that can be deployed in this case are changing the bin size of g or simply early termination. Neither are currently implemented in this variant of the algorithm, but for both this is a straightforward process.

2.3.2 Omitting neighbours

Still, we assume no structure in the edge weights, meaning we still have to store all weights. Because of this, we continue to allow edges to be missing from the samplable graph. Before this was possible because $r_{ij} = 0$ implied $f(r_{ij}) = 0$. For the sake of maintaining $f : \mathbb{R}_+ \to \mathbb{R}_+$, we say that we can shift the given edge weights by 1, and adjust the given target distribution f accordingly. We then still define $f(0) = 0$.

This enables preprocessing strategies, where one can obtain a subset of edges through some other means, and then feed this subset to this algorithm to obtain the final graph. For example, one could use the structure of the edge weights to construct the subset to be roughly according to f. The implementation of the continuous case can then be used both as a refinement step for the distribution, and as a way to sample according to a given degree sequence.

2.3.3 An alternative implementation

When sampling edges, the main implementation we just discussed still follows the same approach as the main implementation of the binary case, just with an additional die roll. If f is very close to g, this additional die roll hardly has any overhead. It will almost always succeed with a probability close to 1.

However, if f and g are very different, one should first ask if there are enough edges to sample a graph with edge weights according f to begin with. If this is the case, one could argue that they are probably not so different after all, and that this overhead is not too high.

Still, if one insists that this overhead is significant, we can eliminate the cost of this die roll, along with that of $\frac{N_i}{\max_k N_k}$, in the first step where we sample from an array. In return, we then gain the die roll $\frac{\hat{d}_i}{\max_k \hat{d}_k}$.

Eliminating $\frac{f(r_{ij})}{g(r_{ij})}$

In the binary case, we simply enumerated all edges for which $f(r_{ij}) = 1$, and we sampled from that array uniformly. This is a special case of the two-step process we use here. First, notice that f does not change throughout the algorithm. This means we can convert the histogram, which represents the probability density function, into a cumulative distribution with a single loop over the array. We can then invert this cumulative distribution in another single loop over the array. We can then sample from f in constant time by sampling from the indices of this inverted cumulative distribution. Sampling a bin from f is the first step in selecting an edge with probability $\frac{f(r_{ij})}{g(r_{ij})}$.

Now bin g and f in the same way. In other words, have them share the same start and end point, the same number of bins, and ensure that each bin has the same size as their counterpart. In short, for every bin in f, there is a corresponding bin in g. Instead of storing the number of samplable edges in each bin, store the edges themselves. In the first step, we sampled a bin b according to f, in the second step, we uniformly select an edge from this bin. Because there are $g(b)$ edges in this bin, and bins are a proxy for the edge weights, this edge (i, j) is sampled with probability approximately $\frac{f(r_{ij})}{g(r_{ij})}$, which gets more accurate the more bins there are.

As an aside, notice that as long as all die rolls are independent, the resulting output will always

have edges sampled according to f. In other words, the scaling factor $\frac{1}{g(r_{ij})}$ is just a consequence of uniformly sampling from each bin, which is not necessary for obtaining a graph of which the edge weight distribution is f.

2.4 An efficient pipeline

We have now discussed a method to sample a graph according to a degree sequence, and with the edge weights following a target distribution. Note that if we were to use all edges in the generation of a graph, then N_i would be n at the start of the algorithm for all vertices i. As a result, the proposed method would require $O(n^2)$ storage and operations. This is too much and something we would like to avoid.

Because of this, we suggest creating a random subset of the edges first, by sampling some number of edges from all edges with probability $\frac{f(r_{ij})}{g(r_{ij})}$. Then from this subset we build a random graph. This is likely not the same as generating a random graph directly, but for a reasonable run time it seems a necessity.

2.4.1 Polar coordinate subsetting

We will now discuss how to obtain such a subset, if all edge weights are distances, and all vertices have 2-dimensional locations. Polar coordinate subsetting takes as main inputs an array of locations of the vertices and a target distribution f, and it outputs a subset of edges with the distances distributed according to f, obtained by sampling edges according to $\frac{f(r_{ij})}{g(r_{ij})}$.

The main idea of the algorithm is to sample a length according to f as many times as we require edges in the subset, and for every length, loop until an edge is found such that the length of the edge sufficiently matches the sampled length. This is similar to what we did in section 2.3.3. There, we sampled uniformly from all edges with a matching length because we had a list of all edges with a matching length. This time we obtain the uniform sample by looping continually until an edge is found. It works because every edge with a matching length is roughly equally likely to be sampled.

Storing the locations

Before we can do any of that, however, we must first store the locations in a way that facilitates sampling edges, and that captures the nature of the edge weights we are trying to exploit. In other words, because we have 2-dimensional locations, the locations are stored in a 2-dimensional grid.

Storing the edges is done in two steps. We first store the number of edges per grid cell in a square $n^{\frac{1}{2}} \times n^{\frac{1}{2}}$ grid such, such that on average, there is approximately only 1 vertex per cell. Then we observe the maximum number of vertices that coincide in one cell, and store the locations as a 3-dimensional array. Note that the method can easily be adapted to use a rectangular grid as well. Optionally, we can also observe how many cells n' are actually filled, and store the vertices in a $n'^{\frac{1}{2}} \times n'^{\frac{1}{2}}$ grid instead. This resolves issues with many vertices having very similar locations.

Sampling an edge

The sampling of edges is done by first sampling a distance from f in constant time. Again, this can be done by computing the cumulative distribution, and then inverting it once, at the start of the algorithm. Then, we uniformly sample a vertex from V_n and an angle from $[0, 2\pi)$. From the location of the vertex, we move in that direction with the sampled distance, and find the corresponding cell.

We then uniformly sample an index in the cell (in the depth dimension). If this entry contains a vertex, we have not sampled this edge before hence $(i, j) \notin E$ and $f(r_{ij}) > 0$, we add the edge to the output E. Note that if we sampled an index from the number of available vertices in the cell, there would be a bias towards vertices that are alone in their cell, which we want to avoid.

If we cannot add the edge to the output, we sample a new vertex, direction and cell index. We only resample the distance f if an edge is accepted. By not resampling the distance from f, the distribution of edge weights is close to f. This is because all distances are accepted, even though it might take a longer time to find an edge for some distances.

Note the similarity with sampling uniformly from all edges in a bin. For a given sampled distance, we can construct a bin by walking over the circle around each vertex, and adding all possible edges for each grid location we come across. Although the probabilities of obtaining different edges vary depending on the length of the curve going through the grid cells, these variations should not affect some edges more than others over many iterations. This is because these variations are an artefact of the way the locations are stored, and the distances are constantly being resampled from a continuous distribution, and measured from the vertex locations, which are also random.

Dealing with $f(r_{ij}) = 0$

In other words, every iteration, we sample from a distribution of edge weights centred around the distance we sampled from f. This means the probability of sampling an edge comes from sampling nearby distances from f. As a consequence, if the f is suddenly set to 0 after some distance, edges close to this cut-off point are undersampled compared to edges with a weight further away. These closer edges are then missing up to half of their source of probability of being selected. Another consequence of this is that distances selected close to this cut-off point can only be sampled in one direction, causing a slight bump in probability at a medium distance from the cut-off point.

To counteract this to some extent, we check if the distance we are about to sample has probability zero. If it does, we set the desired sampled distance from f to be the closest cut-off point and we try again. Because of this, there being no anomalies near $\frac{1}{2}\sqrt{2}$ in figure 3.2, that would otherwise be there.

2.4.2 An implementation without storing edge weights

Having this way of sampling edges according to $\frac{f(r_{ij})}{g(r_{ij})}$ without storing edge weights enables us to implement the alternative implementation of section 2.3.3, without the hurdle of storing $O(n^2)$ edge weights. Still, there are two main differences in the way edges are sampled.

Grid resizing

The first is that as the degrees of vertices are satisfied, they no longer need to be present in the grid. If the algorithm fails to find an edge for a sampled distance by exceeding some number of attempts, this can now also be because the grid is too large for the number of available vertices present in it. This is much like the problem where we eliminate $\frac{f(r_{ij})}{g(r_{ij})}$ by sampling a bin according to f, but then the corresponding bin in g turns out to be empty.

Before, rather than rescaling f and g to have a larger bin size, it could be tempting to simply terminate if that happened and accept the error. However, now this sparsity also affects the run time since it makes finding edges much slower. Because of this, we opt to do the equivalent of rescaling f and g, which is recomputing the grid whenever the number of vertices that still have remaining degree, halves.

This means we remove all vertices which had their degrees satisfied, and we fill a new grid such that again, every grid location has 1 vertex on average. If the algorithm then still fails to find a suitable edge in $\frac{1}{2}m$ iterations, the grid is recomputed one last time, before the algorithm is terminated at m iterations, as it is no longer possible to sample vertices according to f. Generally, this step does not occur, which means the total run time for maintaining the grid is generally $O(n)$, since the grid halves in size every time it is recomputed. Regardless, the algorithm dictates that we continue sampling even if it is no longer possible to sample edges according to f.

Finishing up

Because of this, the algorithm is continued with the subset method, where in this case, the subset of edges that can be sampled are all edges $(i.j)$ that have not been sampled yet, for vertices i, j that do not yet have their degrees satisfied hence $\hat{d}_i, \hat{d}_i > 0$, and for which $f(r_{ij}) > 0$.

If we terminate the method without subset when there are fewer than $\sqrt{m}$ vertices with some remaining degree left, since this only means storing $O(m)$ potential edges. This is relevant because it can occur that there are few remaining edges of high demand lengths, while there are also many vertices left with remaining degree. This makes the grid unsuitable for sampling according to the desired distribution, increasing the run time.

By not waiting for edges to run out, and by instead terminating early, we try avoiding this issue. Still, we do not want to terminate too early either, since it results in storing too many potential edges. The same strategy is applied to the subset method, where we instead terminate early based on the number of edges in the subset, rather than m.

Run time

Regardless of this issue near the end of termination, this implementation without storing edge weights is possible because, on average, there is one vertex per grid location. Ignoring the probability of finding an edge that is already present in the graph, this makes the probability of finding a suitable edge only dependent on the depth of the grid, which in turn is only dependent on the distribution of the vertices. In other words, the probability of failure is very low.

This allows us to increase the threshold, of how many consecutive failures to sample a candidate edge from the grid constitutes a definitive failure, with m, without it affecting the run time much.

2.4.3 Using the binary method for increased speed

The method that uses an initial subset now consists of three phases. Namely, first we build the subset. Then we run the method that takes in this subset, and we terminate early. Lastly, we finish up by running the method again on all edges that are left over. Because we build the first subset such that the edges are distributed according to f, we do not necessarily need to run the continuous implementation, and we can instead use the faster binary implementation, since the edge weight distribution does not need to be corrected. Still, in the last stage we then need to revert to using the continuous implementation because the leftover edges can have any edge weight distribution again.

In total, this yields 3 implementations of the algorithm, one without an initial subset, and two without. In the next chapter, we will empirically evaluate these methods.

Chapter 3

Empirical behaviour

Having discussed the implementations, we now investigate the empirical behaviour of these methods. On the one hand, the goal of this chapter is to show what the algorithm proposed by [Kryven, 2022] is good at in practical experiments. On the other hand, the goal is to show that the implementation for distances does indeed perform as expected. The properties we are interested in are the run time, converge to f, and uniformity, and mainly for large graphs.

3.1 Experimental setup

3.1.1 Various limits

However, what constitutes a large graph is vague. Think of the vertices as random locations according to some distribution, for example, home addresses of people. There are many ways in which the population n in the graph can increase while f remains the same.

For example, there could be more and more people in the same area because we are sampling more and more locations from the same distribution. This is the simplest case.

The area where people are located could also increase with the number of people, meaning the total distribution changes from which we sample locations as there are more people. For example, instead of modelling just a city, we might want to model an increasingly large area such as a country instead, even if not all regions are densely populated. Similarly, the target distribution f could also change with n, as people find their connections closer to home.

It could also be that the people get to know more people as n increases, hence $d_{\max}$ should increase along with m, following the requirement that $d_{\max} < m^{\frac{1}{4}}$, where m increases $O(n^{\frac{4}{3}})$. If we want to use the method at its full capacity, we do not want to be stuck choosing $d_{\max}$ very small for the sake of uniformity, run time or the accuracy of f.

3.1.2 Vertex distribution

To perform these experiments, we need a distribution for the vertex locations. To keep the results as general as possible, we decided on sampling both the x and y coordinates from the product of two uniform $U(-1, 1)$ distributions for its great heterogeneity in density. This is the same as sampling x and y from the probability density function $-\log(z)$ for $z \in [0, 1]$ and choosing the signs of x and y uniformly at random. The blue dots in figure 3.1 are vertex locations sampled according to this distribution.

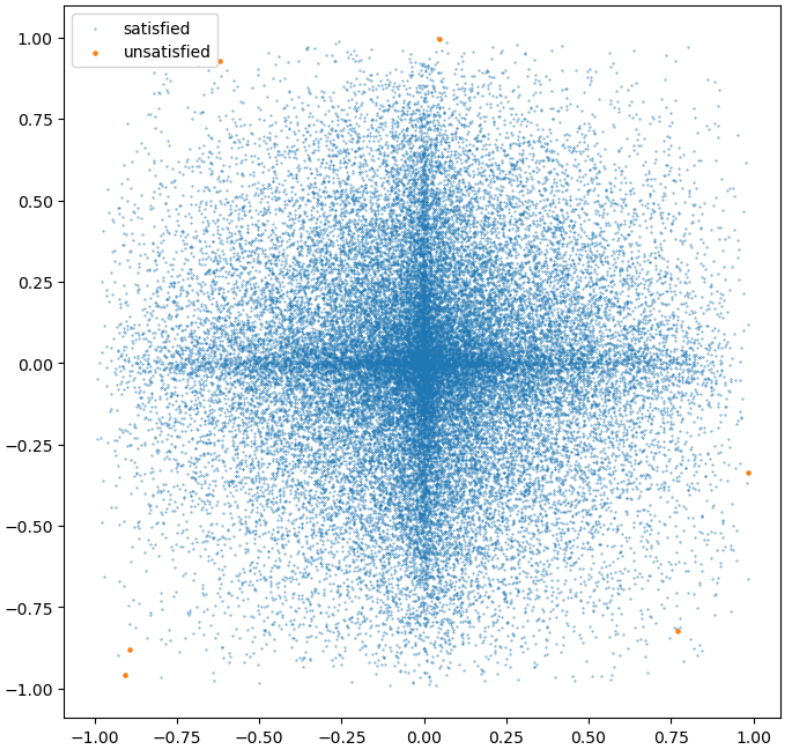


Figure 3.1: Vertex distribution. The distribution of vertices as described in section 3.1.2. The blue dots show the sampled vertices, the orange dots show the vertices left with remaining degree after termination of a run of the algorithm. The degrees of the vertices could not be satisfied due to the distance being more than $\frac{1}{2}$, or because they are already connected.

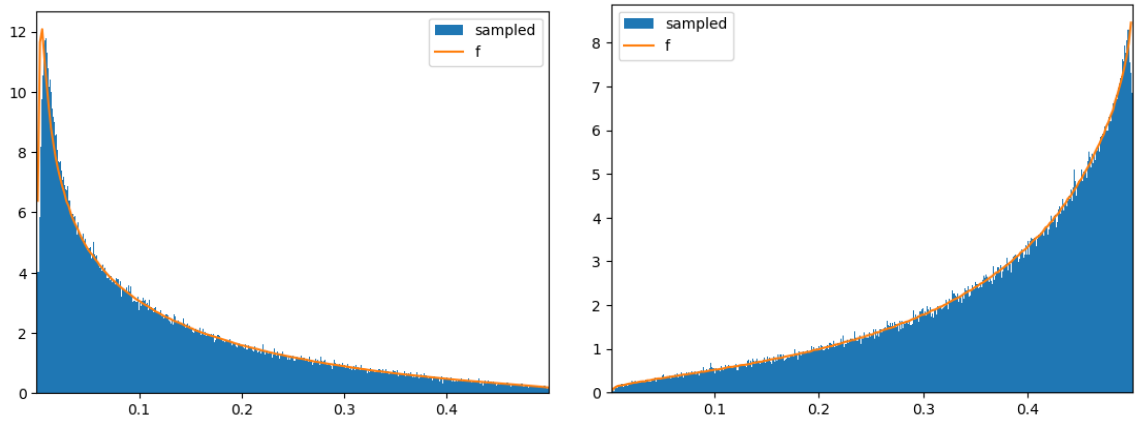
3.1.3 Target distributions

For this distribution, we then measured the edge weight distribution for a vertex situated at $(0, 0)$ and at $(\frac{1}{2}, \frac{1}{2})$ by sampling 10^8 vertices from this distribution and measuring the distances. We smooth these distributions and call them f_0 and $f_{\frac{1}{2}}$. From this we construct the two distributions we use as target distributions.

First, we choose f_{close} to be $\frac{f_0}{f_{\frac{1}{2}}}$, for $r_{ij} < \frac{1}{2}$, and 0 otherwise, since it favours short edges. Second,

we choose f_{far} to be $\frac{f_{\frac{1}{2}}}{f_0}$, for $r_{ij} < \frac{1}{2}$, and 0 otherwise, since it favours long edges. Both have the probability of selecting an edge set to zero for edges longer than $\frac{1}{2}$, because this way we still have enough edges to match the target distribution (to some extent). These density functions are shown as the orange lines in figure 3.2.

The point of selecting the target distributions like this is that the order in which the degrees of vertices are satisfied is different. Most short edges are located near $(0, 0)$, so if f_{close} is used, that is where most edges are first sampled. In other words, the graph is constructed from the center outwards. If f_{far} is used, the graph is constructed from the locations $(\pm\frac{1}{2}, \pm\frac{1}{2})$ towards $(0, 0)$ instead. Because most vertices are located near $(0, 0)$, this can leave a large portion of the vertices stranded, since it is no longer possible to sample the desired edge lengths when the degrees of further away vertices are already satisfied. In turn, this can have an effect of the convergence towards the target distribution.



(a) close

(b) far

Figure 3.2: Target distribution. The target distribution of edge weights as described in section 3.1.3 (in orange), and as sampled in a run of the algorithm (in blue).

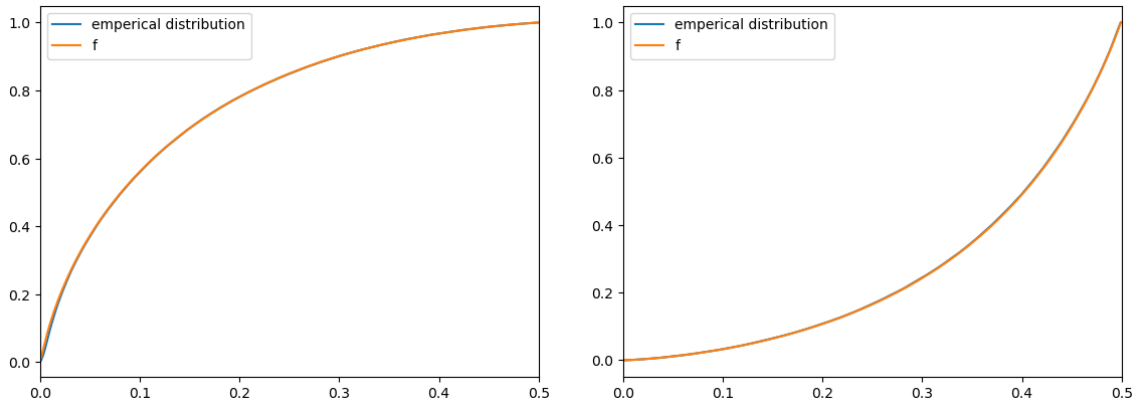
Convergence to the target distributions

The convergence of the distribution of the sampled edge weights to f, is measured as the convergence of the unsigned area between the cumulative distribution of f, and the empirical edge weight distribution, as if the cumulative distribution runs from $(0, 0)$ to $(1, 1)$ in a square. This is done such that if the domain of the target distribution changes, the distance measure would be unaffected. The cumulative target distributions are shown in figure 3.3.

3.1.4 Uniformity

Whereas the run time and the edge weight distribution are easy to directly measure, uniformity is complicated since the method does not compute the probability of the generated graph.

A difference between the methods proposed by [Kryven, 2022] and [Bayati et al., 2010] is that



(a) close

(b) far

Figure 3.3: Cumulative distribution. The cumulative distributions of the distributions shown in figure 3.2.

the latter produces an output with degree sequence d with high probability, whereas the former will frequently terminate without having satisfied all degrees. Because of this, the probability of obtaining each graph varies. This is because graphs with more edges generally have a lower probability of being obtained.

What we can do is report the number of degrees that were failed to be satisfied, the error. If the variance of the error increases with m, and we assume the average log probability per edge to be the same for each graph, then we can assume the graphs do not become uniform. In such a case, it is entirely possible that for the particular combination of edge weights and target distribution, it is simply not possible to sample such a graph, but this is unknown.

3.1.5 Evaluated methods and parameters

We evaluate both the method that samples edges from the grid directly, and the methods that build a subset before starting. This number of edges in the subset is set to be $4d_{\max}\frac{n}{n'}n^{\frac{1}{3}}$, where n' is chosen to be the largest constant such that we always have at least $4d_{\max}n$ edges in the subset.

This subset is kept the same where possible due to it being prohibitively expensive to generate. As a consequence, the cost of generating this subset is not included in the recorded metrics. The degree sequence is always renewed for every generated graph. Each experimental configuration is repeated 30 times.

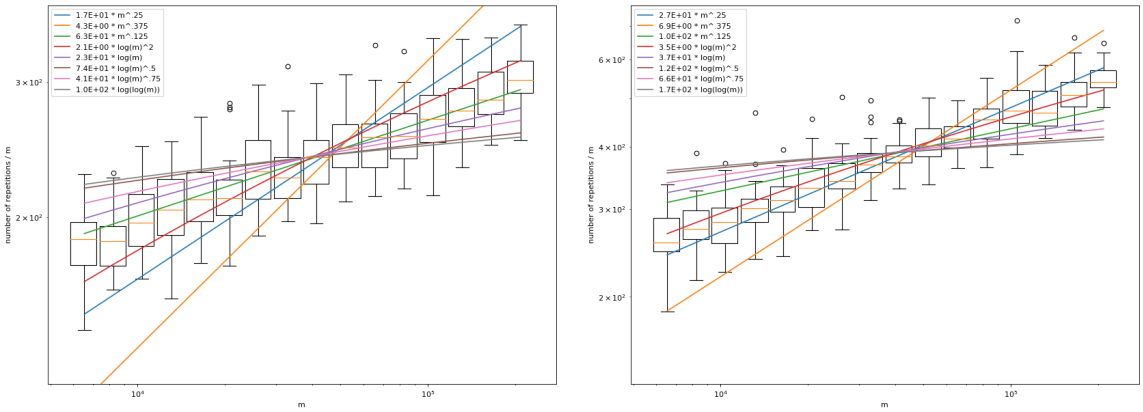
3.2 Results

3.2.1 A denser population with constant $d_{\max}$

The first limit we are interested in is when we fix $d_{\max}$ to be a constant, namely 8, and uniformly sample degrees from $[1, d_{\max}]$ for each vertex, while increasing the number of vertices n that are sampled from the fixed distribution of locations.

Run time

Without the use of a subset, the average number of attempts to sample an edge, and hence the run time per edge, appears to be $O(m^{\frac{1}{8}})$, as can be seen in figure 3.4.



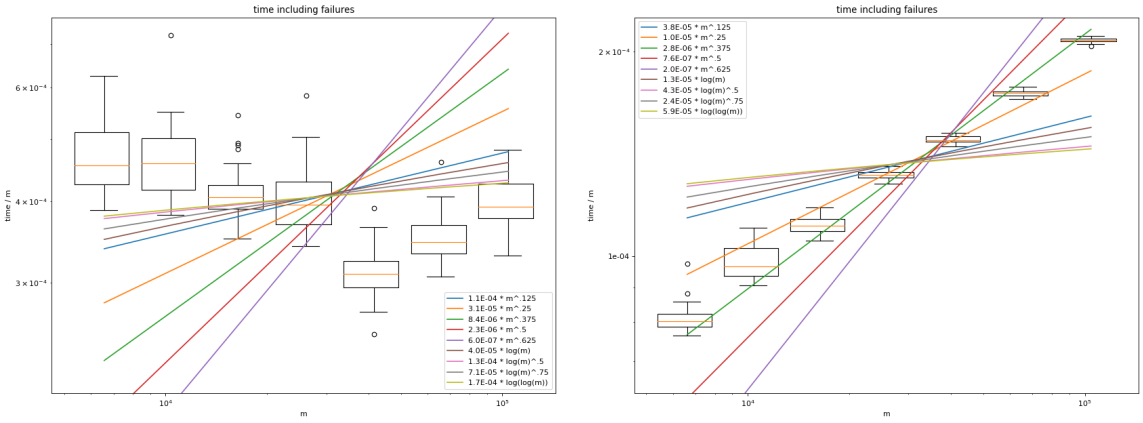
(a) close

(b) far

Figure 3.4: Average number of repetitions per edge without the use of an initial subset.

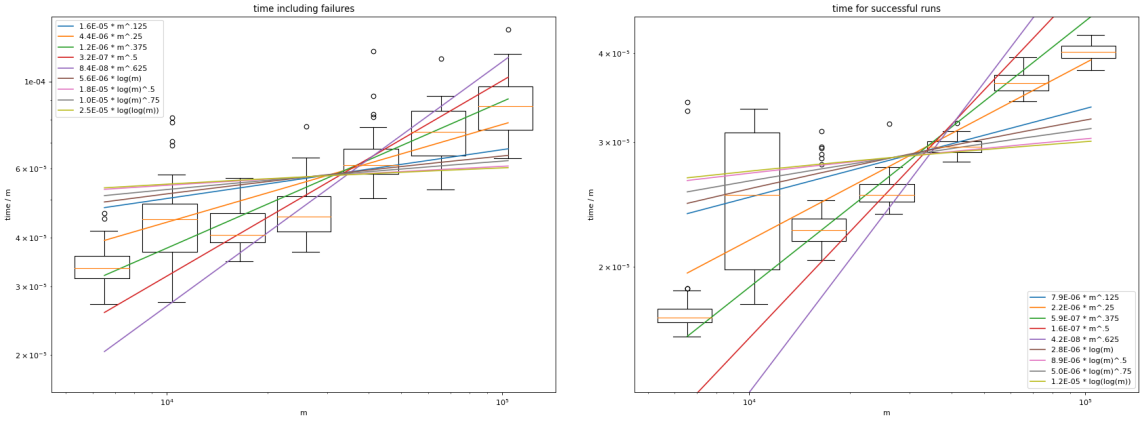
With the use of a subset, the average run time per edge greatly differs between the two target distributions. It is the difference between $O(1)$ for f_{close} and roughly $O(m^{\frac{3}{8}})$ for f_{far}, as can be seen in figures 3.5a and 3.5b. Still, we know the run time should be at least $O(m^{\frac{1}{3}})$ per edge by the design of the experiment.

Adding the use of the binary method reduces the run time roughly by a factor 5. The run time per edge appears to be close to the desired $O(m^{\frac{1}{3}})$ for both f_{far} and f_{close}, as can be seen in figures 3.6a and 3.6b.



(a) close (b) far

Figure 3.5: Average run time per edge with the use of an initial subset.

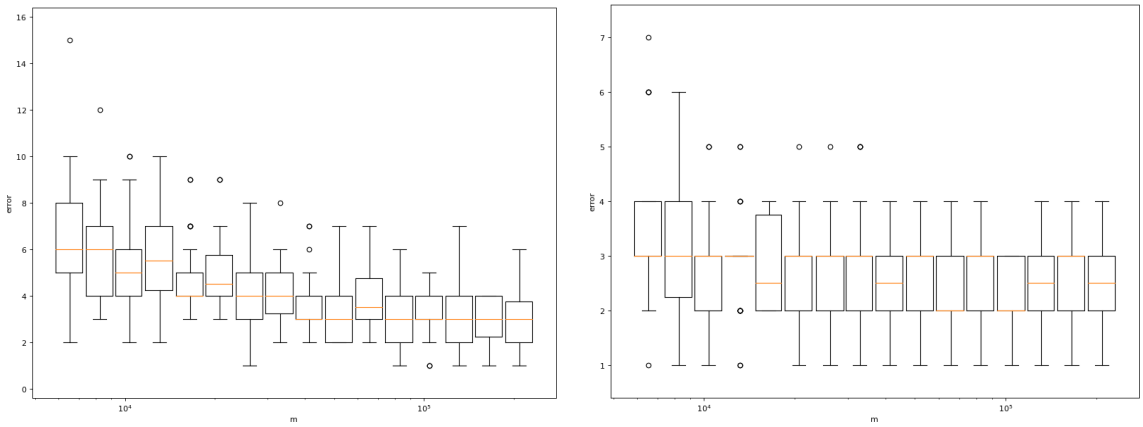


(a) close (b) far

Figure 3.6: Average run time per edge with the use of an initial subset and the binary method.

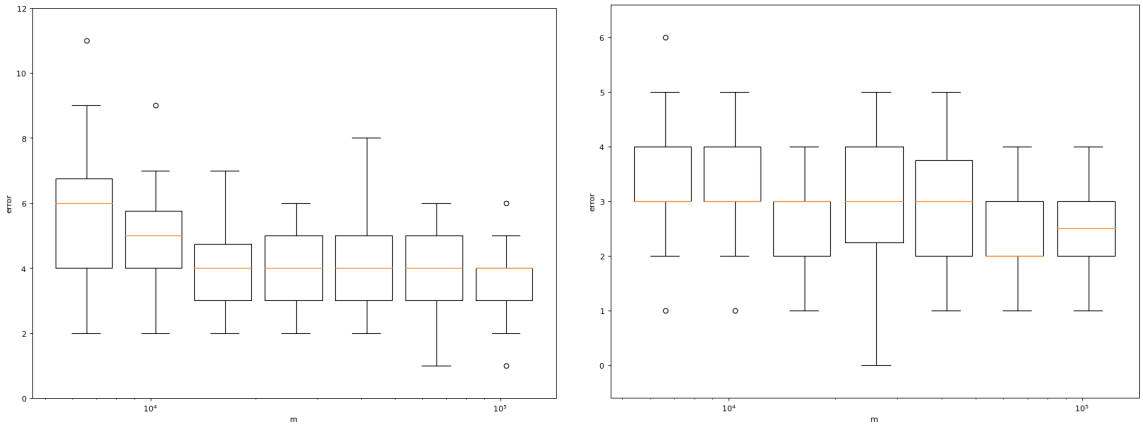
Uniformity

It turns out in all cases, the error does not increase with m, but instead stays $O(1)$, as can be seen in figures 3.7, 3.8, and 3.9. In other words, no evidence is found against some form of uniformity.



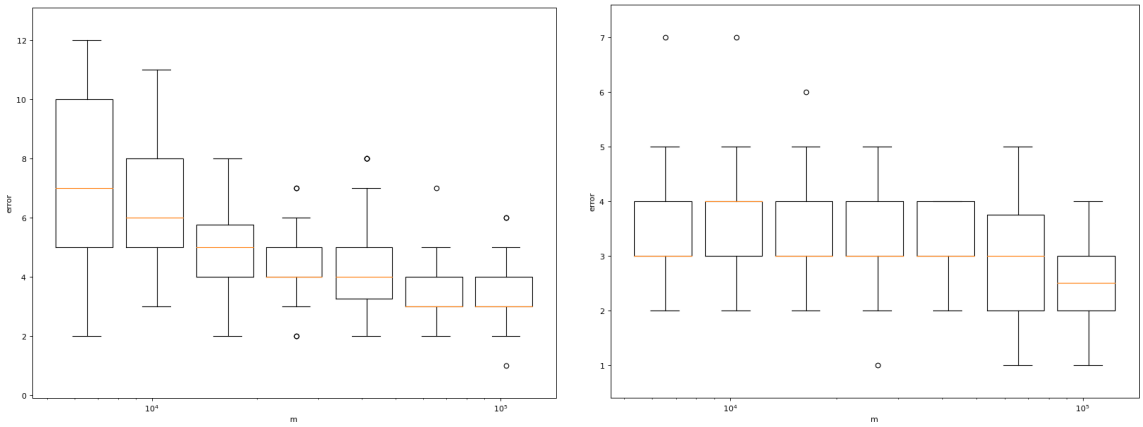
(a) close (b) far

Figure 3.7: The number of unsatisfied degrees without the use of an initial subset.



(a) close (b) far

Figure 3.8: The number of unsatisfied degrees with the use of an initial subset.

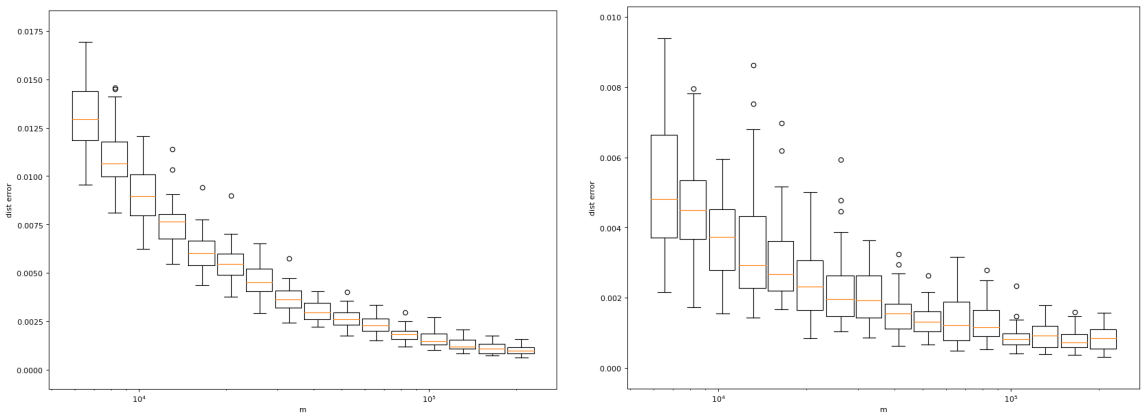


(a) close (b) far

Figure 3.9: The number of unsatisfied degrees with the use of an initial subset and the binary method.

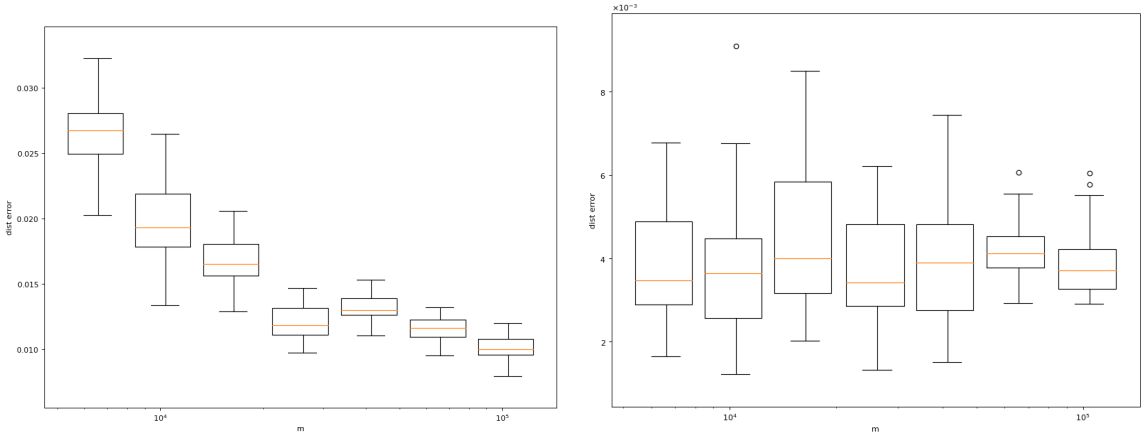
Convergence to the target distributions

The empirical edge weight distribution appears to converge to the target distributions in all cases, except when the target distribution is f_{far} and we apply the method that uses an initial subset of the edges. This can be seen in figures 3.11b and 3.12b.



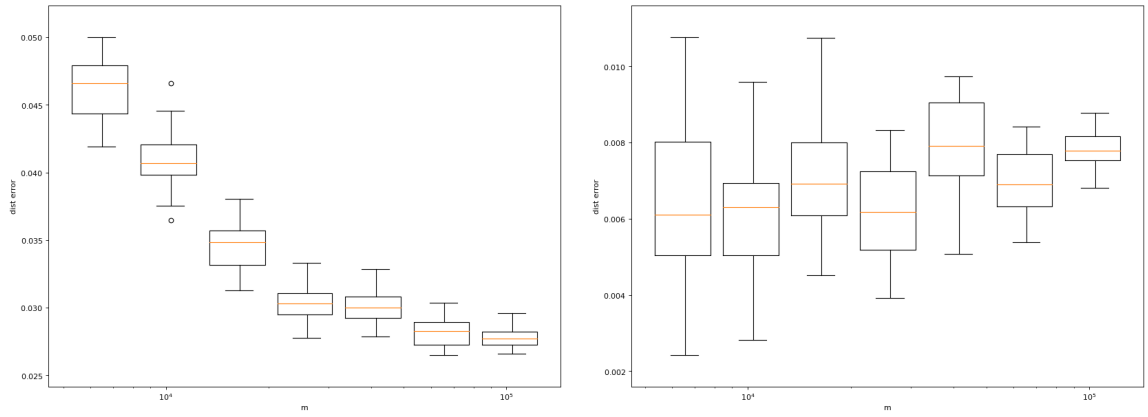
(a) close (b) far

Figure 3.10: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, without the use of an initial subset.



(a) close (b) far

Figure 3.11: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset.



(a) close (b) far

Figure 3.12: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset and the binary method.

3.2.2 A denser population with increasing $d_{\max}$

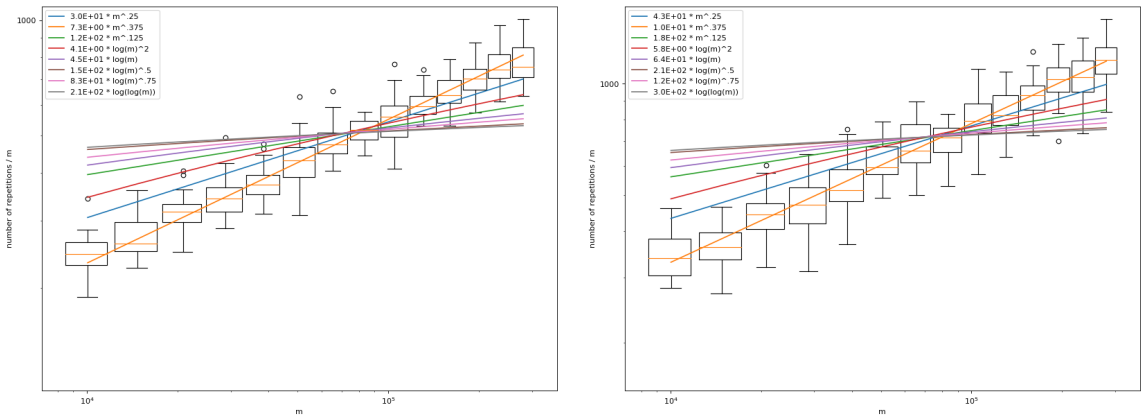
The second limit we are interested in is when we let $d_{\max}$ and n increase together. Such a process can be interesting when wanting to use the method at its full potential, namely whenever $d_{\max} \approx m^{\frac{1}{4}}$.

We investigate this by incrementing $d_{\max}$, and then defining n such that m is on average ${d_{\max}}^4$. We are still sampling the degrees uniformly from $[1, d_{\max}]$, hence we define n as $\frac{4{d_{\max}}^4}{d_{\max}+1}$ rounded to the nearest multiple of 2. We resample a sequence if $d_{\max} \geq m^{\frac{1}{4}}$. $d_{\max}$ is increased from 10 to 22.

Run time

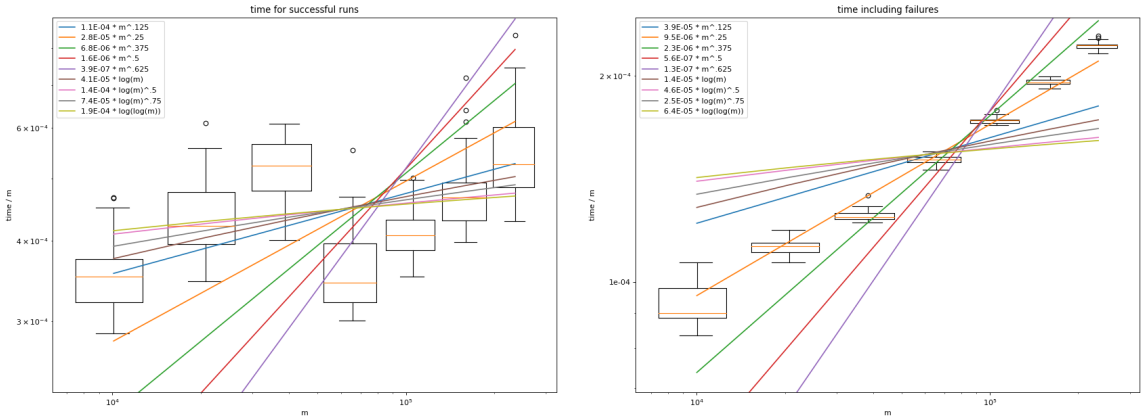
Without the use of a subset, the average number of attempts to sample an edge, and hence the run time per edge, appears to be $O(m^{\frac{1}{8}} d_{\max})$, as can be seen in figure 3.13, and from the observation earlier that the run time appears to be $O(m^{\frac{1}{8}})$ per edge when fixing $d_{\max}$ to be a constant. Although we roll two dice that are scaled with $\max_k \hat{d}_k$, we only obtain $d_{\max}$ as an additional term in the run time, and not ${d_{\max}}^2$. This was already discussed in section 2.1.2, and is why the binary method effectively trading the die roll of $\frac{\hat{d}_i}{\max_k \hat{d}_k}$ for $\frac{N_i}{\max_k N_k}$ is a good idea.

With the use of a subset, the run time again appears to hover around the desired $O(m^{\frac{1}{3}})$, as can be seen in figures 3.14 and 3.15, with the binary method causing a factor 5 speed up.



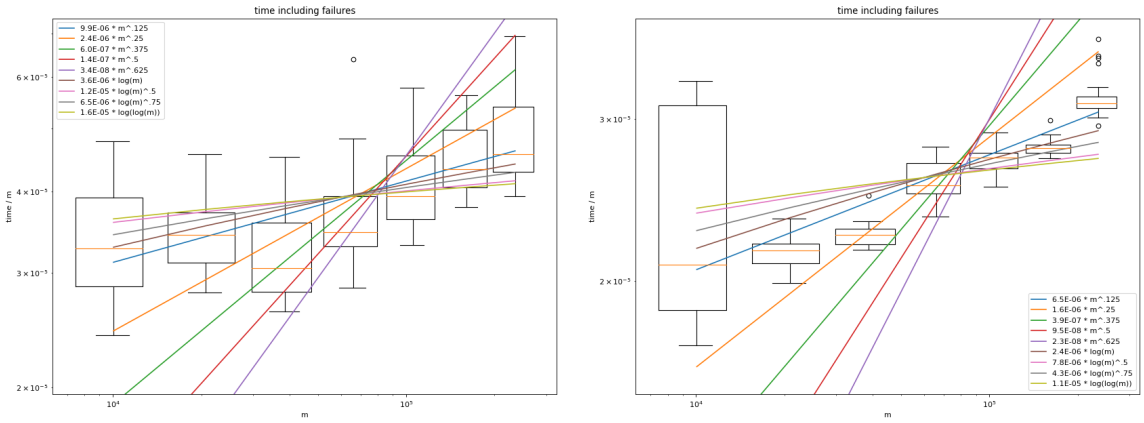
(a) close (b) far

Figure 3.13: Average number of repetitions per edge without the use of an initial subset.



(a) close (b) far

Figure 3.14: Average run time per edge with the use of an initial subset.



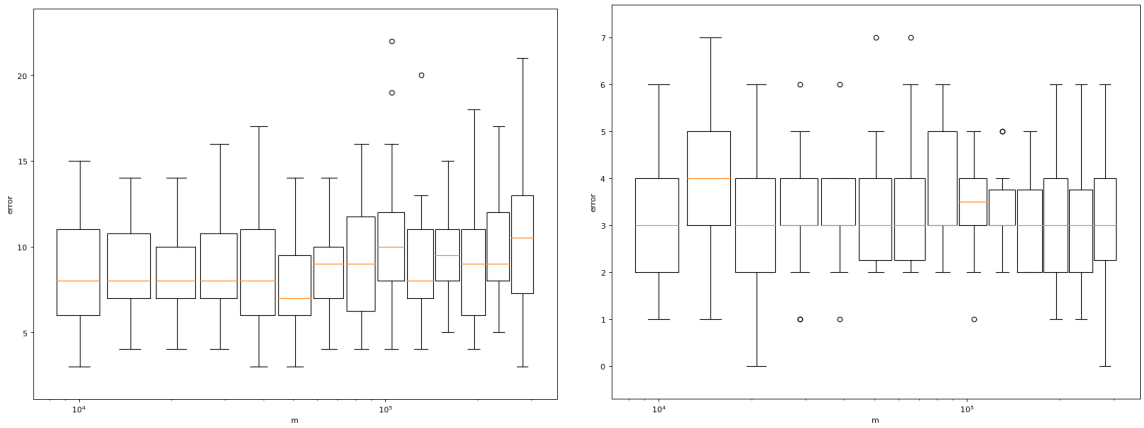
(a) close

(b) far

Figure 3.15: Average run time per edge with the use of an initial subset and the binary method.

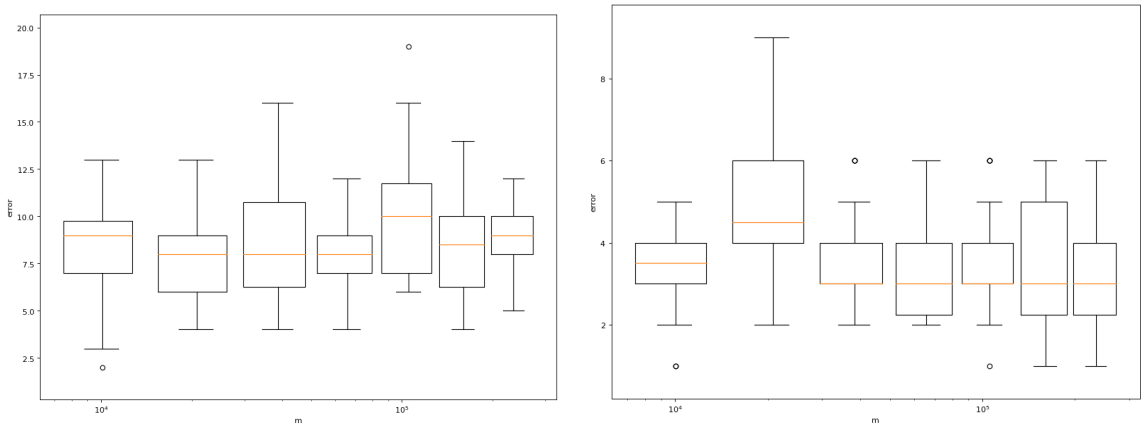
Uniformity

Again, it turns out in all cases, the error does not increase with m, but instead stays $O(1)$, as can be seen in figure 3.17. In other words, no evidence is found against some form of uniformity.



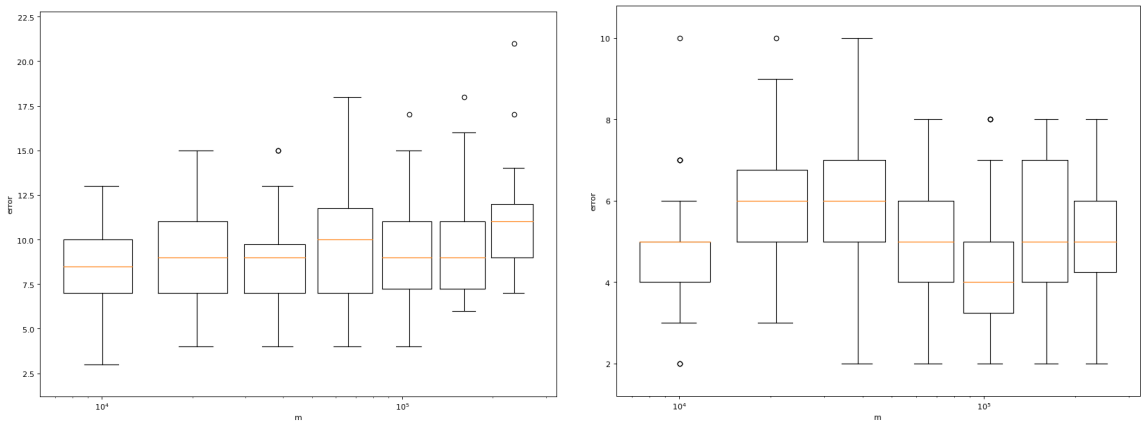
(a) close (b) far

Figure 3.16: The number of unsatisfied degrees without the use of an initial subset.



(a) close (b) far

Figure 3.17: The number of unsatisfied degrees with the use of an initial subset.

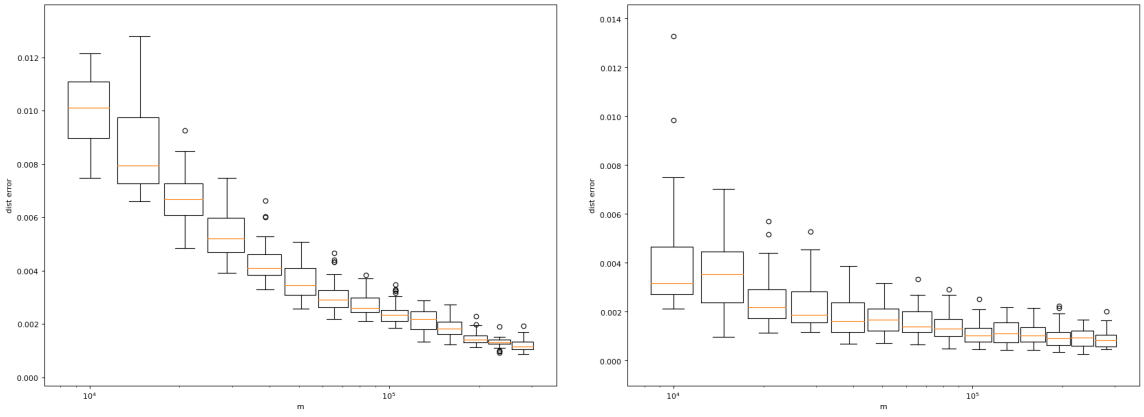


(a) close (b) far

Figure 3.18: The number of unsatisfied degrees with the use of an initial subset and the binary method.

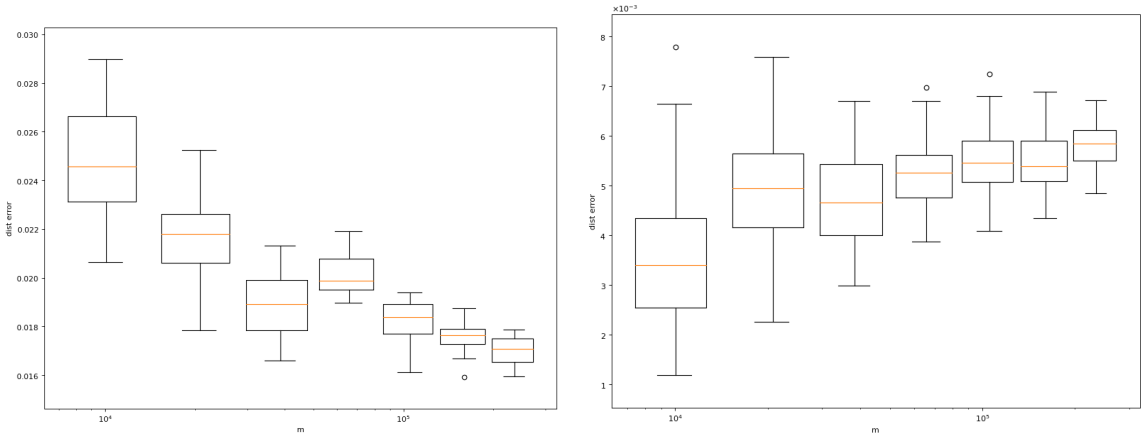
Convergence to the target distributions

The empirical edge weight distribution appears to converge to the target distributions in all cases, except when the target distribution is f_{far} and we apply the method that uses an initial subset of the edges. This can be seen in figure 3.20b.



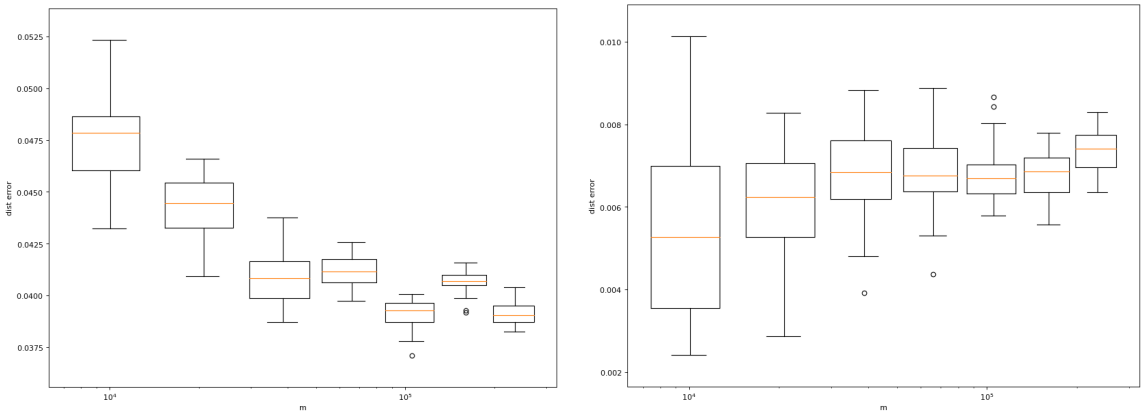
(a) close (b) far

Figure 3.19: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, without the use of an initial subset.



(a) close (b) far

Figure 3.20: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset.



(a) close (b) far

Figure 3.21: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset and the binary method.

3.2.3 A changing distribution with constant $d_{\max}$

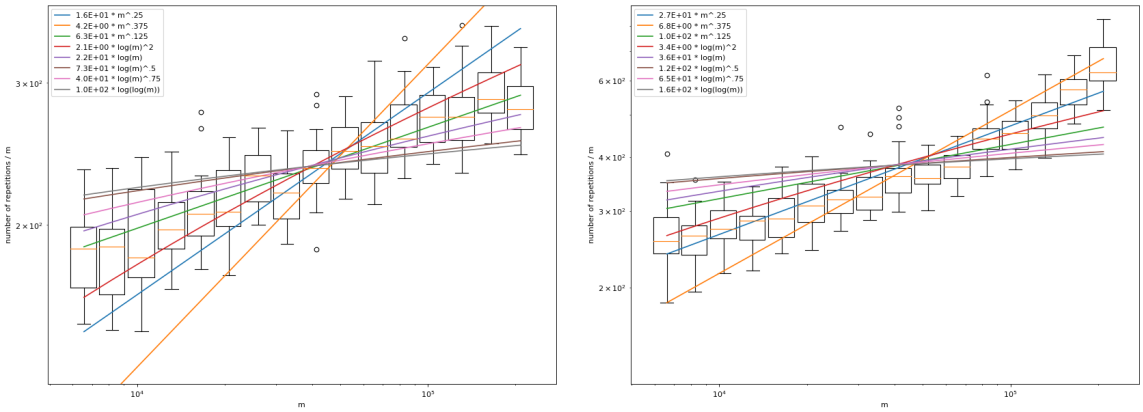
Finally, we are interested in what happens if the number of available edges does not grow as quickly as the number of edges. For f_{close}, we shrink the radius of the disk in which the probability of sampling an edge is not zero. For f_{far}, we sample from an annulus instead, by increasing the minimum distance and decreasing the maximum distance, both at the same rate towards $\frac{1}{4}$.

The radius of the disk and the width of the annulus are chosen to be $\frac{1}{2}\left(\frac{n'}{n}\right)^{\frac{1}{3}}$, with n' being the

constant, such that the radius is $\frac{1}{2}$ for the lowest n. In other words, the number of potential edges now grows as $O(n^{\frac{4}{3}})$, instead of $O(n^2)$.

Run time

Without the use of a subset, the average number of attempts to sample an edge, and hence the run time per edge, appears to again be $O(m^{\frac{1}{8}})$, but only when the error remains near 0, as can be seen in figure 3.22.

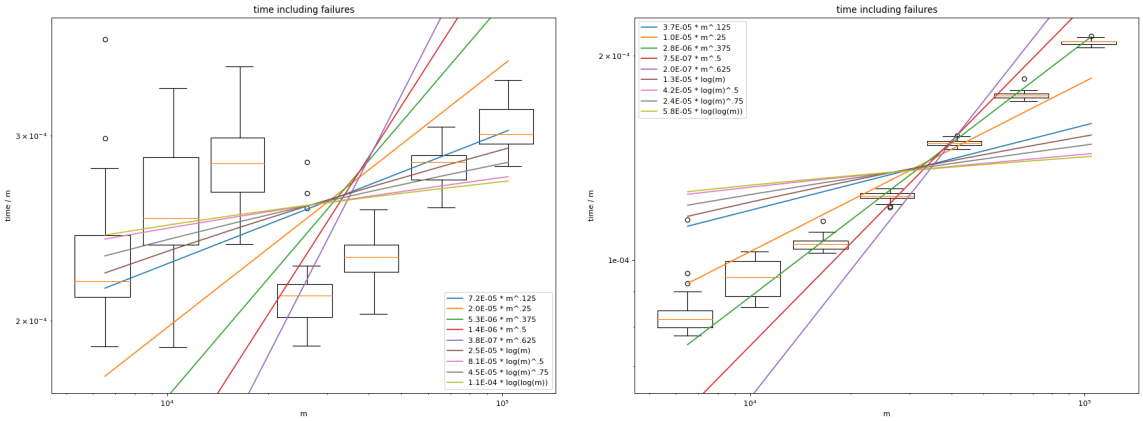


(a) close

(b) far

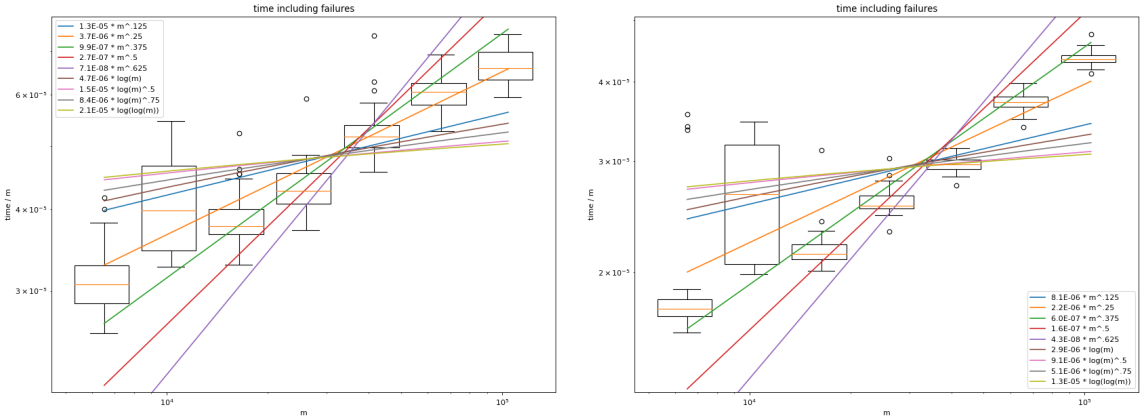
Figure 3.22: Average number of repetitions per edge without the use of an initial subset.

With the use of a subset, the run time again appears to be close to the desired $O(m^{\frac{1}{3}})$ per edge, and the use of the binary method again appears to reduce the run time by a factor 5, as can be seen in figures 3.23 and 3.24.



(a) close (b) far

Figure 3.23: Average run time per edge with the use of an initial subset.

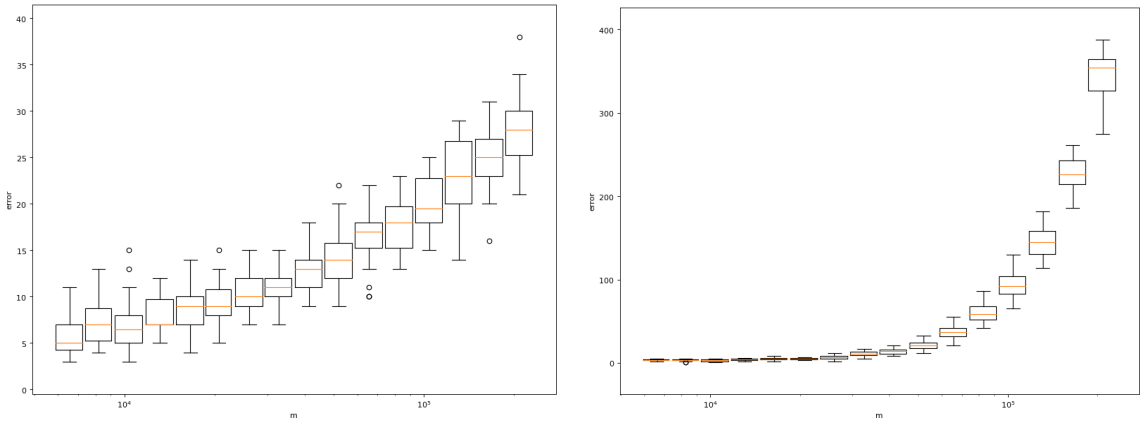


(a) close (b) far

Figure 3.24: Average run time per edge with the use of an initial subset.

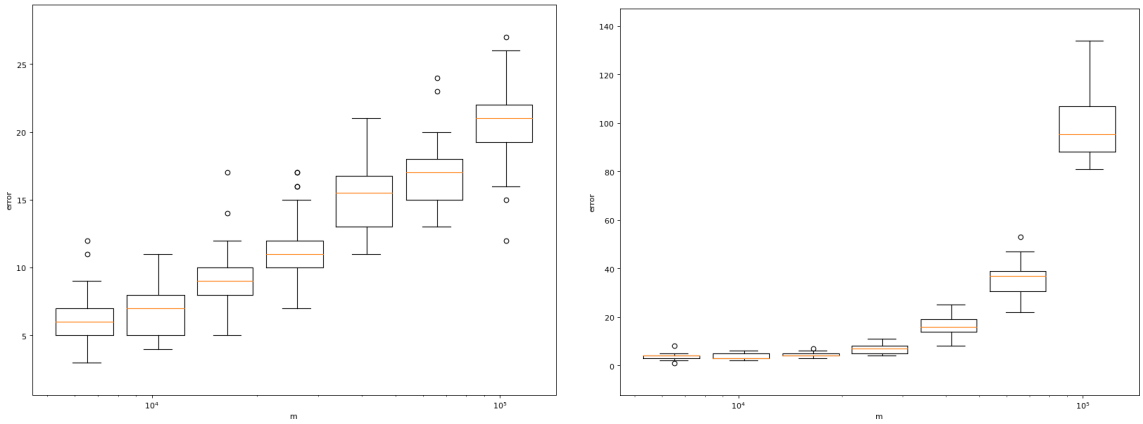
Uniformity

This time, it turns out in all cases, the error does increase with m. For f_{far}, this increase is much larger than for f_{close}. Since the variance of the error also appears to increase with m in all cases, it seems unlikely uniformity holds here.



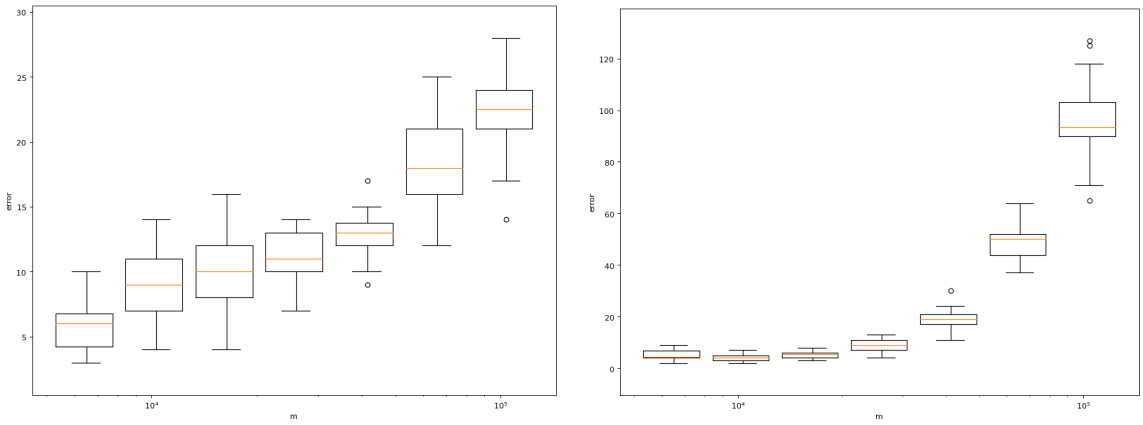
(a) close (b) far

Figure 3.25: The number of unsatisfied degrees without the use of an initial subset.



(a) close (b) far

Figure 3.26: The number of unsatisfied degrees with the use of an initial subset.

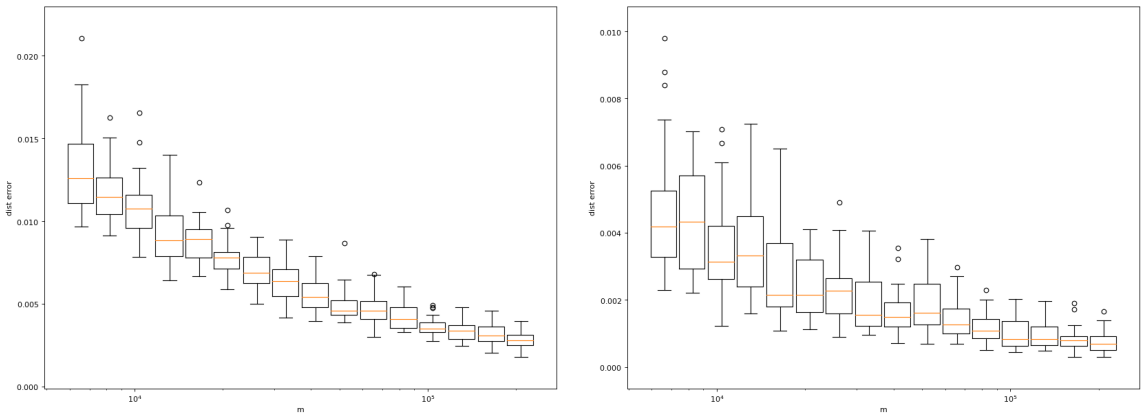


(a) close (b) far

Figure 3.27: The number of unsatisfied degrees with the use of an initial subset.

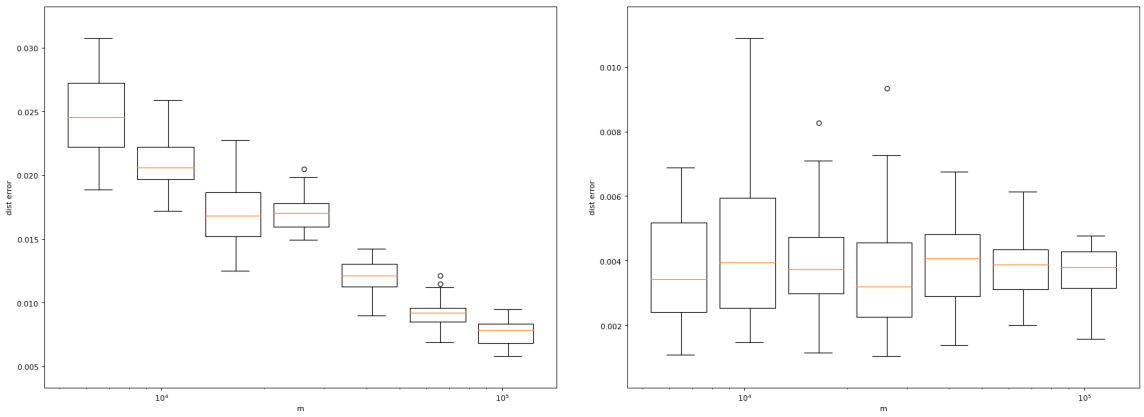
Convergence to the target distributions

Again, the empirical edge weight distribution appears to converge to the target distributions in all cases, except when the target distribution is f_{far} and we apply the method that uses an initial subset of the edges. This can be seen in figures 3.29b and 3.30b.



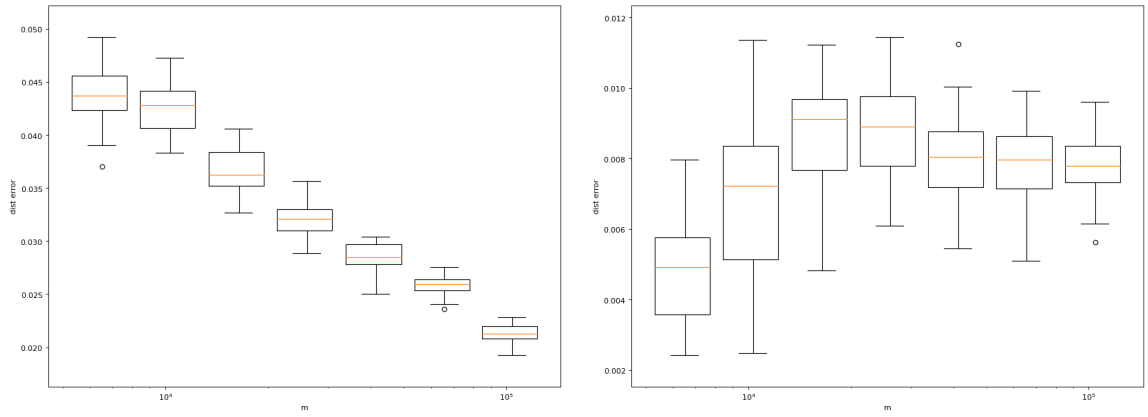
(a) close (b) far

Figure 3.28: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, without the use of an initial subset.



(a) close (b) far

Figure 3.29: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset.



(a) close (b) far

Figure 3.30: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset.

3.3 Summary

3.3.1 Without initial subset

Depending on the limit and the target distribution, the run time of the method without an initial subset varied, and appeared to be $O(m^{1\frac{1}{8}})$ as long as the degree error stayed near 0. The empirical distribution of sampled edge weights always appeared to converge to the target distribution.

3.3.2 With initial subset

The way the run time scaled was often unclear, most likely because it was computationally infeasible to resample a subset every time a graph was generated. On face value, the run time did not appear to deviate significantly from the desired $O(m^{1\frac{3}{8}})$, for both the continuous and binary approaches, although the binary approach was consistently around 5 times faster. The empirical edge weight distribution produced by this method always appeared to converge for the target distribution f_{close}, but never for f_{far}.

3.3.3 Uniformity

We found no evidence against uniformity in the first two limits, i.e. when n was increased individually, and when $d_{\max}$ was increased and n was kept at the bare minimum. We only found evidence against uniformity in the last limit, where we shrunk the range of distances in which we allowed edges. This also means it is unlikely uniformity will persist when an increase in vertices coincides with a broader distribution of locations, if the target distribution is quite limited in its range of allowed edge weights.

Chapter 4

Vertexwise sampling

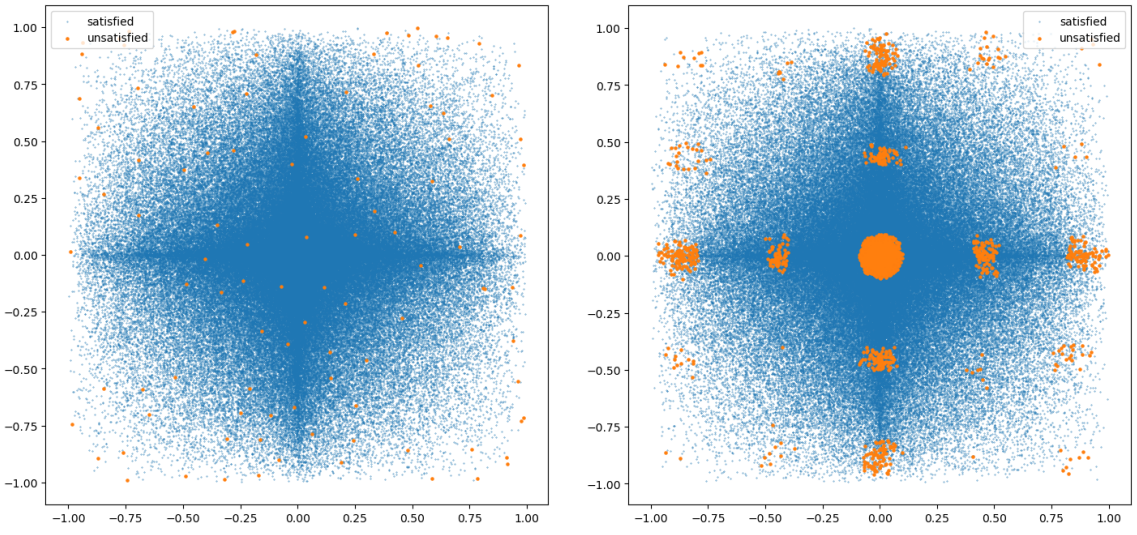
4.1 Introduction

Up until this point, the goal was to sample graphs for which the edge weight distribution followed some target distribution, and such that every sampled graph roughly had the same probability of being sampled. Now we try to do something slightly different. Namely, we still try to sample graphs where the edge weight distributions follows some target distribution, but now we also want control over how this distribution comes to be by also controlling the edge weight distributions of edges connected to individual vertices. More precisely, we want to have the edge weights of individual vertices to follow the target distribution, such that in total, all edges together also follow this distribution.

In the previous chapter, we have seen that if we shrink the allowed range of distances in the graph, the error increases. We have also seen that for f_{far}, this increase is much more drastic than for f_{close}. However, we have not yet looked at how this error manifests itself. In figure 4.1 we show on which vertices the error accumulates, for large graphs, for both f_{close} and f_{far}. The pattern caused by f_{close} can be explained as there simply not being enough edges locally within the permitted range. This is the desired pattern.

The pattern caused by f_{far} can be explained as the high density regions causing the surrounding area to be depleted first, creating large error in lower density regions. This is because the number of edges in a region increases quadratically with the number of vertices. Since the algorithm proposed by [Kryven, 2022] is based on sampling edges with the same probability, there is a bias towards regions where many vertices are located.

In this chapter, we propose an alternative algorithm that samples vertices with the same probability instead, while still converging to f in all the same ways we were used to earlier, and while maintaining the same style of approach and implementation. The goal is to remove this quadratic bias, and have the error be evenly spread across all vertices whenever possible. In practical applications, this is valuable, since the vertices are the entities of interest. If the error consistently accumulates on the same vertices every run of the algorithm, this can make the results be dominated by this artefact and therefore unusable. To contrast figure 4.1, figure 4.2 shows the identical experiment,

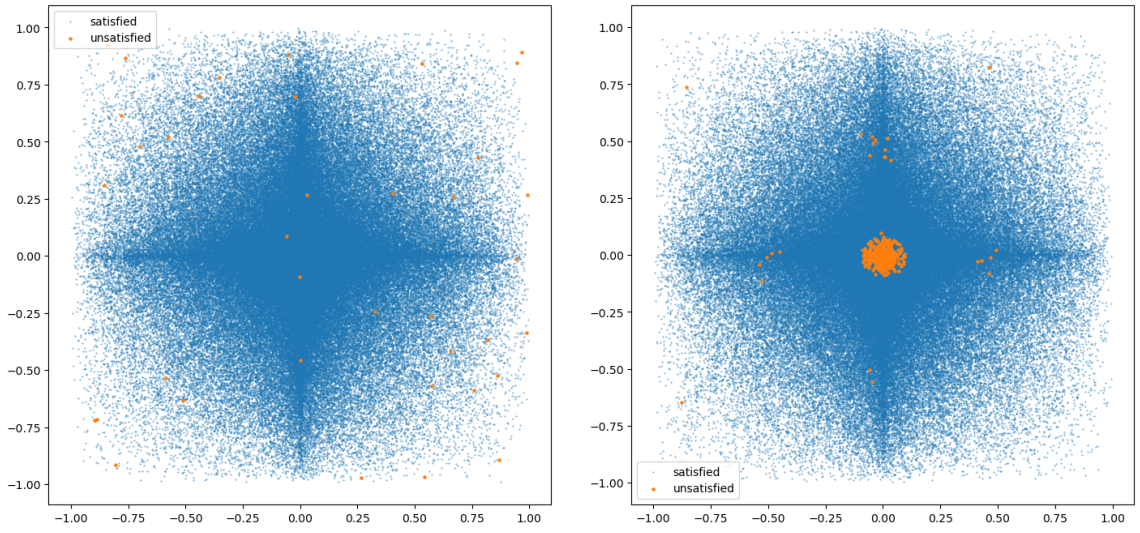


(a) close (b) far

Figure 4.1: The accumulation of error on vertices within the graph for the method proposed by [Kryven, 2022]. The blue dots show the sampled vertices, the orange dots show the vertices left with remaining degree after termination of a run of the algorithm.

but while using the alternative method instead.

The driving narrative will be about how a method was conceived that implements this idea. There will be no proofs, because we do not have proofs. The only validation will be the experimental results at the end in chapter 5. As a consequence, the argumentation will often be questionable, but we still hope to be able to convince the reader that this is a step in the right direction.



(a) close (b) far

Figure 4.2: The accumulation of error on vertices within the graph for the method proposed here. The blue dots show the sampled vertices, the orange dots show the vertices left with remaining degree after termination of a run of the algorithm.

4.2 Designing the new method

A binary perspective

As a starting point, we first turn to the binary case for answers, since it is the simplest case; we only have to deal with edges that are allowed or that are not allowed. We attempt to isolate the effect of $\frac{f(r_{ij})}{g(r_{ij})}$ by performing experiments with a degree sequence of all ones*. Without showing the specific experiments, we noticed that the error is greatly dependent on:

1. The variance in the of number of neighbours $\{N_k | k \in V\}$. A greater variance leads to more error.
2. The fraction of possible neighbours over the number of actual neighbours $\frac{n^2}{\sum_k N_k}$.

In other words, the number of neighbours of vertices has an effect on the error. We have seen some of this pattern already, since we concluded that when decreasing the range of allowed distance with n, the error increases with n. It is not unthinkable that the number of neighbours could have an effect either. After all, for [Bayati et al., 2010], the number of neighbours is always $N_k = n$ for any vertex k, making this a parameter unique to this problem involving edge weights. In conclusion, the number of neighbours is an important parameter that should be kept in mind.

*This turns the problem into finding maximum matchings if the goal is to find an output with an error of 0.

An algorithmic perspective

In section 2.3.3 we looked at what sampling according to $\frac{f(r_{ij})}{g(r_{ij})}$ looks like from an algorithmic standpoint. Let $g'(b)$ be the total number of edges within bin b. As a recap, if we were to choose an edge uniformly at random from the bin, this edge would be sampled from the bin with probability proportional to $\frac{1}{g'(b)}$. Since we first sampled a bin with probability proportional to the $f(b)$[†], we have sampled an edge with probability proportional to $\frac{f(b)}{g'(b)}$. In other words, this is an implementation of sampling with probability proportional to $\frac{f(r_{ij})}{g(r_{ij})}$.

Assume all vertices have at least 1 edge in every bin. Then some vertices may have many edges in this sampled bin b, and some have fewer, but all vertices can be sampled. Let $g'_i(b)$ be the total number of edges within bin b containing vertex i. Then the bias that this binned implementation of $\frac{f(r_{ij})}{g(r_{ij})}$ introduces towards a single vertex i is $\frac{g'_i(b)}{g'(b)}$, where b is the bin sampled according to f.

4.2.1 Formulating a solution

Because bin b is simply a proxy for the edge weight r_{ij}, redefine $g'_i(r_{ij}) = g_i(r_{ij})N_i$, where g_i is the distribution of edge weights of edges containing vertex i, for which $f(r_{ij}) \neq 0$. Similarly, redefine $g'(r_{ij}) = g(r_{ij})\frac{1}{2}\sum_i N_i$. The observed bias towards a single vertex i is then $\frac{g'_i(r_{ij})}{g'(r_{ij})}$, which makes the bias towards both vertices i and j for an edge (i,j)

$$\frac{g'_i(r_{ij})g'_j(r_{ij})}{g'(r_{ij})^2}.$$

This bias can be cancelled out by sampling edges with a probability proportional to

$$\frac{f(r_{ij})}{g(r_{ij})}\frac{g'(r_{ij})^2}{g'_i(r_{ij})g'_j(r_{ij})} = \frac{f(r_{ij})g(r_{ij})\left(\frac{1}{2}\sum_i N_i\right)^2}{N_i g_i(r_{ij})N_j g_j(r_{ij})}$$

instead, rather than $\frac{f(r_{ij})}{g(r_{ij})}$. Since $\frac{1}{2}\sum_i N_i$ is just a constant per iteration, the final probability that edges should be sampled proportional to is

$$\frac{f(r_{ij})g(r_{ij})}{N_i g_i(r_{ij})N_j g_j(r_{ij})}\hat{d}_i\hat{d}_j\left(1 - \frac{d_i d_j}{4m}\right). \tag{4.1}$$

Interpreting this solution

The original way of reasoning that lead to this solution is a much vaguer one, but also insightful. Notice that when sampling proportional to $\frac{f(r_{ij})}{g(r_{ij})}$, we are simply compensating for sampling from g. Every vertex i also has a distribution g_i of edge weights for edges containing vertex i. If we can sample according to $f^{\frac{1}{2}}$ per vertex, we automatically also sample according to f per two vertices, i.e. per edge. Because we see that some vertices are favoured over others, this should be done. In other words, the distribution we should be sampling proportional to should look something like

$$\frac{f(r_{ij})^{\frac{1}{2}}}{g_i(r_{ij})}\frac{f(r_{ij})^{\frac{1}{2}}}{g_j(r_{ij})} = \frac{f(r_{ij})}{g_i(r_{ij})g_j(r_{ij})}.$$

[†]Technically, b is an index of f, and not an edge weight, but since indexes are proxies for edge weights, we abuse notation like this.

However, when $g = g_i$ for all vertices i, the method should be identical to sampling proportional to $\frac{f(r_{ij})}{g(r_{ij})}$, hence the solution is most likely more like

$$\frac{f(r_{ij})g(r_{ij})}{g_i(r_{ij})g_j(r_{ij})}.$$

But now, we still have the problem that if vertex i has many more neighbours than j, we still sample i more, hence the solution should be more like

$$\frac{f(r_{ij})g(r_{ij})}{N_i g_i(r_{ij})N_j g_j(r_{ij})}.$$

This is insightful because it makes clearer that we are sampling proportional to $f(r_{ij})^{\frac{1}{2}}$ per vertex, which then turns out to sample proportional to $f(r_{ij})$ per edge. This insight is useful, because the assumption that every vertex is present in every bin is false, and this has consequences for the resolution of the histograms used to represent g_i'. We solve this problem in section 4.3 using this insight. Still, for in the binary case, there are no such problems.

4.2.2 The binary case

The implementation of the binary case is very similar to the one discussed in section 2.2. This is because $g_i(r_{ij}) = 1$ for all vertices i, since g_i is the distribution over edge weights for which $f(r_{ij}) \neq 0$. In other words, from the edges for which $f(r_{ij}) \neq 0$, we only to have to sample edges with probability proportional to

$$\frac{1}{N_i N_j}\hat{d}_i\hat{d}_j\left(1 - \frac{d_i d_j}{4m}\right).$$

Before the step where we compensate with $\frac{N_i}{\max_k N_k}$, we already sampled with probability proportional to

$$\frac{1}{N_i}\hat{d}_i\hat{d}_j\left(1 - \frac{d_i d_j}{4m}\right).$$

This means that all we need to do, is to replace this die roll with one that succeeds with probability $\frac{\min_k N_k}{N_j}$. The minimum is obtained using the same AVL tree from which we obtained the maximum, and the implementation is no longer split into two phases.

4.3 The continuous case

We now run into the problem that if we split the samplable edges over the n different vertices, and bin these edges in the same way we bin f and g, either the bin size will be impractically large, or the not every vertex will be present in an edge in every bin. These problems are not mutually exclusive, either. As the algorithm progresses, more and more samplable edges are removed, leading to not every vertex being present in every bin anyway. Because of this, we aim to use small bins, and to deal with the zeros by somehow compensating for them. We will not be able to find a solution to this problem entirely, but we will solve it approximately.

4.3.1 Practical problems

Recall that we are attempting to sample according to $f^{\frac{1}{2}}$ per vertex. This means the problem is actually that we are not able to sample edges according to $f^{\frac{1}{2}}$ at all, because we lack edges. Instead, we are only sampling according to $f^{\frac{1}{2}}$ for the bins where we have edges, and according to 0 elsewhere. This means we are undersampling bins if many vertices lack edges in these bins.

Because of this, we change the target distribution that the edges of individual vertices are sampling according to. We do so such that in total, the distribution of all edge weights is still f. Think of the algorithm as if we were to pick a vertex at random, because they should all have the same probability of being chosen, and then for this vertex, select an edge weight roughly according to $f^{\frac{1}{2}}$, deviating in its own way due to missing edges.

Then the probability that a bin is chosen is proportional to the number of vertices for which there is an edge in this bin. If $f^{\frac{1}{2}}$ assigns the same probability to 2 bins, but one occurs among twice as many vertices, this bin will be chosen twice as many times.

Practical solutions

Now assume that this is somewhat representative of what is happening when sampling in our algorithm. Let $h(b)$ count the number of vertices for which there are edges in bin b. Then the claim is there is a bias of $h^2(b)$, squared because we are sampling 2 vertices. Or in other words, proportional to how many vertices are capable of sampling that bin, squared.

Therefore, instead of sampling according to $f^{\frac{1}{2}}$, we sample according to $\frac{f(b)^{\frac{1}{2}}}{h(b)}$. In other words, we scale the bins such that bins with few vertices that contain edges in this bin, are more likely to be sampled.

After scaling with h, the vertices are no longer sampled with the same probability. They are now sampled with probability proportional to $q_i = \sum_b \frac{f(b)^{\frac{1}{2}}}{h(b)} \mathbb{1}_{\text{vertex i has an edge in bin } b}$. This makes our best attempt sampling with probability proportional to

$$\frac{f(r_{ij})}{h(r_{ij})^2 q_i q_j} \frac{g(r_{ij})}{N_i g_i(r_{ij}) N_j g_j(r_{ij})} \hat{d}_i \hat{d}_j \left(1 - \frac{d_i d_j}{4m}\right). \tag{4.2}$$

Imperfections of the solution

The biggest problem with the above equation is that scaling with q_i does not actually fully correct for the bias towards vertices introduced by not having edges in all bins. Imagine having only 2 bins while comparing 2 vertices. Imagine that one vertex has edges in both bins, but that the other vertex only has edges in one bin. The only thing we can then do is scale the probabilities in the one bin where they both occur to the best of our ability, but this still will not make both vertices equally likely to be sampled.

Another problem is that q_i changes for many vertices every time h changes. As such, q_i is not maintainable in a reasonable time as it can take $O(n^3)$ operations. Because of this, we will propose an approximation to q_i that only adds constant time overhead, compared to the cost of the rest of the algorithm.

The third problem is that given this approximation of q_i, it would cost another multiplicative cost of $O(\log(m))$ to maintain an AVL tree of q, and the minimum and maximum would not be very accurate. This is again resolved using an approximation.

4.3.2 Implementation

The implementation of equation 4.2 is structured in the same way as that of the one described in section 2.3. We still sample a vertex i from an array according to $\hat{d}_i$, then sample from its neighbours to obtain j, and then roll a die to compensate for the remaining biases. The difference is that this die roll now looks like this:

$$\frac{N_i}{\max_k N_k} \frac{\hat{d}_j}{\max_{k'} \hat{d}_{k'}} \left(1 - \frac{d_i d_j}{4m}\right) \frac{g'(b_{r_{ij}})}{\max_b g'(b)} \frac{1}{g'_i(b_{r_{ij}}) g'_j(b_{r_{ij}})} \frac{f(b_{r_{ij}}) s^2 u^2}{h(b_{r_{ij}})^2 \tilde{q}_i \tilde{q}_j}$$

We will unpack this slowly. The first term, $\frac{N_i}{\max_k N_k}$, compensates for sampling from the neighbours of i. This might seem counterintuitive because we need N_i in the denominator. However, for this die roll to work, we also need g'_i in the denominator, which also contains N_i implicitly.

Next, we encounter the familiar $\frac{\hat{d}_j}{\max_{k'} \hat{d}_{k'}} \left(1 - \frac{d_i d_j}{4m}\right)$, which we have also seen in most other implementations. After that comes $\frac{g'(b_{r_{ij}})}{\max_b g'(b)}$. Here, the bin $b_{r_{ij}}$ corresponding to the edge weight r_{ij}, is used as an input to g' to emphasize that this is how we approximate g. Recall that $g'(b)$ counts the number of edges that have an edge weight that gets mapped to bin b. Because g and g' only differ by a constant per iteration, it does not matter which we use for the die roll. Since g' is slightly easier to maintain, g' is used.

Second to last, we find $\frac{1}{g'_i(b_{r_{ij}}) g'_j(b_{r_{ij}})}$. Because $g'_i(b)$ and $g'_j(b)$ count the number of edges in bin b, we have that $g'_i(b_{r_{ij}}), g'_j(b_{r_{ij}}) \geq 1$ for every sampled edge. Because of this, $0 < \frac{1}{g'_i(b_{r_{ij}}) g'_j(b_{r_{ij}})} \leq 1$.

Bin compensation

The last term is $\frac{f(b_{r_{ij}}) s^2 u^2}{h(b_{r_{ij}})^2 \tilde{q}_i \tilde{q}_j}$. We have seen f and h before. $f(b_{r_{ij}})$ and $h(b_{r_{ij}})$ are the height of the target distribution, and number of vertices that have an edge in bin $b_{r_{ij}}$, respectively. $\tilde{q}_i, \tilde{q}_j, s$ and u are new. The goal of $\tilde{q}_i, \tilde{q}_j$ and s is that, together, these approximate q_i and q_j. The goal of u is that the entire term remains below 1.

We define s as the number of vertices that still have samplable edges. We then also define $\tilde{q}_i = s q_i = \sum_b f(b)^{\frac{1}{2}} \frac{s}{h(b)} \mathbb{1}_{\text{vertex i has an edge in bin } b}$. The added benefit of this is that $\frac{s}{h(b)}$ stays closer to 1 compared to just $\frac{1}{h(b)}$, which allows us to poorly maintain $\tilde{q}$, without having terms of the sums becoming destructively dominant. We use s over $\max_b h(b)$ since s is much cheaper to maintain.

We maintain $\tilde{q}$ by updating $\tilde{q}_i$ and $\tilde{q}_j$ every time an edge (i, j) is processed in a major way, and we then only update the term involving bin $b_{r_{ij}}$. We do not loop over all bins, and we only partially correct $\tilde{q}_i$ and $\tilde{q}_j$. More specifically, an update occurs after sampling a vertex j from the neighbours of i, and every time a neighbour gets deleted and is therefore no longer samplable. Since the algorithm finishes in much less time than the $O(n^3)$ operations that are required to fully maintain q, the kind of robustness $\tilde{q}$ provides is a must.

To illustrate this, consider what would happen if we did not use $\tilde{q}$. Then, the terms of q would increase in amplitude as the algorithm progressed, meaning that terms in the sum that have not been updated in a while would be drowned out and these sums would quickly be very inaccurate. Also note that if $h(b) = s$ and both decrease by 1, the terms of the sums in $\tilde{q}$ do not need to be updated, but in q they do.

Lastly, we use u in an attempt to keep this term below 1. This sounds much worse than it turns out to be. Notice that we can ensure that the term stays below 1 by scaling by $\max_b \frac{f(b)}{h(b)^2}$ and by $\max_k \left(\frac{s}{\tilde{q}_k}\right)^2$. However, since $\tilde{q}$ is updated very frequently, maintaining an AVL tree adds a lot of overhead. Instead, we can also bound the entries of $\tilde{q}$ itself using the worse (lower) bound

$$\left(\min_b f(b)^{\frac{1}{2}} \frac{s}{h(b)}\right)\left(\min_k \sum_b \mathbb{1}_{\text{vertex } k \text{ has an edge in bin } b}\right),$$

which is the minimum a term in the sum can be, times the minimum number of non-zero entries in the sum. However, this is such a gross underestimation, that one is left wondering if we can do better.

Notice that if we went the route of scaling by upper bounds, we would be multiplying by

$$s^2 \frac{\min_b \frac{f(b)}{h(b)^2}}{\max_b \frac{f(b)}{h(b)^2}} \left(\min_k \sum_b \mathbb{1}_{\text{vertex } k \text{ has an edge in bin } b}\right)^2,$$

and we would define s^2u^2 as such.

However, what we can also do is define u^2 to be just $\frac{1}{c}\left(\min_k \sum_b \mathbb{1}_{\text{vertex } k \text{ has an edge in bin } b}\right)^2$, where c is some integer, initially set to 2, and roll all dice together as one, relying on the accumulation of all overestimations. If the accepting probability never reaches above 1, we conclude that, apparently, this works. If it does not, we increment c and rerun the entire algorithm with a higher constant. However, we have yet to see this fail, with the required constant peaking at 1.5.

4.3.3 Pipelines; polar coordinate subsetting

We now have an implementation that uses an initial subset, but we still need to generate this subset. In section 2.4 we sampled from edges according to $\frac{f(r_{ij})}{g(r_{ij})}$. However, this time, we also require sufficiently many edges per vertex to be able to sample according to the target distribution per vertex. Luckily, it is only a small change to make this happen.

Before we sampled edges by sampling a distance from f and then iterating by repeatedly sampling a random vertex, rotation and grid depth until an edge is found. Now we instead loop over all vertices for a fixed number of edges for vertex, and only then sample edges by sampling a distance from f, after which we iterate by repeatedly sampling rotations and grid depths until an edge is found.

This ensures we have enough edges per vertex to sample according to the desired distribution when we run the algorithm with this subset. However, as we have seen in section 4.3.2, the desired distribution of equation 4.2 does not actually implement the desired idea because if an edge is

simply not present in a bin, no amount of scaling can fix this imbalance. Because of this, it is also important to not just sample edges according to the target distribution, but also according to the uniform distribution to make sure all bins are somewhat evenly filled, which avoids bin compensation as much as possible.

Lastly, we should take into consideration how to choose an adequate bin size for discretizing f, g and g_k for $k \in V_n$. Here, we advocate using the same number of bins as the average number of edges per vertex. This way we expect there to be one edge per bin, per vertex, at the start of the algorithm. This means that if we wish that the output distribution converges to the target distribution, we have to increase the number of edges per vertex over the iterations. If we do not, the resolution of f, g and g_k for $k \in V_n$, remains the same.

Binary method

Again, we can also implement this method using the binary method. This approach is questionable because we know that per vertex, on average half of the edges are sampled according to f, but we have to assume that this also holds for the other half. After all, we loop over the vertices and sampled a set number of edges according to f for each of them, but these edges also get added to the other vertex in the edge and this will still create some imbalance.

If the imbalance is limited to the quantity of edges between vertices, then this is fine because this is corrected for in the binary method. If it also affects the distribution of the edges per vertex, we might not be able to tell from the convergence to f, since the overall edge weight distribution is unaffected.

The subset is also constructed differently from when using the continuous method. There, we want edges in most bins for every vertex, hence we also sample edges according to the uniform distribution. If we use the binary method, we do not sample these additional edges, as we will not be keeping track of these distributions to begin with. We want the distribution of the edge weights in the subset to be the same as those in the output graph.

Chapter 5

Empirical behaviour of vertexwise sampling

We now investigate the empirical behaviour of this vertexwise approach in the same way we have done before in chapter 3. However, this time, we only have a subset based implementation.

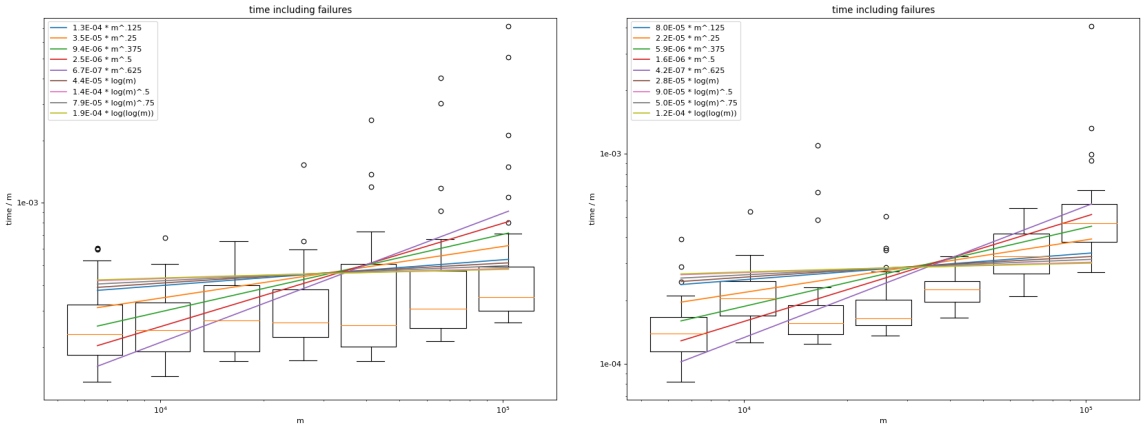
5.1 Results

5.1.1 A denser population with constant $d_{\max}$

Again, the first limit we are interested in is when we fix $d_{\max}$ to be a constant, namely 8, and uniformly sample degrees from $[1, d_{\max}]$ for each vertex, while increasing the number of vertices n that are sampled from the fixed distribution of locations.

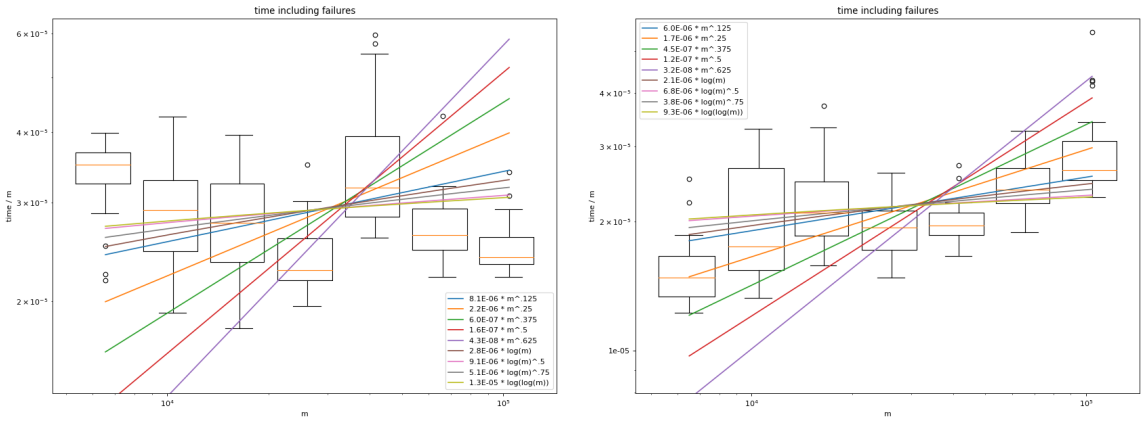
Run time

When using the continuous method exclusively, the results are inconclusive due to the high variance of the run time, as can be seen in figure 5.1. When using the binary method, the run time again does not significantly deviate from $O(m^{\frac{1}{3}})$ per edge, as can be seen in figure 5.2. The binary approach does appear to be roughly a factor 10 faster.



(a) close (b) far

Figure 5.1: Average run time per edge with the use of an initial subset.

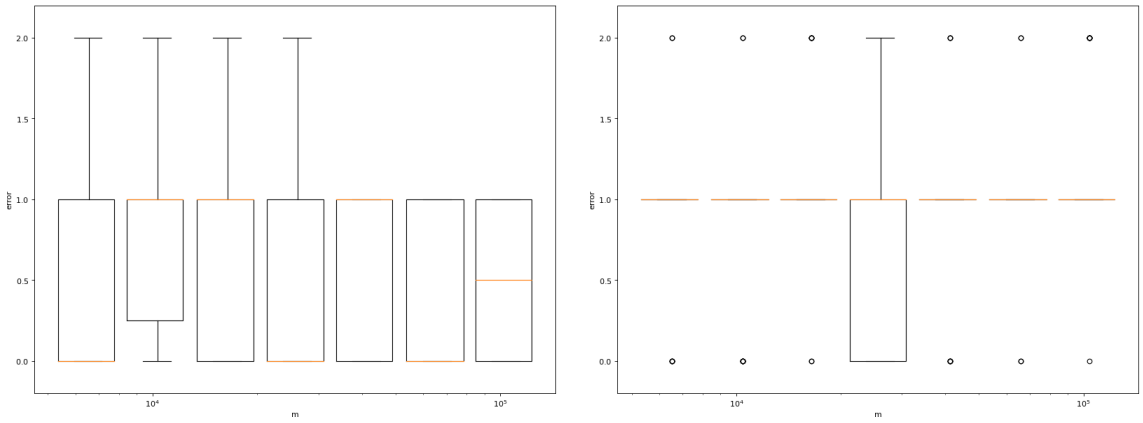


(a) close (b) far

Figure 5.2: Average run time per edge with the use of an initial subset and the binary method.

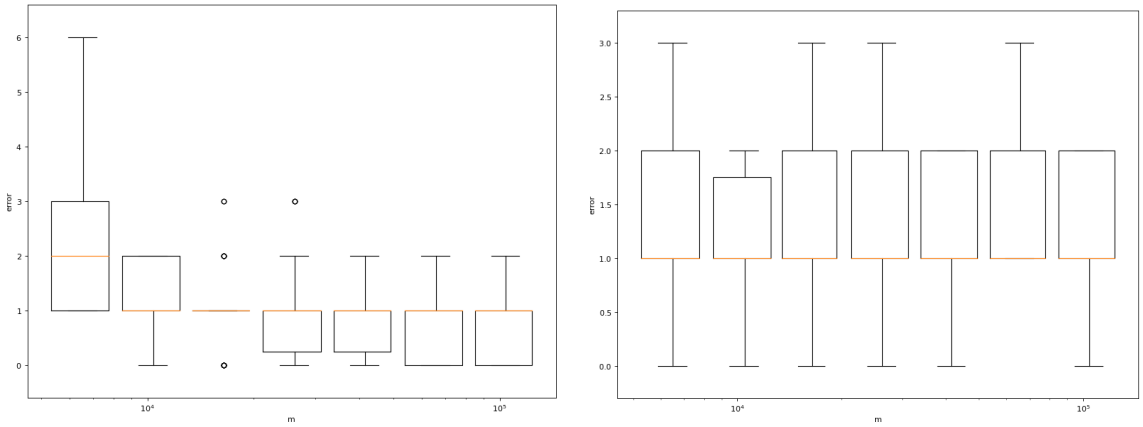
Uniformity

It turns out in all cases, the error does not increase with m, but instead stays $O(1)$, as can be seen in figures 5.3 and 5.4. In other words, no evidence is found against some form of uniformity.



(a) close (b) far

Figure 5.3: The number of unsatisfied degrees with the use of an initial subset.

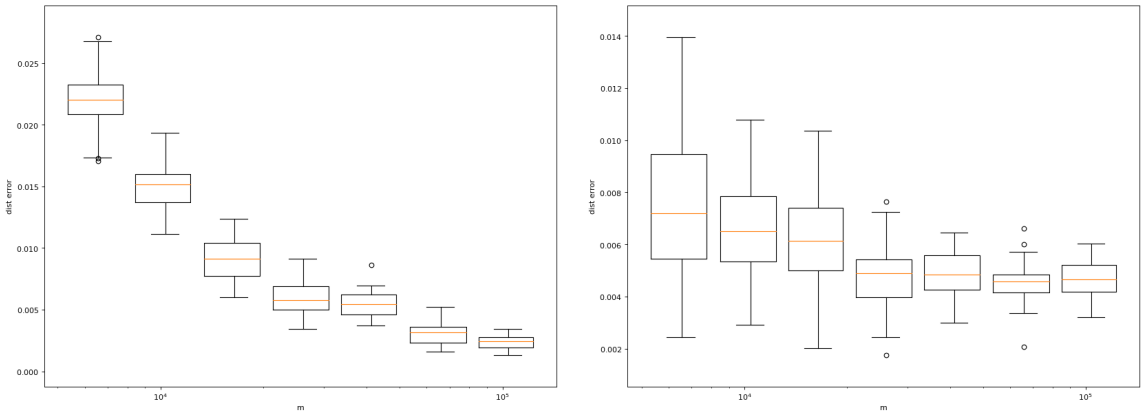


(a) close (b) far

Figure 5.4: The number of unsatisfied degrees with the use of an initial subset and the binary method.

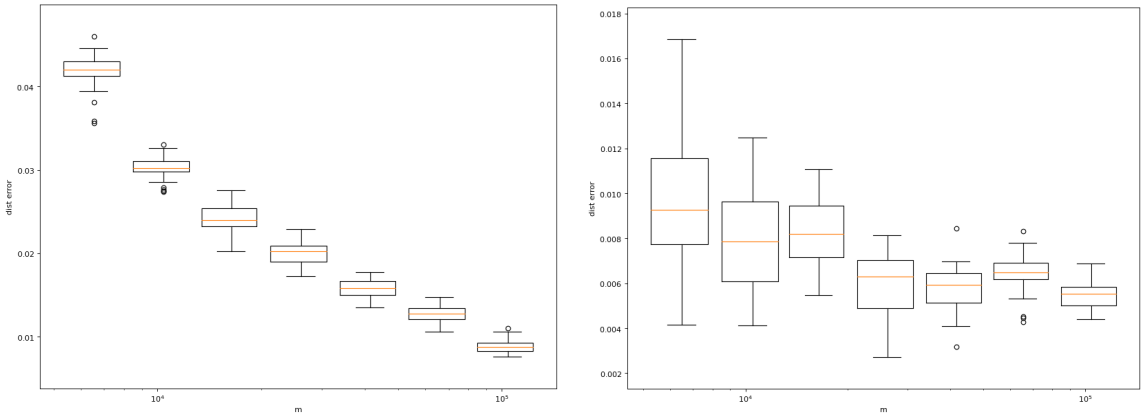
Convergence to the target distributions

Like before, the empirical edge weight distribution appears to converge to the target distributions when using f_{close}. Although the error does decrease as we increase m when using f_{far}, it appears the error is not headed to 0, and stagnates like before. This can be seen in figures 5.5b and 5.6b.



(a) close (b) far

Figure 5.5: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset.



(a) close (b) far

Figure 5.6: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset and the binary method.

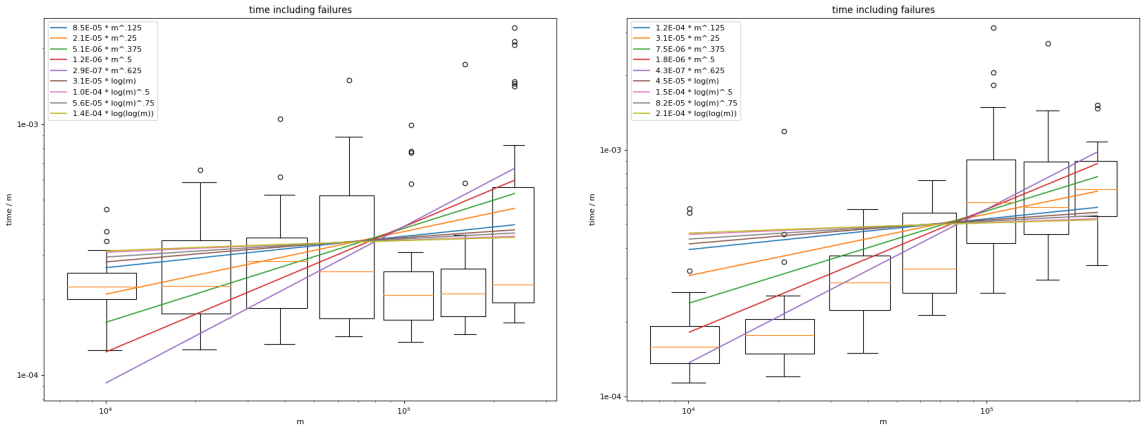
5.1.2 A denser population with increasing $d_{\max}$

The second limit we are interested in is, again, when we let $d_{\max}$ and n increase together. The experimental setup is the same as in section 3.2.2.

Run time

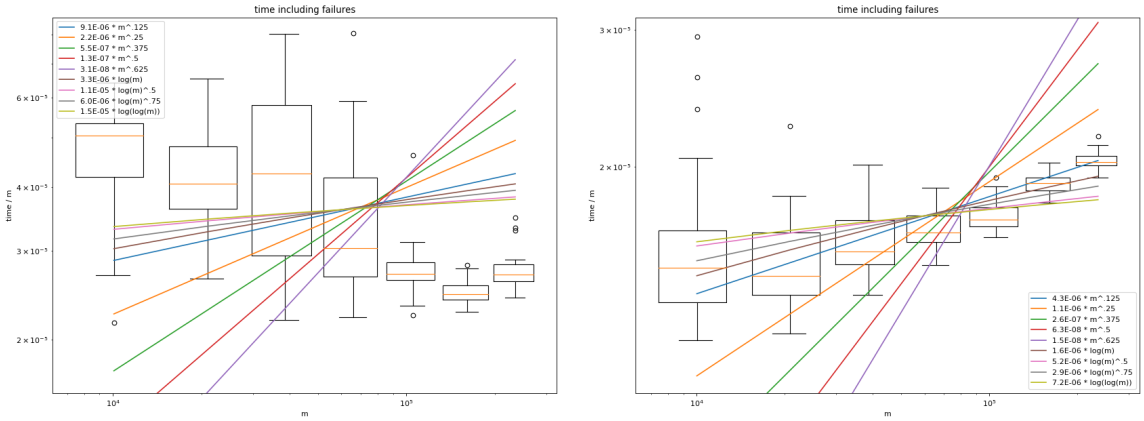
When exclusively using the continuous method, the results are again inconclusive due to the high variance of the run time, as can be seen in figure 5.7. When using the binary method, the run time

again does not significantly deviate from $O(m^{\frac{1}{3}})$ per edge, as can be seen in figure 5.8. The binary approach still appears to be roughly a factor 10 faster.



(a) close (b) far

Figure 5.7: Average run time per edge with the use of an initial subset.

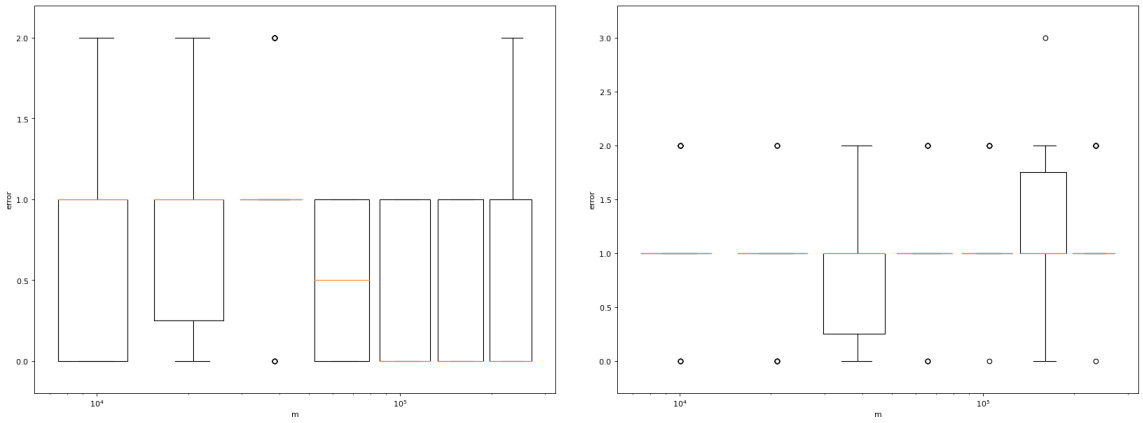


(a) close (b) far

Figure 5.8: Average run time per edge with the use of an initial subset and the binary method.

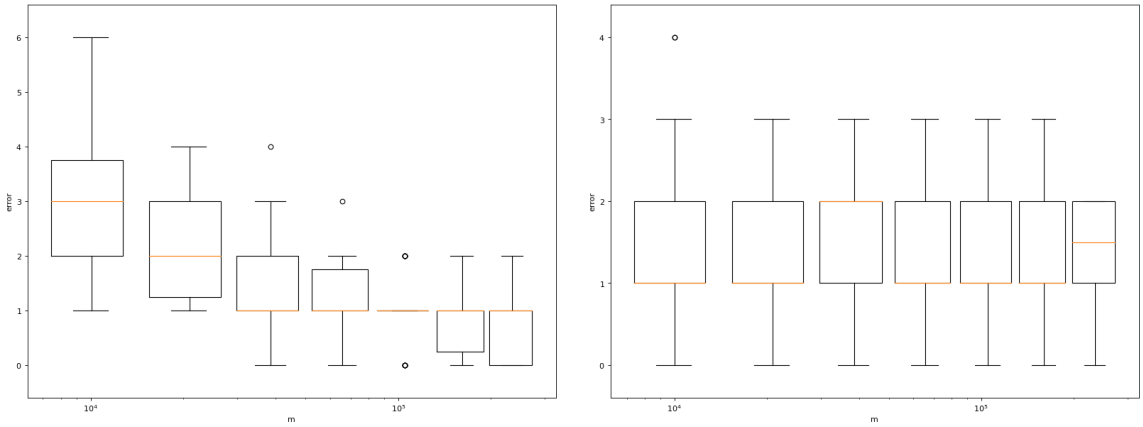
Uniformity

Again, it turns out in all cases, the error does not increase with m, but instead stays $O(1)$, as can be seen in figures 5.9 and 5.10. In other words, no evidence is found against some form of uniformity.



(a) close (b) far

Figure 5.9: The number of unsatisfied degrees with the use of an initial subset.

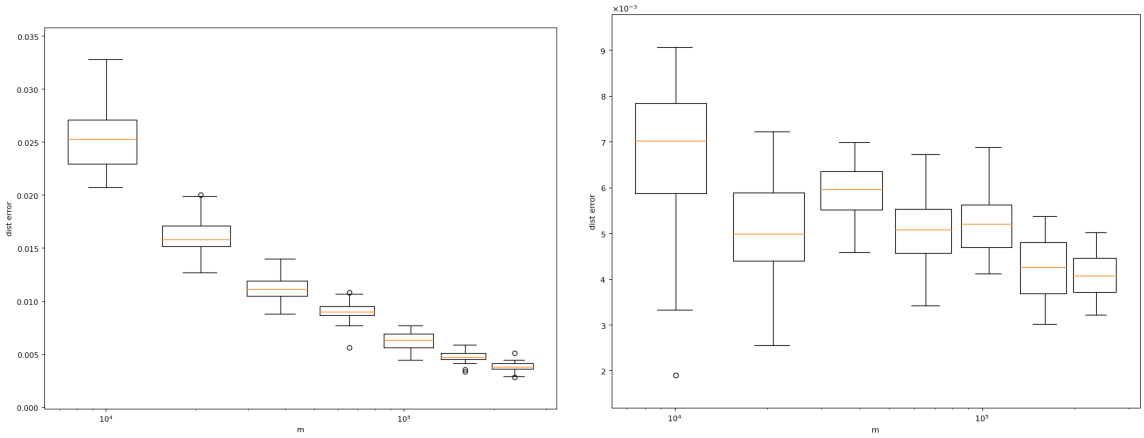


(a) close (b) far

Figure 5.10: The number of unsatisfied degrees with the use of an initial subset and the binary method.

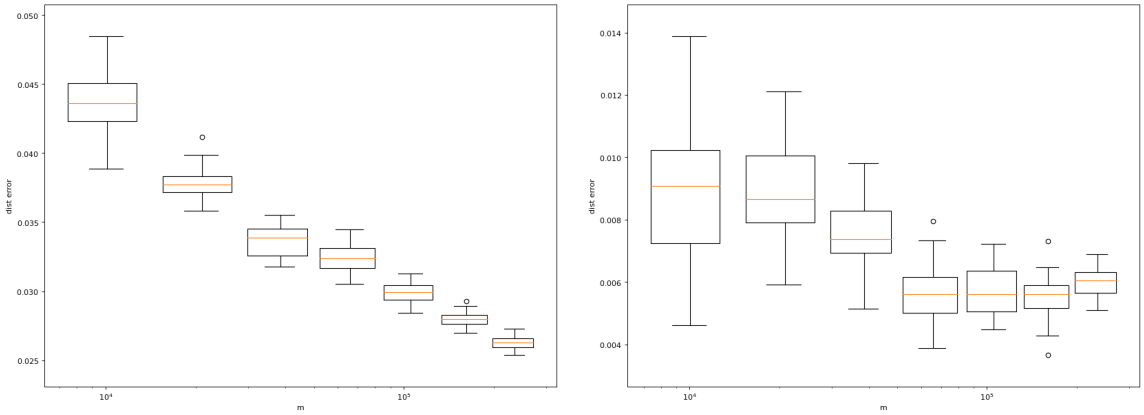
Convergence to the target distributions

Like before, the empirical edge weight distribution still appears to converge to the target distributions when using f_{close}, but when using f_{far}, it appears the error is not headed to 0. This can be seen in figures 5.11b and 5.12b.



(a) close (b) far

Figure 5.11: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset.



(a) close (b) far

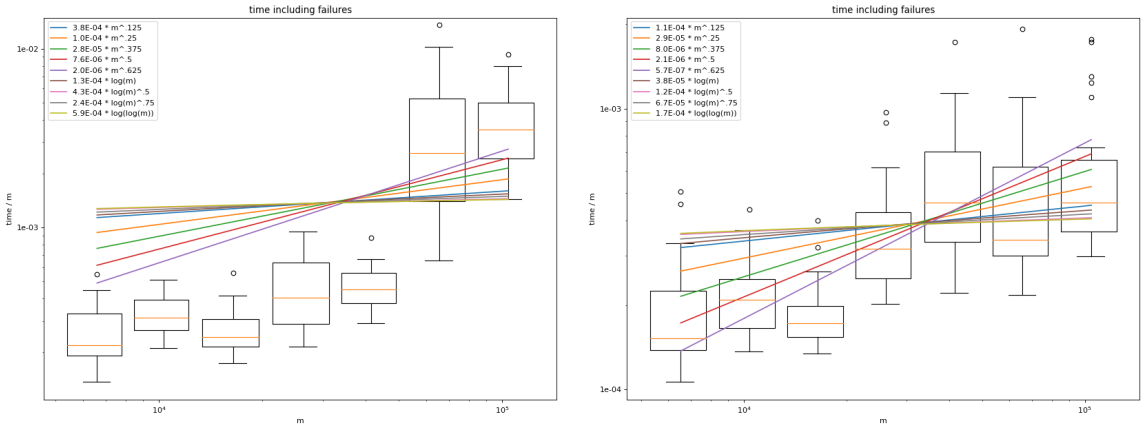
Figure 5.12: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset and the binary method.

5.1.3 A changing distribution with constant $d_{\max}$

Finally, we are again interested in what happens if the number of available edges does not grow as quickly as the number of edges. The experimental setup is the same as in section 3.2.3.

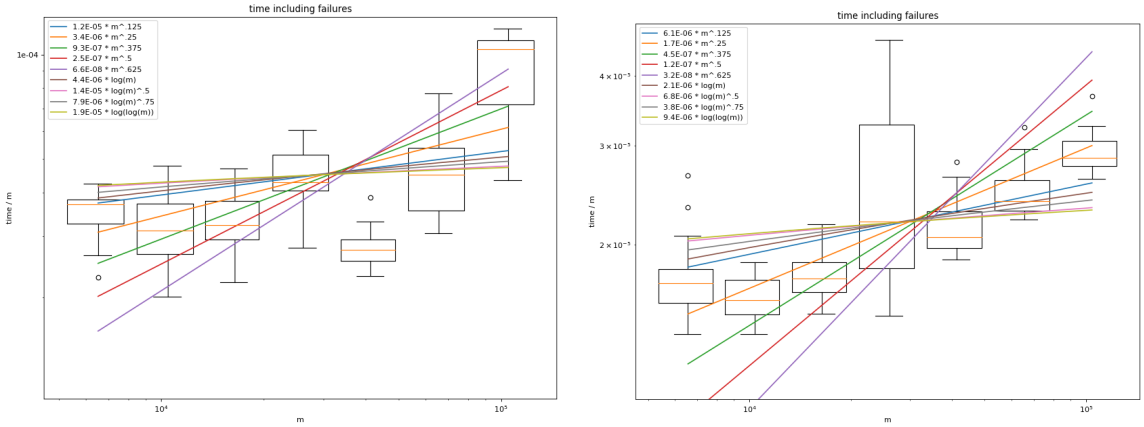
Run time

Here, the results are inconclusive for all runs, as the variance is too large.



(a) close (b) far

Figure 5.13: Average run time per edge with the use of an initial subset.

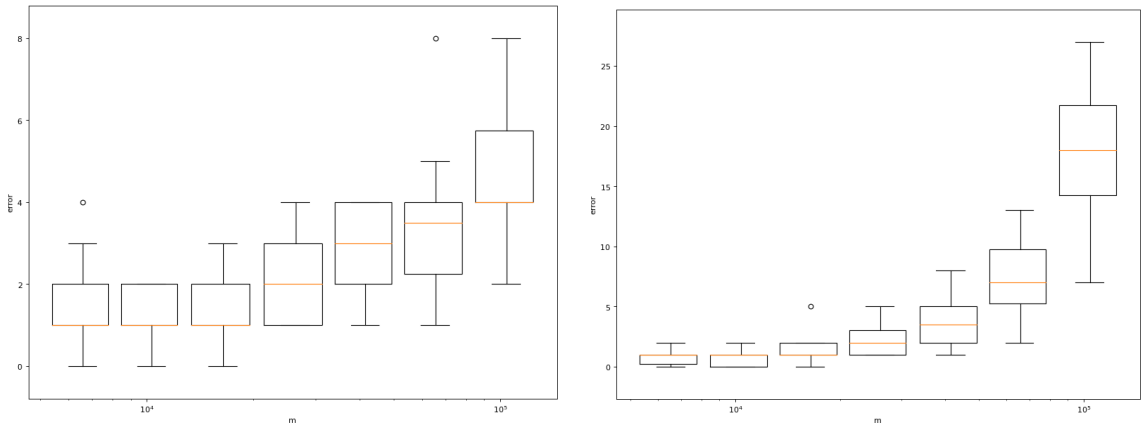


(a) close (b) far

Figure 5.14: Average run time per edge with the use of an initial subset and the binary method.

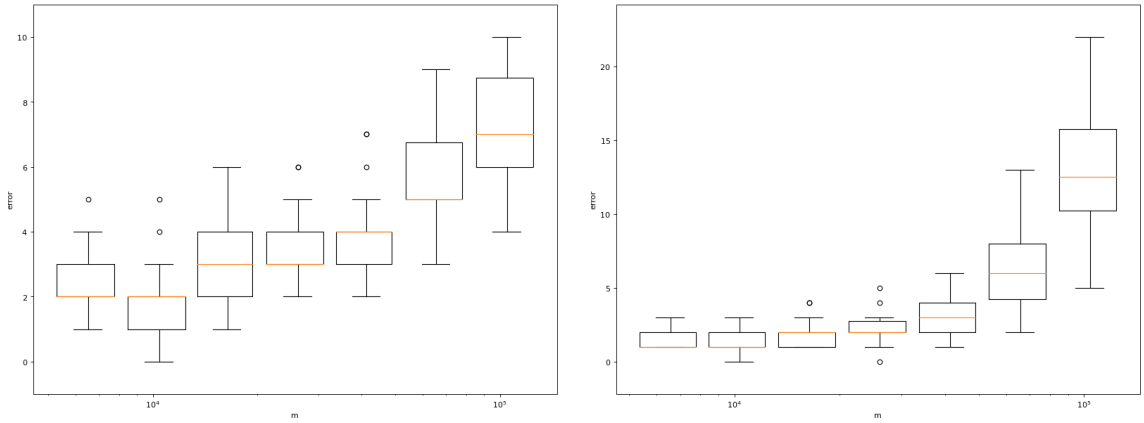
Uniformity

We again find the same pattern as in section 3.2.3, where in all cases, the error does increase with m, but more so for f_{far} than for f_{close}. Although the error does appear to be roughly a factor 4 lower than in section 3.2.3, the variance of the error still appears to increase with m in all cases, hence it seems unlikely uniformity holds here.



(a) close (b) far

Figure 5.15: The number of unsatisfied degrees with the use of an initial subset.

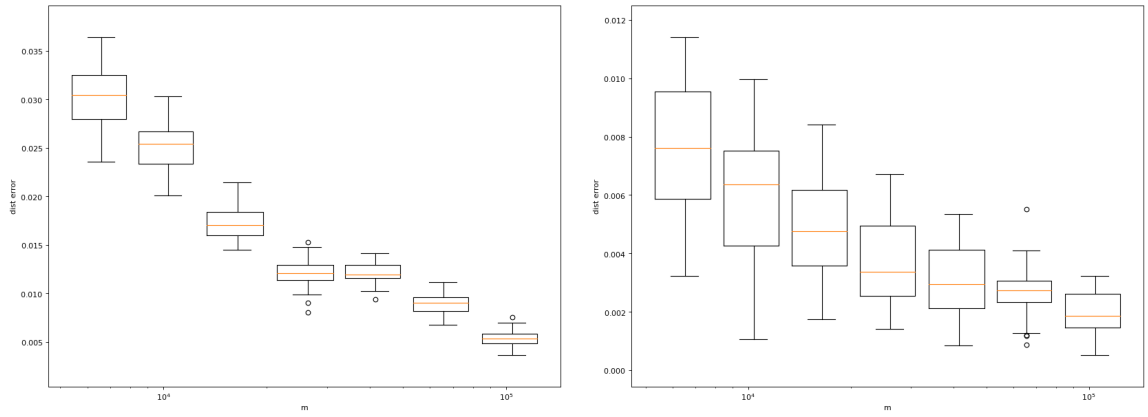


(a) close (b) far

Figure 5.16: The number of unsatisfied degrees with the use of an initial subset and the binary method.

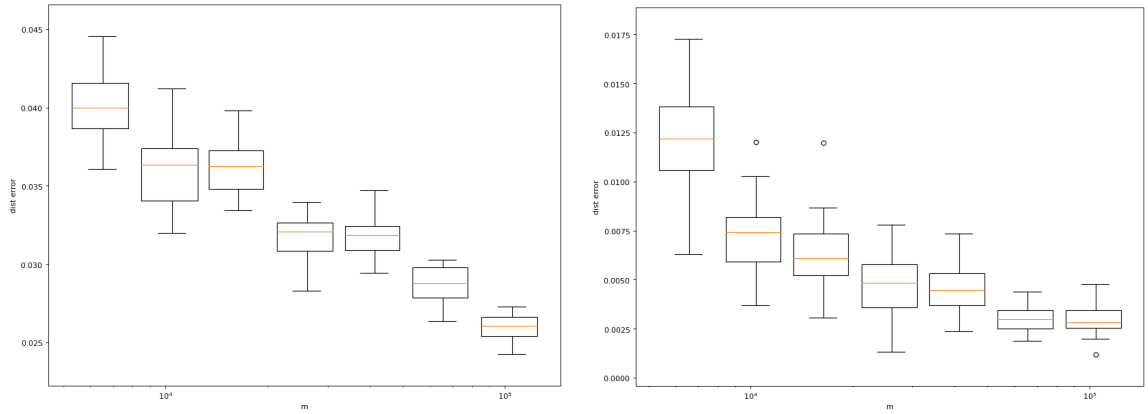
Convergence to the target distributions

Remarkably, this time we do appear to obtain convergens of the empirical edge weight distribution to the target distribution in all cases, even though we did not in section 3.2.3. This can be seen in figures 5.17 and 5.18.



(a) close (b) far

Figure 5.17: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset.



(a) close (b) far

Figure 5.18: The unsigned area between the cumulative target distribution and the empirical edge weight distribution, with the use of an initial subset and the binary method.

5.2 Summary

5.2.1 Run time

The way the run time scaled was unclear when exclusively using the continuous method. When using the binary method, the run time was consistently roughly 10 times faster compared to using the continuous method exclusively. Then, the run time did not notably differ from $O(m^{\frac{1}{3}})$ per edge, although sometimes it did appear to be $O(1)$ for f_{close}, of which we know this cannot hold true for very large n.

5.2.2 Uniformity

Identical dynamics are found as in chapter 3, although here, the error does appear to be some constant factor lower, with this constant ranging from 3 to 10 depending on the experiment. Again, we found no evidence against uniformity in the first two limits, i.e. when n was increased individually, and when $d_{\max}$ was increased and n was kept at the bare minimum. We only found evidence against uniformity in the last limit, where we shrunk the range of distances in which we allowed edges. Because of this, uniformity is unlikely to persist when an increase in vertices coincides with a broader distribution of locations, if the target distribution is quite limited in its range of allowed edge weights.

5.2.3 Convergence to the target distributions

We again found that for f_{close} the empirical edge weight distribution converged to the target distribution every time. Before, the edge weight distribution diverged when using an initial subset for f_{far}. This time, however we found that in the first two limits convergence was plausible. Convergence was too slow to be able to tell if the error was headed to 0 or not. Surprisingly, in the last limit, the empirical distribution did appear to converge to the target distribution.

Chapter 6

Dutch company network

We now apply these network generating algorithms to real data. Namely, we attempt to reconstruct the network of companies in the Netherlands, at Statistics Netherlands. Because reconstructing this network is an active project, data is already readily available in the right format. This data was previously used by a deterministic method that produced a single guess of this network. Since relying on a single network is undesirable, the probabilistic approach of the algorithms in this thesis is favourable.

6.1 Available data

The available data contained the locations of companies, and their estimated number of connections per product group for both supply and use*. Because of this, we split the data per product group, where every vertex has a location, an in-degree and an out-degree. The edge weights are again the distances.

What is not provided is a target distribution. It was decided that the suppliers and users should be more likely to connect if they are closer. This relation, is set to be $\frac{1}{(r_{ij}+C)^t}$, for $t \in \{0, \frac{1}{2}, 1, 2\}$, for distances r_{ij} and constant C. This constant C should be thought of as setting the smallest distance from where we are interested in the dynamics, and was set to be 100 meters in the project. The probability of sampling a company located 2 meters away should not be half of the probability of selecting one that is only 1 meter away. It is also decided that the output distribution should not be independent of the locations of the companies. After all, we are trying to model how companies connect.

For example, imagine there are two large clusters of companies at some distance. We then expect that most trade occurs within the clusters, but we also expect the real network to contain a substantial amount of trade between the clusters. In other words, using a predefined target distribution without considering the vertex distribution is undesirable. Instead, the desired algorithm is

*Incomplete data entries were excluded.

sampling edges with weights proportional to

$$f(r_{ij})\hat{d}_i\hat{d}_j\left(1-\frac{d_i d_j}{4m}\right),$$

where consequently the empirical edge weight distribution no longer converges to f.

The directedness of the graph is implemented by treating every company as two vertices, while excluding connections between these pairs, and also excluding connections between vertices when they are either both suppliers or users.

6.2 Implementation

For this new variant, we can yield an efficient algorithm by adjusting the implementation without subset discussed in section 2.4.2. There we repeatedly resampled the vertex, rotation, and grid depth while keeping the distance the same. This made sure we would find an edge with the desired edge weight, regardless of how likely it would be to find an edge with this weight.

Now we also resample the distance every loop, making it dependent on this probability we tried to avoid earlier. If there are twice as many edges at one distance compared to another, the probability of selecting the former should now be twice as high. However, due to the circle of a given distance covering twice as many grid cells as the distance doubles, we also have to include multiplying $f(x)$ by x to account for this bias. In practise, however, we simply choose $t \in \{-1, -\frac{1}{2}, 0, 1\}$ instead.

The last change is that we need to maintain two grids, one for the users and one for the suppliers. We sample a random vertex from either the users or the suppliers, with a 50% chance. The grids are independently resized whenever the number of vertices in a particular grid halves. We also maintain two AVL trees for tracking $\hat{d}$.

Chapter 7

Discussion and conclusions

7.1 Existing methods

We have discussed different ideas and implementations, all related to sampling graphs. First, there was the method that did involve edge weights, as proposed by [Bayati et al., 2010]. We improved the time complexity of this method from $O(md_{\max})$ to $O(m)$.

Then there was the method proposed by [Kryven, 2022] where we sample a graph according to some output. For 2D distances, we implemented this method faithfully, with a time complexity of roughly $O(m^{\frac{9}{8}}d_{\max})$, where it is unclear what the exact complexity is. For this method, we found that the empirical edge weight distribution converged nicely to the target distribution in all cases, and that we error was $O(1)$ when increasing n, and when increasing $d_{\max}$ while keeping n at a bare minimum. However, the error rapidly increased when decreasing the range in which edges were allowed. The speed of this increase was dependent on the specific target distribution, but in either case, uniformity seemed unlikely to hold because due to the variance of the error increasing as well.

We also implemented this method without assuming any structure of the edge weights. As a consequence, this means storing all edge weights, resulting in at least $O(n^2)$ storage and run time if all edges are used. However, this method also allows cutting down on the edges that are considered by only using a subset of all possible edges instead.

The main struggle for this approach then becomes selecting the edges that are used in such a manner that the edge weights of these edges follow the desired target distribution already, otherwise the method will run out of appropriate edges. Even for 2D distances, the time required for creating such a subset turns out to be prohibitively expensive. As a consequence, using this method is ill-advised from a run time perspective.

Regardless, the approach was tested without building a new subset every repetition, and the run time did not appear to deviate much from $O(m^{\frac{4}{3}})$, when setting the number of edges in the subset to be $O(m^{\frac{4}{3}})$. In other words, handling the subset appeared to be the limiting factor. More damning than the runtime is that this implementation did not converge to the target distribution

as n increased. This is likely caused by the fact that the implementation is not faithful to the intended method due to the use of a subset.

This idea was also implemented in the binary case, where edge weights get reduced to allowing the edge or not allowing the edge. Because the subsets are build to already follow the target distribution, the results turn out to be identical to those of the continuous implementation, except that it speeds up the run time by a factor 5.

7.2 Vertexwise sampling

We then moved on to another way of sampling the edges. Namely, the goal is to sample edges such that vertices are sampled with the same probability, and to have the edge weight distribution each vertex is sampling from be such that the edge weight distributions of the output graphs still converges to the target distribution. This turned out to be difficult to implement, and outright impossible if too many bins of individual vertices were empty.

Regardless, the empirical edge weight distribution converged to f_{close} in all limits. Before, the empirical edge weight distribution did not at all converge to f_{far} when using a subset method. However, when using vertexwise sampling, convergence is plausible for the first two limits, and evident for the last limit.

As for uniformity, the same patterns persisted as without vertexwise sampling. The only difference was that the error was significantly lower, but this difference was only a constant, estimated to be somewhere between 3 and 10. Regardless, it is entirely unclear if this produces graphs with equal probability, since looking at the error can only provide an argument against uniformity. A precise mathematical analysis is still required to make further claims.

The run time was difficult to estimate due to very large variances. Still, it was reduced by a factor 10 through the use of a binary implementation, making it just as fast as the binary approach without vertexwise sampling. Future work could investigate ways of maintaining a constant number of available vertices throughout the algorithm, instead of depleting the provided subset.

7.3 Dutch company network

Lastly, we applied these methods to reconstruct the Dutch company network as a project at Statistics Netherlands. For this project, we implemented a final method, which is an adaptation of the faithful implementation that does not use a subset. This method now produces a directed graph, where the edges are no longer sampled to match a target distribution, but rather to simulate how companies connect.

Bibliography

[Bayati et al., 2010] Bayati, M., Kim, J. H., and Saberi, A. (2010). A sequential algorithm for generating random graphs. *Algorithmica*, 58(4):860–910.

[Kryven, 2022] Kryven, V. (2022). Unbiased sampling of geometric graphs with degree constraints.

[Steger and Wormald, 1999] Steger, A. and Wormald, N. C. (1999). Generating random regular graphs quickly. *Combinatorics, Probability and Computing*, 8(4):377–396.