Uniform Sampling of Negative Weights in Shortest Path Networks

by
Lukas Geis

January 14, 2025

A document submitted in partial fulfillment of the requirements for the degree of
Master of Science
at
Goethe University Frankfurt

supervised by
Dr. Manuel Penschuck

Abstract

This thesis considers a maximum entropy edge weight model for shortest path networks that allows for negative weights. For a given graph $G = (V, E)$ and an edge weight function $w\colon E \to \mathcal{W}$ with weights $\mathcal{W}$, a shortest $u \to v$ path is a $u \to v$ path that minimizes the sum of edge weights along traversed edges. If G is strongly connected and $\mathcal{W}$ contains negative weights, the weight of a cycle can be negative. In that case, a shortest path is no longer well defined which is why this should be avoided. Our model thus selects edge weights $w \in \mathcal{W}^{|E|}$ uniformly at random from all weights that do not introduce a negative cycle.

We first propose an MCMC process and show that it converges to the required distribution before engineering an implementation of the process using a dynamic version of Johnson's algorithm in connection with a bidirectional Dijkstra search. We then try to parallelize the process and show the limits of this approach before slightly altering the original MCMC to allow for a potentially more efficient implementation. Finally, we empirically study the performance characteristics of every proposed implementation of the novel sampling algorithm and the model's output. We also compare all metrics across multiple graph classes, graph sizes, and initial states.

Zusammenfassung

Diese Arbeit betrachtet ein Kantengewichtsmodell mit maximaler Entropie für kürzeste Weg-Netzwerke, das negative Gewichte zulässt. Für einen gegebenen Graphen $G = (V, E)$ und eine Kantengewichtsfunktion $w\colon E \to \mathcal{W}$ mit Gewichten $\mathcal{W}$ ist ein kürzester $u \to v$ Weg ein $u \to v$ Weg, der die Summe der Kantengewichte entlang durchlaufener Kanten minimiert. Wenn G stark zusammenhängend ist und $\mathcal{W}$ negative Gewichte enthält, kann das Gewicht eines Kreises negativ sein. In diesem Fall ist ein kürzester Weg nicht mehr wohldefinitert, weshalb dies vermieden werden sollte. Unser Modell wählt daher Kantengewichte $w \in \mathcal{W}^{|E|}$ gleichmäßig zufällig aus allen Gewichten aus, die keinen negativen Kreise erzeugen.

Wir schlagen zunächst einen MCMC-Prozess vor und zeigen, dass dieser zur erforderlichen Verteilung konvergiert, bevor wir eine Implementierung des Prozesses unter Verwendung einer dynamischen Version des Johnson-Algorithmus in Verbindung mit einer bidirektionalen Dijkstra-Suche entwickeln. Anschließend versuchen wir, den Prozess zu parallelisieren und die Grenzen dieses Ansatzes aufzuzeigen, bevor wir den ursprünglichen MCMC leicht ändern, um eine potenziell effizientere Implementierung zu ermöglichen. Abschließend untersuchen wir empirisch die Leistungsmerkmale jeder vorgeschlagenen Implementierung des neuen Sampling-Algorithmus und die Ausgabe des Modells. Wir vergleichen außerdem alle Metriken über mehrere Graphenklassen, Graphengrößen und Anfangszustände hinweg.

Acknowledgements

First and foremost I want to thank all my co-authors of [56] who contributed a more than significant part to this thesis with their research and feedback in the original paper. Without them, I would have not only had a much harder time finding related literature but also had a significant drop in quality in the first version of this thesis. Special thanks goes out to Manuel Penschuck and Alexander Leonhardt who took on the cruel job of supervising and supporting me in the early stages of this research. Discussions with them were not only always very fruitful but also enjoyable. Both of them also set aside significant time to proofread this thesis. I also want to thank Holger Dell, Manuel Penschuck, and Alexander Leonhardt whose independent work yielded a template for this thesis. I finally want to thank my friends and family and especially my girlfriend for supporting me along the way and allowing me to cool off with them when I needed to.

Contents

1 Introduction

For a given set of nodes and weighted edges that connect two nodes each, the shortest path problem asks to find a path from one node to another in the shortest way possible with respect to the edge weights. This task finds applications in a variety of fields such as network routing, geographical navigation, and other optimization tasks. In the classic variant where each edge e is assigned a non-negative edge $w(e)$, Dijkstra's algorithm [38] is a common choice for a simple and efficient algorithm.

While most applications operate under the assumption of non-negative weights, many allow weights to be negative. The most prominent one is in the context of flow problems [41]. A more recent example that is steadily gaining importance is the routing of recuperating electric vehicles: energy gains downhill and energy expenditures uphill. Henceforth, the need for shortest path networks with negative weights is steadily growing.

For such shortest paths to be meaningful, however, it is important that the underlying network does not contain a negative cycle, namely a cycle with a total length of less than 0. In these cases, shortest paths are unbounded and it is better to traverse such negative cycles as often as possible. Especially in the context of electric vehicles, this is detrimental as negative cycles make no sense in the setting of energy expenditures. Thus, to properly study shortest path algorithms, it is important to have test data with negative weights but no negative cycles.

Classic analytical and empirical studies of shortest path algorithms often involve random graph models [25, 50] as they provide predictable and thus mathematically tractable structure [10, 101]. In almost all frequently studied models, however, graph topology, i.e. nodes and edges, and edge weights are considered separately. Although there is a significant effort put towards "realistic" random graph models (see for example survey [40]), maximum entropy network models are often preferred.

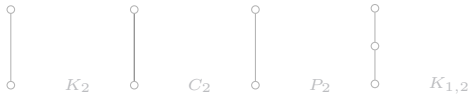
In simple terms, a maximum entropy network model yields a probability distribution over a class of graphs that conform to specified (structural) properties such that this distribution is unbiased. That is, among all graphs that adhere to the desired property, there is no bias towards certain instances. This makes maximum entropy models a great choice for baselines and hypothesis testing [60, 79, 87]. The most famous (and arguably most basic) maximum entropy network models are the $\mathcal{G}(n, p)$ [57] and the $\mathcal{G}(n, m)$ [42] models that parametrize over graph size and density.

The goal of this thesis however is to study and engineer generators of maximum entropy models for random edge weights that can be negative but not introduce

negative cycles in given graphs. We start with a basic sampling algorithm and show its limitations before introducing a dynamic algorithm that approximates the desired maximum entropy model over time. We then engineer this approximate sampling algorithm and discuss possible methods and bottlenecks when trying to utilize parallelism.

Chapter 2 introduces common notation and the most important notions and algorithms used in this thesis. Chapter 3 briefly discusses related work. In Chapter 4, we define our desired model and engineer an approximate sampling algorithm which we try to parallelize in Chapter 5. In Chapter 6, we then extensively study all proposed algorithms and our underlying sampling method empirically. Finally, the appendix (chapters Appendices A to C) provides more detail for Chapter 6 as well as additional insights as part of this thesis.

Disclaimer: Most of this thesis is also the content of a preprint [56]. I was the main contributor to this paper and responsible for almost all experiments, the reference implementation as well as most new algorithmic insights of the following chapters. Sections or results (for example in Appendix A) that are not part of my direct work are marked by an additional citation.



2 Preliminaries

In this chapter, we will give a short introduction into graph theory, shortest path algorithms, random graph models as well as Markov chains. We denote the set of positive real numbers by $\mathbb{R}^+$, the set of consecutive whole numbers $\{i \mid a \leq i \leq n, ßin\mathbb{Z}\}$ by $[a..b]$ and write $[n]$ as shorthand for $[1..n]$. $\mathcal{U}(X)$ refers to a uniform distribution over the ground set X.

2.1 Graph Theory

We only focus on weighted directed graphs as negative weights in undirected graphs are significantly harder to handle [1].

▷ Definition 2.1 (Directed Graph). *A directed graph G is a tuple (V, E) where V is a finite set of nodes (or vertices) and $E \subseteq V \times V$ is a set of directed edges. For edge $(u, v) \in E$, we call u the source and v the target of the edge. If unambiguous from context, we use $n := |V|$ and $m := |E|$ to denote the number of nodes and edges.*

▷ Definition 2.2 (Weight Function). *For graph $G = (V, E)$, a weight function $w\colon E \to \mathcal{W}$ assigns each edge a weight from $\mathcal{W}$ (typically $\mathcal{W} \subseteq \mathbb{Z}$ or $\mathcal{W} \subseteq \mathbb{R}$). For edge $e = (u, v) \in E$, we call $w(e) = w(u, v)$ its edge weight.*

Unless further specified, for the rest of the thesis, $G = (V, E)$ refers to a graph with nodes V and edges E and $w\colon E \to \mathcal{W}$ to a weight function defined on G with $\mathcal{W} \subseteq \mathbb{R}$.

▷ Definition 2.3 (Path). *We call $P = (v_1, \ldots, v_k) \in V^k$ a k-path (or just path), if all edges (v_i, v_{i+1}), $i \in [k-1]$ exist in E. We call P a $u \to v$ path if $v_1 = u$ and $v_k = v$ and say u can reach v.*

▷ Definition 2.4 (Cycle). *A $(k+1)$-path $(v_1, \ldots, v_{k+1})$ is a k-cycle if $v_1 = v_{k+1}$.*

We call a path or cycle *simple* if no node appears more than once (for a cycle this excludes $v_1 = v_{k+1}$). We denote the number of edges in a path (or cycle) P by $|P|$. An acyclic graph is a graph with no cycles.

▷ Definition 2.5 (Tree). *G is a (rooted) tree if it is acyclic and there exists $u \in V$ that can reach every other node $v \in V \setminus \{u\}$.*

▷ Definition 2.6 (SCC). *We call a subset of nodes $S \subseteq V$ a strongly connected component (SCC) if for every $u, v \in S$, u can reach v and vice-versa. We say G is strongly connected if V is a SCC.*

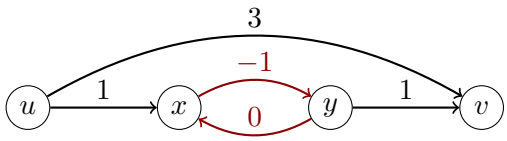


Figure 2.1: A graph with 4 nodes containing a negative cycle (x, y, x).

▷ Definition 2.7 (Subgraph). *We call $G' = (S, E')$ a subgraph of G, if $S \subseteq V$ and $E' \subseteq E \cap S^2$. If $E' = E \cap S^2$, we call G' the subgraph induced by S. If G' is a tree, we call G' a subtree of G.*

▷ Definition 2.8 (Weight of a Path). *For a k-path $P = (v_1, \ldots, v_k)$, we overload w and write*

$$w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

to denote its weight.

We also allow 1-paths, i.e. $P = (u)$ and set its weight to $w(P) = 0$.

▷ Definition 2.9 (Shortest Path). *A shortest $u \to v$ path is defined as a $u \to v$ path with minimum weight among all possible $u \to v$ paths. For an ordered pair of vertices $(u, v) \in V^2$, we denote the distance from u to v by*

$$d_{G,w}(u, v) = d(u, v) = \begin{cases} \min \{w(P) \mid P \text{ is a } u \to v \text{ path}\} & \text{if } u \text{ can reach } v \\ \infty & \text{otherwise} \end{cases}$$

In the unweighted case, i.e. $w \colon E \to \{1\}$, the weight of a shortest $u \to v$ path P is equivalent to the number of edges $|P|$ in it and we only need to traverse the graph using BreadthFirstSearch (Bfs) [106] to compute a shortest path. For general weights, however, the length of a shortest $u \to v$ path can be arbitrary, even negative if we allow negative weights which can lead to complications.

▷ Definition 2.10 (Negative Cycle). *A cycle C is a negative cycle if $w(C) < 0$.*

The notion of a shortest path only makes sense if there is no negative cycle present. For that, consider Fig. 2.1. There exist two simple $u \to v$ paths: $P_1 = (u, v)$ and $P_2 = (u, x, y, v)$ with $w(P_1) = 3 > 1 = w(P_2)$. Thus, P_2 is a shortest **simple** $u \to v$ path. The cycle $C = (x, y, x)$ however has weight $w(C) = -1$ and is thus negative. We can now traverse C multiple times instead of once (as in P_2) to continuously decrease the weight of P_2. Therefore, the weight of a shortest $u \to v$ path is unbounded and there is no finite shortest $u \to v$ path.

▷ Definition 2.11 (Consistent Weight Function). *We say (G, w) are consistent if no negative cycles exist.*

Hence, if we want to have reasonable shortest paths in a graph, we desire (G, w) to be consistent. The rest of the thesis focuses on checking for and maintaining consistency of (G, w) under dynamic weight functions.

Algorithm 1: Dijkstra's SSSP algorithm [38].

Input: graph $G = (V, E)$, weights $w\colon E \to \mathcal{W}$, $\mathcal{W} \subseteq \mathbb{R}^+$, source node u
Output: $\textsc{Spt}(u)$

```
1  for v ∈ V do
2      D[v] ← ∞
3  D[u] ← 0
4  Q ← Min-PriorityQueue
5  add u to Q
6  while Q is not empty do
7      v ← node in Q with minimum D[v]
8      for (v, x) ∈ E do
9          if D[v] + w(v, x) < D[x] then
10             D[x] ← D[v] + w(v, x)
11             if x ∉ Q then add x to Q
12 return D
```

2.2 Shortest Path Algorithms

An important property of shortest paths is that they consists of shortest subpaths.

▷ Observation 2.12. *Let $P = (v_1, \dots, v_k)$ be a shortest $v_1 \to v_k$ path. Then $P_i = (v_1, \dots, v_i)$ is a shortest $v_1 \to v_i$ path for $i \in [k]$.*

▷ Definition 2.13 (Shortest Path Tree). *For source node u, a shortest path tree T from u, denoted by $\textsc{Spt}(u)$ is a subtree of G such that*

$$d_{G,w}(u, v) = d_{T,w}(u, v) \quad \forall v \in V.$$

Note that if G is strongly connected and has no negative cycles, due to Obs. 2.12, $\textsc{Spt}(u)$ exists for every $u \in V$ and contains every node in V. Thus, $\textsc{Spt}(u)$ is well-defined if there is at most one shortest $u \to v$ path for every $v \in V$. Note that in practice we often represent $\textsc{Spt}(u)$ implicitly as a distance mapping $D[v] = d(u, v)$ and do not explicitly store the structure of $\textsc{Spt}(u)$.

▷ Definition 2.14 (SSSP Problem). *The SingleSourceShortestPath problem (SSSP) asks to compute $\textsc{Spt}(u)$ for a given source node u.*

2.2.1 Dijkstra

One of the most famous algorithm for the weighted SSSP problem is Dijkstra's algorithm [38]. Described in Algorithm 1, it starts from a given source node u and stores the tentative distance $D[v]$ for every node $v \in V$. In each round, we visit a node v with smallest tentative distance. Then, we relax all its edges (v, x) by checking whether the $u \to x$ path that visits v directly before x is shorter than the currently found $u \to x$ path. To decide which node to visit, we keep all seen nodes that we have not visited yet in a Min-PriorityQueue which supports queries in logarithmic time.

Algorithm 2: BellmanFord's SSSP algorithm [13, 64].

Input: graph $G = (V, E)$, weights $w\colon E \to \mathcal{W}$, source node u
Output: $\text{Spt}(u)$ or HasNegativeCycle

```
1  for v ∈ V do
2      D[v] ← ∞
3  D[u] ← 0
4  repeat |V| − 1 times
5      for (u, v) ∈ E do
6          if D[u] + w(u, v) < D[v] then
7              D[v] = D[u] + w(u, v)
8  for (u, v) ∈ E do
9      if D[u] + w(u, v) < D[v] then
10         return HasNegativeCycle
11 return D
```

The algorithm however relies on the fact that w only assigns non-negative weights to all edges. If not, Dijkstra would still correctly output $\text{Spt}(u)$ if it exists but would not terminate if G contains a negative cycle. Thus, to use Dijkstra, we require all edges that we can traverse to have a non-negative weight.[1]

▷ Lemma 2.15. *For graph $G = (V, E)$ and weights $w\colon E \to \mathbb{R}^+$, Dijkstra outputs a correct shortest path tree in time*

$$\mathcal{T}_{\text{Dijkstra}} = \mathcal{O}(m + n \log n)\,.$$[2]

2.2.2 BellmanFord

While Dijkstra is very fast for the SSSP problem, it requires edge weights to be non-negative which is not always the case. Especially in the context of this thesis, we almost always allow (and have) negative edge weights making Dijkstra infeasible to use. An alternative for general edge weights that also correctly detects whether there is a negative cycle present in the graph is the BellmanFord algorithm [13, 64].

Similar to Dijkstra, it stores tentative distances for every node and iteratively relaxes all edges. The standard version in Algorithm 2, relaxes every edge $n - 1$ times, leading to a total runtime of $\Theta(nm)$. If there is no negative cycle in G, every shortest path is simple and thus has length at most $n - 1$. Hence, if there is no negative cycle in G, after $n - 1$ relaxations of all edges, no further relaxations

[1] If there exist negative edges but no negative cycle, Dijkstra would still correctly output $\text{Spt}(u)$ but not in the guaranteed runtime (see Lemma 2.15).

[2] With standard priority queues, the runtime is rather $\mathcal{O}((m + n) \log n)$. Using a Fibonacci-Heap [47], we get the stated runtime. Furthermore, if a Radix-Heap [2] is available, we can achieve a runtime of $\mathcal{O}(m + n)$.

are possible and we can correctly return $\textsc{Spt}(u)$. If not, then there is at least one further relaxation that can be done and we return HasNegativeCycle.

Clearly, this method is very inefficient in practice which is why efficient implementations resort to the SPFA heuristic [82]: instead of relaxing every edge $n-1$ times, we keep a queue Q of all nodes for which we have to relax its outgoing edges. If we relax (v, x) with $x \notin Q$, we add x to Q. We repeat this until Q is empty and we can return $\textsc{Spt}(u)$.

This however leads to the same complication we have with Dijkstra and negative cycles: if there is a negative cycle, this algorithm will not terminate. Thus, we can additionally store the predecessor in the tentative shortest path tree for every node and check every n^{th} step if the tentative shortest path tree is in fact acyclic. If there is a negative cycle in G, at some point, this is not the case and we return HasNegativeCycle. Since the check for acyclicity of the tentative shortest path tree runs in time $\mathcal{O}(n)$ [65], this does not lead to a worse asymptotic runtime.[3]

▷ Definition 2.16 (Diameter). *The diameter* $\operatorname{diam}(G)$ *of* G *is defined as*

$$\max_{u,v \in V} |P_{uv}|$$

where P_{uv} *is a shortest* $u \to v$ *path with the minimum number of edges. For* $\textsc{Spt}(u)$*, the diameter* $\operatorname{diam}_{SSSP(u)}(G)$ *is defined as the depth of the tree, i.e.*

$$\max_{v \in V} |P_{uv}| .$$

If there exist multiple shortest path trees, $\operatorname{diam}_{SSSP(u)}(G)$ *is the minimum depth among those.*

If there is no negative cycle in G, $\textsc{Spt}(u)$ exists and we can bound the number of relaxation rounds by the depth of $\textsc{Spt}(u)$.

▷ Lemma 2.17. *The SPFA heuristic runs in time*

$$\mathcal{T}_{\textsc{Spfa}} = \mathcal{O}\Big(\operatorname{diam}_{SSSP(u)}(G) m\Big)$$

Note that possibly $\operatorname{diam}_{SSSP(u)}(G) = \Theta(n)$ and the runtime is still bounded by $\mathcal{O}(nm)$. For the rest of the thesis, we will refer to the SPFA heuristic as BellmanFord.

2.2.3 Johnson's AllPairShortestPath Algorithm

We can extend the SSSP problem to all shortest paths in G.

▷ Definition 2.18 (APSP Problem). *The* AllPairShortestPath *problem (APSP) asks to compute* $\textsc{Spt}(u)$ *for every* $u \in V$*.*

While the naive algorithm of running one SSSP instance for every $u \in V$ might be feasible for graphs with non-negative weights due to faster runtime of Dijkstra, for graphs with general weights, one would have to resort to BellmanFord to

[3] Alternatively, one could just count the number of relaxations and reject if there are too many, i.e. $\omega(nm)$.

Algorithm 3: Johnson's APSP algorithm [63].

Input: graph $G = (V, E)$, weights $w\colon E \to \mathcal{W}$
Output: distance matrix D or HasNegativeCycle

$V' \leftarrow V \cup \{s\}$
$E' \leftarrow E \cup \{(s, v) \mid v \in V\}$
$w' \leftarrow w \cup \{w(s, v) = 0 \mid v \in V\}$
$T \leftarrow \text{BellmanFord}(G' = (V', E'), w', s)$
if $T = \textit{HasNegativeCycle}$ **then**
  return *HasNegativeCycle*
$\phi(u) \leftarrow -T[u] \quad \forall u \in V$
for $u \in V$ **do**
  $D[u, \cdot] \leftarrow \text{Dijkstra}(G, w_\phi, u)$
for $u, v \in V$ **do**
  $D[u, v] \leftarrow D[u, v] + \phi(u) - \phi(v)$
return D

solve the SSSP subproblem. Running n BellmanFord instances however is slow in practice compared to n Dijkstra instances. Johnson used an ingenious workaround to map the SSSP problem with general weights to an SSSP problem with strictly non-negative weights [63]. Note that the framework itself originally came from the work on network-flow problems of Edmonds and Karp [41].

▷ Definition 2.19 (Potential Function). *For any function $\phi\colon V \to \mathbb{R}$, we define w_ϕ as the weight function with*

$$w_\phi(u, v) = w(u, v) + \phi(v) - \phi(u).$$

We call ϕ a potential function and w_ϕ a potential weight function and say ϕ is feasible if $w_\phi(u, v) \geq 0\ \forall (u, v) \in E$. We say an edge $(u, v) \in E$ is broken if $w_\phi(u, v) < 0$.

▷ Lemma 2.20 (Based on [63]). *For any potential function ϕ, any path in G is a shortest path with respect to w iff it is also a shortest path with respect to w_ϕ. Furthermore, the weight of a cycle remains unchanged under ϕ.*

Proof. Let $P = (v_1, \ldots, v_k)$ be a path in G. The path weight $w_\phi(P)$ follows as a telescope sum that cancels out all contributions but the first and the last potential:

$$\begin{aligned} w_\phi(P) &= \sum_{i=1}^{k-1} [w(v_i, v_{i+1}) - \phi(v_i) + \phi(v_{i+1})] \\ &= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) - \sum_{i=1}^{k-1} \phi(v_i) + \sum_{i=2}^{k} \phi(v_i) \\ &= w(P) - \phi(v_1) + \phi(v_k) \end{aligned} \tag{2.1}$$

The remaining potentials $\phi(v_1)$ and $\phi(v_k)$ appear on any $v_1 \to v_k$ path. Thus, a shortest path for w_ϕ remains a shortest path in terms of w (and vice versa).

Notice that if C is a cycle, we have $v_1 = v_k$ and it follows from Eq. (2.1) that $w_\phi(P) = w(P) + \phi(v_1) - \phi(v_1) = w(P)$ and thus the second claim of the lemma. □

Observe that this implies that every shortest path tree with respect to w is also a shortest path tree with respect to w_ϕ. Furthermore, it follows from the previous lemma that if there exists a feasible potential function ϕ, (G, w) are consistent since weights of cycles remain unchanged under potential weights and every potential weight is non-negative.

Thus, if we can compute a feasible ϕ, we not only know that there is no negative cycle in G but can also now use Dijkstra instead of BellmanFord to compute shortest path trees faster.

Algorithm 3 does exactly that using one BellmanFord iteration. The correctness of the step follows from the following lemma using $\phi = 0$ (for which $w_\phi = w$):

▷ Lemma 2.21 (Based on [63]). *Let T be some shortest path tree in G with respect to w_ϕ for some potential function ϕ that contains all nodes. Then, the potential function ϕ' with $\phi'(u) = \phi(u) - T[u]$ for $u \in V$ is feasible for G and w.*

Proof. First note that if T exists, there are no negative cycles in G. By definition of a shortest path tree, we have for every edge $(u, v) \in E$

$$T[v] \leq T[u] + w_\phi(u, v) \quad \iff \quad w_\phi(u, v) + T[u] - T[v] \geq 0.$$

Thus, for $w_{\phi'}$, it holds due to Lemma 2.20 that

$$\begin{aligned} w_{\phi'}(u, v) &= w(u, v) + \phi'(v) - \phi'(u) \\ &= w(u, v) + (\phi(v) - T[v]) - (\phi(u) - T[u]) \\ &= w(u, v) + \phi(v) - \phi(u) + T[u] - T[v] \\ &= w_\phi(u, v) + T[u] - T[v] \\ &\geq 0 \end{aligned}$$

and ϕ' is feasible. □

Notice that while this thesis assumes G to be strongly connected, in general, this is not the case, thus we introduce an additional node s in Algorithm 3 to ensure that we can reach and assign a potential to every node in G.

▷ Observation 2.22. *Johnson's algorithm runs in time*

$$\mathcal{T}_{\textit{Johnson}} = \mathcal{O}(\underbrace{\operatorname{diam}(G)m}_{\textit{BellmanFord}} + n\underbrace{(m + n\log n)}_{\textit{Dijkstra}}) = \mathcal{O}\Big(nm + n^2 \log n\Big)$$

2.3 Random Graph Models

Random graph models are a fundamental tool to provide a flexible and controllable source for synthetic graph data that can be used in experiments to evaluate a variety of graph algorithms. Their study dates back to the 1950s when they were introduced by Edgar N. Gilbert [57] as well as Paul Erdős and Alfréd Rényi [42].

We introduce three random graph models that we will later use in Chapter 6 to evaluate our algorithms. Note that while we discuss some practical optimization details, we only implement an optimized but not *highly* optimized version of the models as they are not the focus of this thesis and our test instances are reasonably small.

2.3.1 Gilbert Graphs

The $\mathcal{G}(n, p)$ model, proposed by Gilbert in 1959 [57], is one of the first and most simplistic random graphs. For a fixed number of n nodes, we insert *every* possible edge into the graph with probability p, independent of other edges. This generalizes the $\mathcal{G}(n)$ model which simply draws a uniform random graph with n nodes.[4]

Since we are only interested in directed graphs in this thesis, there is a total of n^2 possible edges that can exist in a graph with n nodes (if we exclude multi-edges). Because every edge exists with an independent probability of p, the number of edges therefore follows a binomial distribution with parameters n^2 and p. Hence, in expectation, the number of edges in a $\mathcal{G}(n, p)$ graph follows as

$$\mathbb{E}[m] = n^2 \cdot p.$$

Directly parameterizing over p however is often not that practical and does not allow for a good comparison with other models. Instead, we parameterize over the average degree of the graph.

▷ Definition 2.23 (Degree). *The out-degree* $\deg_{\text{out}}(u)$ *of a node* $u \in V$ *is the number of edges* $(u, v) \in E$ *where* u *is the source of the edge. The average (out-)degree* $\overline{d}$ *of* G *is the average over all out-degrees and thus given by*

$$\overline{d} = \frac{1}{n} \sum_{u \in V} \deg_{\text{out}}(u) = \frac{m}{n}.$$

Analogously, we define $\deg_{\text{in}}(u)$ *as the number of edges* $(v, u) \in E$ *where* u *is the target.*

We refer to $\overline{d}$ as the average degree (although it is the average out-degree) for the rest of the thesis. Similar to the number of edges, the degree of a node also follows a binomial distribution — parameterized over n and p (instead of n^2 and p). We can achieve an average degree of $\overline{d}$ in expectation by setting p to $\overline{d}/n$:

$$\mathbb{E}[m] = n^2 \cdot \frac{\overline{d}}{n} = n \cdot \overline{d}$$

We denote the slightly altered model by $\mathcal{GNP}(n, \overline{d})$ and treat both models exchangeably.

2.3.2 DirectedScaleFree Graphs

While the simplistic nature of the $\mathcal{G}(n, p)$ model makes it easier to analyze, it also prohibits random graphs from this model from expressing more complex

[4] For $p = 0.5$, one can prove that $\mathcal{G}(n, p)$ and $\mathcal{G}(n)$ produce the same distribution over graphs.

behaviors. Barabási and Albert showed that the $\mathcal{G}(n,p)$ model fails to accurately model social networks [11]. They coined the term "scale-free" which describes networks whose node degrees follow a power-law distribution:

▷ Definition 2.24 (Scale-Free [11]). *A network is "scale-free" if its node degrees follow a power law distribution given by:*

$$\mathbb{P}[\deg(u) = k] \approx k^{-\gamma}$$

where γ is the power law exponent.

A common algorithm to produce such networks is the PreferentialAttachment algorithm from the same paper which iteratively inserts new nodes into a graph. A new node u then adds an edge to a previous node v where v is chosen randomly with probability proportional to its current degree. This algorithm however is used to create undirected graphs which are not particularly useful for this thesis. Instead of interpreting one undirected edge $\{u,v\}$ as two directed edges $(u,v),(v,u)$, we adopt a directed variant proposed by Bollobás et al. [24].

It works similarly to PreferentialAttachment by iteratively adding new nodes and directed edges to the graph. It is parameterized over the following parameters:

- n: The number of nodes.
- δ_{in}: The bias parameter for drawing nodes proportional to their in-degree:

$$p_{in}(u) = \frac{\delta_{in} + \deg_{\text{in}}(u)}{\delta_{in} n + m}$$

- δ_{out}: The bias parameter for drawing nodes proportional to their out-degree:

$$p_{out}(u) = \frac{\delta_{out} + \deg_{\text{out}}(u)}{\delta_{out} n + m}$$

- α: The probability to add a new node u along an outgoing edge (u,v) directed to an existing node v to the graph where v is selected with probability $p_{in}(v)$.
- γ: The probability to add a new node u along an incoming edge (v,u) directed from an existing node v to the graph where v is selected with probability $p_{out}(v)$.[5]
- β: The probability to add a new edge (u,v) between two existing nodes u,v where u is chosen with probability $p_{out}(u)$ and v is chosen with probability $p_{in}(v)$.

Note that we require $\delta_{in},\delta_{out},\alpha,\beta,\gamma \geq 0$ as well as $\alpha+\beta+\gamma = 1$.
For our purposes, it suffices again to parameterize over the number of nodes n and the average degree $\overline{d}$ of the graph. Thus, we only use $\delta_{out} = \delta_{in} = 1$ as well as $\alpha = \gamma = \frac{1-\beta}{2}$ and control $\overline{d}$ over β. We denote this version by $\mathcal{DSF}(n,\overline{d})$.

[5] Not to be confused with γ in Definition 2.24.

2.3.3 Hyperbolic Graphs

Another "scale-free" model which was studied extensively in recent years is the random hyperbolic graph model [18, 19, 69, 88, 104]. It is rooted in hyperbolic geometry and belongs to the larger random graph model class of geometric graphs. In geometric graphs, an edge between two nodes is only formed if some condition in the underlying geometry is met.

Hyperbolic graphs are generated by sampling n random positions in the hyperbolic disk of radius R and connecting vertices proportional to their distance from each other. More formally, we parameterize over the following:

- n: The number of nodes.
- $\overline{d}$: The average degree of the graph.
- α: The dispersion parameter that controls fraction of nodes near the center of the hyperbolic disk and thereby the skewness of the degree distribution.
- T: The temperature that controls randomness and the clustering coefficient.

Depending on these parameters, a target radius R is computed that leads to the desired properties of the graph. Then, for each $i \in [n]$, we draw a random polar coordinate $p_i = (r_i, \theta_i)$ in the hyperbolic plane where $r_i \sim \mathcal{U}([0, 2\pi))$ and θ_i is drawn according to the density function

$$f(r) = \frac{\alpha \sinh(\alpha r)}{\cosh(\alpha R) - 1}.$$

The distance d_{ij} between two points $p_i = (r_i, \theta_i)$ and $p_j = (r_j, \theta_j)$ in the hyperbolic plane can be computed by

$$d_{ij} = \operatorname{acosh}\left(\cosh(r_i)\cosh(r_j) - \sinh(r_i)\sinh(r_j)\cos(\theta_i - \theta_j)\right).$$

Then, for the case of $T = 0$ which is dubbed the *Threshold* case, p_i and p_j are connected iff $d_{ij} \leq R$. Otherwise, we connect them with probability

$$p_T = \left(e^{(d_{ij}-R)/(2T)} + 1\right)^{-1}.$$

Notice that for $d_{ij} < R$, we have $\lim_{T\to 0} p_T = 1$, and $\lim_{T\to 0} p_T = 0$ for $d_{ij} > R$ and thus the two variants coincide.

Graphs generated with this model have many properties observed in real world networks [20]. They are "scale-free", have a high clustering coefficient [61] and a small diameter [48, 83].

Unfortunately, due to the underlying nature of how an edge is formed, this model can only produce undirected graphs. Therefore, we interpret an undirected edge $\{u, v\}$ as two directed edges (u, v) and (v, u). Our implementation is an adaptation of [104] which partitions the hyperbolic plane into bands of decreasing length to make connection checks more local. This leads to significantly better runtimes — both theoretical and practical — and is used in almost all competitive generators [**?**, 88, 104]. We again only parameterize over n and $\overline{d}$, denote the model by $\mathcal{RHG}(n, \overline{d})$, and set $\alpha = 1, T = 0$.

Fig. 2.2 shows the degree distributions for the three proposed models. It clearly illustrates that $\mathcal{GNP}$ graphs are not fit to model "scale-free" networks as their degree distribution is a binomial distribution.

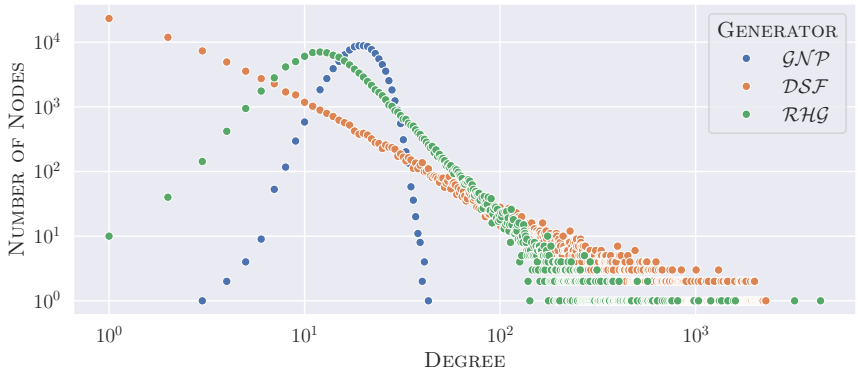


Figure 2.2: Degree distributions for different random graph models with 100000 nodes and average degree 20.

2.4 Parallel Algorithms

Parallel algorithms extend the underlying sequential machine model by allowing multiple operations to be done in parallel at the same time. For that, we can utilize up to p processing units (PUs) that can run computations in parallel. Although there are multiple models with different restrictions and capabilities, we restrict ourselves to a model where multiple PUs can read data in parallel without interfering so long as this data is not currently being modified. We however do not allow multiple PUs to change specific values at the same time.[6] This eliminates data races and is often the way parallelism is implemented in real machines.

We also only make use of the Fork-Join model [35] in our implementations later in Chapter 6. In the Fork-Join model, we extend the sequential RAM model by two operations:

(Fork) Split (*fork*) the current task into N new independent parallel subtasks that can run in parallel — stop the execution of the current task.

(Join) Wait for a subset of subtasks to finish their executions and *join* these parallel branches back into a single task. This can be accompanied by collecting the data computed by the parallel subtasks.

In this model, we can also recursively *fork* subtasks to further subdivide our problem.[7] For us, in an algorithmic context, the exact method of parallelizing is however not that relevant as it often leads only to constant additional operations since we restrict ourselves to constant number of PUs.

For an algorithmic problem $\mathcal{A}(n)$, let $\mathcal{S}$ denote a sequential algorithm solving $\mathcal{A}(n)$ and $\mathcal{P}$ denote a parallel algorithm solving $\mathcal{A}(n)$ using p PUs. Let $\mathcal{T}_{\mathcal{S}}(n)$ and $\mathcal{T}_{\mathcal{P}}(n)$ denote their respective runtimes.

▷ Definition 2.25 (Work). *The work $W_{\mathcal{P}}(n)$ of a parallel algorithm $\mathcal{P}$ is the running time $\mathcal{T}_{\mathcal{P}}(n)$ times the number of PUs p used, i.e.*

$$W_{\mathcal{P}}(n) \coloneqq p \cdot \mathcal{T}_{\mathcal{P}}(n).$$

[6] In the PRAM-Architecture [46], this would equate to the CREW-model.

[7] In common implementations of this model, (Fork) only creates 2 subtasks. Thus to allow for p PUs, we often *fork* $\Theta(\log p)$ times (in parallel) to divide the problem into p subtasks.

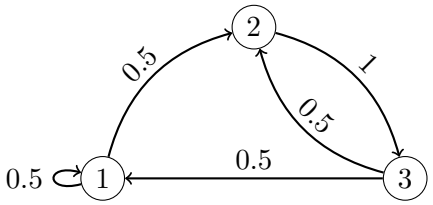


Figure 2.3: A MarkovChain with 3 states.

Thus, $W_{\mathcal{P}}(n)$ is the total number of operations of $\mathcal{P}$ on inputs of size n (asymptotically). This gives a direct bound on the work if $\mathcal{S}$ is the fastest possible algorithm solving $\mathcal{A}(n)$ since simulating the work of p PUs sequentially leads to no more total work.

▷ Observation 2.26. *If $\mathcal{S}$ is the fastest sequential algorithm solving $\mathcal{A}(n)$, then*

$$W_{\mathcal{P}}(n) = \Omega(\mathcal{T}_{\mathcal{S}}(n))$$

for every parallel algorithm $\mathcal{P}$ solving $\mathcal{A}(n)$.

We say a parallel algorithm $\mathcal{P}$ is work-optimal if $W_{\mathcal{P}}(n) = \Theta(\mathcal{T}_{\mathcal{S}}(n))$.

Another way to analyze parallel algorithms is to formalize the gain we get by using a parallel algorithm $\mathcal{P}$ instead of a sequential algorithm $\mathcal{S}$.

▷ Definition 2.27 (Speedup). *The speedup $S_{\mathcal{S},\mathcal{P}}(n)$ of a parallel algorithm $\mathcal{P}$ over a sequential algorithm $\mathcal{S}$ is defined as*

$$S_{\mathcal{S},\mathcal{P}}(n) \coloneqq \frac{\mathcal{T}_{\mathcal{S}}(n)}{\mathcal{T}_{\mathcal{P}}(n)}.$$

Note that in practice the notion of a speedup often only makes sense if either $\mathcal{S}$ is the fastest sequential algorithm for $\mathcal{A}(n)$ or $\mathcal{P}$ is a direct parallelization of $\mathcal{S}$, i.e. $\mathcal{P}$ performs the exact same steps as $\mathcal{S}$ with some possibly additional steps (but not less!). For the rest of the thesis, we assume that at least one of these holds true. We can extend Obs. 2.26 to speedups.

▷ Observation 2.28. *For a parallel algorithm $\mathcal{P}$ with p PUs and a sequential algorithm $\mathcal{S}$, always*

$$S_{\mathcal{S},\mathcal{P}}(n) = \mathcal{O}(p)\,.$$

2.5 Markov Chains

Markov chains are a widely studied and adopted model for dynamic random processes [3,49,58,96]. They are "memoryless" since every transition only depends on the current state and input which often makes them a great modeling choice [70].

▷ Definition 2.29 (Stochastic Matrix). *A matrix $P \in \mathbb{R}^{n\times n}$ with $n \in \mathbb{N}$ is called stochastic if $P_{i,j} \in [0,1]$ for all $i,j \in [n]$ and*

$$\sum_{j=1}^{n} P_{i,j} = 1 \quad \forall i \in [n].$$

We sometimes write $P(i,j)$ instead of $P_{i,j}$.

A stochastic matrix can model transition probabilities. Each row $i \in [n]$ then represents the current state i and each column $j \in [j]$ in row i then represents the probability of transitioning from state i to state j. Fixing an arbitrary order for elements of a set, we can model transition probabilities between elements of arbitrary (finite) sets.

▷ Definition 2.30 (Markov Chain [74]). *A MarkovChain is a tuple $\mathcal{M} = (\mathcal{S}, P)$ where $\mathcal{S}$ is a (finite) set of states and $P \in [0,1]^{|\mathcal{S}|\times|\mathcal{S}|}$ is a stochastic matrix. P defines transition probabilites between states of $\mathcal{S}$, i.e. $P_{i,j}$ is the probability to transition from state $i \in \mathcal{S}$ to $j \in \mathcal{S}$.*

In Fig. 2.3 for example, we have $\mathcal{S} = \{1,2,3\}$ and transition probabilities

$$P = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}.$$

While we formalize MarkovChains here using a finite state set and transition matrix, we can also model transition matrices and consequently MarkovChains for countable infinite sets of states $\mathcal{S}$. Furthermore, we often describe a transition matrix and thus its MarkovChain only with its non-zero transition probabilities without formalizing the complete transition matrix.

▷ Definition 2.31 (Distribution). *A vector $\pi \in [0,1]^{|\mathcal{S}|}$ is a distribution over states of the MarkovChain if $\|\pi\| = 1$.*

A distribution can represent the probabilities in which state the MarkovChain is currently in. Typically, in the beginning, this is deterministic with $\pi_i = 1$ for some i and $\pi_j = 0$ for every other $j \neq i$. After one step, we might end up in different states depending on the transition probabilities. Thus, the distribution is no longer deterministic if we make non-deterministic steps.

▷ Definition 2.32 (Random Walk). *A random walk starts in any distribution over states $\pi^{(0)} \in [0,1]^{|\mathcal{S}|}$ and applies the transition matrix iteratively, i.e. makes a random step based on P*

$$\pi^{(k+1)} = P \cdot \pi^{(k)} = P^2 \cdot \pi^{(k-1)} = \cdots = P^{k+1} \cdot \pi^{(0)}.$$

▷ Definition 2.33 (Stationary Distribution). *A distribution π over states is stationary if $\pi = P \cdot \pi$.*

For example, $\pi = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)$ is a stationary distribution for the MarkovChain in Fig. 2.3. It can be shown that for certain MarkovChains, a stationary distribution π always exists and every random walk converges to π independent of its initial distribution.

▷ Definition 2.34 (Irreducible). *A MarkovChain is irreducible if the underlying state graph is strongly connected.*

▷ Definition 2.35 (Aperiodic). *A MarkovChain is aperiodic if for every state i the greatest common divisor for the length of all cycles starting in i in the underlying state graph is 1.*

We also call a MarkovChain with these two properties *ergodic*.

▷ Observation 2.36. *A irreducible MarkovChain is aperiodic if the underlying state graph has a self-loop.*

▷ Lemma 2.37 ([17, 74]). *If a MarkovChain is irreducible and aperiodic, a unique stationary distribution π exists and every random walk converges to $\pi = \pi^{(\infty)}$.*

If the MarkovChain is not irreducible, there are some states we cannot reach from other states, thus there might be more than one stationary distribution if the underlying state graph consists of multiple SCCs. If the MarkovChain is not aperiodic, then every distribution alternates between different regimes of the MarkovChain and every random walk diverges. There are some MarkovChains that do not have both these properties but have a unique stationary distribution to which every random walk converges[8].

We introduce one other important property that further simplifies the computation of a stationary distribution π.

▷ Definition 2.38 (Symmetric). *A MarkovChain is symmetric if $P_{i,j} = P_{j,i}$ for every $i, j \in S$.*

▷ Lemma 2.39 (Th. 7.10 from [80]). *A MarkovChain that is irreducible, aperiodic, and symmetric converges to the uniform distribution over $\mathcal{S}$.*

While every MarkovChain satisfying these properties eventually converges to its stationary distribution π, for different MarkovChains, it might take longer to get closer to π than others. Thus, we also want to have a measure of how fast a MarkovChain actually converges, i.e. how many steps it takes to get arbitrarily close to π.

▷ Definition 2.40 (MixingTime). *For $\varepsilon > 0$, the mixing time τ_ε of an ergodic MarkovChain with stationary distribution π is defined as*

$$\tau_\varepsilon = \min\left\{\tau \geq 0 \colon \max_{x \in \mathcal{S}} \left[\sum_{y \in \mathcal{S}} |\pi(y) - \sigma_{\tau,x}(y)| \leq \varepsilon\right]\right\}$$

where $\sigma_{\tau,x}$ is the distribution of the MarkovChain after τ steps starting in distribution x.

We say a MarkovChain is rapidly mixing if τ_ε is bounded by a polynomial in

$$\frac{\log|\mathcal{S}|}{\varepsilon}.$$

▷ Example (sketch) 2.41 (τ_ε of Fig. 2.3). *For the MarkovChain in Fig. 2.3, we have*

$$\left\lceil \log_2 \frac{1}{\varepsilon} + \log_2 \frac{2}{3} \right\rceil \leq \tau_\varepsilon \leq \left\lceil \log_2 \frac{1}{\varepsilon} + 1 \right\rceil$$

[8] For example: consider the MarkovChain in Fig. 2.3 and add a new node 4 with edge $(4, 1)$ and transition probability 1. Then, the MarkovChain is no longer irreducible but $\pi = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0\right)$ is (still) the unique stationary distribution to which every random walk converges to.

Proof. Notice that $\sigma_{\tau,x} = P^\tau x$. We can prove by induction that we have

$$P^\tau = \frac{1}{2^k} \begin{cases} \begin{pmatrix} \Delta_k + 1 & \Delta_k + 1 & \Delta_k \\ \Delta_k & \Delta_k & \Delta_k + 2 \\ \Delta_k + 1 & \Delta_k + 1 & \Delta_k \end{pmatrix} & \text{if } k \text{ is odd} \\ \begin{pmatrix} \Delta_k & \Delta_k & \Delta_k + 1 \\ \Delta_k + 1 & \Delta_k + 1 & \Delta_k - 1 \\ \Delta_k & \Delta_k & \Delta_k + 1 \end{pmatrix} & \text{if } k \text{ is even} \end{cases}.$$

for $\Delta_k = \frac{2^k - 1 - (k \mod 2)}{3}$. Then, simply bound $|\pi - P^\tau x|$ from below and above to get the stated bounds. □

3 Related Work

3.1 Shortest Path Algorithms

Apart from the aforementioned shortest path algorithms Dijkstra, BellmanFord, and Johnson, there are a multitude of further algorithms used both in theory in practice. Most importantly, the study of parallel single shortest path algorithms, kicked off by the Δ-Stepping algorithm proposed by Meyer and Sanders [78], is to this day an ongoing field [22, 28, 36, 39, 71, 85].

Δ-Stepping works by creating buckets $B_0, \dots$ of width Δ such that B_i stores nodes with tentative distances $[i \cdot \Delta, (i+1) \cdot \Delta)$.[1] The algorithm then marks all nodes in the current bucket and relaxes its outgoing edges with weight less than Δ in parallel. This process repeats until the current bucket is empty before proceeding with the next bucket. Δ-Stepping generalizes both Dijkstra and BellmanFord as for $\Delta = \min_{e \in E} w(e)$, Δ-Stepping mimics Dijkstra and for $\Delta = \max_{u,v \in V} d(u, v)$ it mimics BellmanFord [78]. While parallel shortest path algorithms are very relevant in practice, they are not of interest to this thesis and thus are not considered in the following.

In the regime of non-negative edge weights, Dijkstra is optimal in the sequential case and thus almost always the chosen algorithm. For negative edge weights, however, BellmanFord still is the most used shortest path algorithm in practice despite a subpar running time. Since the introduction of BellmanFord in the 50s however, many theoretical improvements were achieved, starting with the scaling technique in the 80s and 90s [52, 53, 59] leading to a runtime of $\mathcal{O}(m\sqrt{n} \log W)$ where $W \geq 2$ is the minimum integer such that $w(e) \geq -W$ for all $e \in E$. Further improvements were achieved by focusing on specific graph classes such as planar graphs [44, 67, 68, 73] and dense graphs with small weights [94].

An alternative approach is to model the SSSP problem as a minimum-cost flow problem by assigning each edge $e \in E$ an infinite capacity and cost $w(e)$. Then, when adding a new node t into the graph with zero weighted edges (v, t) for $v \in V$, every minimum-cost flow from some source s to t equals a shortest path tree $\textsc{Spt}(s)$ (assuming that there is no negative cycle). Hence, a multitude of advancements of the last decade for minimum-cost flow also translate to the negative weighted SSSP problem. Notably, a combination of convex optimization techniques and dynamic algorithms lead to runtimes $\tilde{\mathcal{O}}\left(m^{10/7}\right)$ [33], $\tilde{\mathcal{O}}\left(m^{4/3}\right)$ [9], and $\tilde{\mathcal{O}}\left(m + n^{3/2}\right)$ [99, 100] where $\tilde{\mathcal{O}}$ hides polylogarithmic factors. This also

[1] This definition directly implies that we require non-negative weights for Δ-Stepping.

resulted in a near linear runtime of $\mathcal{O}\left(m^{1+o(1)} \log W\right)$ [32].

The most recent breakthrough of Bernstein, Nanongkai, and Wulff-Nilsen [15] and follow up work of Bringmann, Cassis, and Fischer [26] achieved a running time of $\widetilde{\mathcal{O}}(m \log W)$ without a reduction to minimum-cost flow. Their algorithms work by computing a feasible potential function using a combination of the scaling technique and graph decompositions into strongly connected subgraphs with lower diameters. Although only used as a subroutine, they also propose a simple algorithm ElimNeg for computing a feasible potential function using a combination of Dijkstra and BellmanFord which we will briefly highlight in Appendix C. Apart from ElimNeg, we do not go into detail about their algorithms as they are highly complex and unpractical compared to algorithms such as BellmanFord even though they provide better asymptotic guarantees.

3.2 Uniform High Entropy Models

We are unaware of previous work towards high entropy models for generating negative edge weights. Thus, we instead review different high entropy models in the field of graph topology which is still an active field of research today [90].

Common high entropy models in the context of graph generators include $\mathcal{G}(n, p)$ [57], $\mathcal{G}(n, m)$ [42], and the Configuration Model [14, 23, 81, 84]. Due to their simplistic nature, efficient implementations often rely on basic sampling algorithms such as Bernoulli trials and sampling without replacement [12, 51, 86, 93].

We already introduced the $\mathcal{G}(n, p)$ model earlier in Section 2.3.1. The $\mathcal{G}(n, m)$ model draws a uniform graph with n nodes and m edges which can be rephrased as sampling without replacement which leads to very efficient implementations [12, 103].

The Configuration Model is a bit more involved and is parametrized by a degree sequence $(d_1, \ldots, d_n)$ with the goal of generating a graph with n nodes $v_1, \ldots, v_n$ such that node v_i has degree d_i.[2] The model works by placing each node v_i exactly d_i times ínto an urn. Then, until the urn is empty, two random nodes v_i, v_j are drawn uniformly without replacement, and an edge $\{v_i, v_j\}$ is emitted. We can implement this efficiently by modeling the urn as an array and reading it in random order [5]. This is akin to shuffling the array and reading it in order and thus emits very efficient implementations [89].

Note however that this might produce a non-simple graph, i.e. a graph with multi-edges or self-loops. A more complex model is the problem of sampling a uniform simple graph from all simple graphs with a prescribed degree sequence. Exactly sampling from this distribution can be achieved naively by drawing from the Configuration model repeatedly until a simple graph is drawn, i.e. using rejection sampling. While this method is linear in the number of edges of the graph, it is exponential in the maximum degree squared and thus only practical for graphs with small degrees [105].

An efficient alternative is a combination of the Configuration Model and the *switching* method [76, 77]. First, a uniform graph with the prescribed degree

[2] In this case, the graph is undirected and we have $\deg_{\text{in}}(v) = \deg_{\text{out}}(v)$ for every node, and every edge always is counted for both ends.

sequence and not too many self-loops or multi-edges is drawn using the Configuration Model and rejection sampling. Then, a series of *switching* steps that exchange endpoints of two (or more) selected edges, are applied until the graph is simple. This often involves additional rejection steps to counteract the bias that is introduced by the switchings. Efficient generators exist for sampling graphs with degrees up to the fourth root of the number of edges [8, 77], sampling regular graphs with slightly higher degrees [8, 54], as well as graphs with degree sequences that follow a power-law distribution with a sufficiently small power-law exponent [8, 55].

Another efficient approach is to approximate a uniform sampler using the Markov-Chain-Monte-Carlo method [29, 58, 91, 97]. For example, the Edge-Switching method starts with a deterministic simple graph with the prescribed degree sequence and then repeatedly selects two edges uniformly at random and exchanges their endpoints (similar to the *switching* method) if this does not introduce a self-loop or a multi-edge. While there exist many practical implementations of such samplers [4, 5, 16, 31, 62], there tends to be a large gap between rigorous bounds on their mixing times compared to practically sufficient choices [29, 43, 58, 62].

Nonetheless, such MCMC processes are often very simple and thus allow for easily adding additional constraints (e.g. [7, 96, 102]). For example, [102] extend Edge-Switching with graph connectivity tests to draw a uniform random graph from the set of simple connected graphs with a prescribed degree sequence.

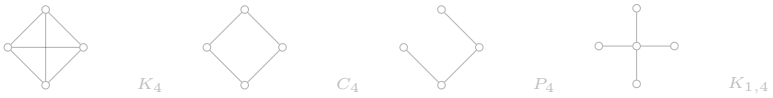
3.3 Parallel MarkovChains

As we are again unaware of previous work towards parallel MarkovChains for generating negative edge weights, we instead briefly discuss common techniques for parallelizing the previous example.

As the basic step of the Edge-Switching method is very simple, a direct parallelization of each step of the MarkovChain provides no real benefit. Instead, simple MarkovChains are commonly not parallelized per step but rather over a range of steps, i.e. trying to execute multiple steps of the MarkovChain in parallel instead of one after another [4, 31]. In the case of the Edge-Switching method, common parallelization tries to perform multiple switches of edges at once [4, 16]. However, as two different switches can affect the same node, complications can arise when trying to perform them in parallel. For example, two switches could independent of each other try to create the same edge. If both switches were to be accepted, this would directly insert a Multi-Edge into the graph which is forbidden in the underlying Prescribed-Degree-Sequence model.

Thus, parallel methods often involve additional checks to prevent such *dependencies* from creating faulty states. They also often contain slight adjustments to the original MarkovChain to make prediction and handling of *dependencies* easier. This however can lead to alterations in the resulting target distribution as in [16] where authors conducted an empirical error analysis to bound this alteration instead. Other results however maintain the same target distribution even in the parallel case with alterations in the MarkovChain [4, 31].

In [4], Allendorf et al. present a parallel Edge-Switching method, dubbed the Global Edge-Switching MarkovChain (G-ES-MC) based on [30] where the sequence of edges that are to be switched is drawn as a permutation π of all edges and a switch-length $\ell \leq \lfloor m/2 \rfloor$. Then, the first ℓ pairs of edges in π try to perform an edge switch. While this method does alter the original Edge-Switching MarkovChain by forcing ℓ consecutive edges to be distinct, the target distribution remains unchanged. They then resolve dependencies using a concurrent hash table to store whether dependencies for a specific switch have already been resolved and the switch can be performed (or rejected).



4 Uniform Edge Weight Sampling

In this chapter, we study the problem of sampling a consistent weight function w for a given graph G. We aim for a uniform model of all weight functions that have weights in a defined interval $\mathcal{W}$ and do not introduce negative cycles:

▷ Definition 4.1. *For a given graph $G = (V, E)$ and set of possible weights $\mathcal{W}$, the set of all consistent edge weights $\mathbb{S}_{G,\mathcal{W}}$ is defined as*

$$\mathbb{S}_{G,\mathcal{W}} = \{w \mid w\colon E \to \mathcal{W} \wedge (G, w) \text{ are consistent}\}.$$

Then, $\mathcal{S}_{G,\mathcal{W}}$ is the uniform distribution over $\mathbb{S}_{G,\mathcal{W}}$.

For the remainder of the chapter, we again fix G and $\mathcal{W}$ and write $\mathcal{S} = \mathcal{S}_{G,\mathcal{W}}$ and $\mathbb{S} = \mathbb{S}_{G,\mathcal{W}}$ as shorthands. Our goal then is to sample a uniform weight function $w \sim \mathcal{S}$.

▷ Observation 4.2. *If G is acyclic or $\mathcal{W} \subseteq \mathbb{R}^+$, then $\mathbb{S} = \mathcal{W}^m$ and every $w\colon E \to \mathcal{W}$ is consistent with G. If G contains at least one cycle and $\mathcal{W} \cap \mathbb{R}^+ = \emptyset$ (as well as $\mathcal{W} \neq \emptyset$), then $\mathbb{S} = \emptyset$.*

Observation 4.2 makes the problem trivial for certain parameters. Thus, we first restrict ourselves to $\mathcal{W}$ that contains negative and positive weights.[1] Second, we restrict ourselves to strongly connected G. This is justified by the following lemma.

▷ Lemma 4.3. *For graph G, let $V_1, \dots, V_s$ denote its SCCs and $G[V_i]$ the induced subgraph of V_i. Then,*

- *every edge $(u, v) \in E$ with $u \in V_i, v \in V_j$ and $i \neq j$ can have arbitrary weights.*
- *for $i \neq j$, edge weights in $G[V_i]$ are independent of edge weights in $G[V_j]$.*

Proof. By definition of SCCs, every node that lies on the same cycle is part of the same SCC. Thus, edge weights of edges $(u_1, v_1), (u_2, v_2)$ are only dependent on each other if $u_1, u_2, v_1, v_2 \in V_i$ for some $i \in [s]$. The first claim directly follows as edges in between two different SCCs are on no cycle and can hence not be part of a negative cycle. □

Lemma 4.3 implies that to sample from $\mathcal{S}_{G,\mathcal{W}}$, it suffices to split G into its SCCs $V_1, \dots, V_s$ in linear time [98], sample from $\mathcal{S}_{G[V_i],\mathcal{W}}$ for every $i \in [s]$ and assign uniform weights from $\mathcal{W}$ to every edge not present in any SCC. Thus, any efficient sampling algorithm on strongly connected components directly translates into an efficient sampling algorithm for general graphs.

[1] Typically, we have $\mathcal{W} = [a..b]$ for some $a < 0 < b$.

Figure 4.1: A double linked path of length 4.

4.1 Hardness

Before discussing our algorithm, we first show that the problem is not trivial.

Rejection Sampling. The simplest approach might be to just draw a uniform weight function from $\mathcal{W}^m$ and reject if w contains a negative cycle. While we can check for a negative cycle using BellmanFord in polynomial time $\mathcal{O}(nm)$, the probability of drawing a consistent w might be exponentially small. For instance, consider a double linked path $D_k = (V_k, E_k)$, defined by

$$V_k = \{v_i \mid i \in [k]\}, \quad E_k = \{(v_i, v_j) \mid i, j \in [k],\ |i - j| = 1\}.$$

An example with $k = 4$ can be seen in Fig. 4.1.

▷ Observation 4.4. *(D_k, w) are consistent iff for every $i \in [k-1]$, we have*

$$w(v_i, v_{i+1}) + w(v_{i+1}, v_i) \geq 0.$$

Now consider $\mathcal{W} = \{-1, 0, 1\}$. Then, every pair of edges $(v_i, v_{i+1}), (v_{i+1}, v_i)$ has $|\mathcal{W}|^2 = 9$ possible weight assignments from which 6 do not introduce a negative cycle, i.e. sum to a non-negative weight.

Hence, if we draw w uniformly from $\mathcal{W}^{2(k-1)}$, the probability of having no negative cycle is given by

$$\left(\frac{6}{9}\right)^{k-1}.$$

Therefore, following a geometric distribution, we expect to draw $\Theta\left(1.5^k\right)$ times in order to generate a consistent w which is far from efficient.

$\mathcal{NP}$-hard Distributions. While in the previous example, it is quite easy to compute the distribution of $\mathcal{S}$, in general, properties of weight functions in $\mathcal{S}$ can be non-trivial. A natural example is the probability of having a certain number of negative edges:

▷ Definition 4.5 (k-NegEdges). *In the k-NegEdges, we have to decide if for graph G and weights $\mathcal{W}$, there exists $w \in \mathbb{S}_{G,\mathcal{W}}$ such that w has k negative edges.*

Unfortunately, we can show that this problem is in fact $\mathcal{NP}$-hard by a reduction from the DirectedFeedbackArcSet problem [66].

▷ Definition 4.6 (DFAS [66]). *The $\mathcal{NP}$-hard DirectedFeedbackArcSet problem (DFAS) asks whether for graph G it suffices to remove k edges from G to make G acyclic.*

▷ Theorem 4.7. *k-NegEdges is $\mathcal{NP}$-hard.*

Algorithm 4: Sampler: an approximate $\mathcal{S}_{G,\mathcal{W}}$-Sampler [56]

Input: graph $G = (V, E)$, possible weights $\mathcal{W}$, number of steps τ
Output: consistent weight function $w\colon E \to \mathcal{W}$

```
w ← arbitrary consistent weight function
repeat τ times
    e ∼ 𝒰(E)
    c ∼ 𝒰(𝒲)
    w' ← w_{e←c}
    if (G, w') are consistent then
        w ← w'
return w
```

Proof. We show that

$$(G, k) \in \text{DFAS} \quad \Longleftrightarrow \quad (G, \mathcal{W}, m - k) \in k\text{-NegEdges}.$$

with $\mathcal{W}$ such that there are $a, b \in \mathcal{W}$ with $a < 0 < b$ and $b \geq (\mathcal{C} - 1) \cdot (-a)$ where $\mathcal{C}$ is the length of the longest simple cycle in G. We do this by interpreting *deleted* edges in DFAS as edges with positive weight in k-NegEdges.

$\Longleftarrow$: Observe that if all edge weights are negative, G must be acyclic (since otherwise trivially every cycle is negative). Thus, if there are $m - k$ negative edges in G, removing the k positive edges makes G acyclic, proving

$$(G, \mathcal{W}, m - k) \in k\text{-NegEdges} \quad \Longrightarrow \quad (G, k) \in \text{DFAS}.$$

$\Longrightarrow$: For the other direction, assume there exists a subset of k edges S such that their removal makes G acyclic. Observe that if G is acyclic, all edges can be set to a. Otherwise, we have to introduce at least one positive edge for each cycle, i.e. the edges in S. Thus, assign b to every edge in S and a to every other edge in $E \setminus S$. Hence, the weight of any cycle C in G is at least

$$b + (|C| - 1) \cdot a \;\geq\; b + (\mathcal{C} - 1) \cdot a \;\geq\; (\mathcal{C} - 1) \cdot (-a) + (\mathcal{C} - 1) \cdot a \;=\; 0$$

by definition of a and b. Thus, the weight function is consistent, completing the proof. □

4.2 A uniform Sampler

The previous example and reduction suggest that, in general, uniform sampling from $\mathcal{S}$ has to be a bit more involved. Thus, instead of a direct method, we propose a process that can sample approximately uniform from $\mathcal{S}$.

The process can be seen in Algorithm 4. We denote a new weight function that differs from w only in one edge weight $w(e) = c$ by $w_{e \leftarrow c}$. The algorithm starts with an arbitrary (deterministic) consistent weight function w and perturbs w for a specified number of steps τ. Following Obs. 4.2, canonical choices for an arbitrary consistent weight function are

- $w_{zero} = (0, \dots, 0)$ (if $0 \in \mathcal{W}$),
- $w_{max} = (\max(\mathcal{W}), \dots, \max(\mathcal{W}))$,
- $w_{unif} = \mathcal{U}(\mathcal{W}_+^m)$ where $\mathcal{W}_+$ are the non-negative weights in $\mathcal{W}$.

We perturb w by drawing a uniform edge $e \sim \mathcal{U}(E)$ and a uniform weight $w \sim \mathcal{U}(\mathcal{W})$ and updating w if no negative cycles are created by doing so. To show that this method in fact approximates a uniform distribution over $\mathbb{S}$, we interpret the algorithm as a MarkovChain over state space $\mathbb{S}$.

Clearly, since we maintain the invariant in Line 6 that (G, w) are consistent, we have $w \in \mathbb{S}$ at any point in time. Furthermore, a weight update in Line 7 can be interpreted as a state transition.

▷ Definition 4.8 (Neighborhood). *For $w \in \mathbb{S}$, its neighborhood $N[w]$ consists of all weight functions that differ in at most one edge weight:*

$$N[w] = \{w' \mid \exists e \in E \; \exists c \in \mathcal{W} \colon w' = w_{e \leftarrow c} \in \mathbb{S}\}.$$

Thus, we perturb w by choosing a random neighbor in $N[w]$ and transitioning to it. Also, every neighbor is equally likely to be the next one:

▷ Observation 4.9. *Let $w_1, w_2 \in \mathbb{S}$ with $w_2 \in N[w_1]$ and $w_1 \neq w_2$. Then, we transition from w_1 to w_2 with probability*

$$\frac{1}{|E| \cdot |\mathcal{W}|}.$$

Note that we have $w \in N[w]$ and that we can also draw a non-consistent w' in Lines 3 to 5 which leads to no change in this specific round. Thus, the probability of staying in state w is at least the probability of transitioning to a specific neighbor $w' \in N[w], w' \neq w$.

▷ Observation 4.10. *Let $w_1, w_2 \in \mathbb{S}$. Then,*

$$w_1 \in N[w_2] \iff w_2 \in N[w_1].$$

Thus, if there is a path from some $w_i \in \mathbb{S}$ to some $w_j \in \mathbb{S}$, there also is one from w_j to w_i (the same one in reverse).

▷ Observation 4.11. *Let $w \in \mathbb{S}$ and $w' = w_{e \leftarrow c}$ with $c \geq w(e)$. Then, $w' \in N[w]$.*

We can now prove that Algorithm 4 in fact approximates $\mathcal{S}$.

▷ Theorem 4.12 ([56]). *For $\tau \to \infty$, w in Algorithm 4 converges to $\mathcal{S}$.*

Proof. The theorem follows if the underlying MarkovChain is irreducible, aperiodic, and symmetric using Lemma 2.39. Aperiodicity follows from the previous observation that $w \in N[w]$ and Obs. 2.36. Symmetry follows from Obs. 4.9 and 4.10. For irreducibility, consider any two states w and w'. Due to Obs. 4.11, we know that both w and w' can reach w_{max} and following Obs. 4.10, there exist paths from w to w' and back. Hence, every state can reach every other state concluding the proof. □

We prove a stronger version of this theorem in Appendix A.1 for the case in which $\mathcal{W}$ consists of sets of possible weights that are uncountable. While we provide empirical evidence in Chapter 6 that this MarkovChain is rapidly mixing, rigorous theoretical bounds are still an open question and thus not part of this thesis. However, we provide a proof in Appendix A that for the simple n-cycle graph, the MarkovChain is rapidly mixing for integer weights $\mathcal{W} = [a, b]$.

4.3 Fast Consistency Checks

Drawing a random edge $e \sim \mathcal{U}(E)$ and weight $c \sim \mathcal{U}(\mathcal{W})$ in Algorithm 4 can be trivially done in expected constant time [**?**]. The only challenging part is the consistency check in Line 6. Thus, we have to determine if w' induces a negative cycle. Fortunately, since we maintain the invariant that w is consistent with G throughout the algorithm, we only need to check whether the weight change of $w(e)$ introduces a new negative cycle.

Following Obs. 4.11, a weight-increase in edge e can not create a negative cycle and thus can be immediately accepted. If we decrease the weight of e, we need to check if e is part of a negative cycle. Since (G, w) are consistent, we know that the length $d_{G,w}(u, v) > -\infty$ of a shortest $u \to v$ path is finite for every pair of vertices $u, v \in V$.

▷ Lemma 4.13. *If we change the weight of edge $e = (u, v)$ to $w'(u, v)$, e is part of a negative cycle iff*

$$d_{G,w}(v, u) + w'(u, v) < 0.$$

Proof. If (u, v) is part of a negative cycle with respect to w', there must exist a simple cycle C with $w'(C) < 0$ consisting of a simple $v \to u$ path P and edge (u, v). Since only $w'(u, v)$ changed and (G, w) were consistent, i.e. every shortest $v \to u$ path is simple and does not traverse (u, v) with respect to w, we have $w'(C) = w'(u, v) + w(P) = w'(u, v) + d_{G,w}(v, u) < 0$. □

Thus, to check for consistency of the weight change, we need to compute the length of a shortest $v \to u$ path. Using BellmanFord, we can do this in time $\mathcal{O}(nm)$[2].

This however is slow for bigger graphs. Utilizing Dijkstra instead of BellmanFord would significantly improve the performance of this algorithm. Since, in general, G might already contain negative weights, it is not possible to directly substitute BellmanFord with Dijkstra. Instead, we use potential functions as in Johnson introduced in Section 2.2.3.

Following Lemmas 2.20 and 2.21, we know that there exists a feasible ϕ with respect to w. Thus, shortest $v \to u$ paths with respect to w_ϕ found with Dijkstra are also shortest $v \to u$ paths with respect to w found with BellmanFord (although we have to account for the additional $\phi(u) - \phi(v)$ term). Since we start with an arbitrary consistent weight function in Algorithm 4, we can either use the technique of Johnson or start with $\phi = 0$ if all weights are non-negative.[3]

[2] Although no asymptotic improvement, since (G, w) are consistent, we need no further amortized checks for negative cycles in the SPFA heuristic as we do not need to consider outgoing edges from u.

[3] Alternative algorithms for the first option are studied in detail in Appendix C.

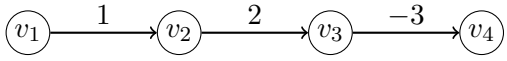


Figure 4.2: A graph with no clear partial shortest path tree for source node v_1 and $\Delta = 2$. v_4 should be part of $\text{SPT}_2(v_1)$ since $d(v_1, v_4) = 0$ — v_3 however not since $d(v_1, v_3) = 3 > 2$ thus making v_4 unreachable from v_1.

Hence, in the following, we discuss the problem of maintaining a feasible potential function ϕ under dynamic edge weight updates.

For the rest of the section, w thus refers to the *current* consistent weight function and ϕ to its correspondent feasible potential function. We denote by $w' = w_{(u,v)\leftarrow c}$ the new weight function which differs from w only in edge (u, v). ϕ' refers to the updated version of ϕ that we get when updating w to w'. Note that we still have to define ϕ' and prove its feasibility (if w' does not induce a negative cycle).

As increasing the weight of an edge also increases its potential weight, we do not need to update ϕ under weight increases. Decreasing $w(u, v)$ however might *break* the edge $(w_\phi(u, v) < 0)$ and necessitate an update of ϕ by increasing the gradient $\phi(v) - \phi(u)$. We have three solutions which we will discuss in order:

(1) Increasing $\phi(v)$.

(2) Decreasing $\phi(u)$.

(3) Both (1) and (2).

▷ Definition 4.14 ($\mathcal{B}_{uv}$). *The broken value $\mathcal{B}_{uv}$ of edge (u, v) is defined as*

$$\mathcal{B}_{uv} \coloneqq -w'_\phi(u, v).$$

We can rephrase the previous observation to see that we only require an update of ϕ if $\mathcal{B}_{uv} > 0$. Otherwise, even under weight decreases, ϕ remains feasible and w' thus consistent with G.

With this definition, we can also rephrase Lemma 4.13 in terms of w_ϕ and $\mathcal{B}_{uv}$ (instead of w and $w(u, v)$). Dijkstra's invariant that every node that we visit increases in distance from the source node motivates the following definition and lemma.

▷ Definition 4.15 (Partial Shortest Path Tree). *For source node u, positive weights w and max distance Δ, a partial shortest path tree T from u, denoted by $\text{SPT}_\Delta(u)$ is a subtree of G such that*

$$d_{T,w}(u, v) = \begin{cases} d_{G,w}(u, v) & \text{if } d_{G,w}(u, v) \leq \Delta, \\ \infty & \text{otherwise.} \end{cases}$$

Thus, a partial shortest path tree only consists of all nodes that are at most a distance of Δ away from u. We denote Dijkstra's variation that only considers nodes with distance at most Δ by Dijkstra_Δ.

Note that we require w to be non-negative: otherwise a partial shortest path tree might be infeasible, i.e. might not be able to *reach* all desired nodes, as seen in Fig. 4.2.

▷ Lemma 4.16. *It suffices to compute $\textsc{Spt}_{\mathcal{B}_{uv}}(v)$ (instead of $\textsc{Spt}(v)$) with respect to w_ϕ to check whether w' introduces a negative weight cycle.*

Proof. Following Lemma 4.13, w' introduces a negative cycle if $d_{G,w_\phi}(v,u) < \mathcal{B}_{uv}$. Now since edge weights are positive and path lengths thus non-decreasing, we only need to consider nodes with distance less than $\mathcal{B}_{uv}$. Since ϕ is feasible for w, w_ϕ is non-negative. □

The partial shortest path tree can be computed by $\textsc{Dijkstra}_{\mathcal{B}_{uv}}$ and can not only determine the consistency of w' but can also be used to maintain feasibility of ϕ. Building on top of Lemma 2.21, partial subtree potential updates are feasible in the tree itself.

▷ Lemma 4.17. *Let T be some partial shortest path tree $\textsc{Spt}_\Delta$ in G with respect to w_ϕ for some feasible potential function ϕ. Then, the potential function ϕ' with $\phi'(u) = \phi(u) - T[u]$ for $u \in T$ ($\phi'(u) = \phi(u)$ otherwise) only breaks incoming edges (x,y) onto T by a value of $T[y]$ — every other edge is not broken. Outgoing edges of T have a potential weight greater than Δ.*

Proof. Due to Lemma 2.21, edges inside T are not broken and do not need to be considered. This is the same with edges not incident to T: both potentials remain unaffected and thus non-negative.

Now consider an incoming edge $(x,y) \in (V \setminus T) \times T$. Since ϕ was feasible before, i.e. $w_\phi(x,y) \geq 0$, we now have $w_{\phi'}(x,y) \geq -T[y]$ since we only decreased $\phi(y)$. For an outgoing edge $(x,y) \in T \times (V \setminus T)$, notice that by definition of T, we have $T[x] + w_\phi(x,y) > \Delta$ and thus after decreasing $\phi(x)$ by $T[x]$, for the new potential weight of (x,y), we have

$$w_{\phi'}(x,y) \geq w_\phi(x,y) - T[x] > \Delta.$$

□

▷ Observation 4.18. *Let ϕ be a feasible potential function with respect to w, and $S \subseteq V$ any subset of nodes. Then, the potential function $\phi'(u) = \phi(u) + \Delta$ for $u \in S$ (and $\phi'(u) = \phi(u)$ otherwise) only breaks outgoing edges (from S) by at most Δ and increases the potential weight of incoming edges by Δ.*

A combination of Lemma 4.17 and Obs. 4.18 leads us to the main theorem allowing partial potential updates.

▷ Theorem 4.19. *Let T be some partial shortest path tree $\textsc{Spt}_\Delta$ in G with respect to w_ϕ for some feasible potential function ϕ. Then, the potential function ϕ' with $\phi'(u) = \phi(u) + \Delta - T[u]$ for $u \in T$ ($\phi'(u) = \phi(u)$ otherwise) is feasible.*

Proof. Following the proof of Lemma 4.17 and Obs. 4.18, we only need to consider outgoing and incoming edges onto T. For outgoing edges (x,y) note that in Lemma 4.17 we showed that decreasing $\phi(x)$ by $T[x]$ increases the potential weight to something greater than Δ. Then, by Obs. 4.18, this potential weight is

decreased by Δ when increasing $\phi(x)$ by Δ and still $w_{\phi'}(x,y) \geq 0$. For incoming edges (x,y), the same arguments apply inversely since $T[y] \leq \Delta$. Thus, ϕ' is feasible. □

Notice that since for $T = \textsc{Spt}_\Delta(x)$, we have $T[x] = 0$, the prior method increases $\phi(x)$ by Δ. This leads to the following corollary which is the main result of this section.

▷ Corollary 4.20. *If decreasing the weight of (u,v) does not introduce a negative cycle, ϕ remains feasible by applying Theorem 4.19 with $\textsc{Spt}_{\mathcal{B}_{uv}}(v)$.*

Following Lemma 2.21, edges inside $\textsc{Spt}_{\mathcal{B}_{uv}}(v)$ now have a potential weight of 0 after the update. Thus, if we had used some value $\Delta > \mathcal{B}_{uv}$ instead of $\mathcal{B}_{uv}$, $\textsc{Spt}_\Delta(v)$ would only increase and size and thus would necessitate more potential updates. And since we require the increase of $\phi(v)$ to be at least $\mathcal{B}_{uv}$ to maintain feasibility in option (1), $\textsc{Spt}_{\mathcal{B}_{uv}}(v)$ is the smallest partial shortest path tree that allows Theorem 4.19 to maintain feasibility. Hence, this method is indeed optimal if we want to minimize the number of potential updates to keep ϕ feasible (and do not want to change $\phi(u)$).

Using Corollary 4.20, we can finally formalize our update procedure for an edge weight change. The optimized version of the algorithm is depicted in Algorithm 5.

Algorithm 5: SamplerPot: an approximate $\mathcal{S}_{G,\mathcal{W}}$-Sampler

Input: graph $G = (V,E)$, possible weights $\mathcal{W}$, number of steps τ
Output: consistent weight function $w\colon E \to \mathcal{W}$

```
w ← arbitrary consistent weight function
φ ← feasible potential function with respect to w
repeat τ times
    (u, v) ~ U(E)
    c ~ U(W)
    w' ← w_{(u,v)←c}
    if B_uv ≤ 0 then
        w ← w'
    else
        T ← Dijkstra_{B_uv}(G, w_φ, v)
        if u ∉ T then
            w ← w'
            φ(x) ← φ(x) + B_uv − T[x]   ∀x ∈ T
return w
```

This concludes solution (1). The runtime of Algorithm 5 follows with Lemma 2.15.

▷ Corollary 4.21. *The approximate uniform sampler can be implemented in time*

$$\mathcal{T}_{\textsc{SamplerPot}} = \mathcal{O}(\tau \cdot (m + n \log n)) .$$

We will later see in Chapter 6 that this worst-case bound is greatly exaggerated in practice and a single $\textsc{Dijkstra}_{\mathcal{B}_{uv}}$ round is much faster than $\mathcal{T}_{\textsc{Dijkstra}}$ due to the additional pruning of the shortest path tree.

Reversing the Direction. To discuss solution (2), consider the inverted graph of G, i.e. the graph we get by reversing every edge (x, y) to (y, x).

▷ Definition 4.22 (Inverted Graph). *For graph $G = (V, E)$, the inverted graph is defined as $G^R = (V, E^R)$ with*

$$E^R := \{(v, u) \mid (u, v) \in E\}.$$

Note that for weight function w (for G), we assign each reversed edge its original edge weight. Then, if ϕ is a potential function for G, the gradient is now inverted for G^R, i.e. $w_\phi(u, v) = w(u, v) + \phi(u) - \phi(v)$ for $(u, v) \in E^R$ since sources and targets are swapped. Therefore, $w_\phi(u, v) = w^R_\phi(v, u)$ and only the interpretation changes for G^R. Thus, every lemma and theorem we proved earlier for potential functions also apply to G^R by including an additional factor of (-1) to every potential change. For a completeness, we restate (the arguably most important) Lemma 2.21:

▷ Lemma 4.23 (Adaptation of Lemma 2.21). *Let T be some shortest path tree in G^R with respect to w^R_ϕ for some potential function ϕ that contains all nodes. Then, the potential function ϕ' with $\phi'(u) = \phi(u) + T[u]$ for $u \in V$ is feasible for G and w.*

Proof. First note that if T exists, there are no negative cycles in G^R and thus also none in G. By definition of a shortest path tree and Lemma 2.20, we have

$$T[u] \leq T[v] + w^R_\phi(v, u) \quad \iff \quad w^R_\phi(v, u) + T[v] - T[u] \geq 0$$

for every edge $(u, v) \in E$. Thus, for $w_{\phi'}$ it holds that

$$\begin{aligned} w_{\phi'}(u, v) &= w(u, v) + \phi'(v) - \phi'(u) \\ &= w(u, v) + (\phi(v) + T[v]) - (\phi(u) + T[u]) \\ &= w(u, v) + \phi(v) - \phi(u) + T[v] - T[u] \\ &= w_\phi(u, v) + T[v] - T[u] \\ &= w^R_\phi(v, u) + T[v] - T[u] \\ &\geq 0 \end{aligned}$$

and ϕ' is feasible. □

Using similar conversion techniques, every following lemma and theorem also follows including Theorem 4.19 and Corollary 4.20 which we also restate for completeness.

▷ Theorem 4.24 (Adaptation of Theorem 4.19). *Let T be some partial shortest path tree $\textsc{Spt}_\Delta$ in G^R with respect to w^R_ϕ for some feasible potential function ϕ. Then, the potential function ϕ' with $\phi'(u) = \phi(u) - \Delta + T[u]$ for $u \in T$ ($\phi'(u) = \phi(u)$ otherwise) is feasible.*

▷ Corollary 4.25 (Adaptation of Corollary 4.20). *When decreasing $\phi(x)$ by Δ, ϕ remains feasible by applying Theorem 4.24 with $\textsc{Spt}_\Delta(x)$.*

We thus can alter Algorithm 5 by running $\textsc{Dijkstra}_{\mathcal{B}_{uv}}(G^R, w^R_\phi, u)$ in Line 10 instead and "increasing" potentials in Line 13 by $T[x] - \mathcal{B}_{uv}$ instead. The runtime of this version however provides no asymptotic (or practical) benefits and thus also falls under Corollary 4.21.

Combining both Directions. Using Corollaries 4.20 and 4.25 we can finally tackle solution (3): decreasing $\phi(u)$ and increasing $\phi(v)$. The application is straightforward: we compute a partial shortest path tree in G from v for some max distance Δ and apply Theorem 4.19. For the other side, we compute a partial shortest path tree in G^R from u for max distance $\mathcal{B}_{uv} - \Delta$ and apply Theorem 4.24. Then, by Corollaries 4.20 and 4.25, ϕ' is feasible again if both trees do not overlap. This however still leaves two problems open:

(i) What if w' induces a negative cycle?

(ii) What value should be picked for Δ?

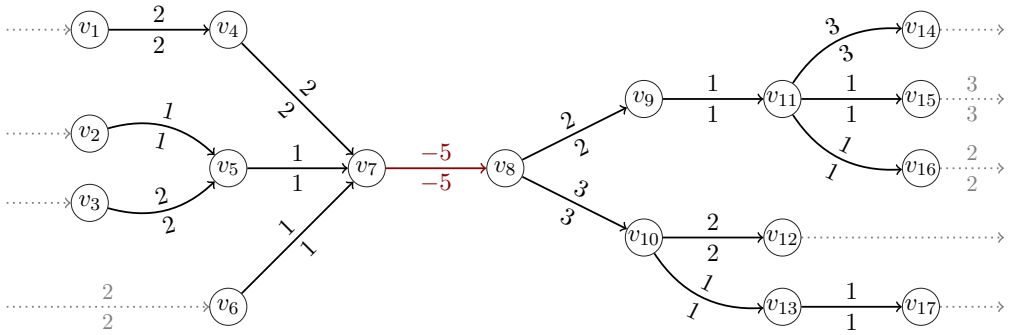
Our solution is a bidirectional search: instead of running both searches (Dijkstra instances) one after the other, we run them in parallel by alternating steps between them. The idea is summarized in Algorithm 6. We start with a source node v and target node u and a positive weight function w_ϕ for graph G as well as a max distance Δ (in our case we will set $\Delta = \mathcal{B}_{uv}$). The algorithm alternates between a forward Dijkstra search from v and a backward Dijkstra search from u. If at any point, we find a $v \to u$ path with distance less than Δ (Lines 15 and 30), we stop the algorithm and return ShorterPathFound. For $\Delta = \mathcal{B}_{uv}$, this indicates a negative cycle by Lemma 4.13. We dub the altered sampling algorithm using BiDijkstra as BiSamplerPot.

If no such path is ever found, at some point, we must have $\Delta_f + \Delta_b \geq \Delta$[4] and have computed partial shortest path trees $\text{Spt}_{\Delta_f}(v)$ in G and $\text{Spt}_{\Delta_b}(u)$ in G^R that do not overlap. Then, we can simply apply Theorems 4.19 and 4.24 to make ϕ' feasible again and accept the weight change.

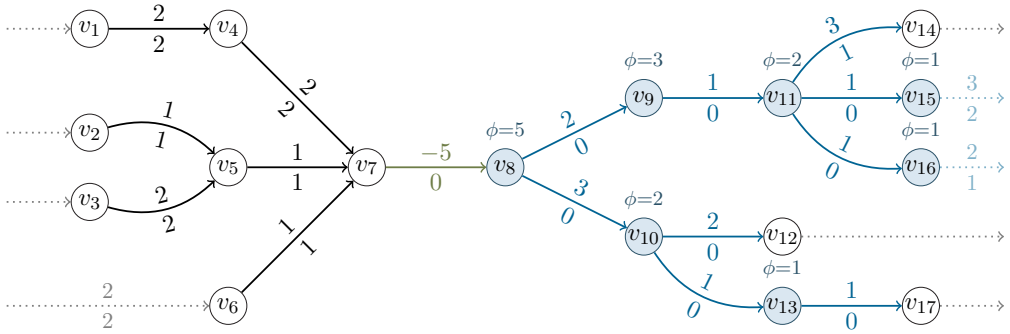
It is obvious that this approach does not incur a worse asymptotic runtime than solutions (1) and (2) since we run two Dijkstra instances instead of one. In fact, a bidirectional search heavily accelerates node-to-node shortest path algorithms in practice (and theory) by reducing the search space drastically [18, 21, 25]. This also translates to potential updates which can possibly be dramatically reduced by using a bidirectional search as illustrated in Fig. 4.3.

We will study more concrete comparisons of all three approaches (Sampler, SamplerPot, and BiSamplerPot) in detail in Chapter 6.

[4] If the graph is strongly connected, i.e. a $v \to u$ path exists which is an assumption for the sampling algorithm.



Solution (1) (as with Dijkstra): only increase $\phi(v_8) + 5$. This causes many cascading updates:



Solution (3) (as with BiDijkstra): split update to $\phi(v_7) - 2$ and $\phi(v_8) + 3$. This causes fewer cascading updates:

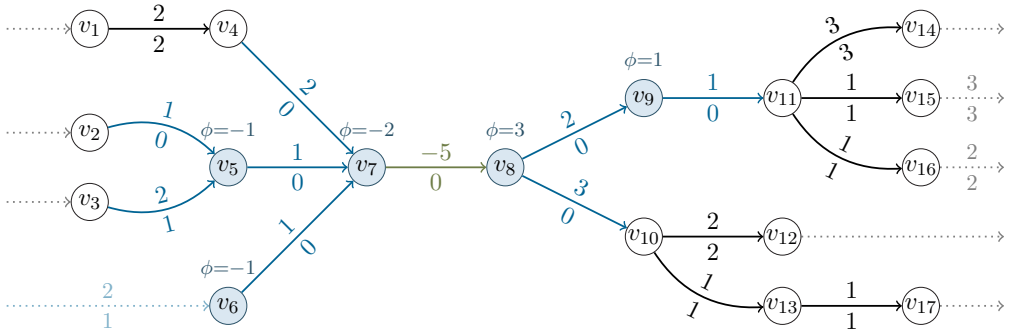


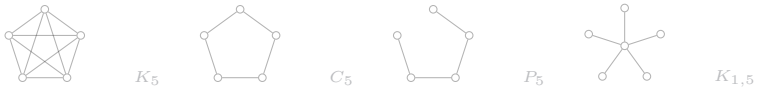
Figure 4.3: Unidirectional (1) and bidirectional (3) potential updates on a graph. The number above an edge (x, y) is $w(x, y)$ and below $w_\phi(x, y) = w(x, y) + \phi(y) - \phi(x)$ where originally $\phi = 0$ (as in the top panel). In the top panel, update $w(v_7, v_8) \leftarrow -5$ broke edge (v_7, v_8), i.e. we need to increase the gradient $\phi(v_8) - \phi(v_7)$ by at least $\mathcal{B}_{v_7 v_8} = 5$ to make ϕ feasible again. Blue nodes in (1) and (3) mark nodes for which we had to update ϕ. Blue edges are traversed by Dijkstra and BiDijkstra respectively to compute potential updates.

Algorithm 6: BiDijkstra: a bidirectional Dijkstra search.

Input: graph $G = (V, E)$, weights $w_\phi \colon E \to \mathcal{W},\ \mathcal{W} \subseteq \mathbb{R}^+$, source node v, target node u, max distance Δ

Output: $\text{SPT}_{\Delta_f}(v)$ and $\text{SPT}_{\Delta_b}(u)$ with $\Delta_f + \Delta_b = \Delta$ or ShorterPathFound.

```
1  for x ∈ V do
2      D_f[x], D_b[x] ← ∞
3  D_f[v], D_b[u] ← 0
4  Q_f, Q_b ← Min-PriorityQueues
5  add v to Q_f and u to Q_b
6  Δ_f, Δ_b ← 0
7  repeat
8      if Q_f is not empty then                          // Forward-Search
9          x ← node in Q_f with minimum D_f[x]
10         Δ_f ← D_f[x]
11         if Δ_f + Δ_b ≥ Δ then
12             Δ_b ← Δ − Δ_f
13             break
14         for (x, y) ∈ E do
15             if D_f[x] + w_φ(x, y) + D_b[y] < Δ then
16                 return ShorterPathFound
17             else if D_f[x] + w_φ(x, y) < min{D_f[y], Δ} then
18                 D_f[y] ← D_f[x] + w_φ(x, y)
19                 if y ∉ Q_f then add y to Q_f
20     else
21         Δ_f ← Δ − Δ_b
22         break
23     if Q_b is not empty then                          // Backward-Search
24         x ← node in Q_b with minimum D_b[x]
25         Δ_b ← D_b[x]
26         if Δ_f + Δ_b ≥ Δ then
27             Δ_f ← Δ − Δ_b
28             break
29         for (y, x) ∈ E do
30             if D_b[x] + w_φ(y, x) + D_f[y] < Δ then
31                 return ShorterPathFound
32             else if D_b[x] + w_φ(y, x) < min{D_b[y], Δ} then
33                 D_b[y] ← D_b[x] + w_φ(y, x)
34                 if y ∉ Q_b then add y to Q_b
35     else
36         Δ_b ← Δ − Δ_f
37         break
38 return D_f, D_b
```



5 Parallelizing Edge Weight Sampling

In this chapter, we want to discuss different methods of parallelization of the previously presented MarkovChains in Chapter 4.

5.1 The Problem of Dependencies

An intuitive parallelization is applying several weight changes at a time instead of one, i.e. we would combine up to ℓ MCMC steps into a single parallel one for some $\ell \in \mathbb{N}$. This however can lead to multiple complications caused by dependencies between single steps.

For exposition, fix a series of weight changes and let δ_1 and δ_2 be two proposed weight changes with $\delta_1 \coloneqq$ "$w(u,v) \leftarrow w_1''$ and $\delta_2 \coloneqq$ "$w(x,y) \leftarrow w_2''$ with $u, v, x, y \in V$ and $w_1, w_2 \in \mathcal{W}$ such that δ_1 happens before δ_2.

▷ Definition 5.1 (Hard-Dependency). *A weight change δ_2 is hard-dependent on a previous weight change δ_1, if the length of a shortest $y \to x$ path with respect to w changes when accepting δ_1.*

Figure 5.1 depicts two possible scenarios on the 4-cycle. The decision on whether to accept $w(u,v) \leftarrow -2$ depends on whether an earlier weight change $w(x,y) \leftarrow \{-1, 2\}$ is accepted because the length of a shortest $y \to x$ path changes when accepting $w(x,y) \leftarrow \{-1, 2\}$.

(i) Earlier Weight-Decrease (ii) Earlier Weight-Increase

Figure 5.1: Hard dependencies between earlier (blue) and later (red) rounds. If $w(x,y) \leftarrow -1$ gets accepted first in (i), $w(u,v) \leftarrow -2$ would be rejected. If not, $w(u,v) \leftarrow -2$ would be accepted. If $w(x,y) \leftarrow 2$ gets accepted first in (ii), $w(u,v) \leftarrow -2$ would be accepted. If not, $w(u,v) \leftarrow -2$ would be rejected.

Note that a hard-dependency does not necessarily imply that the decision on whether to accept δ_2 depends on the acceptance of δ_1. If for example in (i), edge (v,x) would have weight $w(v,x) = 2$ instead of 1, we could accept $w(u,v) \leftarrow -2$ regardless of the weight change of $w(x,y)$. Hard dependencies are difficult too handle in a simple parallelization as without additional computation we do not

know which weight changes are dependent on each other and thus can not be executed in parallel beforehand.

For maximum efficiency, it seems beneficial to include our potential-based shortest-path optimizations SamplerPot or BiSamplerPot in a parallel algorithm. However, since we need to maintain the invariant in SamplerPot (and BiSamplerPot) that every edge has a non-negative potential weight in order to utilize Dijkstra (or BiDijkstra), we not only change the weight of the affected edge but may also need to repair further potentials in the partial shortest path trees. Thus, after every accepted weight change $w(u, v) \leftarrow c$ in SamplerPot (and BiSamplerPot), the length of a shortest $v_1 \to v_2$ path might change even if it does not use the affected edge (u, v).

▷ Definition 5.2 (Soft-Dependency). *A weight change δ_2 is soft-dependent on a previous weight change δ_1 if only potentials but not edge-weights change along a shortest $y \to x$ path.*

We include the condition that the shortest path does not change with respect to w to better differentiate between hard and soft dependencies. Hence, δ_2 is softly-dependent on δ_1 if accepting δ_1 in SamplerPot (or BiSamplerPot) incurs potential changes that affect a shortest $y \to x$ path. Recall that by Lemma 2.20 evey potential but the first and last one along a path cancel out and thus this happens only if potentials of x or y have changed.

▷ Corollary 5.3. *A weight change δ_2 is softly dependent on a previous weight change δ_1 iff accepting δ_1 incurs potential updates of $\phi(x)$ or $\phi(y)$.*

5.2 Resolving Dependencies

In the following, we only focus on parallelization of SamplerPot (and BiSamplerPot). As weight increases are usually relatively easy to handle, we show a framework that can resolve hard and soft dependencies of consecutive weight decreases in parallel. Afterward, we briefly show why weight increases in this specific framework do not work.

Now let us consider a sequence of weight decreases. In order to resolve hard-dependencies, we have to differentiate between shortest paths that use edges for which it is yet not clear what their weight is (i.e. edges with pending weight changes of earlier rounds) and shortest paths that only use edges with fixed weights. One method to do that is to entirely forbid the use of affected edges of earlier rounds in the shortest path trees of later rounds.

▷ Definition 5.4 (Constrained Shortest Path Tree). *For a subset of edges $\hat{E} \subseteq E$ and source node u, a constrained shortest path tree $\textsc{Spt}(u, \hat{E})$ is a shortest path tree in the subgraph $\hat{G} = (V, E \setminus \hat{E})$ from source node u with respect to w.*

Notice that constraining a shortest path tree $\textsc{Spt}(u)$ by a subset of edges $\hat{E}$ only changes shortest $u \to v$ paths (the distances) iff every shortest $u \to v$ path uses at least one edge in $\hat{E}$.

▷ Observation 5.5. *For a subset of edges $\hat{E} \subseteq E$, a shortest $u \to v$ path P in $\hat{G} = (V, E \setminus \hat{E})$ is either also a shortest $u \to v$ path in G or every shortest $u \to v$ path in G traverses at least one edge in $\hat{E}$.*

As illustrated in Fig. 5.2, in the simple case of $\hat{E} = \{(x, y)\}$, Obs. 5.5 implies that a shortest $u \to v$ path either does not use edge (x, y) or consists of a shortest $u \to x$ path (which is part of $\textsc{Spt}(u, \hat{E})$), followed by edge (x, y) and then by a shortest $y \to v$ path (which again is part of $\textsc{Spt}(u, \hat{E})$).[1]

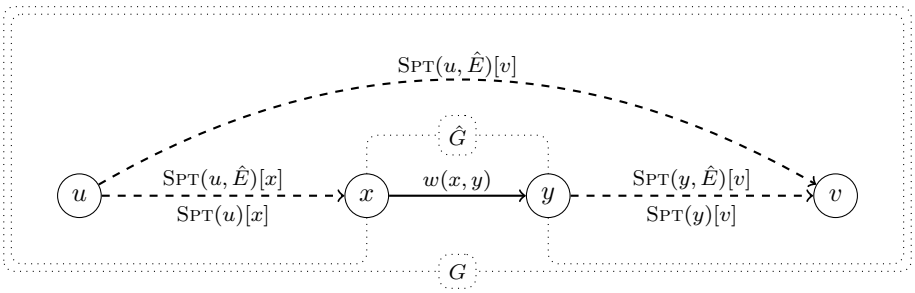


Figure 5.2: Inserting $\hat{E} = \{(x, y)\}$ back into $\hat{G} = (V, E \setminus \hat{E})$ to get $\textsc{Spt}(u)[v]$: Either the shortest $u \to v$ path is unaffected or it changes by now including edge (x, y), i.e. $\textsc{Spt}(u)[v] = \textsc{Spt}(u, \hat{E})[x] + w(x, y) + \textsc{Spt}(y, \hat{E})[v]$. If no negative cycles exist in G, we have $\textsc{Spt}(u, \hat{E})[x] = \textsc{Spt}(u)[x]$ and $\textsc{Spt}(y, \hat{E})[v] = \textsc{Spt}(y)[v]$.

Note that we restrict ourselves to SamplerPot and thus can assume that all (potential) weights are non-negative. Therefore, in this example we also have $\textsc{Spt}(u, \hat{E})[x] = \textsc{Spt}(u)[x]$ and $\textsc{Spt}(y, \hat{E})[v] = \textsc{Spt}(y)[v]$ as every shortest path is simple. We can extend the previous example to constrained graphs with more than one edge restriction. We call the method of Fig. 5.2 *inserting* (x, y) *and* $\textsc{Spt}(y, \hat{E})$ *into* $\textsc{Spt}(u, \hat{E})$ to yield $\textsc{Spt}(u)$.

▷ Lemma 5.6. *For a subset of edges $\hat{E} \subseteq E$, let $\textsc{Spt}(u, \hat{E})$ be a constrained shortest path tree rooted in u. Then, inserting all edges (x, y) and shortest path trees $\textsc{Spt}(y)$ with $(x, y) \in \hat{E}$ back into $\textsc{Spt}(u, \hat{E})$ in any order yields $\textsc{Spt}(u)$.*

Proof. W.l.o.g. assume that every shortest path is unique and arbitrarily fix the order of insertions, i.e. number the edges $(x_1, y_1), \ldots, (x_k, y_k)$ for $k = |\hat{E}|$ and insert them in order. Let $\textsc{Spt}(u, \hat{E}_i)$ denote the tentative shortest path tree after i insertions. Also note that every value of $\textsc{Spt}(u, \hat{E}_i)[v]$ for some i and $v \in V$ corresponds to a $u \to v$ path in G, meaning by definition $\textsc{Spt}(u, \hat{E}_i)[v] \geq d(u, v)$ at all times. Now consider a $u \to v$ path P. If $w(P) = d(u, v)$ for every $v \in V$, then the procedure is correct, i.e. the lemma follows. Thus fix some $v \in V$ and consider the shortest $u \to v$ path P. There exists some $i \in [0..k]$ such that no edge (x_j, y_j) with $j > i$ is present in P. If the statements follows for every $i \in [0..k]$, i.e. regardless of which round P is first fully present in G, P is correct, the lemma holds. We prove by induction over the highest number $i \in [0..k]$.

Base case ($i = 0$): If P does not use any edges in $\hat{E}$, i.e. $i = 0$, then P is part of $\textsc{Spt}(u, \hat{E})$ and thus we have $\textsc{Spt}(u, \hat{E})[v] = d(u, v)$. As our update procedure can only decrease the distance to a node, this value remains unchanged for the entirety of the procedure, meaning $\textsc{Spt}(u)[v] = d(u, v)$ at the end.

[1] By definition, $\textsc{Spt}(u, \hat{E})[v] = \infty$ if no $u \to v$ path exists in $\hat{G}$. Thus, this only includes the case where every shortest $u \to v$ path traverses (x, y).

Inductive step($i \Rightarrow i+1$): Assume that the shortest paths of all nodes whose shortest paths do not use $(x_{i+1}, y_{i+1}), \ldots, (x_k, y_k)$ are all correct. By assumption, P must use (x_{i+1}, y_{i+1}) but not (x_j, y_j) for $j > i+1$. Since we assume all (potential) edge weights to be positive, the shortest $u \to x_i$ path P' can not use edge (x_i, y_i). Furthermore, because we assume shortest paths to be unique, P' must be a subset of P, i.e. P consists of P', followed by (x_i, y_i) and a shortest $y_i \to v$ path. This implies that P' can not use edges (x_j, y_j) for $j > i$ and by assumption must therefore already be correctly computed. Then, due to the presupposition that $\textsc{Spt}(y_i)$ is correct, our procedure should correctly determine $\textsc{Spt}(u)[v] = \textsc{Spt}(u, \hat{E}_{i+1})[v] = \textsc{Spt}(u, \hat{E}_i)[x_i] + w_\phi(x_i, y_i) + \textsc{Spt}(y_i)[v]$, thus completing the proof. □

Lemma 5.6 should serve as an intuition for our parallel superstep, formalized in Algorithm 7. The key idea is to forbid the first $i-1$ affected edges in the i^{th} Dijkstra iteration. Note that we do not use pruning but instead require the complete shortest path tree to be computed.[2] Then, we iteratively decide whether to accept a weight decrease or not. Afterward, we "broadcast" this weight change and the computed shortest path tree of the respective round to all later rounds that have yet to be decided. By induction, when we decide in round i, we have inserted all prior shortest path trees and weight changes and thus have a complete *correct* shortest path tree $\textsc{Spt}(v_i)$. However, at the time we insert (u_i, v_i) and subsequently $\textsc{Spt}(v_i)$ into $\textsc{Spt}(v_j, \cdot)$ for $i < j$, $\textsc{Spt}(v_i)$ still uses unchanged weights of later rounds $i < k < j$, i.e. if we decrease a weight in round $k > i$, this is not reflected in $\textsc{Spt}(v_i)$ which we insert in $\textsc{Spt}(v_j)$ for $j > k$. Hence, technically $\textsc{Spt}(v_i)$ is not completely *correct* for $\textsc{Spt}(v_j)$. Nevertheless, this is in fact not a problem due to the way we design our insert-procedure.

▷ Lemma 5.7. *A sequence of ℓ succeeding weight decreases of Algorithm 5 are correctly simulated by Algorithm 7.*

Proof. First note that by Lemma 2.20, potential changes do not affect shortest path and that Obs. 5.5 works when each edge is equally weighted in both shortest path trees (e.g. each shortest path tree assigns the same edge the same weight). Thus, broadcasting potentials as done in the Algorithm 7 not only does not conflict with any shortest path computation but also ensures that the Spt-insertions are correct.

For the main part, it suffices to show that after iteration $i-1$ of the Update-loop, D_i is *correct* for the current graph, i.e. after the first $i-1$ acceptions/rejections. Because of the correctness of the original proof, correctness of Algorithm 7 then follows. We show by induction over the rounds i.

Base case ($i = 1$): In the first round, no prior edges are forbidden and Dijkstra correctly computes D_1 in Line 6.

Inductive step ($1, \ldots, i \Rightarrow i+1$): Fix some arbitrary $v \in V$ and consider a shortest $v_i \to v$ path P. Let $j_1 < j_2 < \cdots < j_d$ denote the indices of affected edges of

[2] This makes no difference asymptotically in the worst-case but makes this algorithm useless in practice as the best-case is asymptotically significantly worse.

Algorithm 7: Parallel Weight-Decrease-SamplerPot superstep

Input: graph $G = (V, E)$, weight function w, feasible potential function ϕ, number of decrease steps ℓ
Output: w and ϕ after ℓ steps of the MCMC

```
// Sampling Phase
```
for $i = 1, \ldots, \ell$ **do par**
  $(u_i, v_i) \leftarrow$ uniform random edge
  $w_i \leftarrow$ uniform random weight $\leq w(u_i, v_i)$

```
// Computation Phase
```
$E_i \leftarrow \{(u_j, v_j) \mid j \leq i\}$
for $i = 1, \ldots, \ell$ **do par**
  compute $D_i \leftarrow \textsc{Spt}(v_i, E_i)$

```
// Update Phase
```
for $i = 1, \ldots, \ell$ **do**
  `// Broadcast Potential changes if necessary`
  if $D_i[u_i] \geq \mathcal{B}_{u_i v_i}$ **then** `// Accept Weight-Update`
    $\Delta\phi[v] \leftarrow \max\{\mathcal{B}_{u_i v_i} - D_i[v], 0\}\ \forall v \in V$
    $\phi \leftarrow \phi + \Delta\phi$
    for $j = i, \ldots, \ell$ **do par**
      for $v \in V$ **do par**
        $D_j[v] \leftarrow D_j[v] + \Delta\phi[v] - \Delta\phi[v_j]$
    $w(u_i, v_i) \leftarrow w_i$
  `// Broadcast new shortest paths`
  for $j = (i+1), \ldots, \ell$ **do par**
    if $D_j[u_i] + w_i < D_j[v_i]$ **then**
      $D_j[v_i] \leftarrow D_j[u_i] + w_i$
    for $v \in V$ **do par**
      if $D_j[v_i] + D_i[v] < D_j[v]$ **then**
        $D_j[v] \leftarrow D_j[v_i] + D_i[v]$

return w, ϕ

earlier rounds that lie on P. Let π denote the permutation of $j_1, \ldots, j_d$ in which these edges appear in P. Consider the path decomposition of P into these parts, i.e.

$$P_1 = (v_i \to u_{\pi(j_1)}), (u_{\pi(j_1)}, v_{\pi(j_1)}), \ldots, (u_{\pi(j_d)}, v_{\pi(j_d)}), P_{d+1} = (v_{\pi(j_d)} \to v).$$

Now first observe that because we restrict ourselves to only weight-decreases, a shortest path can only change if the *new* shortest path uses an edge whose weight was just decreased. Therefore if P_k is a shortest path in D_i, it must also have been a shortest path in every prior round. Thus, P_k was correctly computed by Dijkstra in round $\pi(j_{k-1})$ for $k \geq 2$.

Let σ denote the function that maps j_c to its direct predecessor in π, i.e. if $\pi^{-1}(j_c) = j_b$, then $\sigma(j_c) = \pi(j_{b-1})$ (if $b = 1$, then $\sigma(\cdot) = i$). Let $\hat{sigma}$ denote σ defined for the immediate successor (return 0 in the edge case with $u_0 = v$). Then, by Obs. 5.5, after round j_1, D_{j_1} was correctly inserted into $D_{\sigma(j_1)}$ and thus $D_{\sigma(j_1)}[u_{\sigma(\hat{j}_1)}]$ is now correct. We can then apply this argument recursively using the induction assumption to finally yield that $D_i[v]$ must be also correct completing the proof. □

▷ Theorem 5.8. *For $p \leq \ell$ PUs, Algorithm 7 performs ℓ consecutive weight-decreases of the MCMC in Algorithm 5 in parallel in time*

$$\mathcal{O}\left((m + n \log n + n\ell) \cdot \frac{\ell}{p}\right).$$

Proof. As mentioned above, the correctness of the algorithm follows using Obs. 5.5 and Lemma 2.20. For runtime, consider the three phases:

- Sampling
  We sample ℓ edges and weights independent of each other using p PUs. Sampling of a single edge and weight needs constant time leading to a runtime of $\mathcal{O}\left(\frac{\ell}{p}\right)$.

- Computation
  We run ℓ Dijkstra instances independent of each other. While some use even fewer edges and nodes (since they are constrained), every individual Dijkstra instance is still upper bounded by $\mathcal{O}(m + n \log n)$. Again, with ℓ instances and p PUs, this yields a runtime of $\mathcal{O}\left((m + n \log n) \cdot \frac{\ell}{p}\right)$.

- Update
  We have to run ℓ iterations of the for-loop sequentially. Here, the i^{th} iteration first resolves the question of acceptance in constant time before potentially updating ϕ and broadcasting that update. Both the broadcasting as well as the updating can be done in parallel. We need to broadcast to $\Theta(\ell - i)$ constrained shortest path trees as well as update $\mathcal{O}(n)$ potentials each in the worst case. Thus, this needs time $\mathcal{O}\left((\ell - i) \cdot n \cdot \frac{1}{p}\right)$.

  Then, the new shortest paths are broadcasted. This takes again the same time as broadcasting potentials (with additional constant work) as we need to broadcast to $\Theta(\ell - i)$ constrained shortest path trees and update at most $\mathcal{O}(n)$ shortest paths in constant time. Again, this yields a runtime of $\mathcal{O}\left((\ell - i) \cdot n \cdot \frac{1}{p}\right)$.

  Hence, Update requires a total runtime of $\mathcal{O}\left(\sum_{i=1}^{\ell}(\ell - i) \cdot \frac{n}{p}\right) = \mathcal{O}\left(n \cdot \frac{\ell^2}{p}\right)$.

Combining the runtime of all three phases and rearranging yields the stated runtime. □

In our previous discussion, we only considered a sequence of weight-decreases. As the original MarkovChain however does not allow forcing of ℓ rounds to be weight-decreases, without changing the original algorithm, we can only parallelize

consecutive weight-decreases if we randomly sample them without bias. Sadly, since we expect very few consecutive weight-decreases at best[3] and thus no real application of this method. Nonetheless, we provide an upper bound for the number of consecutive weight-decreases for which parallelization makes sense, i.e. is work-optimal.

It is easy to see that "setting" ℓ greater than p does not benefit neither runtime nor work as we only get the additional overhead from the greater number of rounds to be parallelized without the benefit of being able to parallelize all this work. Hence, both in a theoretical and practical context, it makes sense to "set" $\ell = p$. For exposition, let $\mathcal{P}$ denote the parallelized sampling algorithm for ℓ weight-decreases of the MCMC.

This leads to the question of choosing p in a way to make $\mathcal{P}$ as work-efficient as possible. For that, substitute p with ℓ and rearrange further to get the runtime of the algorithm as

$$\mathcal{T}_{\mathcal{P}}(n, m, \ell) = \mathcal{O}\left(n \cdot \max\left\{\frac{m}{n}, \log n, \ell\right\}\right).$$

Remember that by Corollary 4.21, the runtime of the sequential algorithm SamplerPot (or BiSamplerPot) for ℓ steps of the MCMC is

$$\mathcal{T}_{\textsc{SamplerPot}}(n, m, \ell) = \mathcal{O}\left(\ell \cdot n \cdot \max\left\{\frac{m}{n}, \log n\right\}\right).$$

Hence to achieve optimal work, ℓ should not be too big:

▷ Corollary 5.9. *Algorithm 7 has optimal work in the worst-case for $\ell = \mathcal{O}(\max\{\frac{m}{n}, \log n\})$ and thus a speedup of $\Theta(p)$ in these cases.*

Notice that algorithms such as SamplerPot and BiSamplerPot use pruning in their shortest path queries which significantly accelerates the best-case of the algorithm. While the original runtime-bounds still hold in the worst case and thus the previous corollary, because Algorithm 7 does not use pruning, this method is not work-optimal in the best-case (or probably even average-case).

For weight-increases, observe that a parallelization of consecutive weight-increases is trivial as we accept every one without changing potentials. Unfortunately, the previous method does not simply work if we mix weight-increases and weight-decreases. The problem lies in later weight-increases. If between two rounds i and j a weight-increase happens in round $i < k < j$, a path P that is a shortest $v_i \to v$ path in $\textsc{Spt}(v_i)$ could no longer be a shortest $v_i \to v$ path in $\textsc{Spt}(v_j)$ not because there now is some other $v_i \to v$ path that is shorter but because P itself get longer due to the weight-increase and a previous path that was longer is now shorter. This can not be reflected in our simple update procedure as we only update shortest paths if they get shorter.

Intuitive approaches such as separating shortest path trees in round i, i.e. splitting the shortest path tree into two versions — one with the current weight of $w(u_k, v_k)$ and one with the future weight of (u_k, v_k) — also do not work. The first

[3] As an intuition: for uniform weights, we expect every second proposed weight update to be a weight-increase if we never reject a change.

one is possibly needed to determine whether to accept a weight-decrease of (u_i, v_i) whereas the second one is then needed in rounds $l > k$. In a sequence of ℓ rounds, there could then be $\Theta(\ell)$ rounds with weight-increases which could then lead to a possible need of $\Theta(\ell^2)$ shortest path trees that need to be computed.[4] This, in turn, would then lead to the same runtime in the parallel case as in the sequential case, but with no pruning and additional overhead due to parallelization.

5.3 Ignoring Dependencies

As resolving dependencies provides neither asymptotic nor practical benefits but rather worsens the performance, we might want to opt for a different approach to tackle dependencies. Hence, instead of trying to resolve dependencies immediately when they happen, we ignore them and resolve them at a later time. This is motivated by the idea that partial shortest path trees in (the arguable faster (see Chapter 6)) BiSamplerPot are often very small. Hence the hypothesis is that, for bigger graphs (with same $\mathcal{W}$), succeeding partial shortest path trees are unlikely not overlap.

▷ Definition 5.10. *For node $v \in V$, the ConflictLength $C(v)$ is defined as the number of consecutive steps in BiSamplerPot (or SamplerPot) in which v is not visited by the underlying search. The conflict-length of the MarkovChain is defined as the number of steps it takes before any node gets visited again.*

Note that the notion of ConflictLengths is loosely defined as it is not specified for which round C is defined as a node most probably has a variety of ConflictLengths across the entire algorithm. This definition is rather meant as an intuition and metric we can later track in Chapter 6 as properties of not only the algorithm but the underlying MarkovChain itself.

Due to possibly (very) large shortest path trees in single rounds, many nodes might get a very low ConflictLength more often. However, we also expect many outliers among those, depending on graph topology and drawn weights; if many weight increases are drawn (for example when starting with w_{zero} in the first m rounds), the shortest path trees are practically non-existent and thus have an abysmal chance of colliding.

For the average conflict-length of the MarkovChain however, a single node conflict suffices to create a round-conflict which is way more likely. Hence, round-conflicts should happen way more frequently.

We propose a parallel algorithm making use of this notion (see Algorithm 8). We briefly study the performance of Batching in Chapter 6. Due to the expected low value of average conflict-lengths of the MarkovChain, we expect the additional overhead of Batching to outweigh the benefits of parallelism for most graphs with reasonable size.

▷ Observation 5.11. *For p PUs, Batching has an expected runtime of*

$$\mathcal{T}_{\textsc{Batching}} = \mathcal{O}\left(\tau \cdot \left(\min\left\{\hat{C}, p\right\}\right)^{-1} \cdot (m + n \log n)\right)$$

where $\hat{C}$ is the average conflict-length of the MarkovChain.

[4] If for example, the rounds alternate between weight-decreases and weight-increases.

Algorithm 8: Batching

Input: graph $G = (V, E)$, weight function w, feasible potential function ϕ, number of PUs p, number of steps τ
Output: w and ϕ after τ steps of the MCMC

$\ell \leftarrow 1$
$Q \leftarrow$ Queue
while $\tau > 0$ **do**
  `// Sampling Phase`
  while $Q.\texttt{length} < \min\{\ell, \tau\}$ **do**
    $(u, v) \leftarrow$ uniform random edge
    $w \leftarrow$ uniform random weight
    Q.push$(\{(u, v), w\})$
  `// Computation Phase`
  for $i = 1, \ldots, \ell$ **do par**
    compute partial shortest path trees D_i using Dijkstra or BiDijkstra for the i^{th} edge and weight in Q
  `// Conflict Phase`
  $i \leftarrow \min\{j \in [\ell] \mid D_j$ has conflict with an earlier round$\}$
  accept/reject first $j - 1$ weight changes in Q and apply changes
  pop first $j - 1$ elements in Q
  if *conflict happened* **then**
    $\ell \leftarrow 2^{\lfloor \log_2(i) \rfloor}$
  else
    $\ell \leftarrow \min\{p, 2 \cdot \ell\}$
  $\tau \leftarrow \tau - j + 1$
return w, ϕ

5.4 Manipulating Dependencies

Unfortunately, the previous two methods (probably) perform subpar in practice which is why we want to study yet another method of resolving dependencies. Instead of strictly preserving the original MarkovChain of Algorithm 4, we change up the sampling procedure to allow weight changes for multiple edges at once. Since two edges in general can be hard-dependent on each other, we restrict the class of edges over which we can parallelize.

▷ Lemma 5.12. *For a fixed node $v \in V$, changing the weight $w(u_1, v)$ of an incoming edge $(u_1, v) \in E$ is oblivious to weight changes $(u_2, v), (u_3, v), \ldots$ of other incoming edges $(u_i, v) \in E$ that happened immediately prior if none introduce a negative cycle.*

Proof. As increasing the weight of an edge is always oblivious to other changes, assume that we decrease $w(u_1, v)$. If we changed the weight of another incoming edge (u_2, v) of v immediately before, then the only edge-weight change that differs is that of (u_2, v): every other edge weight is unaffected. Potentials might differ

but they do not influence the decision on whether to accept the weight change of (u_1, v).

Thus, accepting the weight-decrease of (u_1, v) only depends on the existence of a $v \to u_1$ path P with length at most $-w_\phi(u_1, v)$. As we assume there are no negative cycles, every shortest path is simple, and hence P does not traverse any other incoming edge of v (including (u_2, v)). Therefore, P is oblivious to weight changes of other incoming edges of v. □

Algorithm 9: NodeSampler: an approximate $\mathcal{S}_{G,\mathcal{W}}$-Sampler

Input: graph $G = (V, E)$, possible weights $\mathcal{W}$, number of steps τ
Output: consistent weight function $w\colon E \to \mathcal{W}$

```
1  w ← arbitrary consistent weight function
2  φ ← feasible potential function with respect to w
3  repeat τ times
4      v ∼ U(V)
5      c ∼ U(W^{deg_in(v)})
6      w' ← w_{(u_1,v)←c_1,(u_2,v)←c_2,...}
7      B_uv ← max{ −w'_φ(u_i, v) | i ∈ [deg_in(v)] }
8      T ← Dijkstra_{B_uv}(G, w_φ, v)
9      for i ∈ [deg_in(v)] do
10         if T[u_i] ≥ −w'_φ(u_i, v) then
11             w ← w_{(u_i,v)←c_i}
12     if at least one accepted then
13         φ(x) ← φ(x) + B_uv − T[x]   ∀x ∈ T
14 return w
```

Applying Lemma 5.12 to our MarkovChain yields Algorithm 9. Note that the Dijkstra$_{\mathcal{B}_{uv}}(G, w_\phi, v)$ call in Line 8 works slightly different than before as we now can traverse predecessors u_i of v for which we have a pending weight change of (u_i, v) without early returning. Namely, as we limit the maximum distance of this Dijkstra call by the maximal broken value, we can find other predecessors in fewer steps without rejecting them. Following Lemma 5.12, we can continue the search from these nodes without considering edges that lead back to u.

Since we altered the algorithm, it is not directly clear that important properties of Theorem 4.12 still hold for this variant.

▷ Theorem 5.13. *For $\tau \to \infty$, w in Algorithm 9 converges to $\mathcal{S}$.*

Proof. Most important observations for the original sampler also apply for this one. Most notably, if there is a direct transition from w to w', there is also a direct transition from w' to w. Furthermore, increasing all incoming edge weights is always allowed and thus a possible step. Thus, again, irreducibility follows from the same argument (over w_{max}).

For aperiodicity, we can again observe that it is possible to draw the exact current weight of every incoming edge in one round leading to no change in w. Hence, the underlying state graph has a loop and by Obs. 2.36, this implies aperiodicity.

For symmetry, consider two neighboring states $w \neq w'$. By definition, w and w' can differ only in incoming edge weights of at most one node. Since every node is drawn uniformly regardless of the current weight function, the probability of drawing this exact node is equal in both cases.

Now, due to Lemma 5.12 and by construction of the sampling procedure, we can interpret weight changes of multiple incoming edges independent from each other, i.e. the probability to change the weight of edge e from $w(e)$ to $w'(e)$ is independent of the probability to change the weight of edge e' from $w(e')$ to $w'(e')$. Then, by Obs. 4.9, the probability of transitioning from $w(e)$ to $w'(e)$ is equal to the probability of transitioning from $w'(e)$ to $w(e)$ (although we now have a $\cdot|V|$ instead of $\cdot|E|$). Therefore, the MarkovChain is again symmetric, and by Lemma 2.39, this concludes the proof. □

Although we motivated Algorithm 9 as a parallelized sampling algorithm, the "parallel" should be taken with a grain of salt as the algorithm itself is sequential again and falls under the same shortcomings as the original sampling algorithm. However, we increased the number of edge-weight changes we can do per shortest path query, thus achieving a de-facto parallelization of the original sampling algorithm.

We again conduct empirical evaluations of this algorithm in Chapter 6 where we compare NodeSampler to BiSamplerPot to study whether the loss of the bidirectional search is outweighed by the increased (but biased) number of weight updates.


K_6

C_6

P_6

$K_{1,6}$

6 Experiments

The experiments are split into two different parts that go in order over the two previous chapters:

1. (Sections 6.1 to 6.4): Properties of the underlying MarkovChain of Algorithm 4 that are independent of the chosen algorithm.

2. (Sections 6.5 to 6.7): Properties of the proposed algorithms in Chapters 4 and 5.

We conduct all experiments with the following input parameters and repeat each combination 10 times.

- Random graphs drawn from $\mathcal{GNP}(n, \overline{d}), \mathcal{DSF}(n, \overline{d}), \mathcal{RHG}(n, \overline{d})$ graphs with $n = 10000$ and $\overline{d} \in \{10, 20, 50\}$.

- Initial weight functions $w \in \{w_{max}, w_{unif}, w_{zero}\}$.

- Number of MCMC-steps $\tau = 10^8$.

- Integer weights in $\mathcal{W} = [-100, 100]$.

Due to a very high number of isolated nodes and consequently SCCs in $\mathcal{DSF}$ graphs, we instead extract the largest SCC of the generated $\mathcal{DSF}$ graph and ran all algorithms on its induced subgraph. To match the number of nodes and edges in this subgraph as best as possible with the intended number of nodes and edges, we modify the initial parameters[1] to

- $(n, \overline{d}) = (25000, 6)$ which led to a SCC with roughly 10000 nodes and average degree 10.

- $(n, \overline{d}) = (20000, 14)$ which led to a SCC with roughly 10000 nodes and average degree 20.

- $(n, \overline{d}) = (17000, 47)$ which led to a SCC with roughly 10000 nodes and average degree 50.

We also focus on the following questions for each experiment:

[1] All these values were found experimentelly using a binary search by hand.

What impact does the underlying graph model have? As discussed in Section 2.3, $\mathcal{RHG}$ and $\mathcal{DSF}$ graphs both are "scale-free" and their degrees follow a power-law distribution whereas edges in $\mathcal{GNP}$ graphs are independent of each other and therefore degrees follow a binomial distribution. Additionally, $\mathcal{RHG}$ graphs are originally undirected and thus have a reversed edge for every edge in the graph. Due to this, $\mathcal{RHG}$ graphs have on average the most cycles and the shortest cycles of all three models whereas $\mathcal{DSF}$ graphs are the most acyclic, i.e. have the least number of cycles in expectation.

What impact does the average degree have? Obviously, graphs of the same random graph class have no significant structural change for bigger average degrees. But, a bigger average degree translates into a higher number of edges and thus also a higher number of cycles (and possibly shorter cycles) in the graphs.

What impact does the initial weight function have? If the average weight is higher, the probability of accepting a proposed weight change consequently also rises. While there are definitely some scenarios in which a few select number of edges contribute significantly to the high average weight whereas every other edge has a very low weight, these are definitely highly improbable as we assume some kind of uniform distribution in edge-weights across the graph.

All experiments were implemented in Rust[2] and run on the Goethe-NHR cluster[3]. We only used nodes with two `Intel Xeon Skylake Gold 6148` *processors with a total of* 40 *cores and* 192 *GB RAM per node. The implementation can be found on GitHub[4].*

6.1 Weight Distributions

We first study the distribution of edge weights. Since, for each of the 27 possible settings (3 graph models, 3 average degrees, 3 initial weight functions), we have 10^8 rounds in which we can examine the weight distribution over 201 possible weights, the sheer amound of data warrants a reduction to a few key aspects. For this, we first investigate weight distributions as a whole over time in some fixed intervals.

While weights are uniformly drawn from the symmetric interval $[-100, 100]$, since we reject negative cycles, the average weight in strongly connected graphs must always be non-negative[5]. Because extremely low weights are highly likely to get rejected and extremely high weights are highly likely to be changed in later rounds, we expect distributions to have a peak around some weight and be linearly decreasing in probability mass in both directions. Also, the impact of the initial weight function should be negligible in higher rounds since all instances converge to the same distributions regardless of starting weights.

[2] https://www.rust-lang.org/

[3] https://csc.uni-frankfurt.de/

[4] https://github.com/lukasgeis/negative_edge_weights

[5] Under the assumption that the graph is strongly connected. Otherwise a single edge not part of any SCC could potentially lower the average weight below 0.

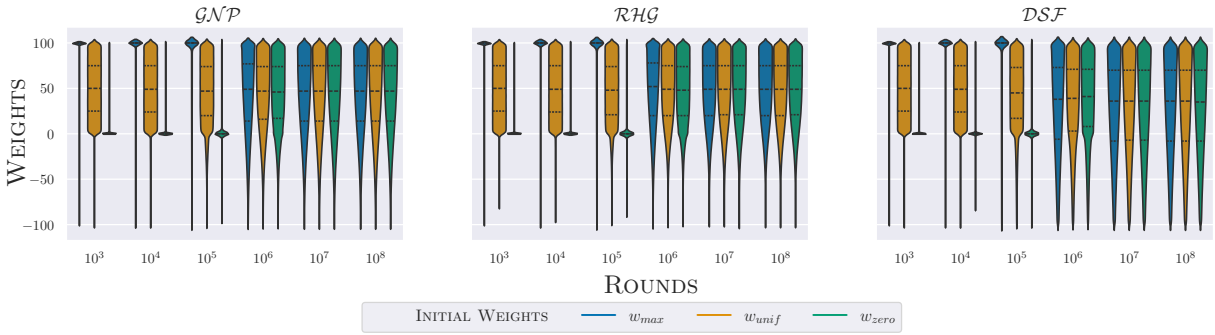


Figure 6.1: Weight distributions over time for different graphs with $n = 10000$ nodes and average degree $\overline{d} = 10$.

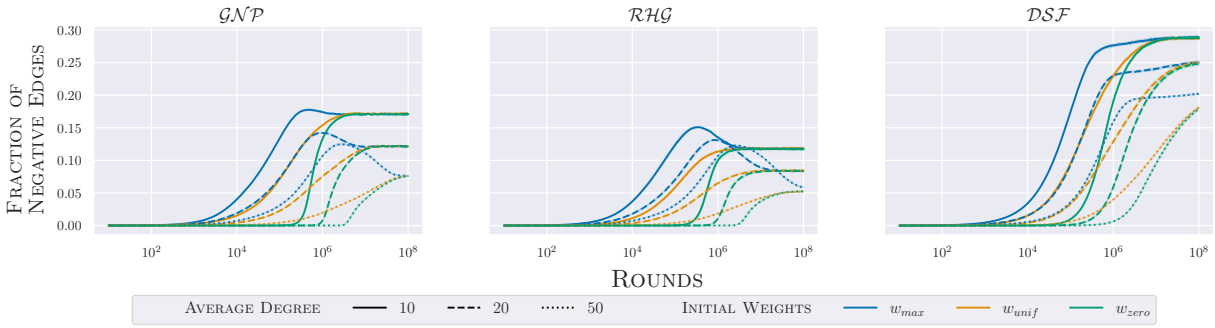


Figure 6.2: Fraction of edges with negative weights over time for different graph models with $n = 10000$ nodes and average degree $\overline{d} \in \{10, 20, 50\}$.

Fig. 6.1 displays the weight distributions at fixed rounds 10^i for $i = 3, \ldots, 8$. The complete dataset can be seen in Fig. B.1. Clearly, independent of initial weight function, in each instance, all weight distributions converge to a similar one in roughly 10^7 steps. This *final* weight distribution is also centered around an average of roughly 50 with slight variations depending on the chosen graph model and average degree: less cycles lead to lower weights as more weight decreases get accepted whereas more and shorter cycles lead to higher weights as more weight decreases get rejected (see $\mathcal{DSF}(\overline{d} = 10)$ and $\mathcal{RHG}(\overline{d} = 50)$ for extremes).

Additionally, while instances that started with w_{max} already have extremely low weights after 10^3 rounds, instances that started with w_{zero} have almost no negative weight for the first 10^5 rounds. This can be further observed in Fig. 6.2 which plots the fraction of edges with negative weight over time. As one would expect, instances of the same model and average degree converge to a fixed fraction independent of the initial weight function. Also, less cycles lead to higher fractions with $\mathcal{DSF}$ graphs achieving fractions almost twice as high as $\mathcal{RHG}$ graphs.

Surprisingly, for initial weight function w_{max}, the MarkovChain seems to "overshoot" the fraction of negative edges as at the beginning we have many extremely high edge weights that allow for many (small) negative weights. As this fraction of extremely high weights gradually decreases over time, the fraction of negative edges concurs and also decreases again until it reaches a stable state. w_{unif} instances fall almost directly in the middle between w_{max} and w_{zero} curves

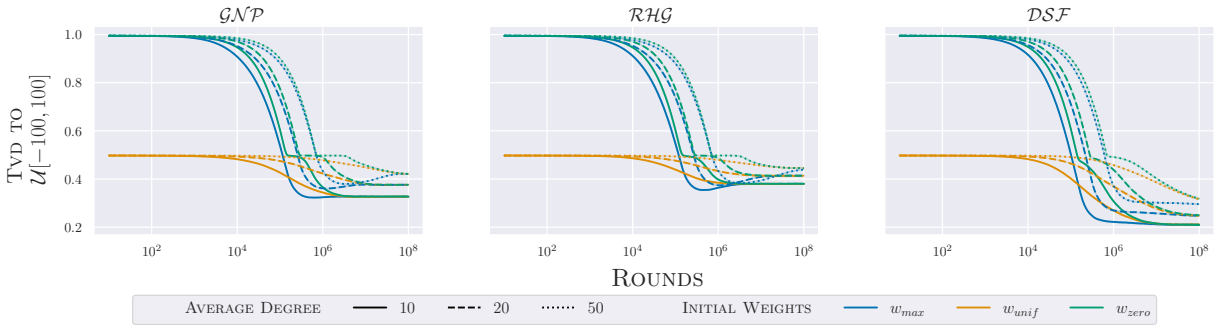


Figure 6.3: Total variation distance to $\mathcal{U}[-100, 100]$ over time for different graph models with $n = 10000$ nodes and average degree $\overline{d} \in \{10, 20, 50\}$.

which suggests that one could parametrize initial weight functions (possibly by average weight) to interpolate between w_{max} and w_{zero} curves.

Total Variation Distance

Another measure for converging distributions is the total variation distance (Tvd)[6] to the distribution in the limit π. But since we established in Section 4.1 that the limit distribution is very hard to compute as it requires some form of enumeration of all feasible states (and thus also of those with a certain number of negative edges), this is computationally unfeasible in this context. Instead, we measure the Tvd not to π but to $\mathcal{U}(\mathcal{W}^m)$, i.e. the distributions from which we draw our weights. While π and $\mathcal{U}(\mathcal{W}^m)$ should be inherently different, this also gives insight into the structure of π as we would expect the distribution after 10^8 steps to be approximately drawn from π. Therefore, the final total variations distances should also give insight into how much π differs from $\mathcal{U}(\mathcal{W}^m)$.

Fig. 6.3 summarizes the results for $\mathcal{GNP}$, $\mathcal{RHG}$, and $\mathcal{DSF}$ graphs as a function of average degree and initial weight function. As expected, w_{max} and w_{zero} start with a Tvd of nearly 1 as they concentrate all their probability mass into a single value whereas $\mathcal{U}(\mathcal{W})$ fairly divides the probability mass among all weights. Furthermore, w_{unif} start with a Tvd of exactly 0.5 because w_{unif} is a uniform distribution over exactly half the elements of $\mathcal{W}$. Still, instances of the same model and average degree converge to the same Tvd after 10^8 steps regardless of initial weight function.

Yet another repeated trend is the correlation between number of cycles and Tvd: more and shorter cycles equate to a higher TVD as lower weights (as seen in Fig. 6.2) are rejected more often and are therefore less likely in contrast to $\mathcal{U}(\mathcal{W}^m)$. Thus, $\mathcal{DSF}$ graphs with average degree 10 achieve the lowest Tvd from almost 0.2 at the end whereas $\mathcal{RHG}$ graphs with average degree 50 converge to a Tvd of roughly 0.45.

This is probably the biggest argument for the high acyclic nature of low degree $\mathcal{DSF}$ graphs compared to a low acyclicity of high degree $\mathcal{RHG}$ graphs. Yet, surprisingly, a uniform distribution of only positive weights is only 5% more different than an approximation of π for $\mathcal{RHG}$ graphs with average degree 50.

[6] Tvd was implicitly defined in Definition 2.40 as the term that should be less than ε.

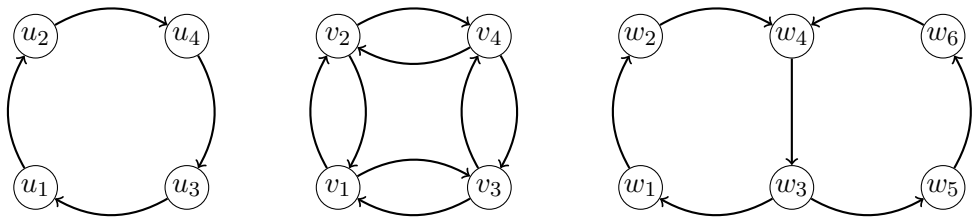


Figure 6.4: A 4-Cycle (left), 4-NestedCycle (middle) and 7-PairedCycle (right).

6.2 Coverage

To further expand on this notion of rapid practical convergence, we conduct three experiments on very small graphs ($m < 20$) that measure the number of steps it takes to see every possible weight distribution once. For that, we restrict ourselves to $\mathcal{W} = \{-1, 0, 1\}$ and the three following graph types, all of which can be seen in Fig. 6.4. They represent the possible ways in which cycles can exist:

- n-Cycle C_n: the simple cycle,
- n-NestedCycle NC_n: cycles nested inside other cycles,
- n-PairedCycle PC_n: cycles that share one edge.

For C_n, we in fact have $m = n$. For consistency, we refer to the number of edges as n (instead of m) for all three graphs (even though the number of nodes might be less). Thus, every graph has 3^n possible weight assignments. In these examples, we then can simplify the consistency-condition for a weight function $w = (w_1, \dots, w_n) \in \mathcal{W}^n$:

- (C_n, w) are consistent iff $\sum_{i=1}^{n} w_i \geq 0$
- (NC_n, w) are consistent iff $w_i + w_{n/2+i} \geq 0$ for all $i = 1, \dots, n/2$ as well as $\sum_{i=1}^{n/2} w_i \geq 0 \wedge \sum_{i=1}^{n/2} w_{n/2+i} \geq 0$.[7]
- (PC_n, w) are consistent iff $\sum_{i=1}^{\lceil n/2 \rceil} w_i \geq 0$ as well as $\sum_{i=\lceil n/2 \rceil}^{n} w_i \geq 0$.[8]

For C_n and PC_n this leads to a constant probability of drawing a consistent weight function if we sample each weight independently uniform (see Table 6.1 for C_n). In the case of NC_n, by transference of Section 4.1 we again have an exponential runtime using rejection sampling. But as we pick $n < 20$, this is negligible as we still have a constant time sampler.

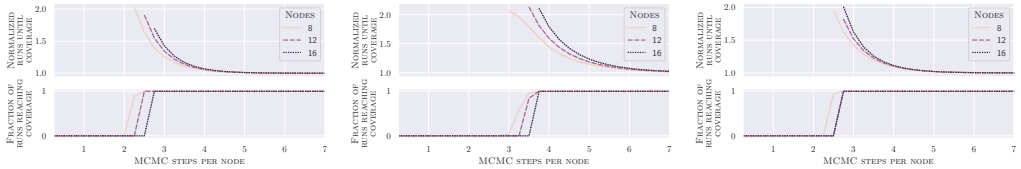
Hence, for each graph, we generate a random weight function $w \in \mathcal{W}^n$ and reject it iff it is not consistent. This algorithm then runs in expected time $\Theta(n)$. We then obtain a large number k of independent samples by running τ steps of

[7] Here, $w_1, \dots, w_{n/2}$ are the weights of clockwise edges, and $w_{n/2+1}, \dots, w_n$ the weights of counter-clockwise edges.

[8] Here, $w_1, \dots, w_{\lceil n/2 \rceil}$ are the weights of edges in the left cycle and $w_{\lceil n/2 \rceil}, \dots, w_n$ weights of edges in the right cycle, $w_{\lceil n/2 \rceil}$ is the weight of the shared edge.

n	$\lvert\mathcal{W}^n\rvert$	$\lvert\mathbb{S}_{C_n,\mathcal{W}}\rvert$	Acc. rate
8	6561	3834	0.584
12	531 441	302 615	0.569
16	43 046 721	24 121 674	0.560

Table 6.1: Number of possible $|\mathcal{W}^n|$ and consistent $|\mathbb{S}_{C_n,\mathcal{W}}|$ weights on the n-cycle [56].



(i) C_n (ii) NC_n (iii) PC_n

Figure 6.5: Runs to see 99% of $\mathbb{S}_{G,\mathcal{W}}$ for $G \in \{C_n, NC_n, PC_n\}$ for $n \in \{8, 12, 16\}$ as function of MCMC steps.

the MarkovChain each starting from initial weights w_{zero}. As we sample with replacement, we might get duplicates, but reminiscent of the Coupon Collector's problem [45], for large $n, k = \Theta(|\mathbb{S}_{G,\mathcal{W}}| \log(\mathbb{S}_{G,\mathcal{W}}))$, samples are expected to be sufficient to observe every item in $\mathbb{S}_{G,\mathcal{W}}$ at least once.

We also approximate a similar threshold by counting the number of samples the exact sampler takes to cover $\mathbb{S}_{G,\mathcal{W}}$. Since the last few elements incur significant noise, we stop earlier when 99% of elements were seen. We name this quantity *runs-until-coverage*.

Fig. 6.5 summarizes the results. As we pick n even, the left cycle of PC_n in fact has one edge less than the right one.

For C_n, we find that the exact sampler uses $(4.6 \pm 0.1)|\mathbb{S}_{C_n,\mathcal{W}}|$ *runs-until-coverage*. For NC_n, the sampler rather uses $(7.0 \pm 0.1)|\mathbb{S}_{NC_n,\mathcal{W}}|$ and for PC_n $(5.5 \pm 0.1)|\mathbb{S}_{PC_n,\mathcal{W}}|$ *runs-until-coverage*. As each sample is a new item with probability of at least 0.01, this is plausible.

We then replace the exact sampler with the MCMC process for different numbers of steps τ and measure their *runs-until-coverage*. This allows us to reject the hypothesis that the MCMC process is identical to the exact sampler if τ is too small. If coverage does not occur within $10|\mathbb{S}_{G,\mathcal{W}}|$ steps (i.e. more than twice the exact sampler's *runs-until-coverage*), we say that the run does not reach coverage. We observe a sharp transition from no runs achieving coverage to all runs completing succesfully between $2n \leq \tau \leq 3n$ for C_n, between $3n \leq \tau \leq 4n$ for NC_n and again between $2n \leq \tau \leq 3n$ for PC_n. In all cases we see a slight increase in *runs-until-coverage* for bigger n which is consistent with a suplinear mixing time.

6.3 Acceptance Rates

The final measure of Sampler we want to investigate is the the acceptance rate over time, i.e. the fraction of rounds in which we accepted the proposed weight change. This does include weight decreases as well as weight increases. Therefore,

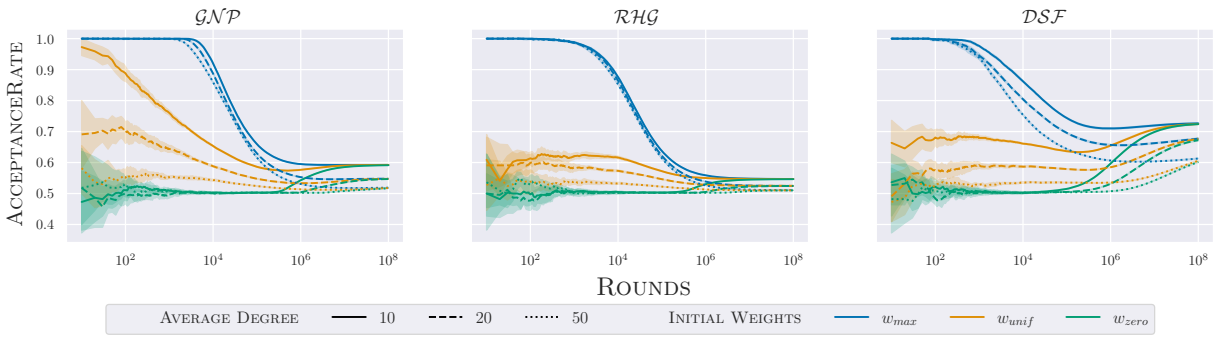


Figure 6.6: Acceptance-Rate over time for different graph models with $n = 10000$ nodes and average degree $\overline{d} \in \{10, 20, 50\}$

we expect the acceptance rate to converge to some value greater than 0.5 as we expect weight increases — which are immediately accepted — to happen in roughly 50% of the time due to $\mathcal{W}$ being symmetric around 0. Second, higher number of cycles should result in more rejections and thus a smaller acceptance rate as there are more ways in which a weight decrease could produce a negative cycle. As higher degrees equate to higher number of cycles, the higher the degree, the lower the acceptance rate. Lastly, we expect that the higher the average weight of the inital weight function, the higher the acceptance rate in the beginning. This influence however should diminish over time such that all graphs of the same model with the same number of nodes and edges reach a similar convergence point.

Fig. 6.6 supports our hypotheses. Most importantly, all instances of the same model with the same number of nodes and edges converge to the same acceptance rate greater than 0.5.

Furthermore, regardless of graph model or average degree, instances with an initial weight function w_{max} have an acceptance rate of nearly 1.0 at the beginning before quickly descending between 10^4 and 10^5 steps. Similarly, instances with initial weight function w_{zero} have an acceptance rate of roughly 0.5 at the start as roughly every second weight change is a weight increase. Finally, instances with initial weight function w_{unif} have starting acceptance rates between those of instances with initial weight functions w_{zero} and w_{max}. Surprisingly, for $\mathcal{GNP}$ graphs with average degree 10, instances with starting uniform weights also have a nearly 100% acceptance rate at the start. For $\mathcal{RHG}$ and $\mathcal{DSF}$ graphs, starting acceptance rates for w_{unif} tend to be centered around 0.6 (rather 0.5 - 0.7).

Additionally, as expected, higher average degrees in fact equate to lower acceptance rates. This can be seen not only at the final convergence points but also at the beginning and even more exaggerated for $\mathcal{GNP}$ graphs with initial weights w_{unif}.

Finally, convergence points of $\mathcal{RHG}$ graphs are very close to each other, while the convergence points for $\mathcal{GNP}$ and $\mathcal{DSF}$ graphs are slightly more spread out. In a similar fashion, $\mathcal{RHG}$ have the smallest convergence points, followed by $\mathcal{GNP}$ and then $\mathcal{DSF}$ graphs. This could be explained by the fact that in $\mathcal{RHG}$ we have many 2-cycles which consequently lead to many rejections. As these 2-cycles exist regardless of average degree, acceptance rates tend to be closer to each other.

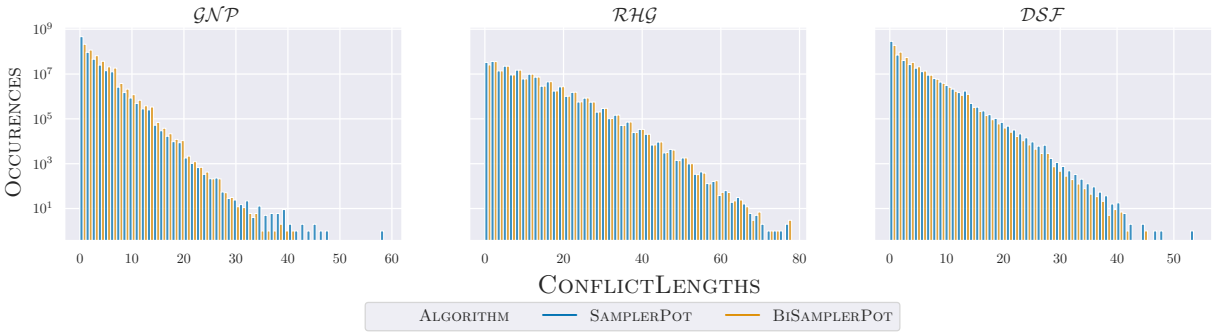


Figure 6.7: Occurences of ConflictLengths between rounds after 10^8 rounds for different graphs with $n = 10000$ nodes, average degree $\overline{d} = 10$ and starting weights w_{max}.

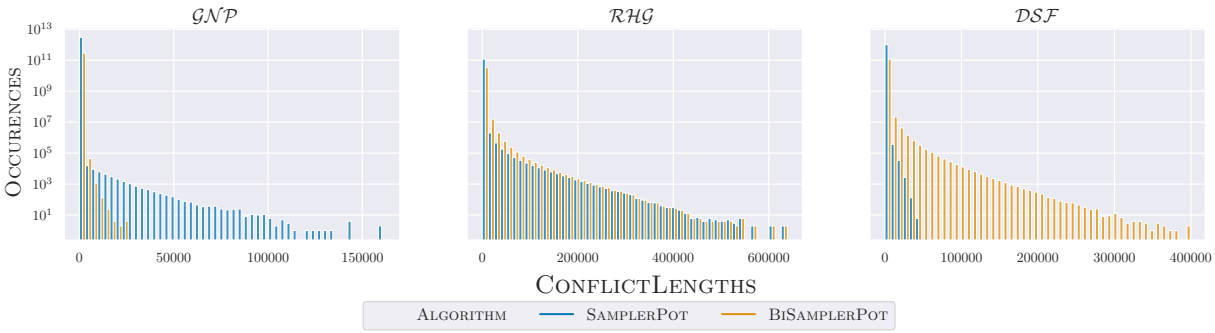


Figure 6.8: Occurences of ConflictLengths between nodes after 10^8 rounds for different graphs with $n = 10000$ nodes, average degree $\overline{d} = 10$ and starting weights w_{max}.

In $\mathcal{GNP}$ graphs, there are longer and fewer cycles which is amplified by a lower average degree whereas in $\mathcal{DSF}$ graphs there are even fewer cycles which leads to even higher acceptance rates.

While all previous measures are no direct proof for a rapid mixing time, they still suggest that the mixing time is bounded by a small polynomial in m. This and the proof in Appendix A.2 provide compelling evidence for a small mixing time in practice.

6.4 Conflict Lengths

The final metrics of the underlying MarkovChain we want to study are ConflictLengths which we introduced in Chapter 5. Note that accepting rounds in Algorithm 4 however are always global as BellmanFord does not use pruning and therefore only terminates (accepts) after every node has been visited at least once. Thus, every weight-decrease that gets eventually accepted generates a round-conflict with at least the most recent accepted weight-decrease.

Hence, Figs. 6.7 and 6.8 only depict the distribution of ConflictLengths for SamplerPot and BiSamplerPot for graphs with average degree 10 and initial weight function w_{max}. The complete dataset can be found in the appendix (Figs. B.2 to B.7). Although the decline of occurrences for bigger conflict lengths

is less steep on $\mathcal{RHG}$ graphs, the general trend is consistent across all different graph classes. Note that the y-axis in the plots is log-scaled while the x-axis is not. Thus, ConflictLengths of rounds seem to follow an almost perfect exponential distribution. Outliers in all three distributions can be explained by the chosen initial weight function w_{max}: in early rounds, we almost always can immediately accept a weight change without conducting a thorough search — a round-conflict basically becomes a node-conflict in these rounds.

While $\mathcal{GNP}$ graphs have one more outlier than $\mathcal{DSF}$ graphs, the distributions are almost identical. Only $\mathcal{RHG}$ graphs achieve higher average ConflictLengths which is consistent with observations from the previous experiment. Small cycles and lower acceptance rates lead to siginificantly smaller shortest path trees on average, thus reducing the number of conflicts.

As with all prior metrics, the resulting data is invariant to the initial weight function with the exception of a few outliers. Also again, a higher average degree leads to more cycles and thus also to lower ConflictLengths on average across all three graph classes.

In the case of node-conflicts, the patterns are less consistent. While, again, the rough trend distributions are invariant to the initial weight function, the impact is way more apparent as for w_{max} node-conflicts are incredibly rare in the first $\mathcal{O}(m)$ rounds which is more apparent than for round-conflicts (since round-conflicts depend on all nodes).

The average degrees and graph topology also play a more siginificant role this time. Surprisingly, for $\mathcal{GNP}$ graphs with average degree 10, SamplerPot has far fewer node-conflicts than BiSamplerPot— almost one order of magnitude less. This however immediately flips for average degrees 20 and 50 where BiSamplerPot now has far fewer node-conflicts on average than SamplerPot which can probably be attributed to the fact that partial shortest path trees in SamplerPot are more likely to have a bigger depth. Whereas for $\overline{d} = 10$, most partial shortest path trees are sparse and thus do not include many nodes despite a bigger depth, for larger average degrees this is not the case and a partial shortest path tree with bigger depth includes significantly more nodes leading to more node-conflicts. For BiSamplerPot, which has smaller partial shortest path trees on average and on the extreme, this effect is thus not that apparent.

The same steep decline of SamplerPot for higher average degrees can also be observed for $\mathcal{RHG}$ graphs. Although, BiSamplerPot performs equally good for average degree 10 in that case. Finally, for $\mathcal{DSF}$ graphs, BiSamplerPot has far fewer node-conflicts in all cases, regardless of average degree and weight function.

Notice however that this decline also happens for BiSamplerPot. Higher average degrees and thus more edges lead to smaller ConflictLengths for both algorithms in $\mathcal{GNP}$ and $\mathcal{RHG}$ graphs. This decrease however has an higher impact on SamplerPot compared to BiSamplerPot which again is consistent with prior observations. Also, due to the very small cycles in $\mathcal{RHG}$ graphs, the choice of algorithm does make a big impact for low average degrees.

Finally, the most surprising observation is the fact that ConflictLengths seem to increase for higher average degrees in $\mathcal{DSF}$ graphs. One possible explanation points again back at acceptance rates where $\mathcal{DSF}$ graphs had by far

the biggest decrease in acceptance rate for higher average degrees (more than 12% from $\overline{d} = 10$ to $\overline{d} = 50$). This, coupled with the already very acyclic nature of $\mathcal{DSF}$ graphs could then lead to very short rejecting paths which incur less node-conflicts overall.

Nonetheless, a more detailed investigation of ConflictLengths seems warranted to better understand the behaviour of the underlying MarkovChain as well as the distributions of shortest paths in such random graphs.

6.5 Sequential Algorithms

In this section, we want to study performance metrics of the various sequential algorithms proposed in Chapter 4, namely Sampler, SamplerPot, and BiSamplerPot. For each parameter setting, we run all three algorithms simultaneously in each round (back to back) to obtain the desired performance metrics on the same dataset for a fair comparison.[9]

As Sampler uses BellmanFord as a subroutine which is inherently slow, it is not feasible to run all 10^8 rounds of the MCMC in each instance for Sampler. Instead, we opt for an amortized approach and only ran BellmanFord every 10^3 rounds for $\tau > 10^6$. For $\tau \leq 10^6$, we run BellmanFord every $\max\left\{1, 10^{\lceil \log_{10}(\tau) \rceil - 4}\right\}$ rounds, i.e. for $10^i < \tau \leq 10^{i+1}$, we obtain $\min\left\{10^4 - 10^3, 10^{i+1} - 10^i\right\}$ datapoints of BellmanFord. Metrics such as runtime are then averaged out. Further discussed in Section 6.5.3, this has no significant impact on the obtained data and is coherent with test runs that executed BellmanFord every round.

6.5.1 Queue Insertions

The first metric we want to investigate is the number of queue insertions of the underlying shortest path queries (BellmanFord, Dijkstra, BiDijkstra). For BellmanFord, this is the number of queue insertions in the SPFA heuristic, for Dijkstra and BiDijkstra the number of priority queue insertions. We further differentiate between *accepted* and *rejected* rounds.

It is a common observation that in practice, worst-case bounds of BellmanFord tend to be overly pessimistic. Additionally, as mentioned before, bidirectional searches perform significantly better than their unidirectional variants. This is supported by Fig. 6.9 where we can see the distribution of Queue-Insertions over time for BellmanFord, SamplerPot, and BiSamplerPot for graphs with average degree 10 and initial weights w_{max}. Again, the complete dataset is found in the appendix (Figs. B.8 to B.10).

BellmanFord is by far the worst algorithm among all three. As Dijkstra and BiDijkstra can also significantly prune their search space, this is even more apparent for *accepted* rounds where a BellmanFord iteration only ends after the complete shortest path tree was discovered whereas Dijkstra and BiDijkstra can stop after computing partial shortest path trees. Surprisingly, in *accepted*

[9] This also served as an additional correctness check for these algorithms as we could assert equality of their output (*accepted* or *rejected*) in each round.

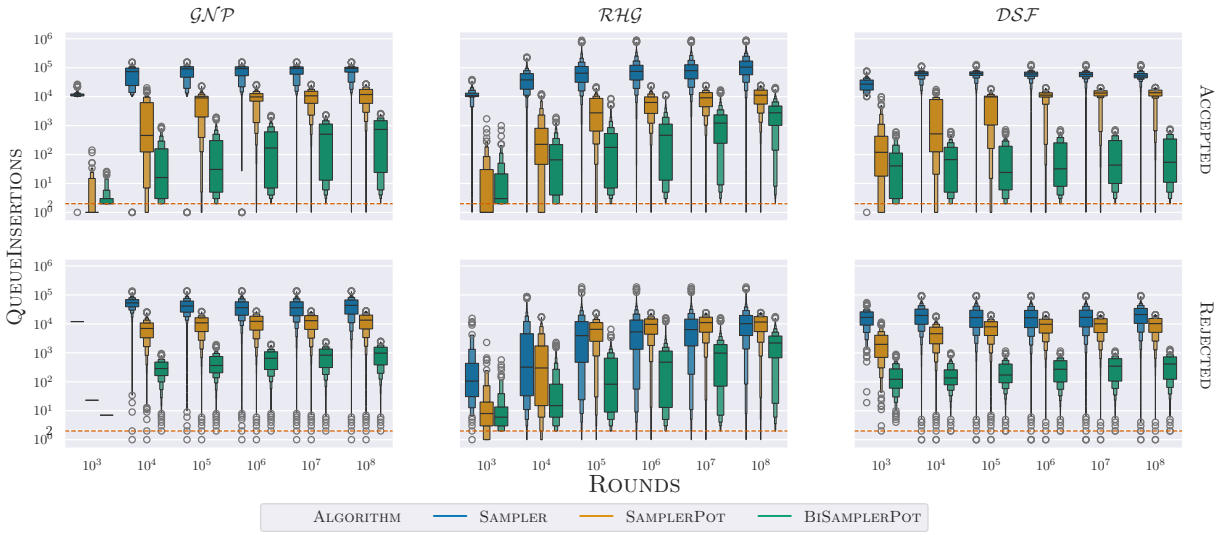


Figure 6.9: Number of Queue-Insertions per algorithm for different graphs with $n = 10000$ nodes and average degree $\overline{d} = 10$ with starting weights w_{max}.

rounds, Dijkstra performs in the median better than BiDijkstra, although this can be attributed to super short rounds with only 1 insertion (the starting one). While Dijkstra only terminates when its queue is empty, BiDijkstra terminates when *rejecting* path is disproven - often with leftover elements in its queue. Nonetheless, as we will see later in Fig. 6.12, the average number of Queue-Insertions is much smaller for BiDijkstra as it is for Dijkstra overall.

For *rejected* rounds however, the order of medians is flipped and BiDijkstra performs significantly better than Dijkstra; whereas in *accepted* rounds, both Dijkstra and BiDijkstra performed on average less than 10 insertions, in *rejected* rounds, BiDijkstra is a whole order of magnitude better ($10^2 \sim 10^3$ compared to $10^3 \sim 10^4$). Still, BellmanFord performs the worst, although the difference to Dijkstra is not that apparent as before anymore: instead of multiple orders of magnitude, BellmanFord is at most 10 times worse than Dijkstra. This is coherent with theory as both BellmanFord and Dijkstra need to find the same path that leads to *rejection*: pruning no longer allows for early returns.

6.5.2 Potential Updates

A similar point of comparison is the number of potential updates incurred by an *accepted* weight change. As BellmanFord does not rely on potentials, this only applies to Dijkstra and BiDijkstra in *accepted* rounds. Fig. 6.10 show similar trends observed in Fig. 6.9.[10]

While Dijkstra seems to perform better in the median, on average, BiDijkstra still outperforms Dijkstra as it has far fewer and smaller extremes. For example, ignoring all datapoints of ≤ 2 potential updates actually leads to a smaller median for BiDijkstra.

[10]The complete data can be seen in Figs. B.11 to B.13.

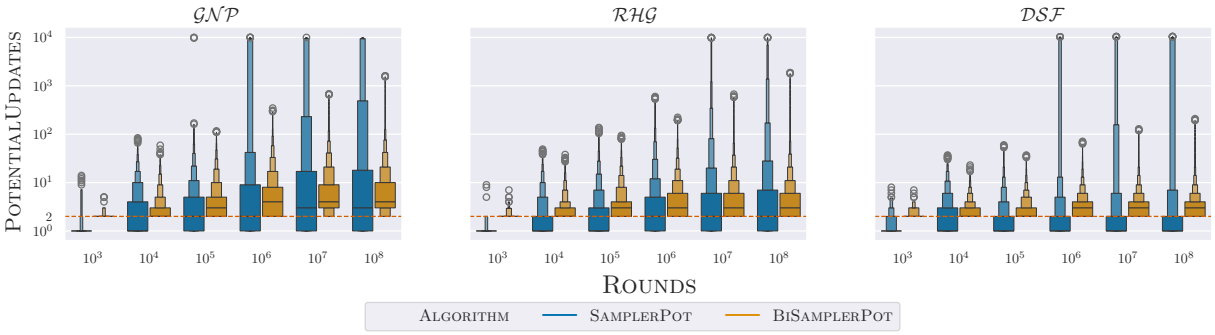


Figure 6.10: Number of Potential-Updates per algorithm for different graphs with $n = 10000$ nodes and average degree $\overline{d} = 10$ with starting weights w_{max}.

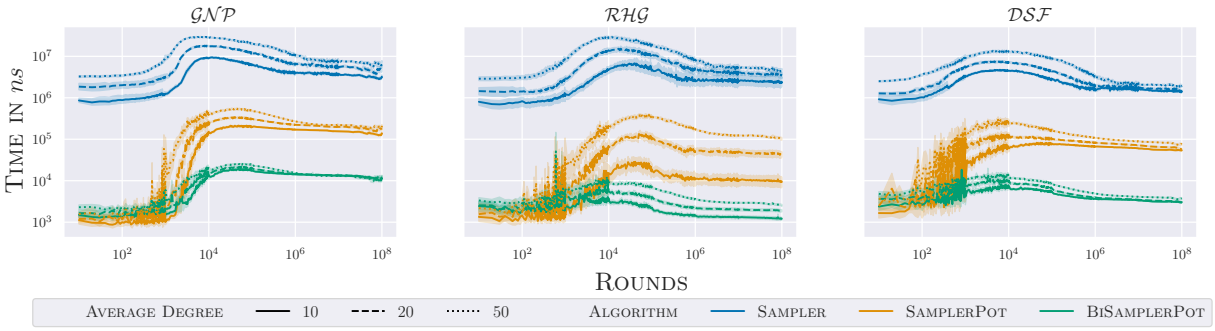


Figure 6.11: Runtime per round over time of different algorithms for different graph models with $n = 10000$ nodes and average degree $\overline{d} \in \{10, 20, 50\}$ with initial weight function w_{max}.

The occurences of extremes are further amplified by higher average degrees as a potential change can incur significantly more cascading potential changes. The choice of an initial weight function also impacts the number of potential updates as for w_{max}, both algorithms had a significant number of direct *accepts* (1 or 2 potential updates) in the first 10^5 rounds: if that is not the case, i.e. for w_{unif}, the median and subsequently the average rises for both algorithms, although only at the beginning. At the end, the number of potential updates seems to be again independent of initial weight function. Note that for w_{zero} there are almost no potential changes in the first 10^6 rounds as almost all accepted weight changes are weight increases while almost all weight decreases are rejected.[11]

6.5.3 Average Runtime

Lastly, we study the performance of our implementations through a time measurement. Fig. 6.11 shows the average runtime per round over time for all three algorithms and different average degrees.[12]

They further emphasize earlier observations for Queue-Insertions and Potential-Updates. BiDijkstra is by far the best algorithm: being roughly 10 times

[11] Remember that in our setting, we have at least 10^5 edges and we expect to have seen every edge with high probability after $m \cdot \log(m) > 10m \geq 10^6$ steps.

[12] Fig. B.14 shows the data for all initial weight functions.

faster on average than Dijkstra and $100 \sim 1000$ faster than BellmanFord. Furthermore, Dijkstra is also $10 \sim 100$ times faster than BellmanFord confirming our expectations.

Also, runtime appears to increase with higher average degrees as more edges are present and thus traversed in the algorithms. Following the previous observations on Queue-Insertions, instances that start with w_{zero} are faster at the beginning for BellmanFord as *rejecting* rounds are computationally faster due to early returns. Surprisingly, algorithms seem to perform better on average on $\mathcal{RHG}$ graphs compared to $\mathcal{GNP}$ and $\mathcal{DSF}$ graphs: probably due to shorter cycles.

The measurements are very noisy at the beginning — probably due to very short rounds on average that lead to many inconsistencies in the very few instances that we run. However, with time, they even out as the MarkovChain settles into a stable state.

Finally, the initial weight function again has no siginificant impact on the runtime of the algorithms.

6.6 Batching

We give a short overview over the performance of Batching with 64 threads compared to BiSamplerPot. Table 6.2 summarizes all total runtimes on all possible input parameters.

Consistent with the results on round-conflicts, regardless of chosen graph or weight function, Batching is multiple times ($64 \sim 190$) slower than BiSamplerPot due to the sheer amount of additional work and overhead. On $\mathcal{RHG}$ and $\mathcal{DSF}$ graphs, the slowdown is twice as bad as on $\mathcal{GNP}$ graphs which indicate that "scale-free" graphs are harder to handle for Batching. While our implementation is not highly optimized and one could probably make the algorithm a bit faster, the sheer difference in performance as well as the theoretical limit imposed by round-conflicts pose significant challenges for Batching to ever reach BiSamplerPot's performance in practice. The additional overhead of parallelization can only then be counteracted if another way of resolving conflicts is found. In the current state, this is not a feasible solution.

6.7 Neighborhood Updates

In our final experiments, we want to compare BiSamplerPot (and SamplerPot) to NodeSampler. While NodeSampler is no true parallel algorithm as it is again sequential in its core, it "parallelizes" weight-updates by performing multiple in one round. Thus, as our goal in the MarkovChain is to permute the weights as much as possible to get a coherent sample of the target distribution, we measure the performance of NodeSampler with regard to succesfull weight-updates. Note that since NodeSampler is siginificantly slower per round than BiSamplerPot or SamplerPot as we perform multiple updates instead of one, we only ran NodeSampler for 10^7 rounds compared to 10^8 for BiSamplerPot and SamplerPot.

Graph	Average Degree	Initial Weights	BiSamplerPot time in ms	Batching time in ms	Average Slowdown
$\mathcal{DSF}$	10	Maximum	10414 ∼ 11257	1796183 ∼ 1925397	172.43
		Uniform	10752 ∼ 11385	1821850 ∼ 1958901	169.408
		Zero	10684 ∼ 11191	1850585 ∼ 1956106	174.093
	20	Maximum	12712 ∼ 13314	2026321 ∼ 2174308	160.966
		Uniform	13494 ∼ 14126	2119665 ∼ 2241692	157.143
		Zero	13389 ∼ 13780	2142839 ∼ 2247288	161.368
	50	Maximum	19154 ∼ 20743	2332546 ∼ 2456868	121.462
		Uniform	20246 ∼ 21162	2445529 ∼ 2568598	121.048
		Zero	19558 ∼ 20160	2449341 ∼ 2564978	125.72
$\mathcal{GNP}$	10	Maximum	33441 ∼ 35232	2187103 ∼ 2322490	65.001
		Uniform	34187 ∼ 35661	2199807 ∼ 2316110	64.215
		Zero	33137 ∼ 34544	2182611 ∼ 2315062	65.882
	20	Maximum	33721 ∼ 34966	2344612 ∼ 2484474	69.878
		Uniform	34453 ∼ 35873	2383430 ∼ 2482532	68.833
		Zero	33026 ∼ 34098	2343049 ∼ 2480880	71.109
	50	Maximum	33689 ∼ 36420	2503518 ∼ 2650046	73.677
		Uniform	34396 ∼ 36178	2496346 ∼ 2614062	72.805
		Zero	29944 ∼ 31783	2333914 ∼ 2514262	79.856
$\mathcal{RHG}$	10	Maximum	3300 ∼ 5704	474456 ∼ 1592833	181.366
		Uniform	3167 ∼ 10026	496726 ∼ 1199446	175.603
		Zero	3219 ∼ 7115	432652 ∼ 1345578	176.604
	20	Maximum	4587 ∼ 12695	717543 ∼ 1324089	171.328
		Uniform	4738 ∼ 7312	689240 ∼ 1619917	190.424
		Zero	4407 ∼ 18096	697825 ∼ 1677583	185.791
	50	Maximum	7698 ∼ 13562	1114689 ∼ 1872249	143.872
		Uniform	7574 ∼ 13659	1069077 ∼ 1864137	146.277
		Zero	6721 ∼ 16275	971226 ∼ 1959928	148.194

Table 6.2: Runtime of BiSamplerPot and Batching (64 Threads) with 10^8 rounds for different graphs with 10^4 nodes.

6.7.1 Queue Insertions per Weight Change

We first compare Queue-Insertions per weight-update for all three algorithms.

See Fig. 6.12 for the results on graphs with average degree 10. The complete dataset can be seen in Fig. B.15. As one would expect, the initial weight function again has no significant impact.

Since SamplerPot and BiSamplerPot operate on the same underlying MarkovChain, the number of succesfull weight updates is similar[13] and the relation between BiSamplerPot and SamplerPot remains similar to those in the previous Queue-Insertion experiment. Although, we now get more insight into the average number of Queue-Insertions as compared to distributions as a

[13]In our case, it is even equal as our experiment for BellmanFord, SamplerPot, and BiSamplerPot (where this data was obtained from) ran on one graph at a time, meaning acceptance of weight-updates was invariant to the sampling algorithm.

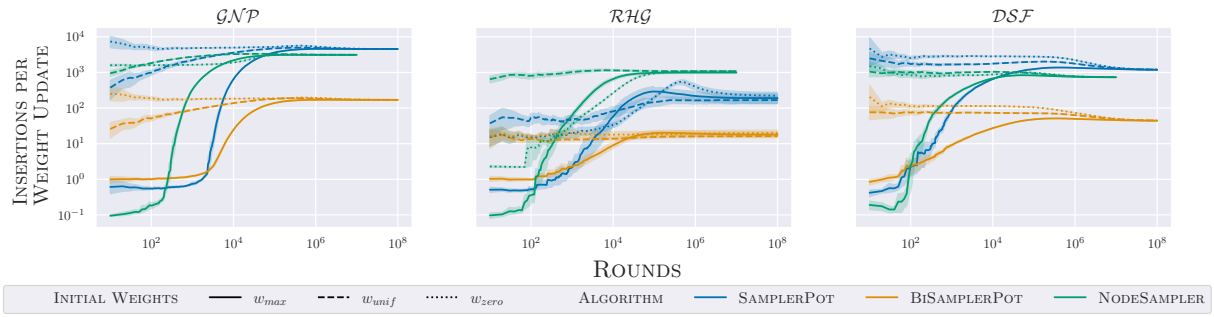


Figure 6.12: Number of Queue-Insertions per succesfull Weight-Update over time of different algorithms on different graphs with $n = 10000$ nodes and average degree $\overline{d} = 10$ for starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

whole. Therefore, it is even more apparent that BiSamplerPot is far better than SamplerPot.

Surprisingly, while BiSamplerPot is also far better than NodeSampler regardless of graph class, NodeSampler is slightly better for $\mathcal{GNP}$ and $\mathcal{DSF}$ graphs whereas for $\mathcal{RHG}$ graphs, SamplerPot slightly has the edge. Nonetheless, BiSamplerPot is almost always at least one order of magnitude better than both SamplerPot and NodeSampler.

This observation also extends to higher average degrees where BiSamplerPot still is by far the best algorithm. Only for $\mathcal{DSF}$ graphs with an average degree of 50, NodeSampler gets reasonably close to BiSamplerPot (although still worse).

For $\mathcal{GNP}$ and $\mathcal{RHG}$ graphs, the order of SamplerPot and NodeSampler is flipped only for average degree 20. Similarly, while NodeSampler is better on $\mathcal{DSF}$ graphs than SamplerPot, for average degree 20 this difference is the smallest among all three (average degrees). This can probably be attributed to the fact that for very small average degrees, the shortest path trees of NodeSampler are not much bigger than those of SamplerPot because the graphs are very sparse. Nonetheless, NodeSampler performs more weight-updates for a single shortest path tree on average. This is no longer true for higher average degrees (20) as now the shortest path trees get bigger which is more noticeable in NodeSampler where the pruning parameter is bigger on average. For even bigger average degrees (50) however, the number of incoming edges of a node and thus also the number of edges for which a weight-change is proposed is significantly higher again. At this point, the number of succesfull weight-updates grew faster than the size of the computed shortest path trees, hence NodeSampler now performing better than SamplerPot again.

6.7.2 Potential Updates per Weight Change

This follow up experiment studies the number of potential updates per succesfull weight-update.

Fig. 6.13 again depicts the results for graphs with average degree 10. See Fig. B.16 for the complete data. Note that since w_{zero} has almost no succesfull weight-decreases at the beginning due to the already very low weight, potential

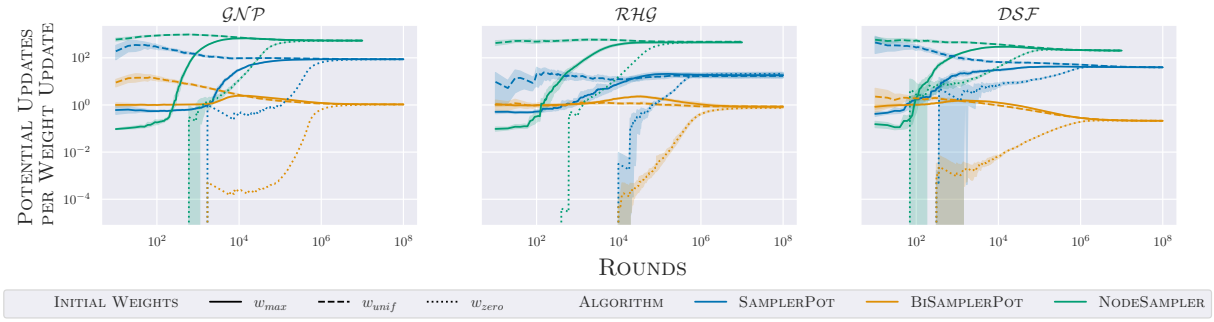


Figure 6.13: Number of Potential-Updates per succesfull Weight-Update over time of different algorithms on different graphs with $n = 10000$ nodes and average degree $\overline{d} = 10$ for starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

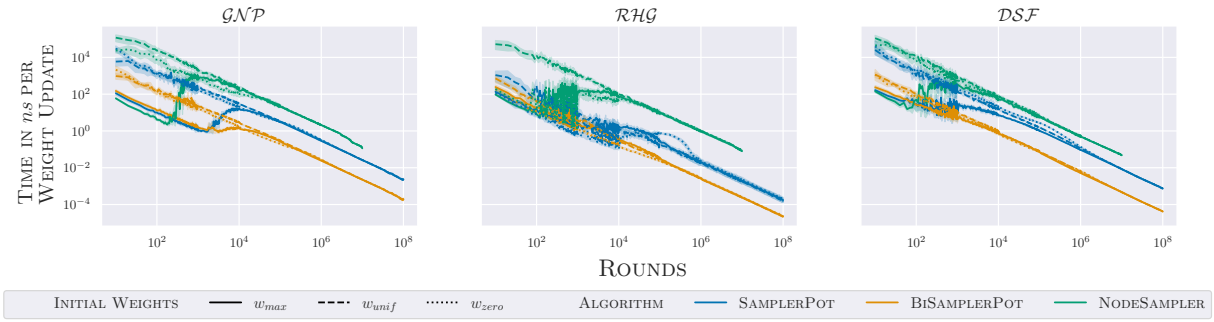


Figure 6.14: Runtime per succesfull Weight-Update over time of different algorithms on different graphs with $n = 10000$ nodes and average degree $\overline{d} = 10$ for starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

updates only happen after roughly 10^3 rounds.

Again, similar trends as for Queue-Insertions per succesfull weight-update can be observed. The only very apparent difference is that now SamplerPot performs far better than NodeSampler across all categories. Since, potential updates are only performed in accepted rounds in SamplerPot, the number of Queue-Insertions in SamplerPot is inflated with rounds where no potential-update was performed afterwards. In comparison, in NodeSampler, we almost always accept at least one weight change which in turn incurs potential-updates.

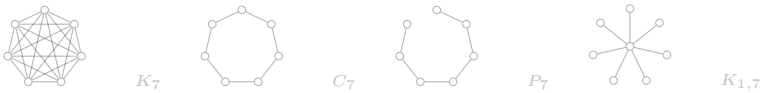
Nonetheless, BiSamplerPot is again by far the best algorithm among all three.

6.7.3 Runtime per Weight Change

The final experiment compares the average runtime per succesfull weight-update (see Figs. 6.14 and B.17).

The results are basically an extension of the previous experiment: BiSamplerPot is again the fastest algorithm, followed by SamplerPot and finally NodeSampler— regardless of average degree or initial weight function. The only upset to this happens in $\mathcal{DSF}$ graphs where NodeSampler appears to be slightly faster at the end than SamplerPot which could warrant further experi-

ments. However, since SamplerPot fares much better in the first 10^6 rounds, this speedup probably does not matter much in practice since SamplerPot is significantly faster in the beginning. Furthermore, BiSamplerPot should still always be the preferred sampling algorithm.



7

Conclusion and Outlook

Conclusion. In this thesis, we introduced a maximum entropy model that allows sampling of signed edge weights for static graph networks which do not introduce negative cycles. In Chapter 4, we proposed a Monte-Carlo-MarkovChain sampler that starts from allowed edge-weights and dynamically permutes edge-weights for a given number of steps. In each step, we ensure that the proposed weight change does not create a negative cycle in the graph. We then showed that this method in fact converges to the desired target distribution of the model.

We then engineer multiple algorithms step-by-step to allow for an efficient practical implementation of this sampling algorithm. Starting with a naive implementation using a very slow BellmanFord, we introduce dynamic node potentials that allow for the use of the much faster Dijkstra before going even one step further and incorporating a bidirectional search BiDijkstra to speed up the algorithm even further.

In Chapter 5, we show that a parallization of the previous algorithms is in fact not simple and requires meticulous techniques to not deviate from the original target distribution. We show that we can parallelize a consecutive number of rounds under certain conditions with optimal work but also that this method is in no way practical. We then propose a naive implementation without any theoretical guarantees before sidestepping the original sampling algorithm and proposing an alternative method NodeSampler that works the same at its core but forces certain edge-weight updates to be consecutive. Nonetheless, we also prove that this new method also converges to the desired target distribution.

In Chapter 6, we conduct a series of experiments on three different random graph models to not only study the behaviour of the proposed algorithms but also of the underlying MarkovChain itself. These range from simple weight distributions and total variation distances to meta metrics such as acceptance rate of the sampler, coverage time, as well as the number of conflicts between two rounds across the algorithm. We benchmark our implementations not only with runtime but also algorithm-specific metrics such as number of Queue-Insertions and Potential-Updates.

Coherent with the shown convergence of the underlying sampling algorithm, we find that the choice of an initial weight functions plays no significant role in the final (distribution of) metrics apart from a few outliers attributed to early rounds where the difference is still very apparent between two initial weight functions. Also, a higher number of edges and therefore a higher number of cycles in a graph lead to worse performances across almost all metrics: higher runtime,

more Queue-Insertions, more Potential-Updates, lower acceptance rate and higher number of conflicts. Nonetheless, the general trend remains unchanged, not only for varying number of edges but also for different graph classes where we can observe similar trends with slight deviations.

We find that BiSamplerPot is by far the best implementation across all metrics and should always be the preferred implementation in practice. Even the parallelized algorithms can not compare to BiSamplerPot and most of the time not even to the much slower SamplerPot.

Outlook. Especially the attempt to parallelize the sampling procedure warrants further investigation and new techniques as in the current state, no parallization-technique can rival the performance of the sequential BiSamplerPot. To this end, the method of NodeSampler could be particularly interesting because it is the only simple case in which two edge-weight changes are guaranteed to not conflict with each other.[1] One could also propose a technique similar to the Configuration model paired with the *edge-switching* technique: draw random edge-weights with possibly negative cycles and randomly *repair* edges to eventually yield the desired target distribution.

It is also possible to parallelize the search in NodeSampler using Δ-Stepping. Since SamplerPot and BiSamplerPot have very small shortest path trees regardless of original graph size, Δ-Stepping provides no benefit. For NodeSampler however, the size of the shortest path tree is expectedly much bigger for bigger graphs as we prune over the maximum broken value (instead of a single one). Hence, for NodeSampler, Δ-Stepping might be beneficial at some point.[2]

Another big point of study are the theoretical bounds on the mixing time of this sampling algorithm. We provide a proof in Appendix A.2 that this method is in fact rapidly mixing on the n-Cycle but the bound is far from practical and not easily extendable to general graphs. Thus, a general bound on the mixing time of the underlying MarkovChain is definitely an important next step. A generalization to certain graph classes would also be a considerable progress.

We also provide a general algorithm in Appendix C for computing consistent potentials that builds upon the technique of NodeSampler but show that this is no better than current techniques in practice. Nonetheless, further investigation in this direction could prove beneficial and maybe even speed up state-of-the art algorithms. At this point however, this seems unlikely.

Finally, and probably most important, we could further extend the experiments. Not only in quantity but also in quality. Currently, we only investigate random graph models with low diameter whereas road networks in the real world have a bigger diameter. These road networks are certainly a point of interest for this maximum entropy model as laid out in Chapter 1. Due to time and space restrictions, they were not included as part of the experiments. In [56], we ran some of the experiments of this thesis on a real-world road network.

[1] Apart from edge-weight changes in two different SCCs.

[2] [39] proposed a practical optimization that runs Bfs to collect (at most) 4096 nodes as the neighborhood of a node. Thus, running a pruned Δ-Stepping on a graph with only $n = 10000$ nodes has no practical benefit and was thus not considered.

Additional Proofs

Although the following two sections are not part of my work, they were obtained as part of the research on [56]. I consider them too important to not be added at least in the appendix even though they are in no way my direct contribution. I also worked extensively on the problem in Appendix A.2, however with no significant results.

A.1 Convergence of the General MCMC [56]

The underlying Markov Chain of Algorithm 4 is defined on the state space $\mathbb{S}_{G,\mathcal{W}}$. In the proof of Theorem 4.12, we assumed that $\mathcal{W}$ is countable and that we can assign a non-zero probability to every value in $\mathcal{W}$, i.e. $\frac{1}{|\mathcal{W}|}$. However, if $\mathcal{W}$ is uncountable, e.g. we have real-valued $\mathcal{W} = [a, b]$, the state space $\mathbb{S}$ might also be uncountable and the proof of Theorem 4.12 does not apply. Not only that but also the notions of aperiodicity and irreducibility do not apply to such a Markov Chain. For that, we need similar but more general definitions. We thus assume for the remainder of this section that $\mathcal{W}$ consists of an arbitrary finite union of intervals.

A Markov Chain on a measureable set $\mathcal{X}$ is defined by transition probabilities $P(x, dy)$ where for each $x \in \mathcal{X}$ and measureable subset $A \subseteq \mathcal{X}$, the value $P(x, A)$ is the probability to move from x to somewhere in A. We again denote i steps of the Markov Chain by $P^{(i)}$. Starting with $P^{(1)}(x, dy) = P(x, dy)$, higher-order transition probabilities are then given by

$$P^{(n+1)}(x, A) = \int_{\mathcal{X}} P^{(n)}(x, dy) P(y, A).$$

▷ Definition A.1 (ϕ-irreducibility [92]). *A general Markov Chain is ϕ-irreducible if there exists a non-zero σ-finite measure ϕ on $\mathcal{X}$ such that for all $A \subseteq \mathcal{X}$ with $\phi(A) > 0$, and for all $x \in \mathcal{X}$, there exists a positive integer $n = n(x, A)$ such that $P^{(n)}(x, A) > 0$.*

▷ Definition A.2 (Aperiodicity [92]). *A general Markov Chain with stationary distribution $\pi(\cdot)$ is aperiodic if there do not exist $d \geq 2$ and disjoint subsets $\mathcal{X}_1, \ldots, \mathcal{X}_d \subseteq \mathcal{X}$ with $P(x, \mathcal{X}_{i+1} = 1$ for all $x \in \mathcal{X}_i$ ($i \in [d-1]$), and $P(x, \mathcal{X}_1) = 1$ for all $x \in \mathcal{X}_d$ such that $\pi(\mathcal{X}_1) > 0$.*

▷ Theorem A.3 (Theorem 4 of [92]). *If a Markov chain on a state space with countably generated σ-algebra is ϕ-irreducible and aperiodic and has a stationary distribution $\pi(\cdot)$, then for π-a.e. $x \in \mathcal{X}$,*

$$\lim_{n \to \infty} \|P^n(x, \cdot) - \pi(\cdot)\| = 0.$$

Verifying Theorem A.3 for our proposed MC yields a proof for the general version of Theorem 4.12.

▷ Theorem A.4 (Generalization of Theorem 4.12). *Let $G = (V, E)$ be a directed graph and $\mathcal{W}$ a real weight interval. Then, the MCMC process converges to a uniform density on $\mathbb{S}_{G,\mathcal{W}}$.*

Proof. Subsets of $\mathbb{R}^m$ equipped with the standard Borel σ-algebra are countably generated by open balls with rational centers and rational radii [92].

Furthermore, since the Markov chain is symmetric, detailed balance is fulfilled, i.e. for probability densities s, s' it holds

$$\pi(s)P(s, s') = \pi(s')P(s', s),$$

for a stationary density π and in particular for the uniform density. By symmetry it holds $\int_{x\in A} P(x, B)dx = \int_{y\in B} P(y, A)dy$ for measurable $A, B \subseteq \mathcal{X}$ which extends to a weighting of π when chosen as the uniform density. Here, stationary follows directly from detailed balance, i.e. integrating over $\mathcal{X}$ yields

$$\begin{aligned}\int_{\mathcal{X}} \pi(x)P(x, y)dx &= \int_{\mathcal{X}} \pi(y)P(y, x)dx \\ &= \pi(y) \int_{\mathcal{X}} P(y, x)dx \\ &= \pi(y).\end{aligned}$$

With the existence of π, it remains to verify ϕ-irreducibility and aperiodicity. Let $x \in \mathbb{S}_{G,\mathcal{W}}$ and $\pi(A) > 0$. We emulate the process of changing the weights to values exceeding some of the values of A (with positive measure) to subsequently enable a move to A with positive probability. Due to continuity there exists constant $\varepsilon > 0$ where some point $y \in A$ emits $B(\varepsilon, y) \subseteq A$. Now, with positive probability we successively change the coordinates of x from smallest to largest to a value that is larger than any coordinate of any point in $B(\varepsilon, y)$ (guaranteed to exist due to continuity) and returning to a value in $B(\varepsilon, y)$ after a total of $2m$ steps, yielding

$$P^{(2m)}(x, A) \geq P^{(2m)}(x, B(\varepsilon, y)) > 0,$$

where π is a probability measure and as such non-zero and σ-finite.

Suppose $\mathcal{X}_1$ and $\mathcal{X}_2$ are disjoint subsets of $\mathcal{X}$ both of positive π measure and wlog. $\mathbf{0} \in \mathbb{S}_{G,\mathcal{W}} \cap \mathcal{X}_1$, with $P(x, \mathcal{X}_2) = 1$ for all $x \in \mathcal{X}_1$. This essentially represents an emigration from $\mathcal{X}_1$ to $\mathcal{X}_2$ with probability 1. Observe that in case G contains at least one cycle, the probability to introduce a negative cycle is positive, hence $P(\mathbf{0}, \mathcal{X}_1) > 0$ leads to a contradiction as therefore $P(\mathbf{0}, \mathcal{X}_2) < 1$.

Otherwise, since no cycles exist in G, the set $\mathbb{S}_{G,\mathcal{W}}$ coincides with $\mathcal{W}^m$ and the MCMC process degenerates to choosing uniform values in $\mathcal{W}$ and assigning them to randomly sampled edges. Since $\pi(\mathcal{X}_1) > 0$ by definition we can find a subset $A \subset \mathcal{X}_1$ where for $x \in A$ it holds $P(x, \mathcal{X}_1) \geq P(x, A) > 0$. □

A.2 Rapid-Mixing on the n-Cycle [56]

We showed in Theorems 4.12 and A.3 that our algorithms converge to a uniform distribution over $\mathcal{W}$ and saw in Chapter 6 that in practice this happens in

reasonable polynomial time. However, we still lack a theoretical upper bound on the mixing time. As such proofs are generally hard and very involved, we restrict ourselves to the n-Cycle C_n and prove that our MCMC is rapidly mixing on it for a set of integer weights $\mathcal{W} = [a, b]$.

Consider an ergodic Markov chain M with state space[1] $\mathbb{S}$ and stationary distribution π (i.e. the unique configuration the ergodic M converges to). The mixing time τ_ε of M describes the smallest number of steps to reach a distribution that has at most ε distance from π (in L_1 norm) — even for an adversarial starting configuration. Formally, the mixing time is defined as

$$\tau_\varepsilon = \min\Big\{\tau \geq 0 : \max_{x\in\mathbb{S}} \Big[\sum_{y\in\mathbb{S}} |\pi(y) - \sigma_{\tau,x}(y)| \leq \varepsilon\Big]\Big\},$$

where $\sigma_{\tau,x}$ is the distribution of states after τ steps when starting on state x.

To show rapid mixing of the Markov chain underlying Algorithm 4 for the n-cycle, we consider a variant that is simpler to analyze. This chain $\mathcal{M}$ is defined as follows. Its state space $\mathbb{S}$ is the state space of the chain underlying Algorithm 4. From the current state, the chain $\mathcal{M}$ transitions to the next state as follows:

- With probability $1/2$: remain in the current state.
- With probability $1/4$: choose edge e and $b \in \{-1, 1\}$ uniformly at random. If adding b to the weight $w(e)$ yields a state in $\mathbb{S}$, move to this state.
- With probability $1/4$: choose edges e_1, e_2 uniformly at random. If incrementing $w(e_1)$ and decrementing $w(e_2)$ results in a state in $\mathbb{S}$, move to this state.

Since the same transitions are performed by the chain underlying Algorithm 4 with a probability bounded from below by a polynomial in n and $(b - a)$, the mixing time of both chains can be related by a polynomial in n and $(b - a)$ (cf. [72, Remark 13.19]).

The main result of this section is as follows.

▷ Theorem A.5. *The mixing time τ_ε of the Markov chain $\mathcal{M}$ for integer weights in $[a, b]$ on the n-cycle satisfies $\tau_\varepsilon \leq 2^9 n^{11} b^2 (b - a) \left(\log(b - a) + \log \varepsilon^{-1}\right)$.*

We employ a common argument to show Theorem Theorem A.5. Let $1 = \lambda_0 > \lambda_1 \geq \cdots \geq \lambda_{|\mathbb{S}|} > -1$ denote the eigenvalues of the transition matrix P of $\mathcal{M}$. Since $\mathcal{M}$ is lazy, e.g. we remain in the same state with probability at least $1/2$, all eigenvalues are non-negative, and it holds that $\tau_\varepsilon \leq (1 - \lambda_1)^{-1} \left(\log |\mathbb{S}| + \log \varepsilon^{-1}\right)$, e.g. [95, p. 4]. Thus it suffices to bound $1 - \lambda_1$ from below.

Our idea is to decompose $\mathcal{M}$ into multiple Markov chains $\mathcal{M}'$ and $\mathcal{M}_i$ where $i \in [0, nb]$. Intuitively, $\mathcal{M}'$ moves between states that correspond to the different sums of the weights in the n-cycle, and each $\mathcal{M}_i$ moves between states that correspond to the same sum i. Precisely, we partition $\mathbb{S}$ into the sets $\Omega_0, \ldots \mathbb{S}_{nb}$ where $\mathbb{S}_i$ contains all states with a sum of weights i (n.b. since we consider a cycle graph, the sum of consistent weights is always non-negative). We then define the chain $\mathcal{M}_i$ with state space $\mathbb{S}_i$ and transition matrix $P_i(x, y) = P(x, y)$ where

[1] Commonly, this is denoted by Ω in existing literature as opposed to $\mathbb{S}$.

$x \neq y \in \mathbb{S}_i$ (with the total remaining probability, the chain stays in the current state). In addition, we define the chain $\mathcal{M}'$ with state space $\mathbb{S}' = [0, nb]$ and transition matrix $P'(i,j) = \sum_{x\in\mathbb{S}_i}\sum_{y\in\mathbb{S}_j} P(x,y)$ if $|i-j|=1$ where $i,j \in \mathbb{S}'$ and where again, with the total remaining probability, the chain stays in the current state.

▷ Lemma A.6 (Corollary 3.3 of [75]). *Let $\beta > 0$ and $\gamma > 0$ such that $P(x,y) \geq \beta$ for all $x, y \in \mathbb{S}$ where $P(x,y) > 0$, and $\pi(\delta_i(\mathbb{S}_j) \geq \gamma\pi(\mathbb{S}_i)$ for all $i \neq j$ where $P(i,j) > 0$ and $\delta_i(\mathbb{S}_j)$ is the subset of states in $\mathbb{S}_j$ where $P(i,j) > 0$. Then* $\mathrm{Gap}(P) \geq \beta\gamma\,\mathrm{Gap}(P')\min_i \mathrm{Gap}(P_i)$.

▷ Lemma A.7 (Proposition 6 of [37]). *For a MC with state space $\mathbb{S}$ and transition matrix P, define h as*

$$h = \min_{\substack{X\subset\mathbb{S}\\ \pi(X)\leq 1/2}} \frac{\sum_{x\in X}\sum_{y\in\mathbb{S}\setminus X}\pi(x)P(x,y)}{\pi(X)},$$

where $P(x,y)$ is the probability of moving from x to y and $\pi(X) = \sum_{x\in X}\pi(x)$. Then, $1-2h \leq \lambda_1 \leq 1-h^2$.

Proof of Theorem A.5. By Lemma A.6, the result follows by showing sufficient lower bounds for $\beta, \gamma, \mathrm{Gap}(P')$ and $\min_i \mathrm{Gap}(P_i)$.

We start with $\min_i \mathrm{Gap}(P_i)$. A generalized variant of this kind of chain called the load-exchange Markov chain has been analyzed in [6]. In fact, by combining [6, Theorem 4.2] and [6, Cor. 2.5], it immediately follows that $\min_i \mathrm{Gap}(P_i) \geq 1/(n^3(b-a)2)$.

To bound $\mathrm{Gap}(P')$, we use Lemma A.7. Consider any adversary set of states $X \subset \mathbb{S}'$ as required for Lemma A.7. Observe that if $i \in X$ is a state such that $P'(i,j) > 0$ for some $j \notin X$, then $P'(i,j) \geq 1/8n$. In addition, for any i, $\pi(i)$ is proportional to the number of weights on the n-cycle which sum to i. Precisely, this number is given by the polynomial coefficient $\binom{n}{i}_{b-a+1}$, see for example [27,34]. For our purposes, it suffices to observe that $\pi(i)$ has a peak around some value i, and monotonically decreases in both directions. Now, let i^* be the state in X such that $\pi(i^*)$ is maximal, and assume that the only state i such that $P'(i,j) > 0$ for some $j \notin X$ is a state $i < i^*$ (the other cases all result in better bounds or are symmetric). Then it holds that $\pi(0) \leq \pi(1) \leq \cdots \leq \pi(i) \leq \cdots \leq \pi(i^*)$. Moreover, Lemma A.7 requires $\pi(0) + \pi(1) + \cdots + \pi(i-1) \geq 1/2$, and thus $\pi(i) \geq 1/2nb$ and $\pi(i)/\pi(X) \geq 1/nb$. Combining this estimate with the bound on $P'(i,j)$ then gives $\mathrm{Gap}(P') = 1 - \lambda_1' \geq h^2 \geq 1/64n^4b^2$.

Finally, observe that $\beta \geq 1/4n^2$ and $\gamma \geq 1/n$, which concludes the proof. □

B

Supporting Figures

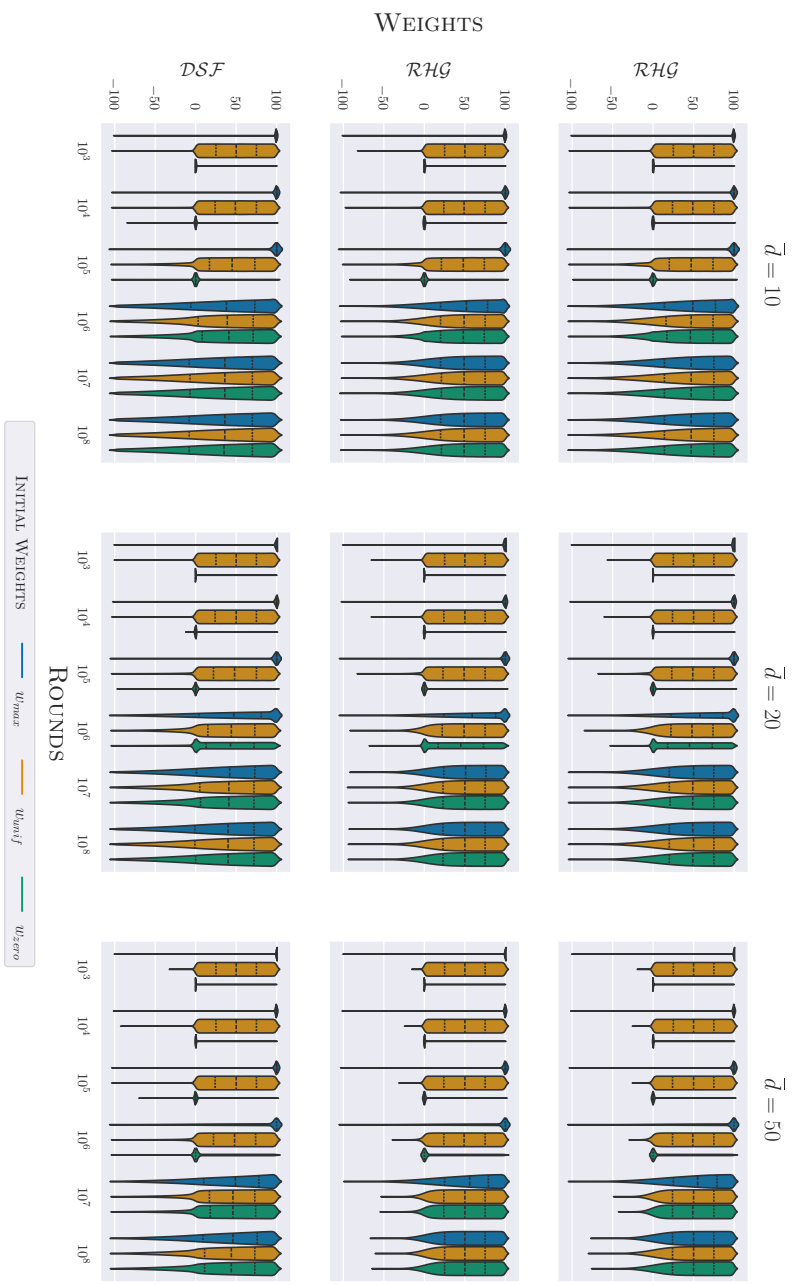


Figure B.1: Weight distributions over time for different graphs with $n = 10000$ nodes and average degrees $\overline{d} \in \{10, 20, 50\}$.

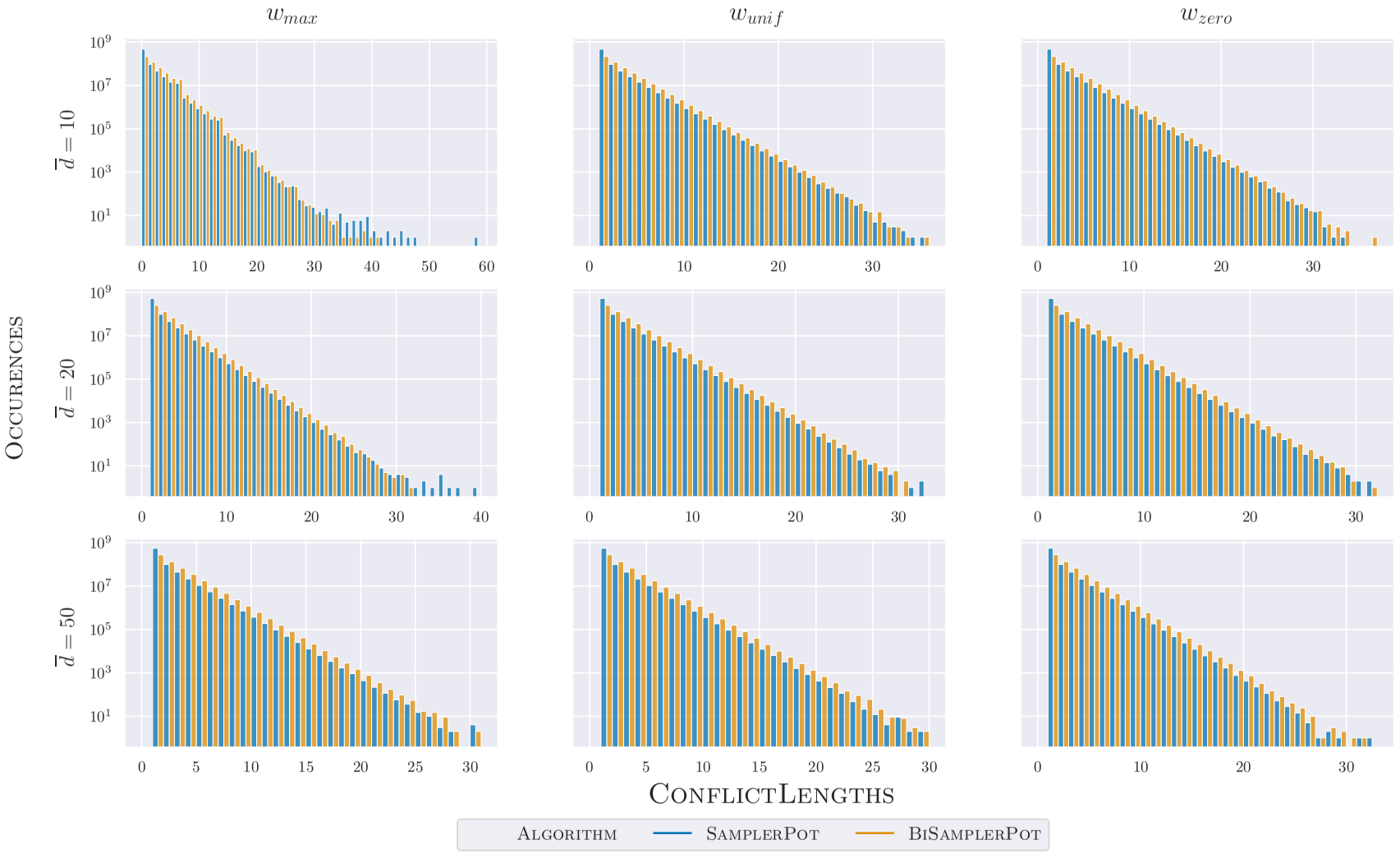


Figure B.2: Occurences of ConflictLengths between rounds after 10^8 rounds for $\mathcal{GNP}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

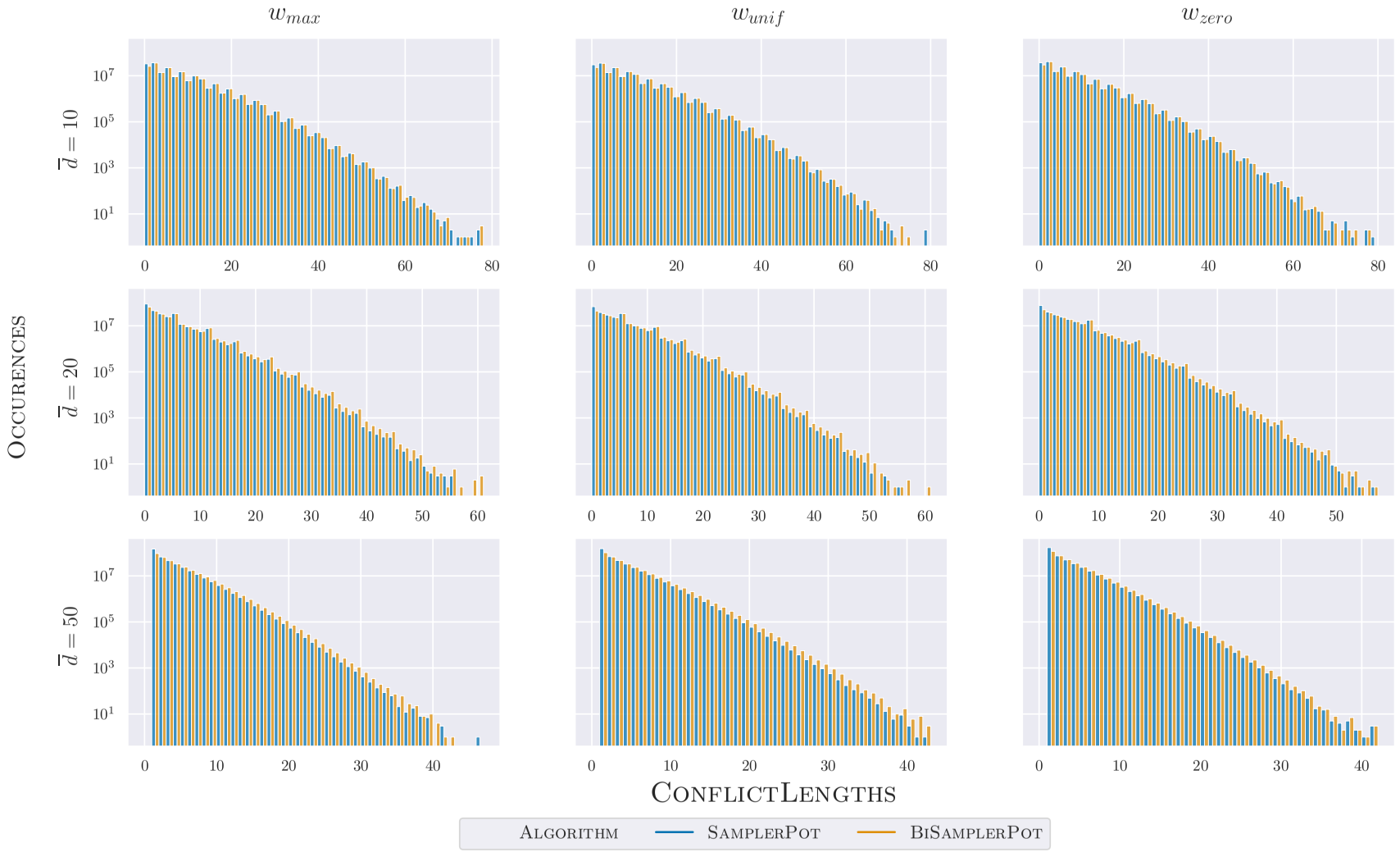


Figure B.3: Occurences of ConflictLengths between rounds after 10^8 rounds for $\mathcal{RHG}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

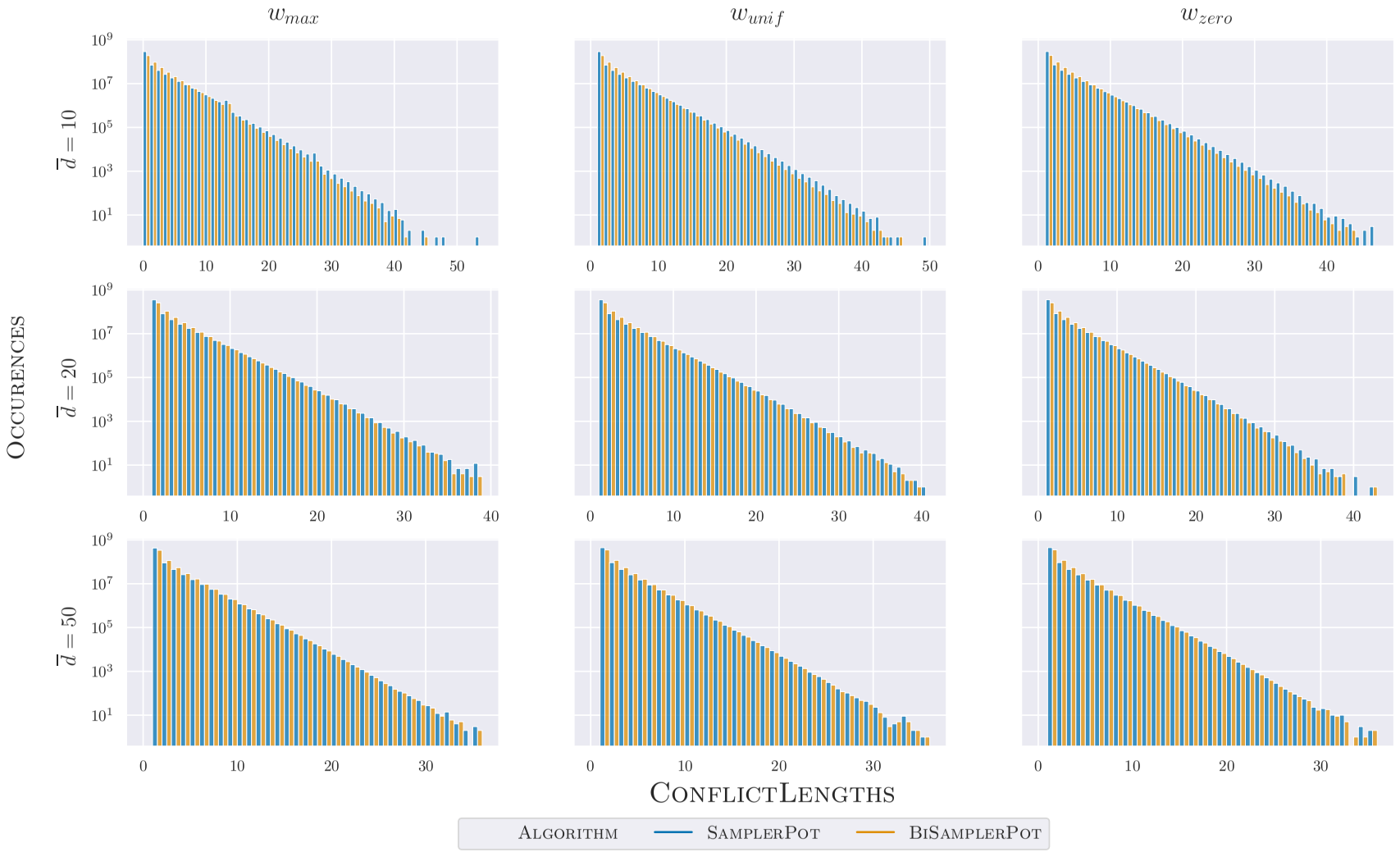


Figure B.4: Occurences of ConflictLengths between rounds after 10^8 rounds for $\mathcal{DSF}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

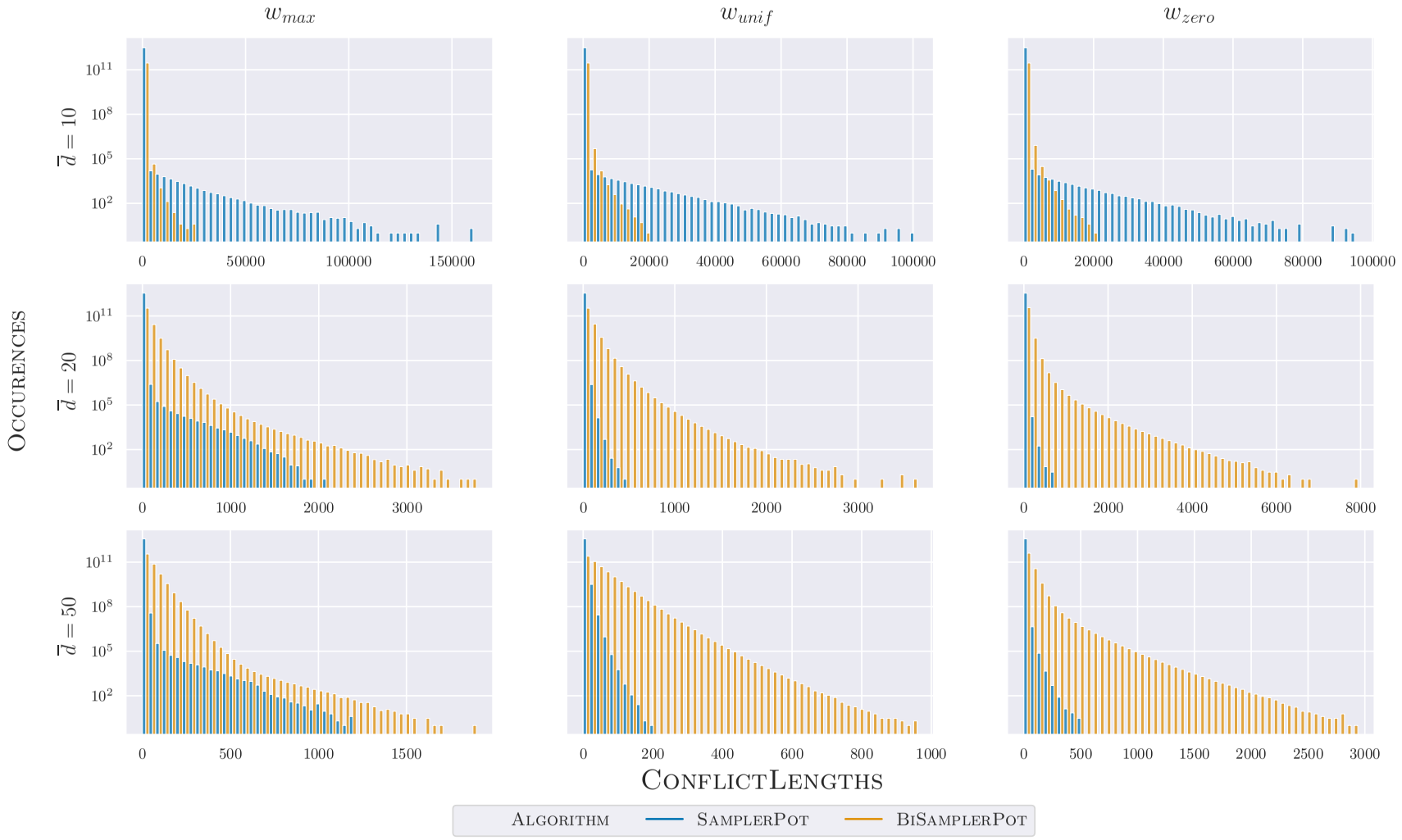


Figure B.5: Occurences of ConflictLengths between nodes after 10^8 rounds for $\mathcal{GNP}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

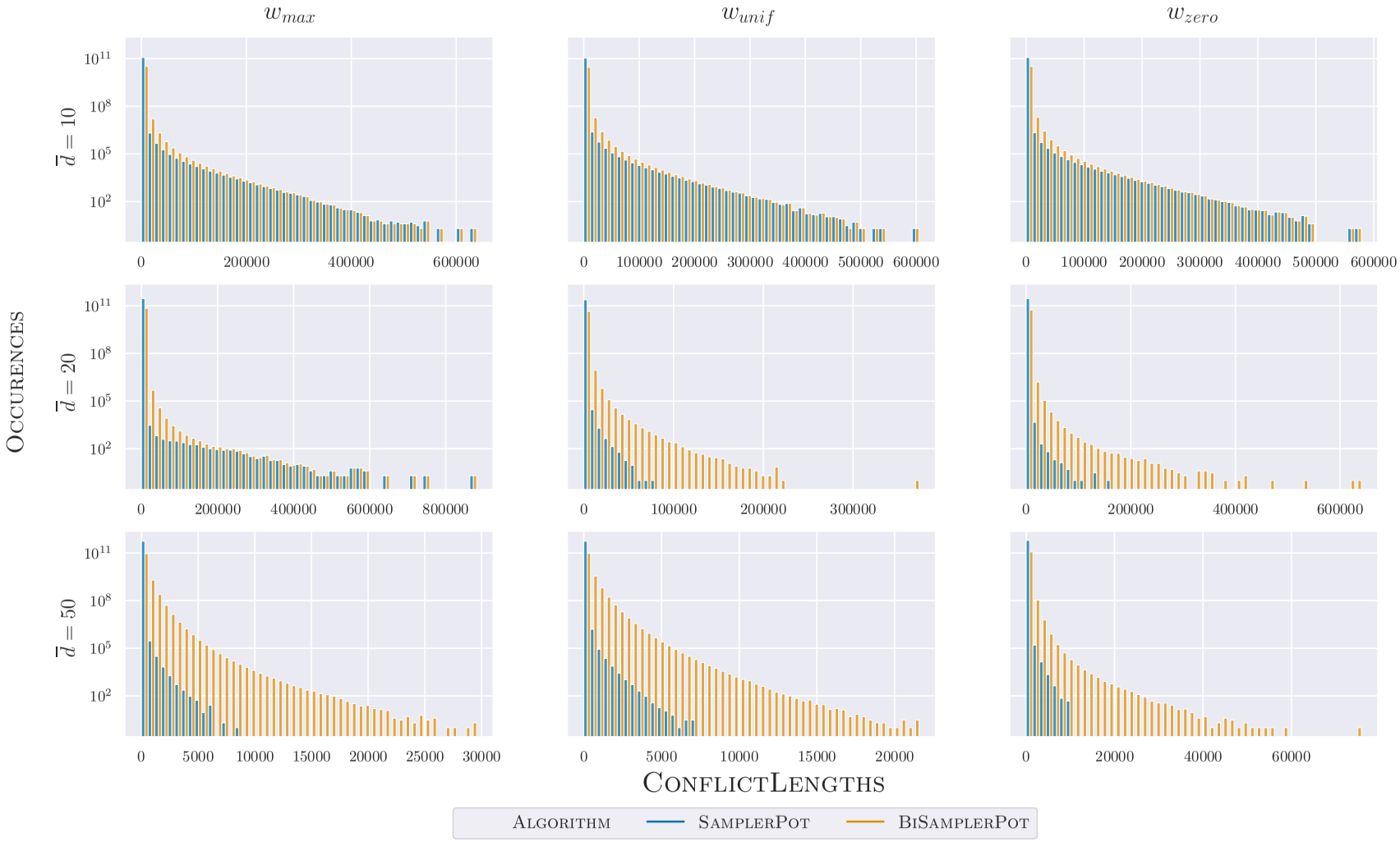


Figure B.6: Occurences of ConflictLengths between nodes after 10^8 rounds for $\mathcal{RHG}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

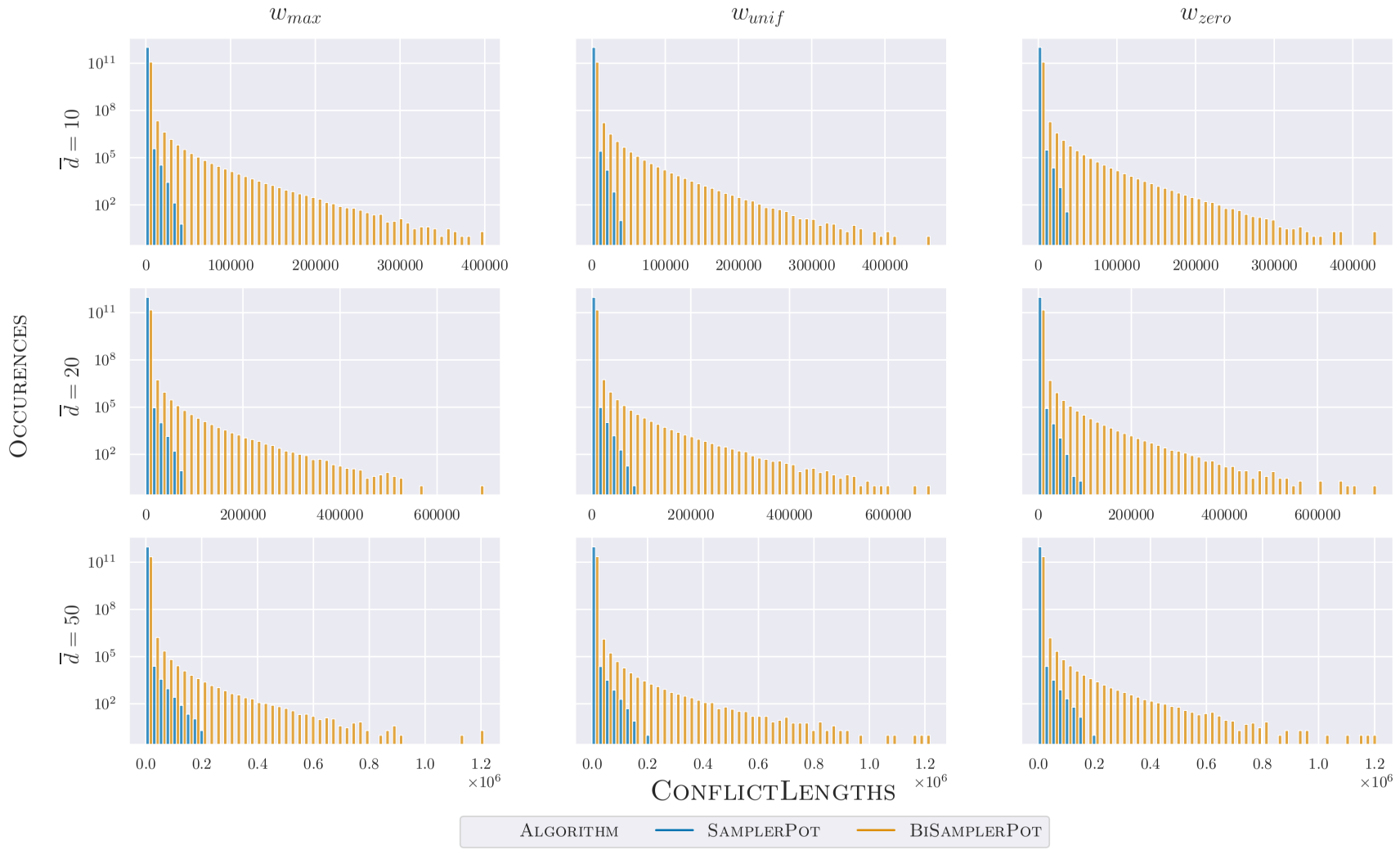


Figure B.7: Occurences of ConflictLengths between nodes after 10^8 rounds for $\mathcal{DSF}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

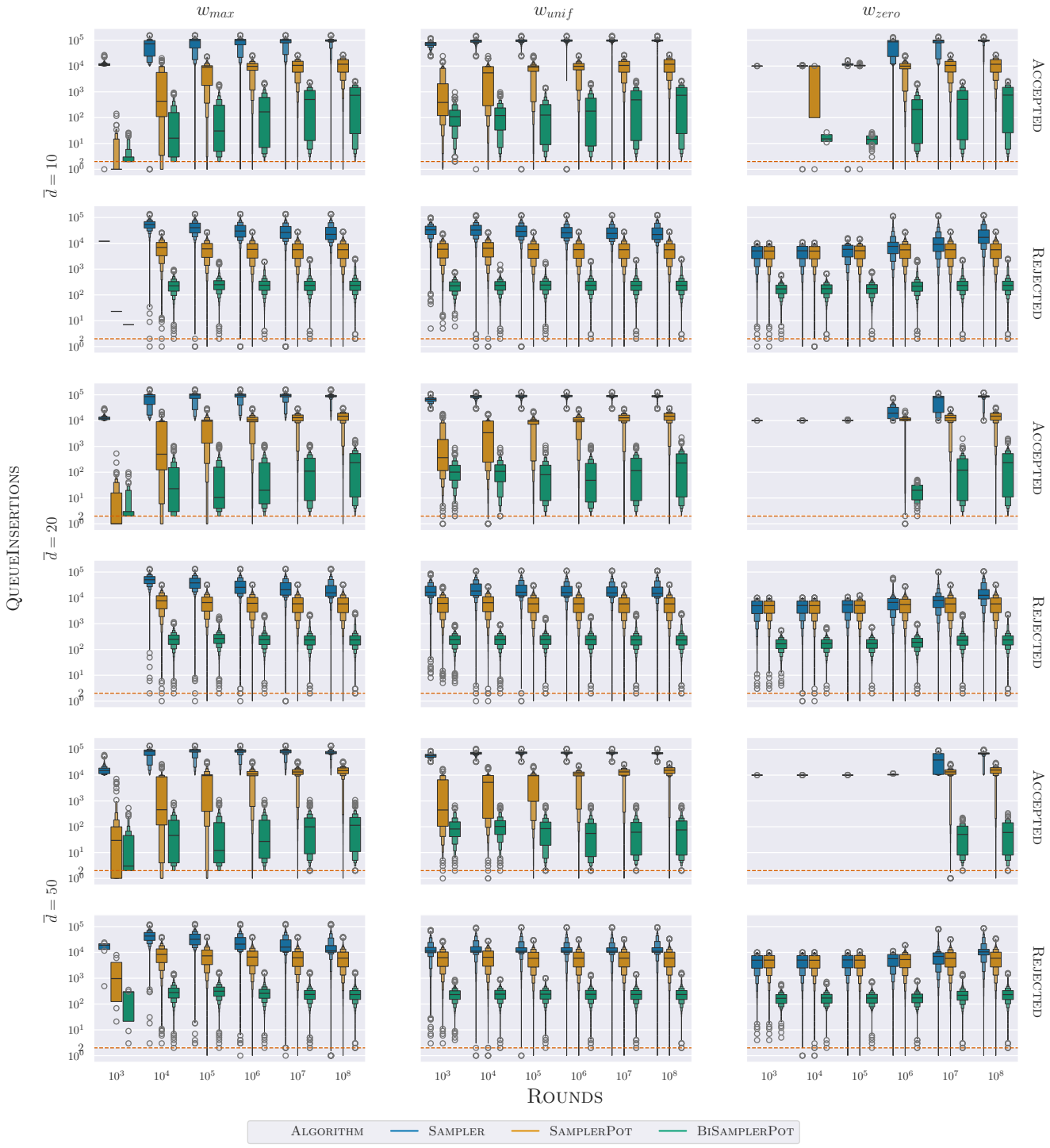


Figure B.8: Number of Queue-Insertions per algorithm for $\mathcal{GNP}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting functions $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

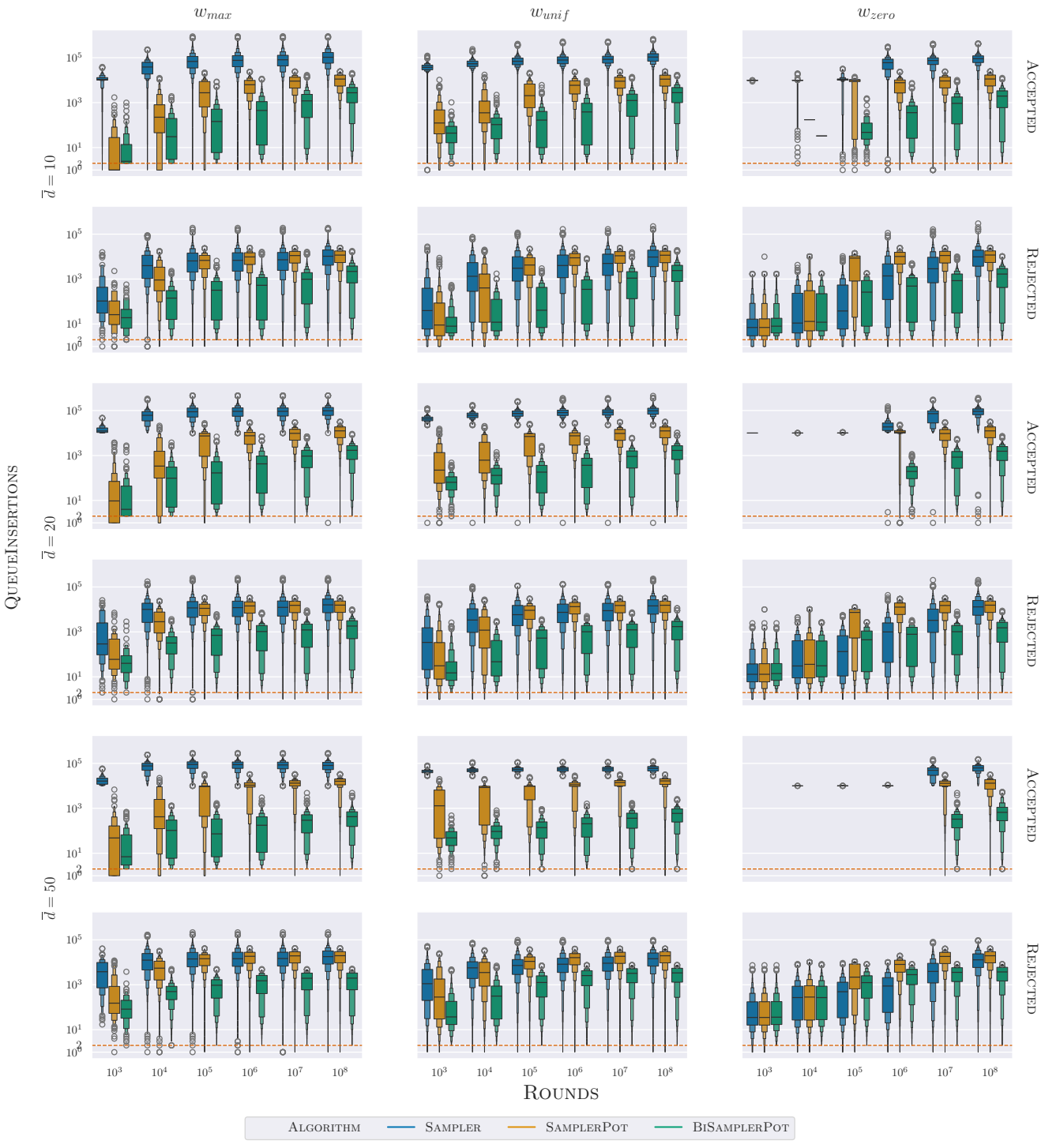


Figure B.9: Number of Queue-Insertions per algorithm for $\mathcal{RHG}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting functions $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

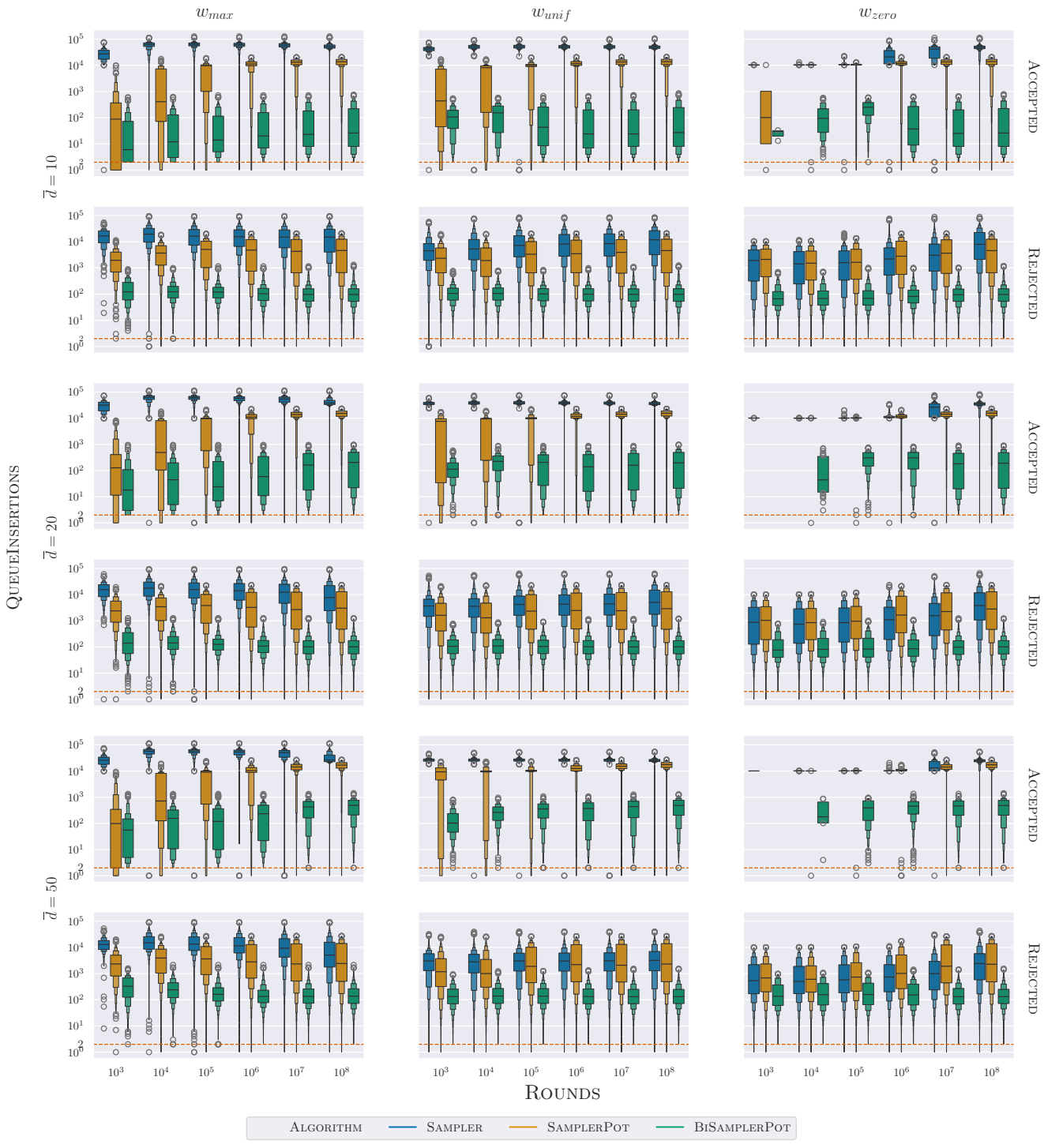


Figure B.10: Number of Queue-Insertions per algorithm for $\mathcal{DSF}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting functions $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

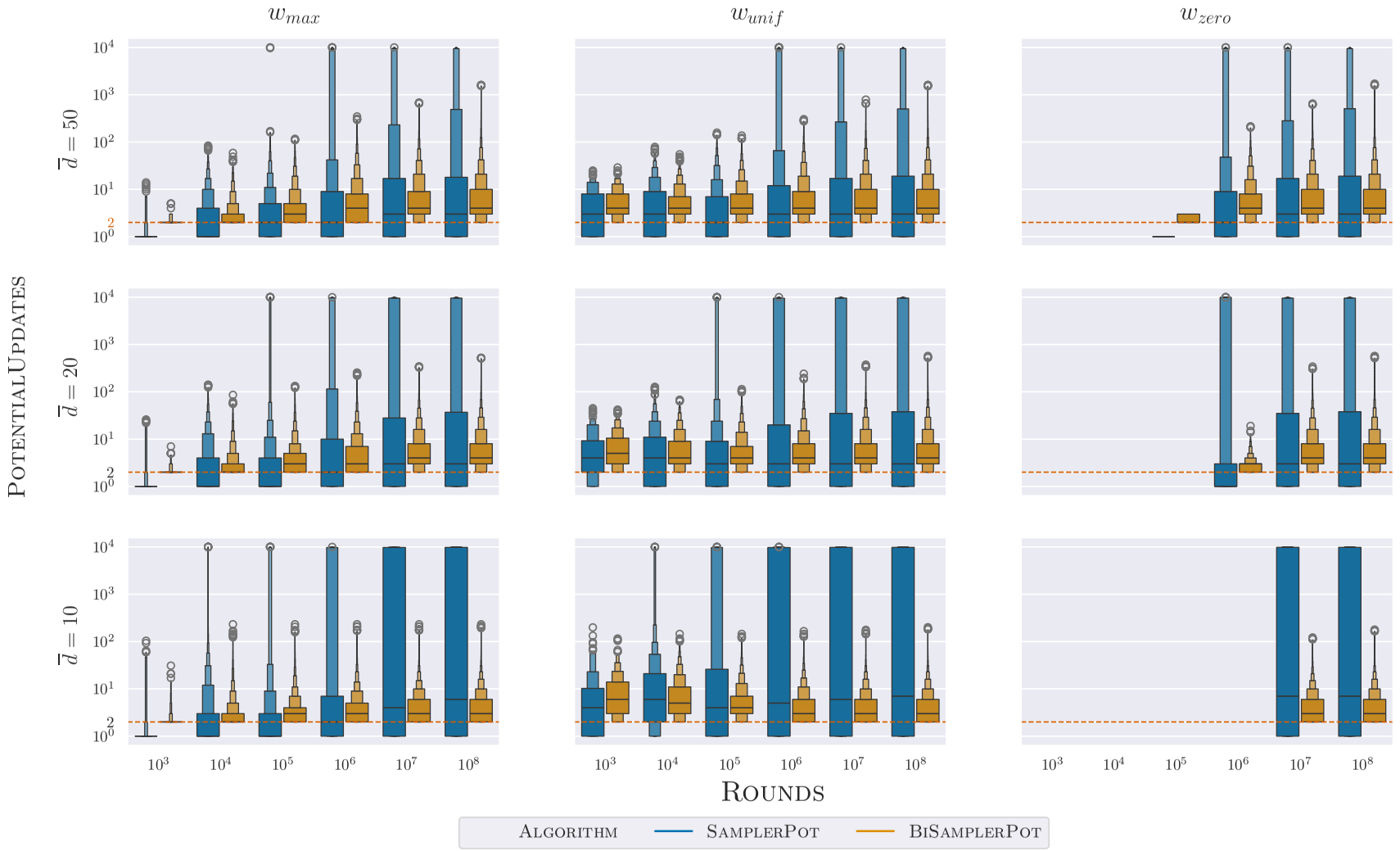


Figure B.11: Number of Potential-Updates per algorithm for $\mathcal{GNP}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting functions $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

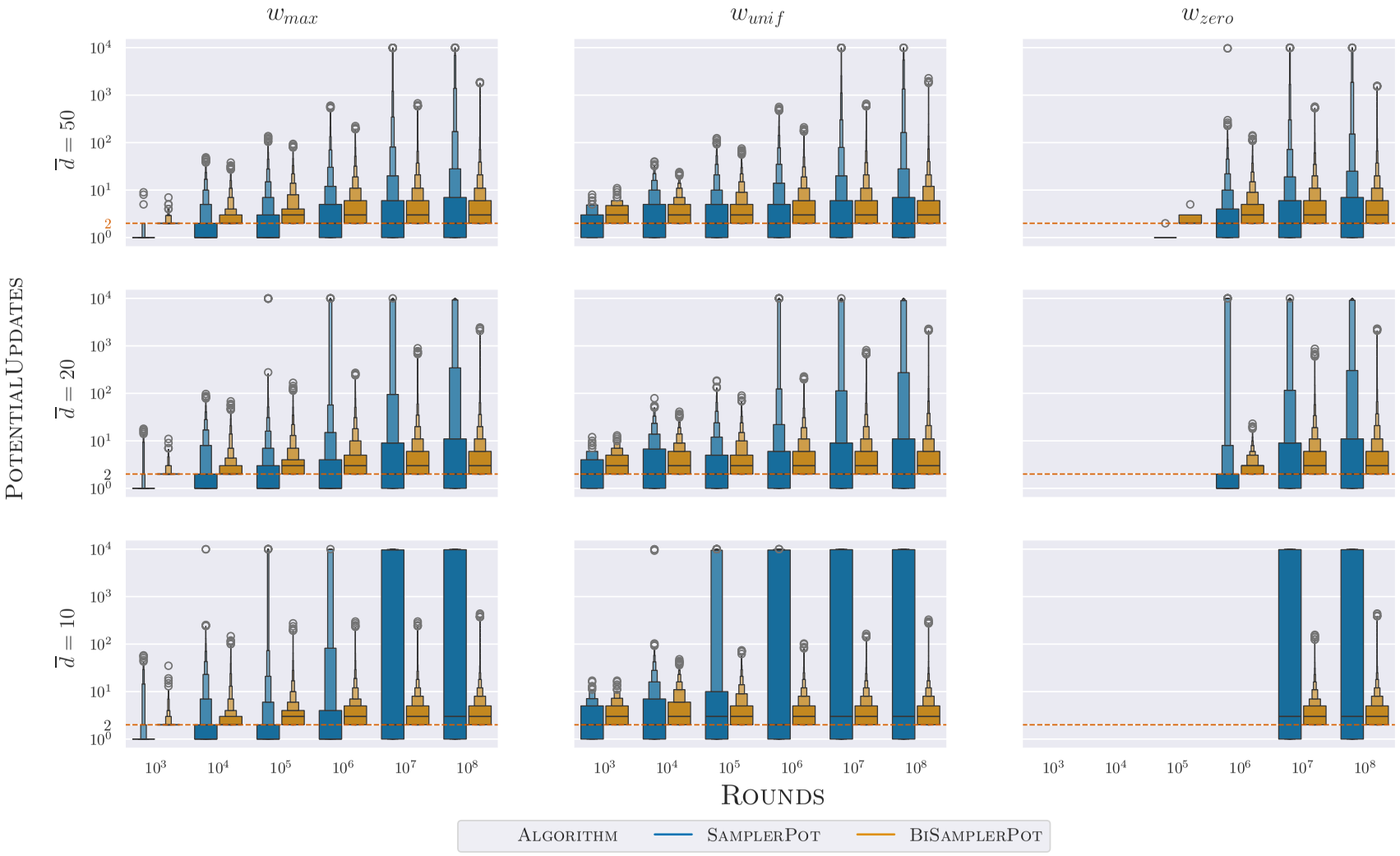


Figure B.12: Number of Potential-Updates per algorithm for $\mathcal{RHG}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting functions $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

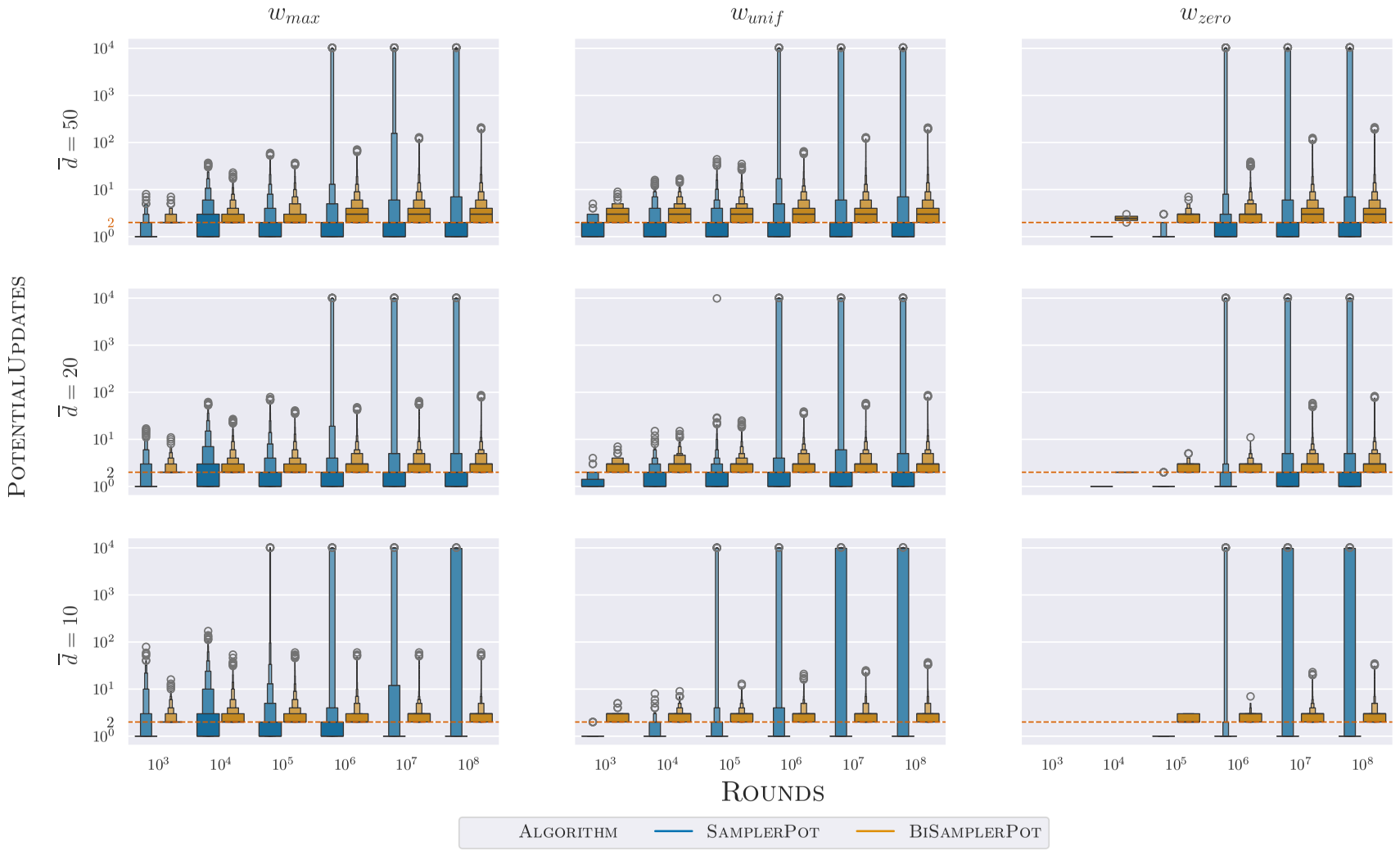


Figure B.13: Number of Potential-Updates per algorithm for $\mathcal{DSF}$ graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting functions $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

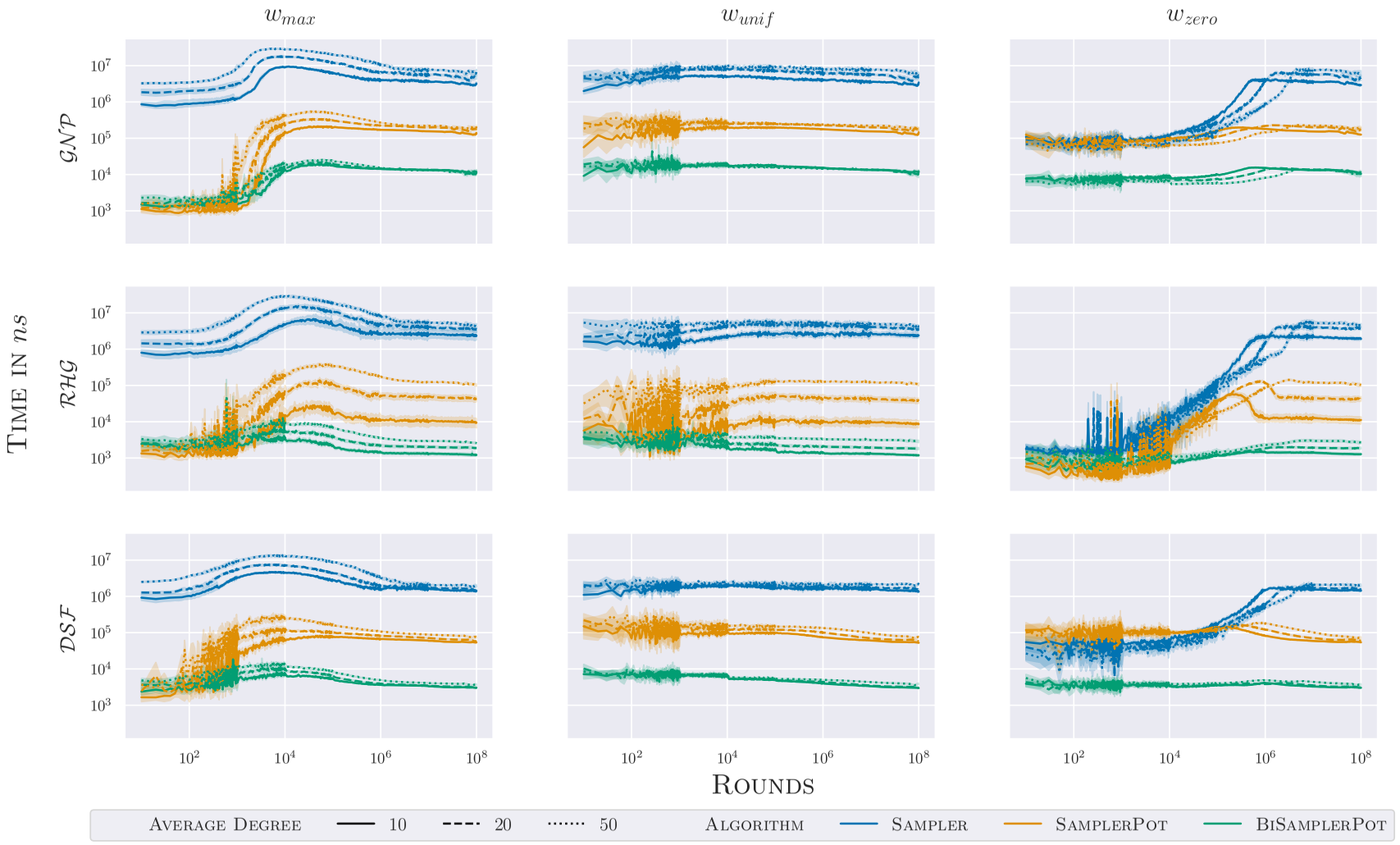


Figure B.14: Runtime per round over time of different algorithms for different graph models with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

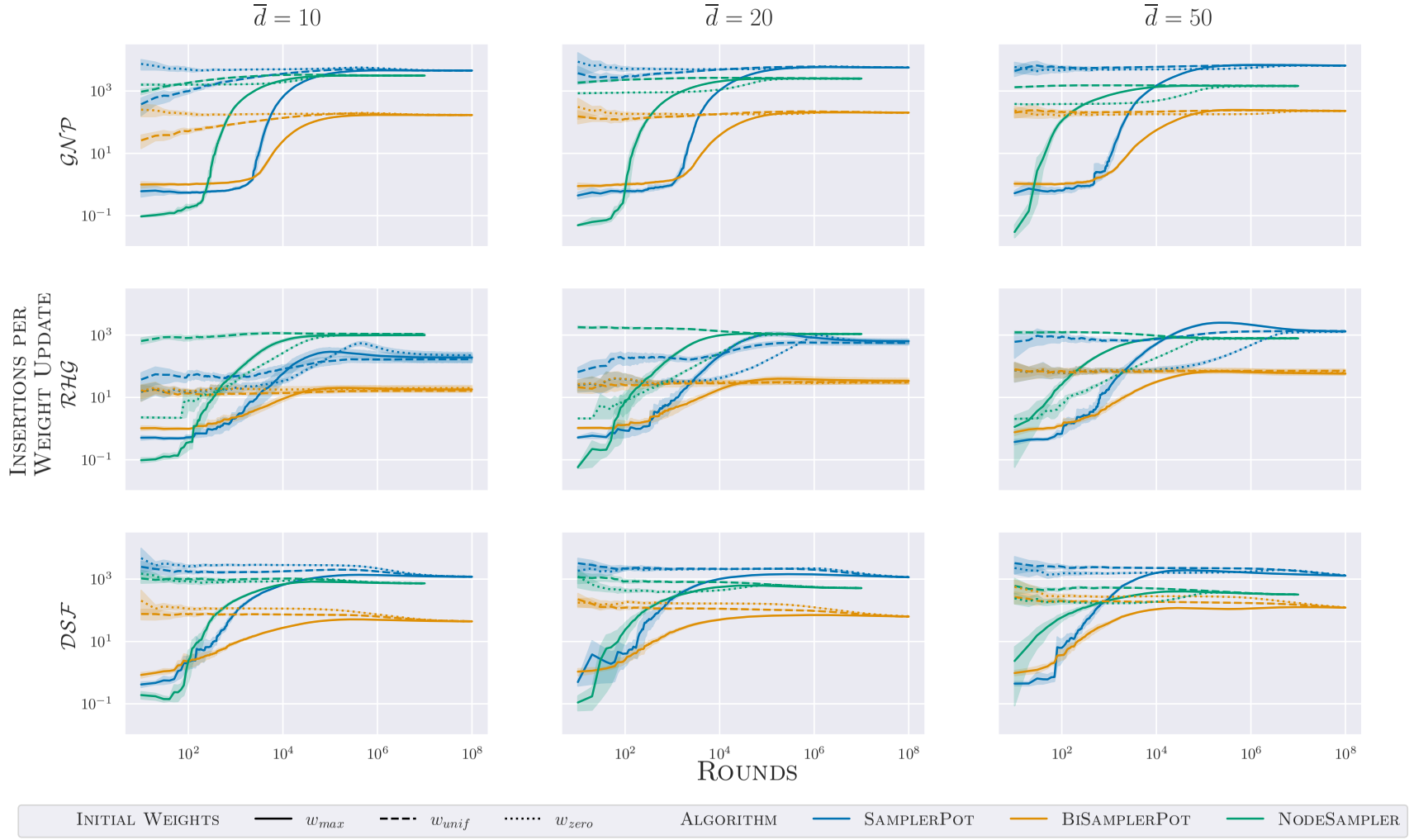


Figure B.15: Number of Queue-Insertions per succesfull Weight-Update over time of different algorithms on different different graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

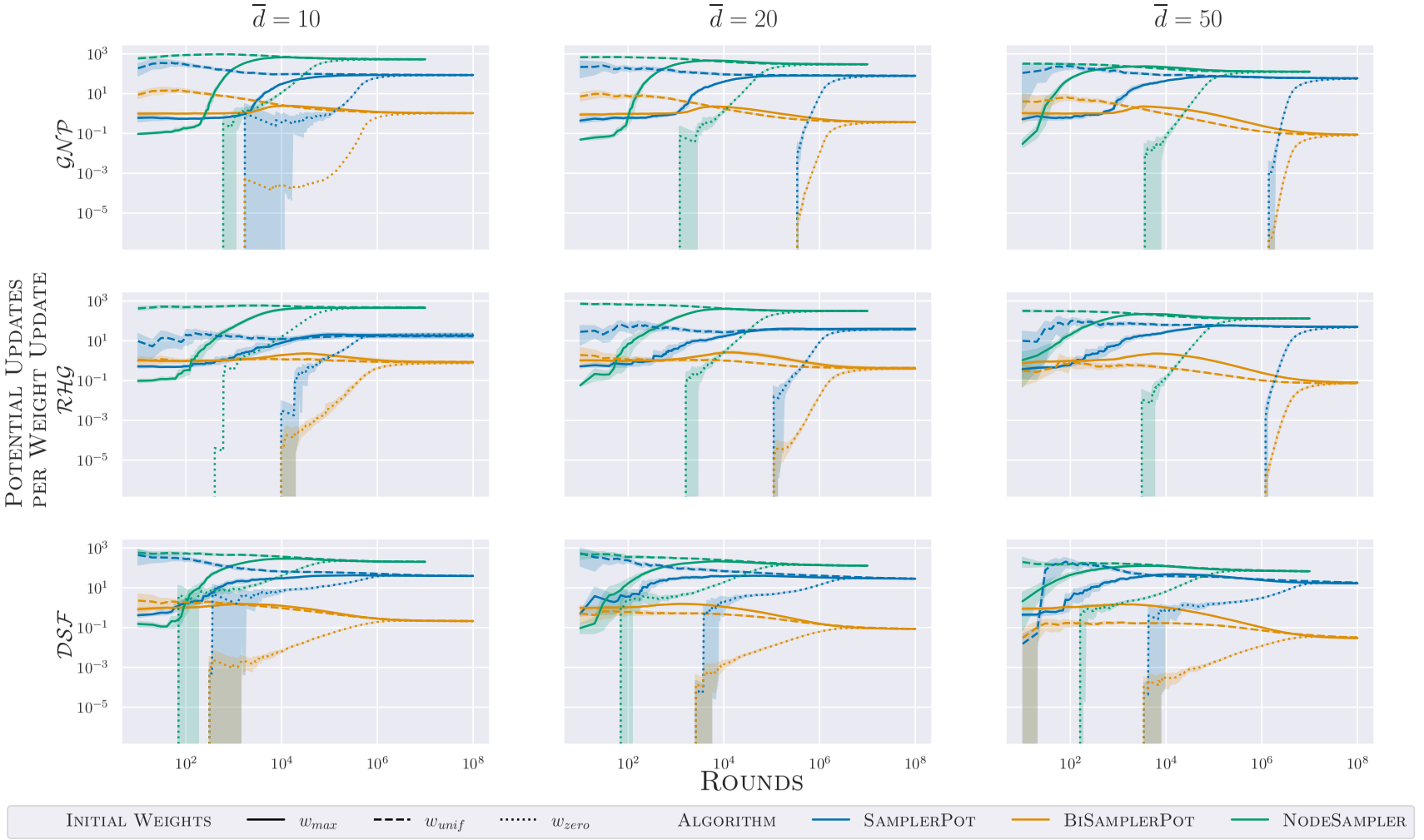


Figure B.16: Number of Potential-Updates per succesfull Weight-Update over time of different algorithms on different different graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

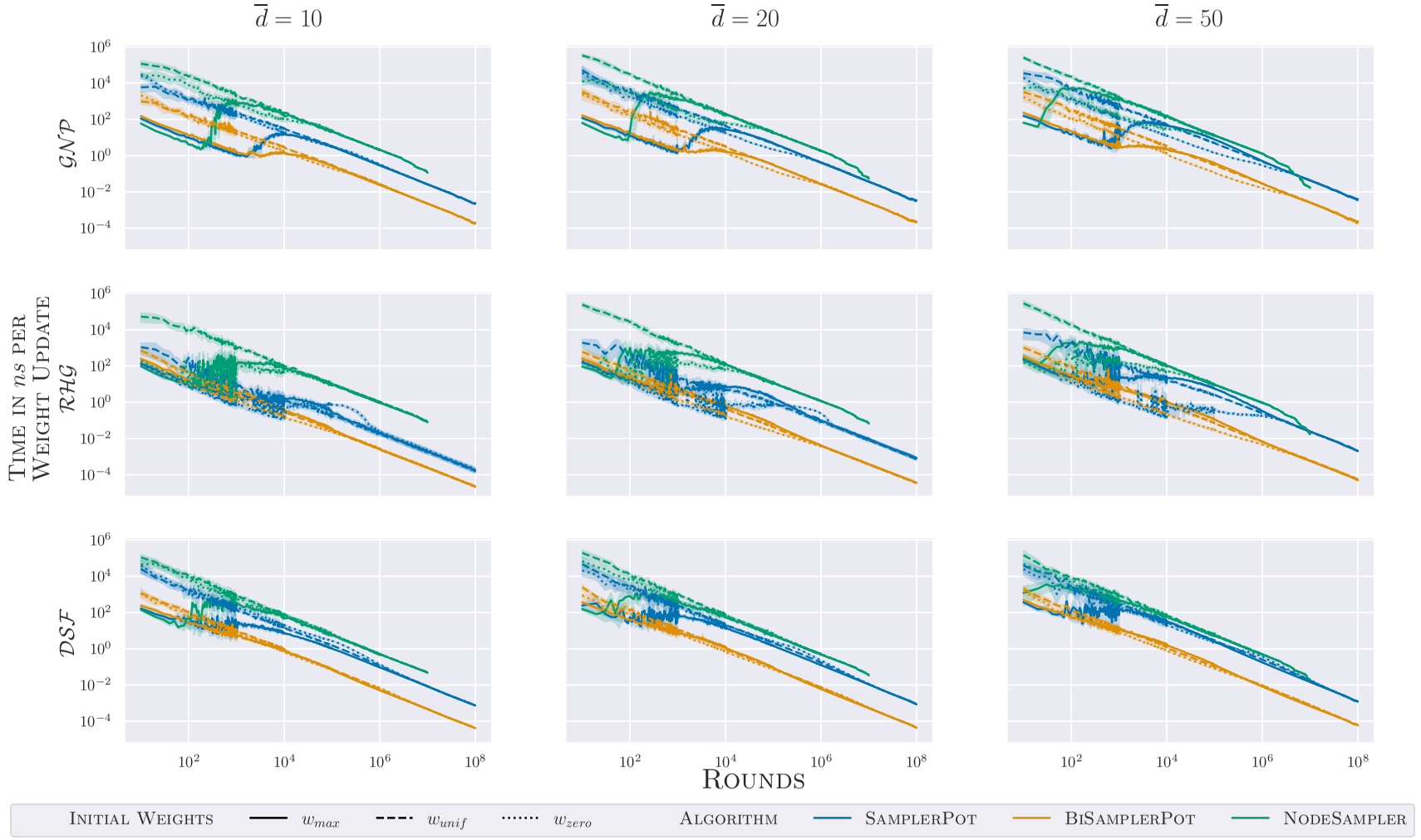


Figure B.17: Runtime per succesfull Weight-Update over time of different algorithms on different different graphs with $n = 10000$ nodes, average degrees $\overline{d} \in \{10, 20, 50\}$ and starting weights $w_{init} \in \{w_{max}, w_{unif}, w_{zero}\}$.

C

Feasible Potential Computation

Notice that in neither of our proposed MarkovChains, the engineered check for negative weights requires the weight update itself to be randomized. In this direction, one could also propose a deterministic sequence of weight updates for a given graph to achieve a desired weight function if no negative cycles are created. Even better, as our improved MarkovChains such as SamplerPot, BiSamplerPot, and NodeSampler also compute a feasible potential function ϕ for the final weight function w, we can utilize these methods to compute a feasible potential function ϕ for a deterministic w if possible or output that a negative cycle exists.

As briefly highlighted in Chapter 3, this is the standard method for theoretical state-of-the-art SingleSourceShortestPath algorithms with negative edge-weights: first compute a feasible potential function ϕ or output that a negative cycle exists in which case SingleSourceShortestPath is infeasible. Then run Dijkstra for weight function w_ϕ which adheres to optimal runtime bounds to compute the shortest path tree.[1]

Hence, we propose a deterministic variant of NodeSampler that makes use of Lemma 5.12 to compute a feasible potential function ϕ for a given weight function w on graph $G = (V, E)$.

▷ Theorem C.1. *For a given graph $G = (V, E)$ and weight function $w\colon E \to \mathbb{R}$, let n^- denote the number of nodes in V with incoming negative-weighted edges. Then there is an algorithm that computes a feasible potential function ϕ in time*

$$\mathcal{O}(n^- \cdot (m + n \log n))$$

or outputs HasNegativeCycle if there is a negative cycle in the graph.

Proof. We adapt NodeSampler by starting with weight function

$$w_+\colon E \to \mathbb{R}^+, \qquad e \mapsto \max\{0, w(e)\}.$$

As $w_+(e) \geq 0$ for every $e \in E$, $\phi = 0$ is feasible and (G, w_+) are consistent. We then iteratively re-insert negative edge-weights into w_+ to eventually yield w (or abort earlier). We can do this one node at a time where we set the weight of all incoming edges of this node to their original weight (i.e. resetting the weight of negative-weighted edges) and checking for consistency using the method of NodeSampler. This also includes the updates to ϕ that are needed to maintain feasibility throughout the algorithm.

[1] One could then subtract the potential gradient to get correct distances with respect to w.

Let $w_+^{(i)}$ denote the (tentative) weight function after i steps of this algorithm. Then, by previous observations in Obs. 4.11 and Theorem 5.13, if (G, w) are consistent, so are $(G, w_+^{(i)})$ as $w_+^{(i)}(e) \geq w(e)$ for every $e \in E$. Correctness then follows from the correctness of NodeSampler whereas runtime follows by definition of n^- (as we only need to update incoming edges of nodes that have negative-weighted incoming edges) as well as Lemma 2.15. □

Although Theorem C.1 is a neat result that shows a more broad way of utilizing our engineered dynamic consistency checks, it is probably of not much practical use in this exact context. Common methods for potential computation in a practical context include the method of running one BellmanFord iteration as done so in Johnson (see Chapter 2) or more recent methods such as ElimNeg [15, 26] that alternate between BellmanFord and Dijkstra (see Algorithm 10). E^{neg} in Line 12 refers to the set of edges with negative weight with respect to w. Similar to Johnson's method, we start with an additional vertex s that has a zero-weighted edge to every other node in the graph and compute the shortest path tree $\textsc{Spt}(s)$ that by Lemma 2.21 is a feasible potential function.

As a personal addition, we also include the amortized acyclicity check used in the SPFA heuristic of BellmanFord to prevent an endless loop. Here, we interpret $P[v] = u$ as an edge (u, v) in a tentative shortest path tree (or no edge if $P[v] = \textsc{Null}$): if this ever creates a cycle, this must correspond to a negative cycle in the original graph. We call this subgraph the P-induced subgraph. Again, this checks runs in time $\mathcal{O}(n)$ [65] since the tentative shortest path tree has $\mathcal{O}(n)$ edges.

Although [15] assumes constant degree for all nodes, i.e. $m = \Theta(n)$, in their runtime proof, we can upper bound their runtime for general graphs.

▷ Lemma C.2 (Runtime ElimNeg). *ElimNeg correctly returns $\textsc{Spt}(s)$ or HasNegativeCycle in time*

$$\mathcal{O}(n^- \cdot (m + n \log n)) .$$

Proof. We only show general runtime and correctness of the additional negative cycle check as correctness of the original algorithm is done in [15]. The original authors also prove an upper bound of

$$\mathcal{O}\left(\log n \cdot \left(n + \sum_{v \in V} \eta(v)\right)\right)$$

where

$$\eta(v) \coloneqq \begin{cases} \infty & , \text{if } d(s, v) = -\infty \\ \min\{|E^{neg} \cap P| \mid P \text{ is a shortest } s \to v \text{ path in } G\} & \text{otherwise,} \end{cases}$$

i.e. the minimum number of negative edges on any shortest $s \to v$ path.

We can mostly disregard their original analysis and only focus on [15, Lemma A.3] which states that "after iteration i of the `Dijkstra Phase`, $D[v] = d(s, v)$ for every v where $\eta(v) \leq i$" if no negative cycle is present in the graph (otherwise $D[v] \leq d(s, v)$). Then, v can not be added again to Q in the `Bellman-Ford Phase`

Algorithm 10: ElimNeg [15, 26]

Input: graph $G = (V, E)$, weights $w\colon E \to \mathbb{R}$, helper node s
Output: $\text{Spt}(s) = \phi$ or HasNegativeCycle

```
1  for v ∈ V do
2      D[v] ← ∞
3      P[v] ← Null
4  D[s] ← 0
5  Q ← Min-PriorityQueue
6  S ← Stack
7  add s to Q
8  repeat
       // Dijkstra Phase
9      while Q is not empty do
10         v ← node in Q with minimum D[v]
11         S.push(v)
12         for (v, x) ∈ E \ E^neg do
13             if D[v] + w(v, x) < D[x] then
14                 D[x] ← D[v] + w(v, x)
15                 P[x] ← v
16                 if x ∉ Q then add x to Q
       // Bellman-Ford Phase
17     while S is not empty do
18         v ← S.pop()
19         for (v, x) ∈ E do
20             if D[v] + w(v, x) < D[x] then
21                 D[x] ← D[v] + w(v, x)
22                 P[x] ← v
23                 if x ∉ Q then add x to Q
       // Correct Distances
24     if Q is empty then
25         return D
       // Check for Acyclicity of SPT
26     if last check is Θ(n) edge-relaxations ago or this is iteration i > n⁻
        then
27         if P is not acyclic then
28             return HasNegativeCycle
```

as the minimum distance has been found. Thus, the runtime of the algorithm for general graphs is upper bounded by the number of iterations times the runtime of both phases combined. The `Dijkstra Phase` trivially runs in $\mathcal{O}(m + n \log n)$ and the `Bellman-Ford Phase` in time $\mathcal{O}(m + n)$ as we only iterate over each node

and its neighbors at most once. Again, by [15, Lemma A.3] it follows that the total runtime is therefore upper bounded by

$$\mathcal{O}\left((m + n \log n) \cdot \max_{v \in V} \eta(v)\right) = \mathcal{O}(n^- \cdot (m + n \log n)) ,$$

since after $\max_{v \in V} \eta(v)$ iterations, every node has found its correct shortest path and Q is empty. Also, trivially $\max_{v \in V} \eta(v) \leq n^-$ because if no negative cycles exist, every shortest path is simple and thus only uses every node at most once. The number of negative edges along a shortest path is therefore upper bounded by the number of nodes with incoming negative-weighted edges which is n^-.

In the case where a negative cycle exists, notice that the original algorithm never terminates. Therefore, we must at some point reach iteration $n^- + 1$ in which there is at least one node v with $D[v] < d'(s, v)$ where $d'(s, v)$ is the length of the shortest simple $s \to v$ path. Otherwise, the algorithm would have already terminated and by the original analysis correctly returned a shortest path tree. Then, since $D[v] < d'(s, v)$, the shortest $s \to v$ path must have traversed one node (of a negative cycle) at least twice which implies that the P-induced subgraph can not be acyclic. Also by definition, the P-induced subgraph (shortest path "tree") can only be acyclic if a negative cycle is present in the graph. Thus, the additional check is correct and incurs no worse asymptotic runtime due to amortization.

Lastly, the latest check that would eventually return HasNegativeCycle happens in $\mathcal{O}(n^-)$ iterations which also does not incur a worse asymptotic runtime. □

Initial experiments indicated that the proposed method of Theorem C.1 is by far the worst technique among the discussed three in practice for common graph models. Thus, at this time, no further research was conducted on this specific method (or similar bounds on Johnson's method).

Bibliography

[1] *Wiley-interscience series in discrete mathematics and optimization*. John Wiley & Sons, Inc., 2014.

[2] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37(2):213–223, 1990. doi:10.1145/77600.77615.

[3] Fawaz Al-Anzi and Dia AbuZeina. A survey of markov chain models in linguistics applications. pages 53–62, 2016. doi:10.5121/csit.2016.61305.

[4] Daniel Allendorf, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Parallel global edge switching for the uniform sampling of simple graphs with prescribed degrees. *J. Parallel Distributed Comput.*, 174:118–129, 2023.

[5] Daniel Allendorf, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Nick Wormald. Engineering uniform sampling of graphs with a prescribed power-law degree sequence. In *ALENEX*, pages 27–40. SIAM, 2022.

[6] Georgios Amanatidis and Pieter Kleer. Approximate sampling and counting of graphs with near-regular degree intervals. In *STACS*, volume 254 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl, 2023.

[7] Naheed Anjum Arafat, Debabrota Basu, Laurent Decreusefond, and Stéphane Bressan. Construction and random generation of hypergraphs with prescribed degree and dimension sequences. In *DEXA (2)*, volume 12392 of *LNCS*, pages 130–145. Springer, 2020.

[8] Andrii Arman, Pu Gao, and Nicholas C. Wormald. Fast uniform generation of random graphs with given degree sequences. *Random Struct. Algorithms*, 59(3):291–314, 2021.

[9] Kyriakos Axiotis, Aleksander Mądry, and Adrian Vladu. Circulation control for faster minimum cost flow in unit-capacity graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 93–104, 2020. doi:10.1109/FOCS46700.2020.00018.

[10] Albert-Laszlo Barabasi. *Network Science*. Cambridge University Press, 2016.

[11] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi:10.1126/science.286.5439.509.

[12] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Phys. Rev. E*, 71(3), 2005. doi:10.1103/physreve.71.036113.

[13] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[14] Edward A. Bender and E. Rodney Canfield. The asymptotic number of labeled graphs with given degree sequences. *J. Comb. Theory A*, 24(3):296–307, 1978.

[15] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 600–611, 2022. doi:10.1109/FOCS54457.2022.00063.

[16] Md Hasanuzzaman Bhuiyan, Jiangzhuo Chen, Maleq Khan, and Madhav V. Marathe. Fast parallel algorithms for edge-switching to achieve a target visit rate in heterogeneous graphs. In *ICPP*, pages 60–69. IEEE Computer Society, 2014.

[17] Somenath Biswas. Various proofs of the fundamental theorem of markov chains, 2022. arXiv:2204.00784.

[18] Thomas Bläsius, Cedric Freiberger, Tobias Friedrich, Maximilian Katzmann, Felix Montenegro-Retana, and Marianne Thieffry. Efficient shortest paths in scale-free networks with underlying hyperbolic geometry. *ACM Trans. Algorithms*, 18(2):19:1–19:32, 2022.

[19] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. Efficiently generating geometric inhomogeneous and hyperbolic random graphs. *CoRR*, abs/1905.06706, 2019.

[20] Thomas Bläsius, Tobias Friedrich, and Anton Krohmer. Hyperbolic Random Graphs: Separators and Treewidth. In *24th Annual European Symposium on Algorithms (ESA 2016)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 15:1–15:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.15.

[21] Thomas Bläsius and Marcus Wilhelm. Deterministic performance guarantees for bidirectional BFS on real-world networks. In *IWOCA*, volume 13889 of *LNCS*, pages 99–110. Springer, 2023.

[22] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. Parallel shortest paths using radius stepping. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, page 443–454. Association for Computing Machinery, 2016. doi:10.1145/2935764.2935765.

[23] B. Bollobás. *Random graphs*. Academic Press, 1985.

[24] Béla Bollobás, Christian Borgs, Jennifer T. Chayes, and Oliver Riordan. Directed scale-free graphs. In *SODA*, pages 132–139. ACM/SIAM, 2003.

[25] Michele Borassi and Emanuele Natale. KADABRA is an adaptive algorithm for betweenness via random approximation. *ACM J. Exp. Algorithmics*, 24(1):1.2:1–1.2:35, 2019.

[26] Karl Bringmann, Alejandro Cassis, and Nick Fischer. Negative-weight single-source shortest paths in near-linear time: Now faster! pages 515–538, 2023. doi:10.1109/FOCS57990.2023.00038.

[27] Camila C. S. Caiado and Pushpa N. Rathie. Polynomial coefficients and distribution of the sum of discrete uniform variables. In *Proc. of SSFA*, 2007.

[28] Nairen Cao, Jeremy T. Fineman, and Katina Russell. Efficient construction of directed hopsets and parallel approximate shortest paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2020, page 336–349. Association for Computing Machinery, 2020. doi:10.1145/3357713.3384270.

[29] C. J. Carstens. *Topology of Complex Networks: Models and Analysis*. PhD thesis, RMIT University, 2016.

[30] Corrie Jacobien Carstens, Annabell Berger, and Giovanni Strona. A unifying framework for fast randomization of ecological networks with fixed (node) degrees. *MethodsX*, 5:773–780, 2018. doi:10.1016/j.mex.2018.06.018.

[31] Corrie Jacobien Carstens, Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. Parallel and I/O-efficient randomisation of massive networks using global curveball trades. In *ESA*, volume 112 of *LIPIcs*, pages 11:1–11:15. Schloss Dagstuhl, 2018.

[32] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 612–623, 2022. doi:10.1109/FOCS54457.2022.00064.

[33] Michael B. Cohen, Aleksander Mądry, Piotr Sankowski, and Adrian Vladu. *Negative-Weight Shortest Paths and Unit Capacity Minimum Cost Flow in Õ (<italic>m</italic>^{10/7} log <italic>W</italic>) Time (Extended Abstract)*, pages 752–771. doi:10.1137/1.9781611974782.48.

[34] Louis Comtet. *Advanced Combinatorics*. Springer Dordrecht, 1974.

[35] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), page 139–146. Association for Computing Machinery, 1963. doi:10.1145/1463822.1463838.

[36] Laxman Dhulipala, Guy Blelloch, and Julian Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*,

SPAA '17, page 293–304. Association for Computing Machinery, 2017. doi:10.1145/3087556.3087580.

[37] Persi Diaconis and Daniel Stroock. Geometric Bounds for Eigenvalues of Markov Chains. *The Annals of Applied Probability*, 1(1):36 – 61, 1991. doi:10.1214/aoap/1177005980.

[38] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[39] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. Efficient stepping algorithms and implementations for parallel shortest paths. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '21, page 184–197. Association for Computing Machinery, 2021. doi:10.1145/3409964.3461782.

[40] Mikhail Drobyshevskiy and Denis Turdakov. Random graph modeling: A survey of the concepts. *ACM Comput. Surv.*, 52(6):131:1–131:36, 2020.

[41] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972. doi:10.1145/321694.321699.

[42] P. Erdős and A. Rényi. On Random Graphs I. *Publicationes Mathematicae (Debrecen)*, 6, 1959.

[43] Péter L. Erdös, Catherine S. Greenhill, Tamás Róbert Mezei, István Miklós, Daniel Soltész, and Lajos Soukup. The mixing time of the swap (switch) markov chains: a unified approach. *CoRR*, abs/1903.06600, 2019.

[44] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 232–241, 2001. doi:10.1109/SFCS.2001.959897.

[45] Philippe Flajolet, Danièle Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992. doi:10.1016/0166-218X(92)90177-C.

[46] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 114–118. Association for Computing Machinery, 1978. doi:10.1145/800133.804339.

[47] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. doi:10.1145/28869.28874.

[48] Tobias Friedrich and Anton Krohmer. On the diameter of hyperbolic random graphs. *SIAM Journal on Discrete Mathematics*, 32(2):1314–1334, 2018. doi:10.1137/17M1123961.

[49] Alan Frieze and Eric Vigoda. 53A SURVEY ON THE USE OF MARKOV CHAINS TO RANDOMLY SAMPLE COLOURINGS. In *Combinatorics, Complexity, and Chance: A Tribute to Dominic Welsh*. Oxford University Press. doi:10.1093/acprof:oso/9780198571278.003.0004.

[50] Alan M. Frieze and Geoffrey R. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discret. Appl. Math.*, 10(1):57–77, 1985.

[51] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200–217, 2019.

[52] Harold N. Gabow. Scaling algorithms for network problems. *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 248–258, 1983.

[53] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, 1989. doi:10.1137/0218069.

[54] Pu Gao and Nicholas C. Wormald. Uniform generation of random regular graphs. *SIAM J. Comput.*, 46(4):1395–1427, 2017.

[55] Pu Gao and Nicholas C. Wormald. Uniform generation of random graphs with power-law degree sequences. In *SODA*, pages 1741–1758. SIAM, 2018.

[56] Lukas Geis, Daniel Allendorf, Thomas Bläsius, Alexander Leonhardt, Ulrich Meyer, Manuel Penschuck, and Hung Tran. Uniform sampling of negative edge weights in shortest path networks, 2024. arXiv:2410.22717.

[57] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4), 1959. doi:10.1214/aoms/1177706098.

[58] Christos Gkantsidis, Milena Mihail, and Ellen W. Zegura. The markov chain simulation method for generating connected power law random graphs. In *ALENEX*, pages 16–25. SIAM, 2003.

[59] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995. doi:10.1137/S0097539792231179.

[60] N. J. Gotelli and G. R. Graves. *Null models in ecology*. Smithsonian Institution, 1996.

[61] Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering. In *Automata, Languages, and Programming*, pages 573–585. Springer Berlin Heidelberg, 2012.

[62] Michael Hamann, Ulrich Meyer, Manuel Penschuck, Hung Tran, and Dorothea Wagner. I/O-efficient generation of massive graphs following the *LFR* benchmark. *ACM J. Exp. Algorithmics*, 23, 2018.

[63] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[64] Lester R. Ford Jr. *Network Flow Theory*. RAND Corporation, 1956.

[65] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962. doi:10.1145/368996.369025.

[66] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Plenum Press, New York, 1972.

[67] Philip Klein, Shay Mozes, and Oren Weimann. Shortest paths in directed planar graphs with negative lengths: a linear-space o(n log2 n)-time algorithm. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, page 236–245. Society for Industrial and Applied Mathematics, 2009.

[68] Philip Klein, Satish Rao, Monika Rauch, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, page 27–37. Association for Computing Machinery, 1994. doi:10.1145/195058.195092.

[69] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguñá. Hyperbolic geometry of complex networks. *Phys. Rev. E*, 82:036106, 2010. doi:10.1103/PhysRevE.82.036106.

[70] N. : Markov-Ketten Kurt. *In: Stochastik für Informatiker: Eine Einführung in einheitlich strukturierten Lerneinheiten.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2020.

[71] Alexander Leonhardt, Ulrich Meyer, and Manuel Penschuck. Insights into (k, ρ)-shortcutting algorithms. In *32nd Annual European Symposium on Algorithms (ESA 2024)*, page TBD. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[72] David A Levin and Yuval Peres. *Markov chains and mixing times*, volume 107. American Mathematical Soc., 2017.

[73] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, 1979. doi:10.1137/0716027.

[74] A. A. Markov. *Extension of the law of large numbers to dependent quantities (in Russian).* Izvestiia Fiz. -Matem. Obsch, 1906.

[75] Russell A. Martin and Dana Randall. Disjoint decomposition of markov chains and sampling circuits in cayley graphs. *Comb. Probab. Comput.*, 15(3):411–448, 2006.

[76] B. D. McKay. Asymptotics for symmetric 0-1 matrices with prescribed row sums. *Ars Combinatoria*, 19:15–25, 1985.

[77] Brendan D. McKay and Nicholas C. Wormald. Uniform generation of random regular graphs of moderate degree. *J. Algorithms*, 11(1):52–67, 1990.

[78] Ulrich Meyer and Peter Sanders. δ-stepping : A parallel single source shortest path algorithm. In *Algorithms — ESA' 98*, pages 393–404. Springer Berlin Heidelberg, 1998.

[79] R. Milo. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594), 2002. doi:10.1126/science.298.5594.824.

[80] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[81] Michael Molloy and Bruce A. Reed. A critical point for random graphs with a given degree sequence. *Random Struct. Algorithms*, 6(2/3):161–180, 1995.

[82] E.F. Moore. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959.

[83] Tobias Müller and Merlijn Staps. The diameter of kpkvb random graphs. *Advances in Applied Probability*, 51(2), 2019.

[84] M. E. J. Newman. *Networks — An Introduction*. Oxford University Press, 2010.

[85] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 456–471. Association for Computing Machinery, 2013. doi:10.1145/2517349.2522739.

[86] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *EDBT*, pages 331–342. ACM, 2011.

[87] T. P. Peixoto. Model selection and hypothesis testing for large-scale network models with overlapping groups. *Phys. Rev. X*, 5(1), 2015. doi:10.1103/physrevx.5.011033.

[88] Manuel Penschuck. Generating Practical Random Hyperbolic Graphs in Near-Linear Time and with Sub-Linear Memory. In *16th International Symposium on Experimental Algorithms (SEA 2017)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 26:1–26:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.SEA.2017.26.

[89] Manuel Penschuck. Engineering Shared-Memory Parallel Shuffling to Generate Random Permutations In-Place. In *21st International Symposium on Experimental Algorithms (SEA 2023)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 5:1–5:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SEA.2023.5.

[90] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation1. In *Massive Graph Analytics*, pages 333–376. Chapman and Hall/CRC, 2022.

[91] A. R. Rao, R. Jana, and S. Bandyopadhyay. A markov chain monte carlo method for generating random (0, 1)-matrices with given marginals. *Sankhyā: The Indian J. Statistics, Series A*, 1996.

[92] Gareth O. Roberts and Jeffrey S. Rosenthal. General state space Markov chains and MCMC algorithms. *Probability Surveys*, 1(none):20 – 71, 2004. doi:10.1214/154957804100000024.

[93] Peter Sanders, Sebastian Lamm, Lorenz Hübschle-Schneider, Emanuel Schrade, and Carsten Dachsbacher. Efficient parallel random sampling - vectorized, cache-efficient, and online. *ACM Trans. Math. Softw.*, 44(3):29:1–29:14, 2018.

[94] Piotr Sankowski. Shortest paths in matrix multiplication time. In *Algorithms – ESA 2005*, pages 770–778. Springer Berlin Heidelberg, 2005.

[95] Alistair Sinclair. Improved bounds for mixing rates of markov chains and multicommodity flow. *Combinatorics, probability and Computing*, 1(4):351–370, 1992.

[96] Isabelle Stanton and Ali Pinar. Sampling graphs with a prescribed joint degree distribution using markov chains. In *ALENEX*, pages 151–163. SIAM, 2011.

[97] G. Strona, D. Nappo, F. Boccacci, S. Fattorini, and J. Miguel-Ayanz. A fast and unbiased procedure to randomize ecological binary matrices with fixed row and column totals. *Nature Commun.*, 2014. doi:10.1038/ncomms5114.

[98] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[99] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, mdps, and ℓ1-regression in nearly linear time for dense instances. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, page 859–869. Association for Computing Machinery, 2021. doi:10.1145/3406325.3451108.

[100] Jan van den Brand, Yin-Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 919–930, 2020. doi:10.1109/FOCS46700.2020.00090.

[101] Remco van der Hofstad. *Random Graphs and Complex Networks*, volume 43 of *Cambridge Series in Statistical and Probabilistic Mathematics*. Cambridge University Press, 2016.

[102] Fabien Viger and Matthieu Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. *J. Complex Networks*, 4(1):15–37, 2016.

[103] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985. doi:10.1145/3147.3165.

[104] Moritz von Looz, Mustafa Özdayi, Sören Laue, and Henning Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. *CoRR*, abs/1606.09481, 2016.

[105] Nicholas C. Wormald. Generating random regular graphs. *J. Algorithms*, 5(2):247–280, 1984.

[106] K. Zuse. *Der Plankalkül*. Berichte der Gesellschaft für Mathematik und Datenverarbeitung. Gesellschaft für Mathematik und Datenverarbeitung, 1972.

Erklärung zur Abschlussarbeit

gemäß § 34, Abs. 16 der Ordnung für den Masterstudiengang Informatik vom 17. Juni 2019

Hiermit erkläre ich

(Nachname, Vorname)

Die vorliegende Arbeit habe ich selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst.

Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Zudem versichere ich, dass die von mir eingereichten schriftlichen gebundenen Versionen meiner Masterarbeit mit der eingereichten elektronischen Version meiner Masterarbeit übereinstimmen.

Frankfurt am Main, den

Unterschrift der/des Studierenden