Optimal Uniform Shortest Path Sampling through Random Walk

Simon Dreyer, Antoine Genitrini, Mehdi Naima

Optimal Uniform Shortest Path Sampling through Random Walk

Simon Dreyer* Antoine Genitrini* Mehdi Naima*

Abstract

Random generation of shortest paths in graphs is utilized across various domains, including traffic-flow simulation and network topology exploration. In this paper, we address the challenge of uniform shortest path sampling in graphs from an algorithmic perspective. We introduce a novel uniform shortest path sampling algorithm that uses a biased random walk operating in two stages. We demonstrate that our algorithm, when combined with Alias sampling, is optimal in terms of worst-case running time among all algorithms in its class. Furthermore, we present an efficient implementation of our algorithm in a low-level programming language and evaluate it on both real-world and synthetic datasets. We compare our theoretically optimal algorithm with other variants to assess its practical performance and discuss its application in traffic modeling.

1 Introduction

Graphs play a crucial role in understanding and addressing problems associated with networks and interconnected systems. They provide a theoretical basis for analyzing various structures, including social networks, communication systems, and transportation networks.

Sampling shortest paths is essential in numerous contexts, such as simulating traffic flow, studying the topology of large networks (like the internet and social networks), and assessing network damage. For instance, the authors of [5] employ this method to evaluate network damage, while [8] analyzes shortest path sampling as an approximation of traceroute paths. The practice of approximating graphs through shortest path sampling has been explored in [16, 20, 1, 23], where the authors investigate the biases introduced by exploring a graph via random shortest paths. In [14], the authors discuss which graph properties can be well approximated using shortest path sampling and the influence of the number of samples on these properties. In the realm of traffic flow, the authors of [27] apply shortest path sampling to enhance network transmission capacities.

Despite extensive research, there is a noticeable gap in the explicit study of the algorithmic complexity of shortest path sampling. Typically, this problem involves fixed source and target nodes, aiming to sample a shortest path between them. Some studies, as mentioned in [8, 5, 21, 27, 18, 7], suggest randomly selecting one shortest path from all possible paths without detailing the implementation. This approach faces challenges due to the potentially exponential number of shortest paths between two nodes in some graph families, such as grid graphs, making it impractical for simulating traffic in city graphs, which often resemble grid graphs.

Other studies recommend assigning random weights to edges to resolve ties among multiple shortest paths, then returning the unique shortest path left, as detailed in [16, 6, 26, 11]. Each change of weights generates a new shortest path. However, this method introduces a bias, meaning that some shortest paths are more likely to be selected than others. To our knowledge, no theoretical or experimental studies have addressed this bias, which could affect the outcomes of experimental studies relying on biased sampling methods.

The paper is organized as follows: In Section 2 after presenting our formalism we give the problem statement for the uniform sampling of shortest path between fixed source and target nodes and review the current state of the art in this field. Then, in Section 3 we present our main contribution, a generic two-stage random walk algorithm, and demonstrate that, with appropriate weight settings, this algorithm uniformly samples shortest paths. Next, in Section 4 we discuss four different implementations of our proposed algorithm, conducting a detailed analysis to identify the optimal version. Section 5 extends our algorithm to generate uniform shortest path in general. Finally, in Section 6, we assess the performance of our algorithm and its various implementations using both real-world and synthetic graphs. We also explore an application of our algorithm in traffic modeling. All missing proofs in the main paper can be found in Appendix A.

2 Context of the problem and state of the art

In this article, we focus on graphs represented by $G = (V, E)$, where V is a finite set of nodes and $E = \{(u, v) \mid u, v \in V, u \neq v\}$ denotes the set of directed edges. Therefore, the graphs under consideration are directed and simple (no self-loops or multiple edges

*Sorbonne Université, CNRS, LIP6 - UMR 7606, F-75005 Paris, France.

are allowed). We denote $n = |V|$ as the number of nodes and $m = |E|$ as the number of edges. An undirected graph can be viewed as a directed graph where if $(u, v) \in E$ then $(v, u) \in E$ as well. An edge $(u, v) \in E$ will be denoted as $u \to v$, where u is referred to as the starting point and v as the end point. A *walk* W in a graph is a sequence of edges such that the end point of one edge is the starting point of the next edge. The *length of a walk* W, denoted $|W|$, corresponds to the number of edges it contains. For a given walk, $s \in V$ usually denotes *the source node* and $t \in V$ denotes *the target node*. The *distance* from s to t, denoted $d(s, t)$, represents the minimal length among all walks from s to t.

A *shortest path* from s to t is a walk W of minimal length, that is $|W| = d(s, t)$. We also denote by $\mathbf{W}_{st}$ the set of all shortest paths from s to t and by $\sigma_s(t) = |\mathbf{W}_{st}|$ the number of shortest paths. The set of all shortest paths starting from s is written as $\mathbf{W}_{s\bullet}$. Therefore, $\mathbf{W}_{s\bullet} = \cup_{v \in V} \mathbf{W}_{sv}$ and $\sigma_{s\bullet} = |\mathbf{W}_{s\bullet}|$. Finally, we denote by $\mathbf{W}$ the set of all shortest paths in the graph: thus $\mathbf{W} = \cup_{v \in V} \mathbf{W}_{v\bullet}$ and $\sigma = |\mathbf{W}|$. Given a graph and fixed source s and target t, we are interested in:

Problem 1: source-target uniform shortest path: Nodes s, t being fixed, give a random generation algorithm satisfying $\forall W \in \mathbf{W}_{st}, \mathbb{P}(W) = 1/\sigma_s(t)$ and for all $W \notin \mathbf{W}_{st}, \mathbb{P}(W) = 0$.

2.1 Naive Algorithm The simplest approach involves precomputing all shortest paths from s to t and storing them in a list during a preprocessing stage. Queries can then be answered by returning a randomly selected element from this list. This approach is implicitly suggested by [8, 5, 21, 27, 18, 7]. The preprocessing stage running time depends on the number of paths in $\mathbf{W}_{st}$, which can be exponential for certain graph families. See for example Figure 6. Therefore, we aim to design polynomial-time algorithms that still ensure uniform sampling.

2.2 Random Weights To create a random generator for shortest paths, one idea used in works such as [16, 6, 26, 11] is to assign small random real weights to the edges of the graph. This ensures that only one shortest path has the minimum weight, which can then be found and returned using Dijkstra's Algorithm with backtracking. The detailed procedure is outlined in Algorithm 1 where `shortest_path_dijkstra`(G, s, t) is a function that returns the shortest path from s to t in G by running a Dijkstra algorithm with backtracking. Therefore, redistributing random weights on the edges of the graph ensures that every shortest path from s to t has some probability of being sampled. However, the probabilities of the different shortest paths are not equal. We illustrate this fact with the following family of graphs $\forall k \geq 1, G_k = (V_k, E_k)$ as shown in Figure 1. We denote by W_0 the shortest path from α to ω, given by $W_0 = \alpha \to 1 \to 2 \to \omega$.

Algorithm 1 Random Weights

Input: $G = (V, E)$: a graph, s: source node, t: target node
Output: a random shortest path

```
function RANDOM_WEIGHTS(G, s, t)
    for e ∈ E do
        ε = uniform(−1/n, 1/n)
        w(e) = 1 + ε                ▷ Assigns a weight to e
    end for
    return shortest_path_dijkstra(G, s, t)   ▷ Dijkstra with backtracking
end function
```

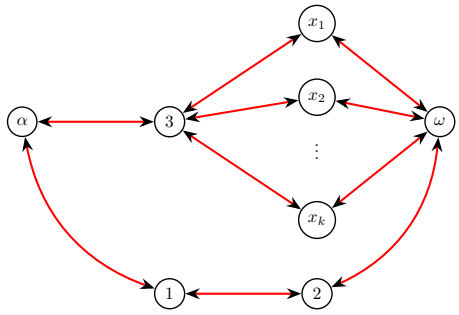


Figure 1: The family of graphs G_k. There are $k + 1$ shortest paths in total. One shortest path $\alpha \to 1 \to 2 \to \omega$, and k shortest paths $(\alpha \to 3 \to x_i \to \omega)_i$.

LEMMA 2.1. *The probability of W_0 being sampled by Algorithm 1 when $k = 2$ is:*

$$\mathbb{P}(W_0) = \frac{737}{2016} \approx 0.37 > \frac{1}{3}, \tag{2.1}$$

when k grows to infinity we have

$$\mathbb{P}(W_0) \overset{k \to +\infty}{\longrightarrow} \frac{1}{24} \neq 0. \tag{2.2}$$

PROPOSITION 2.1. *Random weights is not uniform.*

Proof. Given G_k, the shortest path W_0 should have a probability $\mathbb{P}(W_0) = \frac{1}{k+1}$ for unbiased sampling, since there are $k + 1$ shortest paths from s to t. However, as shown in Lemma 2.1, when $k = 2$, $\mathbb{P}(W_0) \neq \frac{1}{3}$. Additionally, as k increases, $\mathbb{P}(W_0)$ tends towards the constant $\frac{1}{24}$ rather than 0. ∎

Algorithm	Distribution	Preprocessing	Space	Query	Section
Random Weights	biased	x	x	$O(m + n \log n)$	2.2
URW	biased	$O(m + n)$	$O(m + n)$	$O(\ell)$	3.2
Naive	uniform	$\Omega(\lvert\mathbf{W}_{st}\rvert)$	$O(\text{diam} \cdot \lvert\mathbf{W}_{st}\rvert)$	$O(1)$	2.1
BRW-linear	uniform	$O(m + n)$	$O(m + n)$	$O(n)$	4.1
BRW-Ordered	uniform	$O(m + n \log n)$	$O(m + n)$	$O(n)$	4.2
BRW-Binary	uniform	$O(m + n)$	$O(m + n)$	$O(\ell \log(\frac{n}{\ell}))$	4.3
BRW-Alias	uniform	$O(m + n)$	$O(m + n)$	$O(\ell)$	4.4

Table 1: Overview of the discussed random sampling methods for fixed source s and target t, running times correspond to the worst-case scenario given a graph G with n nodes and m edges. The value ℓ represents the length of the sampled path, $\mathbf{W}_{st}$ the set of shortest paths from s to t and diam is the diameter of G.

3 Contribution : a Random Walk Approach

In this Section, source and target nodes are fixed and we design algorithms to tackle the source-target uniform shortest path problem.

3.1 Two stages algorithm We present a generic two-stage random walk algorithm to ensure the sampling of all shortest paths. The two phases are:

- *Preprocessing*: Compute the distributions, among the edges of each node, for the random walk. This is done only once.
- *Queries*: For each query, generate a random walk in the graph.

We will discuss two random walk schemes: an *unbiased* one and a *biased* one. We first need to ensure that we never take edges in the graph that are not on a shortest path from s to t.

This type of two stages problems is usually called repetitive-mode [22] as opposed to single-shot. There are many problems on shortest paths that operate with this two stages procedure for instance to compute distances or all-pairs shortest paths. We refer to [24] for a review of these methods.

Definition 3.1. (Predecessor Set [4]) *Let $G = (V, E)$ be a given graph and fix a node $s \in V$. For $v \in V$, the predecessor set of v is defined as:*

$$pre_s(v) = \{w \in V \mid s \to \cdots \to w \to v \in W_{sv}\}.$$

Definition 3.2. (Successor Graph) *The successor graph is a directed graph $G_s = (V_s, E_s)$ defined as follows:*

$$E_s = \{(w, v) \mid \forall v \in V, w \in pre_s(v)\}.$$

The set V_s corresponds to the nodes belonging to an edge from E_s which is all the nodes accessible from s.

Given a successor graph $G_s = (V_s, E_s)$ and $v \in V_s$, we denote by $\mathbf{N}_v^-$ the list of incoming edges to node v. This list of elements is arbitrarily ordered, thus $\text{set}(\mathbf{N}_v^-) = pre_s(v)$.

Remark 3.1. *The successor graph G_s is the union of all the BFS-trees rooted in s. It is acyclic. It contains exactly one source s and several sinks. Moreover, edges (u, v) of the successor graph go from a node at distance $k = d(s, u)$ to a node at distance $k + 1 = d(s, v)$.*

The successor graph from s is computed in linear time in the size of the graph $O(n+m)$ by using a breadth-first search (BFS) that keeps all the optimal predecessors of a node. Figure 2 introduces a graph-example and its associated successor graph G_0.

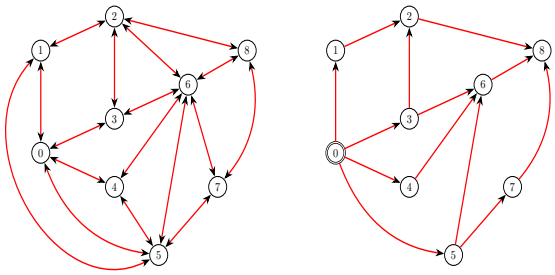


Figure 2: (left) A graph and (right) the associated successor graph for $s = 0$.

The successor graph from s ensures that starting a random walk with target t and taking the edges of G_s in a reversed way, the source s will be reached after exactly $k = d(s, t)$ transitions. The *generic random walk* approach is presented in Algorithm 2 where `BFS_with_predecessors`(G, s) computes the successor graph G_s from node s. We will discuss the different implementations of `compute_weights` and `rand_pred` in Section 4. The algorithm runs in two stages with a *preprocessing step* computing the successor graph from s

and a *random walk step* that starts from t and goes backwards until reaching s. Different random walks could be considered on $G_s = (V_s, E_s)$; these different schemes can be set by the *weight function* $\mathscr{W}$ that assigns weights to the edges E_s. Then, the random walk standing on node $v \in V_s$ chooses a predecessor w given by the list $\mathbf{N}_v^- = [w_0, \ldots, w_k]$ according to the weight function $\mathscr{W}$.

PROPOSITION 3.1. (GENERIC RAND. WALK COMPLEX.) *Any implementation of Algorithm 2 needs $\Omega(m+n)$ operations and $\Omega(m+n)$ space for the* preprocessing step *and $\Omega(\ell)$ operations for the* random walk step *where ℓ is the length of the walk.*

Proof. During the preprocessing step we compute $\mathbf{N}_v^-$ for all $v \in V$ with an edge-distribution for node from $\mathbf{N}_v^-$. Thus the complexity is at least $\Omega(n+m)$ since $\sum_{v \in V} |\mathbf{N}_v^-|$ may be equal to m for some graphs and since we construct a list for each node $v \in V$. For space complexity the successor graph G_s is stored together with a weight on every edge. Thus the space complexity is $\Omega(n+m)$. Now, let W be the generated random walk, in the second step, with $|W| = \ell$. Then W needs at least ℓ operations to be constructed as the predecessors are computed one by one until reaching s. ∎

Algorithm 2 Generic Random Walk

Input: G: a graph, s: source node, t: target node
Output: G_s: Successor Graph from s and the values of $\mathscr{W}$ computed

function PREPROCESSING(s, t)
  G_s = `BFS_with_predecessors`(G, s)
  $\mathscr{W}$ = `compute_weights`(G_s)
  return $(G_s, \mathscr{W})$
end function

Input: s: source node, t: target node, $G_s = (V_s, E_s)$: Successor Graph from s and $\mathscr{W}$ a weight function assigning weights to E_s
Output: path: A shortest path from s to t

function RANDOM_WALK($s, t, G_s, \mathscr{W}$)
  path $= [\,]$, $v = t$
  while $v \neq s$ **do**
    path $= [v] +$ path
    v = `rand_pred`$(\mathbf{N}_v^-, \mathscr{W})$ ▷ chooses a random predecessor of v according to $\mathscr{W}$
  end while
  return $[s] +$ path
end function

The most straightforward random walk that we consider is the *Unbiased Random Walk*.

3.2 Unbiased Random Walk: *URW* In this situation the *weight function* $\mathscr{W}$ assigns weight 1 to all the edges in the successor graph (see the left graph of Figure 3 for an illustration). The random walk is thus unbiased. Using this approach, we get an explicit formula for the probability of sampling a shortest path since all weights are equal to 1. Such a random walk is called *isotropic* in the literature (cf. e.g. [19, 13]).

PROPOSITION 3.2. URW *necessitates $O(m+n)$ for the preprocessing step and $O(\ell)$ operations for the random generation step (ℓ being the length of the sampled walk).*

Proof. The preprocessing step requires the construction of the successor graph, done in $O(m+n)$, and the random walk step then needs to sample a node uniformly in the list $\mathbf{N}_v^-$ which can be done in $O(1)$ and this is repeated ℓ times. ∎

PROPOSITION 3.3. *Let $W \in \mathbf{W}_{st}$ be a shortest path from s to t. The probability of sampling W by* URW *is:*

$$\mathbb{P}(W) = \prod_{v \in W \setminus \{s\}} \frac{1}{|\mathbf{N}_v^-|}.$$

COROLLARY 3.1. (URW IS NOT UNIFORM) *Consider the graphs G_k, and let $(s,t) = (\omega, \alpha)$ the probability of $W' = \omega \to 2 \to 1 \to \alpha$ being sampled by the URW is $\mathbb{P}(W') = 1/2$ for any $k \geq 2$ instead of $\mathbb{P}(W') = 1/(k+1)$.*

In Proposition 2.1 and Corollary 3.1, we have shown that *Random Weights* and *URW* are biased. One might then ask whether one approach yields results closer to the uniform distribution. We demonstrate that there is no universally superior method that works for all graphs. For example, consider the graph G_2 in Figure 1. If we fix $(s,t) = (\alpha, \omega)$, then *URW* assigns a probability of $1/3$ to each of the three shortest paths, while *Random Weights* assigns approximately 0.37 to $\alpha \to 1 \to 2 \to \omega$ and around 0.315 to each of the other two shortest paths, $\alpha \to 3 \to x_i \to \omega$ with $i \in 1, 2$. Therefore, in G_2 with $(s,t) = (\alpha, \omega)$, *URW* is better than *Random Weights*. Conversely, considering $(s,t) = (\omega, \alpha)$, *URW* assigns a probability $1/2$ to $\omega \to 2 \to 1 \to \alpha$ and $1/4$ to the others, while the probabilities of the three shortest paths by *Random Weights* remain unchanged. Therefore, *Random Weights* is closer to the uniform distribution than *URW* in terms of Wasserstein-1 distance.

3.3 Biased Random Walk: *BRW* Now we present how to set the weights of $\mathscr{W}$ such that the uniformity of the sampling is guaranteed. The idea is to assign weights to the edges $(u, v) \in E_s$ of G_s based on

the number of shortest paths arriving at u. Then, *Biased Random Walk (BRW)* is defined by assigning the weights of $\mathscr{W}$ as follows:

$$\forall (u,v) \in E_s, \mathscr{W}(u \to v) = \sigma_s(u). \tag{3.3}$$

PROPOSITION 3.4. (BRW IS UNIFORM) *The biased random walk* BRW *solves the source-target uniform shortest path problem.*

Proof. Let W be the sampled random walk. Since it is computed on the successor graph G_s, $W \in \mathbf{W}_{st}$ is a shortest path. Now, let $W = v_0(:= s) \to v_1 \to \cdots \to v_k(:= t)$. The probability of sampling W can be computed by induction.

$$\mathbb{P}(W) = \prod_{i=1}^{k} \frac{\sigma_s(v_{i-1})}{\sigma_s(v_i)} = \frac{1}{\sigma_s(v_k)} = \frac{1}{\sigma_s(t)}.$$

The probability of going from v_k to v_{k-1} is $\sigma_s(v_{k-1})/\sigma_s(v_k)$. The same reasoning applies by induction (obviously $\sigma_s(s) = 1$), and then terms are telescoping. ∎

4 Implementations of BRW

Now we discuss the algorithmic complexity of computing *BRW*. The functions `compute_weights` and `rand_pred` are implemented in different ways depending on the amount of preprocessing done by `compute_weights` and the computations necessary in `rand_pred` afterward. The function `rand_pred` generates a random predecessor following a given discrete probability distribution. We propose four implementations (linear, ordered, binary, alias) and prove that alias sampling provides the optimal algorithm for both stages. The *preprocessing phase* consists of computing the successor graph G_s and the weights of $\mathscr{W}$. This phase is done only once and stored in memory. The differences between implementations lie in the chosen ordering for $\mathbf{N}_v^-$ and in the computation of weights. The *queries* involve performing random walks on G_s following the distribution $\mathscr{W}$. The differences between implementations are for the function `rand_pred`, which selects a predecessor w of v, where $w \in \mathbf{N}_v^-$.

For all implementations, we compute the values of $\sigma_s(v)$ for $v \in V$. These values are then used for the weight function $\mathscr{W}$. From the successor graph, it is possible to dynamically compute the values of $\sigma_s(z)$ for all $z \in V$ by recurrence, noting that a shortest path from s to v is the concatenation of a shortest path from s to w with the edge (w,v), where w is a predecessor of v (i.e., $w \in \mathrm{pre}_s(v)$). Thus, we have:

$$\sigma_s(s) = 1 \text{ and } \forall v \neq s,\ \sigma_s(v) = \sum_{w \in pre_s(v)} \sigma_s(w). \tag{4.4}$$

This recurrence can be found in [4, Lemma 3]. The values of σ_s will be used in `rand_pred` implemntations of *BRW* as we shall see.

4.1 Linear Implementation In the following implementation, the *preprocessing step* consists in computing the successor graph G_s from node s, and then computing the values of $\sigma_s(v)$ for all $v \in V$ using Equation 4.4. An illustration of the computed successor graph with weights is given in second from left graph of Figure 3. The *query step* then defines a predecessor by sampling a value $r \in [\![\sigma_s(v)]\!]$ and iterating through the predecessors of v one by one to determine where r falls within the predecessor values.

Algorithm 3 Linear implementation of `rand_pred`

Input: $\mathbf{N}_v^-$: List of predecessors of each node v, $\mathscr{W} : E_s \to \mathbb{R}$ a weight function and σ_s: an array with the number of sh. paths from s to each node v
Output: $w \in \mathbf{N}_v^-$ according to the weights $\mathscr{W}$

```
1: function RAND_PRED_IT(N_v^-, W, σ_s)
2:     r = uniform([[σ_s[v]]])
3:     i = 0, w = N_v^-[i]
4:     r = r − W(w → v)
5:     while r ≥ 0 do
6:         i = i + 1, w = N_v^-[i]
7:         r = r − W(w → v)
8:     end while
9:     return w
10: end function
```

To run this procedure we implement the method `rand_pred`$(\mathbf{N}_v^-, \mathscr{W})$ that returns a predecessor w of v according the number of shortest paths to each $w \in \mathbf{N}_v^-$.

PROPOSITION 4.1. (LINEAR PREPROCESSING) *The preprocessing stage of* linear *to build the successor graph from $s \in V$ and to compute the values of $\sigma_s(z)$ for all $z \in V$ is done in time $O(n+m)$.*

Proof. A BFS is done, starting from the node s. This costs $O(n+m)$ operations. Then the values of $\sigma_s(v)$ are computed by the Recurrence (4.4) in linear time of the successor graph size. ∎

The complexity of one query step corresponds to the biased random walk *BRW* and the random choice of predecessors in function `rand_pred_it`.

PROPOSITION 4.2. (LINEAR QUERY) *The worst-case time complexity of the generation phase is $O(n)$ and this bound is tight.*

Proof. Every node is checked at most once by the random walk. In the worst case, when the current node

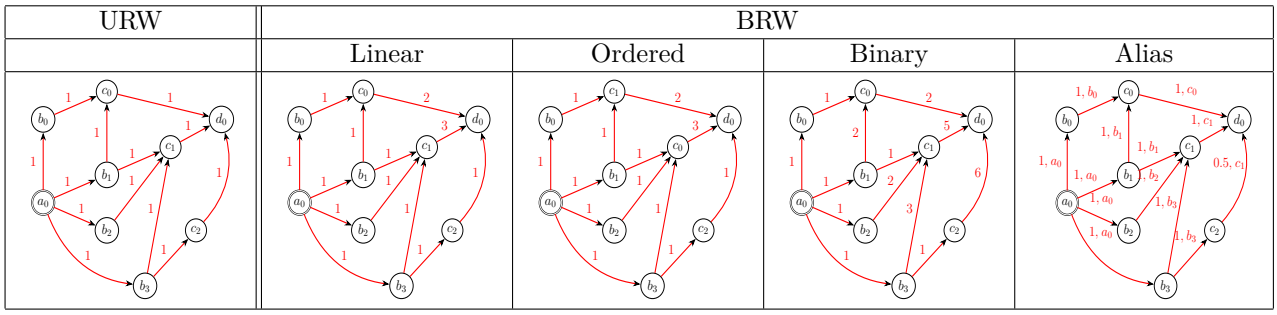


Figure 3: Successor Graph from $s = 0$ of the left graph of Figure 2 with the different weight functions $\mathscr{W}$. The leftmost graph correspond to *URW*. The four remaining weights correspond to the four implementations of *BRW*.

is at distance ℓ from the source s, we visit every node at distance $\ell - 1$ from s to find a predecessor. Moreover the complete bipartite graph is an example where can visit $n/2$ nodes to reconstruct a shortest path of length 2. So this bound is tight. □

4.2 Ordered Implementation In the *linear* implementation, we assumed an implicit order on the predecessor set $pre_s(v)$ of node v given by $\mathbf{N}^-$. However, the predecessors of a node can have any ordering, and thus for each node $v \in V_s$, there are $|\mathbf{N}_v^-|!$ possible orderings. The ordering can be optimized by minimizing the number of predecessor checks made by the function `rand_pred_it`. To achieve this, we order the list $\mathbf{N}_v^- = [w_0, \ldots, w_k]$ in decreasing order of the values of $\mathscr{W}(w_i \to v)$, so that $\mathscr{W}(w_0 \to v)$ is the largest value. For instance, in the successor graph given in Figure 3, the node c_1 is renamed as c_0 in the ordered version. This minimizes the number of checks made in Line 5 of the function `rand_pred_it`. Let π be the ordering where the list $\mathbf{N}_v^-$ is ordered by decreasing values of $\sigma_s(w)$ where $w = \mathbf{N}_v^-[i]$ and $i \in [\![\mathbf{N}_v^-]\!]$.

Proposition 4.3. (Optimal order) *Assuming uniform sampling of shortest paths, the ordering π is the order that minimizes the number of predecessor checks for each possible values $r \in [\![\sigma_s(t)]\!]$.*

Therefore, the ordering π is better than any other ordering and improves the average running time. However, the worst-case time complexity is still $O(n)$ regardless of the length of the shortest path.

Proposition 4.4. (Ordered Preprocessing) *The worst-case time of preprocessing is $O(m + n \log n)$.*

Proposition 4.5. (Ordered query) *The worst-case time complexity of the generation phase is $O(n)$ and this bound is tight.*

Proof. The proof is similar to the one of Proposition 4.2, the complete bipartite graph has of a path of length 2 whose reconstruction requires visiting $n/2$ nodes. □

4.3 Binary Implementation The procedure `rand_pred` used in Algorithm 2 corresponds to finding the position of an element r in a list. Thus, rather than using an iterative approach as it is the case now, it is possible to use binary search instead by adding few operations in the preprocessing stage. As we shall see this addition does not affect the theoretical running time of the preprocessing stage but improves the theoretical bounds of the queries afterward.

In the function `rand_pred_it`, we draw a value $r \in [\![\sigma_s(v)]\!]$ and if $w_0, \ldots, w_{k-1} = \mathbf{N}_v^-$, we want to return the node w_i such that

$$\sum_{j=0}^{i-1} \sigma_s(w_j) \leq r < \sum_{j=0}^{i} \sigma_s(w_j).$$

To optimize this, it is natural to work with partial sums of the weights of the incoming edges and perform a binary search on it. Thus, we assign new weights to the edges. In this scheme, the weights are defined as follows: for $v \in V_s$, let $[w_0, \ldots, w_{k-1}] = \mathbf{N}_v^-$. Then for each $w_i \to v \in E_s$, we assign the weight $\mathscr{W}(w_i \to v) = \sum_{j=0}^{i} \sigma_s(w_j)$. An illustration is given in the second from right graph of Figure 3.

Thus, the number of shortest paths $\sigma_s(t)$ from s to t is just the largest label of the incoming edges of t. We will denote this edge by t^*. Now we can determine the edge taken by the random walk more efficiently. Given a node v and a value r we execute a function `bin_search_index` (see Appendix A) that runs a binary search on $\mathbf{N}_v^-$ to find the index i such that

$$\mathscr{W}(w_{i-1} \to v) \leq r < \mathscr{W}(w_i \to v). \tag{4.5}$$

The predecessor of v we look for is then w_i. Thus, we only need to replace the function `rand_pred` of Algorithm 2 with its optimized version `find_pred_bin`

given in Algorithm 4 and to modify the weights $\mathscr{W}$ computed during the preprocessing.

Algorithm 4 Binary implementation of `rand_pred`

Input: $\mathbf{N}_v^-$: List of predecessors of each node v, $\mathscr{W} : E_s \to \mathbb{R}$ a weight function and σ_s: an array with the number of sh. paths from s to each node v
Output: $w \in \mathbf{N}_v^-$ according to the weights $\mathscr{W}$

```
1: function RAND_PRED_BIN(N_v^-, W, σ_s)
2:     r = uniform(⟦σ_s[v]⟧)
3:     i = bin_search_index(N_v^-, W, r)
4:     return w = N_v^-[i]
5: end function
```

PROPOSITION 4.6. (BINARY PREPROCESSING) *The preprocessing time complexity of Binary is $O(n+m)$.*

Proof. Additionally to $O(n+m)$ for the successor graph construction G_s and the values of σ_s. The weights on G_s is done in $O(m)$ operations using an array containing the values of $\sigma_s(t)$ for all $t \in V$. We compute the partial sum step by step writing the weight of $(w_i \to v)$ at the step when we add $\sigma_s(w_i)$ to the partial sum. ∎

The following proposition proves that using `rand_pred_bin` instead of `rand_pred_it` gives better results in the worst-case scenario.

PROPOSITION 4.7. (BINARY QUERY) *Let W be the sampled shortest path, let $\ell = |W|$ then the generator's worst-case time complexity is $O\left(\ell \log\left(\frac{n}{\ell}\right)\right)$.*

Proof. Suppose that $W = x_0 \to \cdots \to x_\ell$. To find the predecessor of x_i we do a binary search on all the incoming edges to find the edge satisfying Equation (4.5). This costs $O\left(\log(|\mathbf{N}_{x_i}^-|)\right)$ operations. So the total complexity of the random walk is of order of

$$\sum_{i=1}^{\ell} \log(|\mathbf{N}_{x_i}^-|) = \log\left(\prod_{i=1}^{\ell} |\mathbf{N}_{x_i}^-|\right).$$

Let n_k be the number of nodes at distance k from x_0. With the Remark 3.1, we obtain $|\mathbf{N}_{x_i}^-| \leq n_{i-1}$. Then, it follows

$$\prod_{i=1}^{\ell} |\mathbf{N}_{x_i}^-| \leq \prod_{i=1}^{\ell} n_{i-1} \leq \left(\frac{n_0 + \cdots + n_{\ell-1}}{\ell}\right)^{\ell},$$

by using the comparison between arithmetic and geometric means in the last step. We get:

$$\sum_{i=1}^{\ell} \log(|\mathbf{N}_{x_i}^-|) \leq \ell \log\left(\frac{n}{\ell}\right),$$

by noting that $n_0 + \cdots + n_{\ell-1} \leq n$ ∎

Thus, the complexity of a query of the generator depends on the length ℓ of the generated shortest path. The binary implementation has a better worst-case running time than the linear and ordered. For instance, if the size of the sampled path $\ell = \log n$ (which is typical in many random graph models), the generation time in worst case is $O(\log^2 n)$ with binary implementation and $O(n)$ for the linear and ordered one.

4.4 Alias Implementation We want to further improve the generation of the random walk. Note that for each node $v \in V_s$, we compute the distribution of the predecessors $\mathbf{N}_v^-$ already during the preprocessing stage. We use this information to design a constant-time random generator that selects a predecessor according to the distribution. We use the *Alias method* [25].

To each edge $(w_i, v) \in E$, we now associate a pair $\mathscr{W}(w_i \to v) = (t_i, al_i) \in [0,1] \times \mathbf{N}_v^-$ where t_i is a real number in $[0,1]$ called *threshold* and al_i is a predecessor in $\mathbf{N}_v^-$ called *alias*. We compute these weights (t, al) such that the following condition holds

$$\forall w_j \in \mathbf{N}_v^-,\ t_j + \sum_{\substack{i \in [\![|\mathbf{N}_v^-|]\!] \\ al_i = w_j}} (1 - t_i) = \frac{\sigma_s(w_j)}{\sigma_s(v)} \cdot |\mathbf{N}_v^-|. \tag{4.6}$$

Then, to choose a predecessor $w \in \mathbf{N}_v^-$ using the alias method, we draw an index $i \in [\![|\mathbf{N}_v^-|]\!]$ uniformly at random. Let $(t_i, al_i) = \mathscr{W}(w_i \to v)$, we choose the predecessor w_i with probability t_i and the predecessor al_i with probability $1 - t_i$. Thus, the determination of the predecessor is done in constant time. Once again we replace `rand_pred` of Algorithm 2 with its alias version `find_pred_al` given in Algorithm 5.

As an example, in the rightmost graph of Figure 3, we have $\mathscr{W}(c_2 \to d_0) = (0.5, c_1)$ which means that in the case where the edge $(c_2 \to d_0)$ is drawn (case $i = 2$), we walk to c_2 with probability 0.5 and to c_1 otherwise.

Algorithm 5 Alias implementation of `rand_pred`

Input: $\mathbf{N}_v^-$: predecessors of each v, $\mathscr{W}$: weight function
Output: $w \in \mathbf{N}_v^-$ using the alias method

```
1: function RAND_PRED_AL(N_v^-, W)
2:     i = uniform(⟦|N_v^-|⟧) , t = uniform([0,1])
3:     w = N_v^-[i], (t', al) = W(w → v)
4:     if t ≤ t' then return w else return al
5: end function
```

PROPOSITION 4.8. *For $v \in V$ and for all $w \in \mathbf{N}_v^-$:*

$$\mathbb{P}(\texttt{rand\_pred\_al}(\mathbf{N}_v^-, \mathscr{W}) = w) = \frac{\sigma_s(w)}{\sigma_s(v)}.$$

Proof. The proposition is a consequence of the Condition (4.6). Let $w_j \in \mathbf{N}_v^-$. Denote by X the uniformly drawn index in $[\![|\mathbf{N}_v^-|]\!]$. The probability law gives:

$$\mathbb{P}(w_j) = \frac{1}{|\mathbf{N}_v^-|}\left(\mathbb{P}(w_j|X=j) + \sum_{i\neq j}\mathbb{P}(w_j|X=i)\right).$$

For $i \neq j$ if $al_i \neq w_j$ then the probability $\mathbb{P}(w_j|X=i)$ is 0 else it is $1-t_i$. Thus we have

$$\mathbb{P}(w_j) = \frac{1}{|\mathbf{N}_v^-|}(t_j + \sum_{\substack{i\neq j \\ al_i = w_j}}(1-t_i)) = \frac{\sigma_s(w_j)}{\sigma_s(v)}.$$

□

Proposition 4.9. (Alias Preprocessing) *The preprocessing stage of Alias to compute the successor graph from s the associated weight function $\mathscr{W}$ for the Alias method is computed in time $O(n+m)$.*

Proof. The computation of the threshold and alias with respect to Condition (4.6) is done in linear time in the size of $\mathbf{N}_v^-$. To use the alias method at node $v \in V$, we compute $\mathscr{W}(w \to v)$ for all $w \in \mathbf{N}_v^-$, which costs $O(|\mathbf{N}_v^-|)$. Doing this for each node v in the successor graph costs $O(\sum_{v\in V}|\mathbf{N}_v^-|) = O(m)$ time. Since the rest of the preprocessing remains unchanged, the running time is then $O(n+m)$. □

Proposition 4.10. (Alias Query) *Let W be the sampled shortest path, let $\ell = |W|$ then the generator's worst-case time complexity is $O(\ell)$.*

Proof. To construct the path W, we have to call ℓ times the function `rand_pred_al`. This function executes in constant time, since we only draw two uniform random variables. Thus the total time complexity is $O(\ell)$. □

Theorem 4.1. *The alias implementation is an optimal implementation of Algorithm 1 in terms of the asymptotic worst case complexity for both the preprocessing and the sampling stages.*

Proof. It is a consequence of the Proposition 3.1 with the results from Propositions 4.9 and 4.10. □

5 Uniform sampling among all shortest paths

In the previous section, we have shown how to sample uniformly a shortest path with two fixed extremities (a source and a target node). Here, we extend our results to two related problems: Sampling uniformly a shortest path among all shortest paths starting at a fixed $s \in V$; Sampling uniformly a shortest path among all shortest paths in the graph. More formally:

Problem 2: source uniform shortest path: Node s being fixed, give a random generation algorithm satisfying $\forall W \in \mathbf{W}_{s\bullet}, \mathbb{P}(W) = 1/\sigma_{s\bullet}$ and for $W \notin \mathbf{W}_{s\bullet}, \mathbb{P}(W) = 0$.

Problem 3: uniform shortest path: Give a random generation algorithm satisfying $\forall W \in \mathbf{W}, \mathbb{P}(W) = 1/\sigma$ and for $W \notin \mathbf{W}, \mathbb{P}(W) = 0$.

For both problems, the naive algorithms consist in computing all shortest paths in the graph, storing them in a list and drawing one uniformly. Again these algorithms can be of exponential complexity.

5.1 Source uniform shortest path Here we suppose that a source node s is fixed and we want to sample a shortest path $W \in \mathbf{W}_{s\bullet}$ uniformly at random. One method would be to sample $t \in V$ uniformly at random and call `random_walk`$(s, t, G_s, \mathscr{W})$ after doing a preprocessing. However, this method fails to sample uniformly shortest paths of $\mathbf{W}_{s\bullet}$ since some target nodes admit more shortest paths, starting at s and ending to them, than others. Instead, we must sample a target node t according to the distribution $\sigma_s(v)$ for all $v \in V$. Therefore, as before, the preprocessing needs to compute the successor graph G_s and the weights of $\mathscr{W}$ (using for example Alias sampling from Section 4). But additionally, we use an Alias preprocessing on a list src$[s]$ containing $(v, \sigma_s(v))$ for all $v \in V$: this step adds a term $O(n)$ to the preprocessing: thus the overall preprocessing step running time is $O(m+n)$ and the space requirement stays the same. Then, the query step only adds a constant time $O(1)$ to define the appropriate target node t and calls the function `random_walk`$(s, t, G_s, \mathscr{W})$. Thus the query step executes in $O(\ell)$. We call this method `source_random_walk`$(s, G_s, \mathscr{W}, \text{src}[s])$.

5.2 Uniform shortest path Now, we aim at sampling a shortest path W in the set of all shortest paths of the graph $\mathbf{W}$. It is tempting to once again choose uniformly a source s and a target t nodes and then call *BRW* to uniformly sample a shortest path from s to t. We call this algorithm `uniform_s_t`(G). Although every shortest path can be drawn, this algorithm is again biased and we show that any algorithm that samples uniformly a source and target nodes is biased. The average length of a shortest path drawn with this algorithm corresponds to the average distance in the graph. This happens regardless of the weights chosen for the random walk. We define the two following measures:

Definition 5.1. (Average distance [17]) *Let $G = (V, E)$ be a connected and unweighted graph. We define*

the average distance in G to be

$$d_G = \frac{1}{n^2} \cdot \sum_{(s,t)\in V^2} d(s,t).$$

DEFINITION 5.2. (AVERAGE SHORTEST PATH LENGTH) *Let $G = (V, E)$ be a connected and unweighted graph. We define the average length of a shortest path in G by*

$$\ell_G = \frac{1}{\sigma} \sum_{(s,t)\in V^2} \sigma_s(t) \cdot d(s,t)$$

In the literature, the average distance d_G is often called *average path length* or *average shortest path length* [2, 12]. This can be somewhat confusing since the average shortest path length corresponds to ℓ_G and not to d_G. To our knowledge there has been no systematic research studies on ℓ_G. From the Definition of ℓ_G any Algorithm solving the uniform shortest path problem will output paths of average length ℓ_G. The values of d_G and ℓ_G do not coincide in general as we see next:

LEMMA 5.1. *In the 4-cycle graph C_4 we have $d_{C_4} = 1$ and $\ell_{C_4} = 1.2$*

LEMMA 5.2. *Let $\mathcal{A}$ be an algorithm that generates shortest paths drawing $(s,t) \in V^2$ uniformly at random. The mean length of paths sampled by $\mathcal{A}$ is d_G.*

PROPOSITION 5.1. *Any algorithm that draws $(s,t) \in V^2$ uniformly at random and next generates a path $W \in \mathbf{W}_{st}$ cannot solve the uniform shortest path problem.*

Proof. By contradiction. Suppose we have a uniform shortest path generator, then the mean length of the generated shortest path is ℓ_G. However the Lemma 5.2 shows that such the average length of the generated paths is d_G in the graph and Lemma 5.1 states that $d_G \neq \ell_G$ in general. ∎

REMARK 5.1. *Proposition 5.1 is true regardless of how we choose the path $W \in \mathbf{W}_{st}$.*

The bias comes from the fact that some source-target have a larger number of shortest paths than others. For a fixed $W \in \mathbf{W}_{st}$, the probability to sample W with `uniform_s_t` is $\mathbb{P}(W) = \frac{1}{n^2} \cdot \frac{1}{\sigma_s(t)}$. Then the larger is $\sigma_s(t)$ the smaller is $\mathbb{P}(W)$.

This last result gives the intuition that, in a non-biased shortest path sampling algorithm, the probability to have (s,t) as extremities should be proportional to $\sigma_s(t)$. We then define the `biased_s_t`(G) algorithm that given a graph determines a source node s with respect to the distribution $\sigma_{s\bullet}/\sigma$ for all $s \in V$ with Alias sampling. Then the algorithm calls `source_random_walk`$(s, G_s, \mathscr{W}, \mathrm{src}[s])$.

	Preprocess.	Space	Query	Sect.
SUSP	$O(m+n)$	$O(m+n)$	$O(\ell)$	5.1
USP	$O(mn+n^2)$	$O(mn+n^2)$	$O(\ell)$	5.2

Table 2: Overview of the sampling algorithms complexities for Problem 2, Source uniform shortest path (SUSP) and Problem 3, Uniform shortest path (USP).

PROPOSITION 5.2. *The algorithm `biased_s_t` solves the uniform shortest path problem*

Proof. Let $W \in \mathbf{W}$ and (s,t) its source-target pair. The probability of sampling W with `biased_s_t` is

$$\mathbb{P}(W) = \mathbb{P}(s).\mathbb{P}(t|s).\mathbb{P}(W|(s,t)) = \frac{\sigma_{s\bullet}}{\sigma}.\frac{\sigma_s(t)}{\sigma_{s\bullet}}.\frac{1}{\sigma_s(t)} = \frac{1}{\sigma}$$

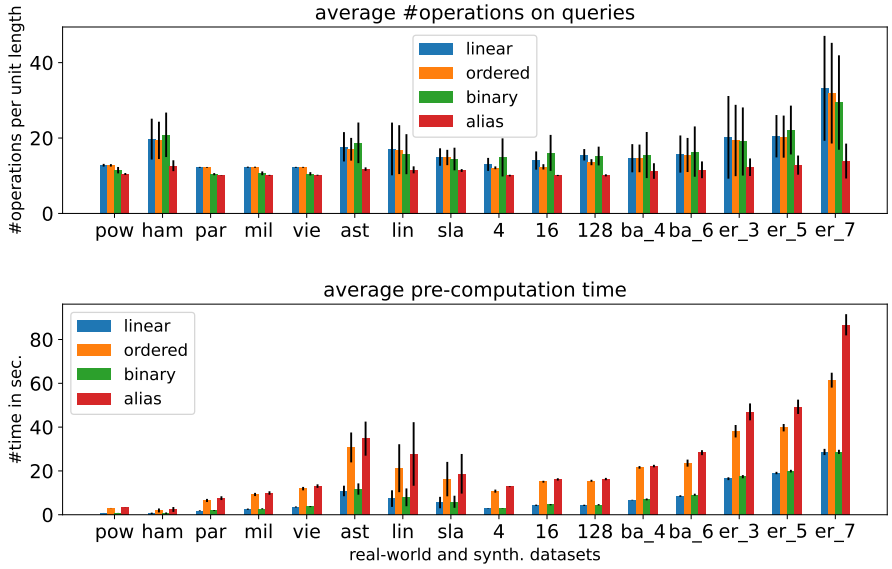
∎

The preprocessing step now requires the values $\sigma_s(v)$ for all $s \in V$ and for all $v \in V$. Therefore, the overall preprocessing needs $O(mn+n^2)$ since we have to compute of the successor graph for each node and then compute the recurrence from Equation (4.4). The space requirement is the same as well $O(mn+n^2)$. Finally the query step now requires one additional constant $O(1)$ time sampling to determine the starting node s and then, the rest is done in $O(\ell)$.

6 Experimental evaluation and application

In our experiments, we compare *BRW* and its various implementations to assess their performance in practice on both real-world and synthetic datasets. We evaluate these variants in their two stages: preprocessing and querying. Our algorithms are implemented in C to enhance efficiency. The code is open-source and will be provided in the final version of this paper. We conducted our experiments on an Intel(R) Xeon(R) Silver 4210R CPU at 2.40GHz without parallel processes. In fact, the preprocessing can be fully parallelized since constructing successor graphs from each node is an independent task. We used several real-world datasets from different domains, including scientific collaboration networks, city networks, social networks, and power grids. For synthetic networks, we used Erdős-Rényi, Barabási-Albert, and 2-dimensional grids.

Figure 4 summarizes our experiments. On the preprocessing stage results show that the ordered implementation takes more time to compute than the linear and binary implementations due to the additional sorting step required. The Alias implementation also requires more computations since we need to associate a pair of values with each edge. Although the theoretical worst-case running times are similar for all variants,



data	dir.	#nodes	#edges	ref.
power_grid	u	4.94K	6.59K	[15]
hamster_full	u	2.43K	16.6K	[15]
paris	d	9.52K	18.3K	[3]
milan	d	12.9K	25.3K	[3]
vienna	d	16.1K	35.7K	[3]
astro_ph	u	18.8K	198K	[15]
linux_mail	d	26.9K	237K	[15]
slashdot	d	51.1K	130K	[15]
4	u	16.4K	28.7K	x
16	u	16.4K	31.7K	x
128	u	16.4K	32.5K	x
ba_4	u	16.4K	65.5K	x
ba_6	u	16.4K	98.3K	x
er_3	u	16.4K	238K	x
er_5	u	16.4K	398K	x
er_7	u	16.4K	558K	x

Figure 4: (Up Left): Query complexity on the different graphs. The y-axis corresponds to the average number of operations (arithmetic, tests, affectation) made by 50 randomly selected source-target (s,t) pairs. For each (s,t) we generated 50 000 shortest paths and summed the number of operations made and divided them by $(50\ 000 \times d(s,t))$ the black bars gives the standard deviation. (Down Left): Preprocessing complexity on the different graphs. The y-axis corresponds to the average running time per node in seconds with the black bars giving the standard deviation. (Right): Real-world and synthetic datasets. Upper part corresponds to the real-world dataset and the bottom part corresponds to the synthetic one. Columns from left to right indicate whether the graph is directed 'd' or undirected 'u', the number of nodes and the number of edges. Our synthetic dataset contains from top to bottom: bi-dimensional grid graphs of 16 384 nodes with 4, 16 and 128 rows, Barabási-Albert graphs with parameter m equals to 4 and 6, Erdős-Rényi graphs with $p = \frac{i \log(n)}{n}$ for $i = 3.5$ and 7.

in practice, the Alias and ordered implementations are slower by a factor of at least 2.

On the query stage, random walks run instantly once the preprocessing is complete. We therefore count the average number of operations performed by each query (sampling), normalized by the distance between pairs of sampled nodes to avoid having large differences between the dataset results. The Alias implementation consistently outperforms the others, confirming our theoretical analysis. The binary implementation performs well when the average distance is short. Therefore, for 2-dimensional grids where the average shortest path is linear, the binary implementation performs the worst. Finally, the ordered implementation is always at least as efficient as the linear one and sometimes better.

Finally, we explore a potential application of our scheme in traffic modeling. One of the most classical models for determining transportation forecasts is the *four-step transportation model* [9]. The second step of this model, known as *Trip Distribution*, matches tripmakers' origins and destinations, creating a matrix $\mathbf{T}$ where the coefficient t_{ij} represents the number of trips made from origin i to destination j per time unit. These trips are typically assumed to follow the shortest paths. The fourth step in the model, called *Traffic Assignment*, assigns specific routes to each trip. The simplest assignment model is the *all-or-nothing* assignment, where all trips from i to j are assigned to the same shortest path. Other route assignment models distribute trips across different paths based on a probability distribution that reflects their relative likelihood. A classical probabilistic assignment model is presented in [10]. This model, known as the two-pass Markov model, includes a heat parameter $\theta \in \mathbb{R}^+$. It calculates node/link transition probabilities during an initial pass through the network and assigns trips during a subsequent pass. Trips are assigned to what the author calls "reasonable paths," which are not necessarily the shortest. When $\theta = \infty$, traffic is assigned exclusively on shortest paths. Therefore, *BRW* can be used in this context to assign traffic uniformly. Once an assignment is done for each of the t_{ij} trips for all origin-destination pairs, the congestion $c(e)$ of a given edge e can be measured by to the number of paths passing

through it.

Figure 5 shows the histogram of edge congestion when paths are assigned using uniform sampling compared to the all-or-nothing heuristic. It is evident that the uniform heuristic results in less congested edges compared to the all-or-nothing heuristic. Specifically, as congestion increases (along the x-axis), the values of the uniform heuristic bars (red) are consistently lower than those of the all-or-nothing bars (blue). The all-or-nothing heuristic results in edges with congestion ranging from over 18 000 to nearly 25 000, while the uniform heuristic has no edges with such high congestion at all.

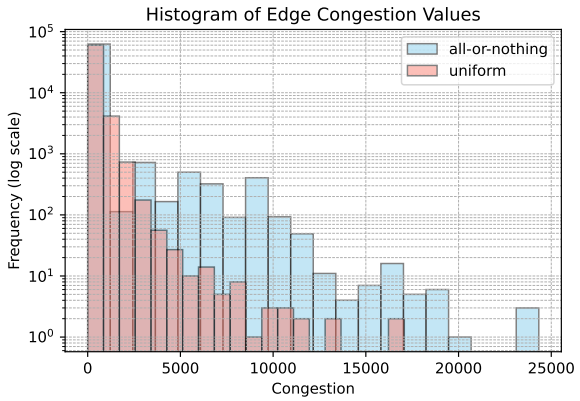


Figure 5: Congestion histogram of the edges on the graph 128 (128×128 grid). We chose randomly 10 nodes and constructed a matrix of origin-destination $\mathbf{T}$ of size 10×10 between these nodes with values for each t_{ij} sampled uniformly in $[\![10\ 000]\!]$.

References

[1] Dimitris Achlioptas, Aaron Clauset, David Kempe, and Cristopher Moore. On the bias of traceroute sampling: or, power-law degree distributions in regular graphs. *Journal of the ACM*, 56(4):1–28, 2009.

[2] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[3] Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, environment and urban systems*, 65:126–139, 2017.

[4] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[5] Fabio Ciulla, Nicola Perra, Andrea Baronchelli, and Alessandro Vespignani. Damage detection via shortest-path network sampling. *Physical review E*, 89(5):052816, 2014.

[6] Aaron Clauset and Cristopher Moore. Traceroute sampling makes random graphs appear to have power law degree distributions. *arXiv preprint cond-mat/0312674*, 2003.

[7] Christophe Crespelle and Fabien Tarissan. Evaluation of a new method for measuring the internet degree distribution: Simulation results. *Computer Communications*, 34(5):635–648, 2011.

[8] Luca Dall'Asta, Ignacio Alvarez-Hamelin, Alain Barrat, Alexei Vázquez, and Alessandro Vespignani. Exploring networks with traceroute-like probes: Theory and simulations. *Theoretical Computer Science*, 355(1):6–24, 2006.

[9] Juan de Dios Ortúzar and Luis G Willumsen. *Modelling transport*. John wiley & sons, 2024.

[10] Robert B Dial. A probabilistic multipath traffic assignment model which obviates path enumeration. *Transportation research*, 5(2):83–111, 1971.

[11] Abraham D. Flaxman and Juan Vera. Bias reduction in traceroute sampling–towards a more accurate map of the internet. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 1–15. Springer, 2007.

[12] Agata Fronczak, Piotr Fronczak, and Janusz A Hołyst. Average path length in random networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 70(5):056110, 2004.

[13] Antoine Genitrini, Martin Pépin, and Frédéric Peschanski. A quantitative study of fork-join processes with non-deterministic choice: application to the statistical exploration of the state-space. *Theor. Comput. Sci.*, 912:1–36, 2022.

[14] Jean-Loup Guillaume and Matthieu Latapy. Relevance of massively distributed explorations of the internet topology: Simulation results. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 2, pages 1084–1094. IEEE, 2005.

[15] Jérôme Kunegis. Konect: the Koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*, pages 1343–1350, 2013.

[16] Anukool Lakhina, John W. Byers, Mark Crovella, and Peng Xie. Sampling biases in ip topology measurements. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 332–341. IEEE, 2003.

[17] Vito Latora, Vincenzo Nicosia, and Giovanni Russo. *Complex networks: principles, methods and applications*. Cambridge University Press, 2017.

[18] Jérémie Leguay, Matthieu Latapy, Timur Friedman, and Kavé Salamatian. Describing and simulating internet routes. *Computer Networks*, 51(8):2067–2085, 2007.

[19] Johan Oudinet, Alain Denise, Marie-Claude Gaudel, Richard Lassaigne, and Sylvain Peyronnet. Uniform Monte-Carlo model checking. In *International Conference on Fundamental Approaches to Software Engineering*, pages 127–140. Springer, 2011.

[20] Thomas Petermann and Paolo De Los Rios. Exploration of scale-free networks: Do we measure the real exponents? *The European Physical Journal B*, 38:201–204, 2004.

[21] Márton Pósfai, Attila Fekete, and Gábor Vattay. Shortest-path sampling of dense homogeneous networks. *Europhysics Letters*, 89(1):18007, 2010.

[22] Franco P. Preparata and Michael I. Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[23] Alireza Rezvanian and Mohammad Reza Meybodi. Sampling social networks using shortest paths. *Physica A: Statistical Mechanics and its Applications*, 424:254–268, 2015.

[24] Christian Sommer. Shortest-path queries in static networks. *ACM Computing Surveys (CSUR)*, 46(4):1–31, 2014.

[25] Alastair J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 8(10):127–128, 1974.

[26] Huijuan Wang and Piet Van Mieghem. Sampling networks by the union of m shortest path trees. *Computer Networks*, 54(6):1042–1053, 2010.

[27] Guo-Qing Zhang, Shi Zhou, Di Wang, Gang Yan, and Guo-Qiang Zhang. Enhancing network transmission capacity by efficiently allocating node capability. *Physica A: Statistical Mechanics and its Applications*, 390(2):387–391, 2011.

A Appendix

Figure 6 shows that the number of shortest paths between two nodes can be exponential. Therefore the naive algorithm presented in Section 2 consisting of listing all shortest paths in the graph should be avoided.

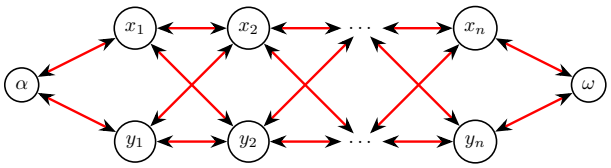


Figure 6: Graph with $2n+2$ nodes and 2^n shortest paths between α and ω.

In the following proof we compute the probability of each shortest path between α and ω of the graph G_k 1 when we use the algorithm random weights to sample a path. This will show that random weights leads to a biased distribution on the sampled shortest paths.

Proof. [of Proposition 2.1] We denote $a, b_1, \dots, b_k, c_1, \dots, c_k, d, e, f$ the random weights on the edges as in figure 7. In the algorithm 1 these weights follows a continuous uniform distribution on $[1-\frac{1}{n}, 1+\frac{1}{n}]$. For the sake of simplicity, we recenter and reduce the variables. This does not change the selected shortest path (we just subtract by $1-\frac{1}{n}$ every weights and then multiply them all by $\frac{n}{2}$). From now on, we suppose that all the weights follow the standard uniform distribution $U(0,1)$.

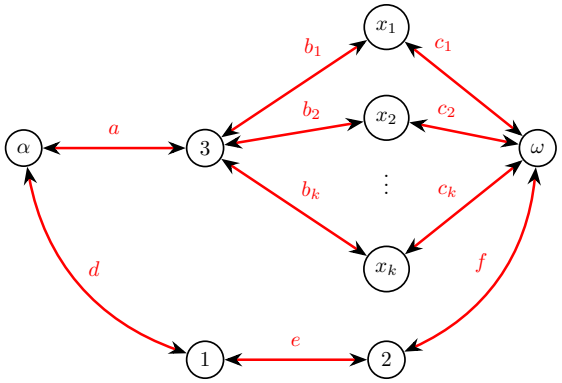


Figure 7: The family of graphs G_k with the random variables defining the weights of the edges.

We define the following random variables:

- $Y_i = b_i + c_i$ for all $1 \leq i \leq k$
- $Z = d + e + f$: total weight of the bot path.

Let $W_0 = \alpha \to 1 \to 2 \to \omega$ and $W_i = \alpha \to 3 \to x_i \to \omega$ for $1 \leq i \leq k$. We want to compute the probability

$$\mathbb{P}(W_0) = 1 - \sum_{i=1}^{k} \mathbb{P}(W_i)$$

All the weights follow the same distribution, then by symmetry :

$$\mathbb{P}(W_0) = 1 - k\mathbb{P}(W_1)$$

Note that

$$\begin{aligned}\mathbb{P}(W_1) &= \mathbb{P}\left((a + Y_1 < Z) \cap \left(\bigcap_{i=2}^{k} (a + Y_1 < a + Y_i)\right)\right) \\ &= \mathbb{P}\left((a + Y_1 < Z) \cap \left(\bigcap_{i=2}^{k} (Y_1 < Y_i)\right)\right)\end{aligned}$$

We condition on $t \in [0,2]$ the value of Y_1. Denote by f_Y the probability density function of Y_1.

$$\mathbb{P}(W_1) = \int_0^2 \mathbb{P}\left((a + t < Z) \cap \left(\bigcap_{i=2}^{k} (t < Y_i)\right) \middle| Y_1 = t\right) f_Y(t)dt$$

The weights are drawn independently thus the variables $(Y_i)_{1 \leq i \leq k}$ and $Z - a$ are independent. It follows :

$$\begin{aligned}&\mathbb{P}\left((a + t < Z) \cap \left(\bigcap_{i=2}^{k} (t < Y_i)\right) \middle| Y_1 = t\right) \\ &= \mathbb{P}(t < Z - a). \prod_{i=2}^{k} \mathbb{P}(t < Y_i)\end{aligned}$$

All the variables Y_i follow the same distribution (sum of two independent standard uniform variables). We denote F_Y the cumulative density function of the Y_i and F_{Z-a} that of $Z - a$. Then

$$\mathbb{P}(W_1) = \int_0^2 (1 - F_{Z-a}(t))(1 - F_Y(t))^{k-1} f_Y(t)dt$$

Because the density function is the derivative of the cumulative density function we can do an integration by parts.

$$k\mathbb{P}(W_1) = 1 - F_{Z-a}(0) - \int_0^2 (1 - F_Y(t))^k f_{Z-a}(t)dt$$

Where f_{Z-a} is the density function of $Z - a$. Finally

$$\mathbb{P}(W_0) = F_{Z-a}(0) + \int_0^2 (1 - F_Y(t))^k f_{Z-a}(t)dt$$

a and Z are independent. Y and Z follow the *Irwin-Hall distribution* for $n = 2$ and $n = 3$ respectively then

$$F_{Z-a}(0) = \mathbb{P}(Z < a) = \int_0^1 F_Z(t)dt = \frac{1}{4!} = \frac{1}{24}$$

and the density of $Z - a$ is the convolution product of the density of Z and $-a$

$$f_{Z-a}(t) = \int_{-1}^{0} f_Z(t-u)du = \int_{0}^{1} f_Z(t+u)du$$

The final formula for $\mathbb{P}(W_0)$ is then

$$\text{(A.1)}\quad \mathbb{P}(W) = \frac{1}{24} + \int_0^2 (1-F_Y(t))^k \int_0^1 f_Z(t+u)du\,dt$$

For $k = 2$ the Equation A.1 gives $\mathbb{P}(W_0) = \frac{737}{2016}$. As k grows to infinity the term $(1-F_Y(t))^k$ converge to 0 for $t \in]0, 2]$. Moreover the function $t \mapsto (1-F_Y(t))^k f_{Z-a}(t)$ is continuous on $[0, 2]$ then it is dominated by a constant. Then the dominated convergence theorem ensures that the integral converge towards 0. Therefore Equation A.1 gives $\mathbb{P}(W_0) \overset{k\to+\infty}{\longrightarrow} \frac{1}{24}$.

□

The following proof stresses out that the order that minimizes the number of predecessors checks is the order where the nodes with the largest $\sigma_s(v)$ are those with the smallest index.

Proof. [of Proposition 4.3] Fix a node $v \in V$. Let $\mathbf{N}_v^- = [w_0, \ldots, w_k]$. Note that `rand_pred_it`$(\mathbb{N}_v^-, \mathscr{W})$ returns the predecessor w_i for exactly $\sigma_s(w_i)$ different values of $r \in [\![\sigma_s(v)]\!]$. When the function returns w_i, it has iteratively checked all the i first predecessors. Then the average number of predecessors checked before finding the right one in `rand_pred_it` is

$$\frac{1}{\sigma_s(v)} \sum_{i=1}^{k} i \cdot \sigma_s(w_i).$$

To minimize the sum the largest values of $\sigma_s(w_i)$ should be matched with the smallest indexes i. That is the ordering π. □

We give the proof of the preprocessing complexity of the ordered algorithm.

Proof. [of Proposition 4.4] We need to ensure that the function `rand_pred_it` traverses the values of $\mathbf{N}_v^- = [w_0, \ldots, w_k]$ in decreasing order of the values of $\sigma_s(w_i)$. To achieve this, we rewrite $\mathbf{N}_v^-$ after computing all the values $\sigma_s(v)$. We start by ordering the nodes $w \in V$ by decreasing values of $\sigma_s(w)$, which costs $O(n \log n)$. For each $w \in V$ we store its successors in $\mathbf{N}_w^+$. That is $v \in \mathbf{N}_w^+$ if and only if $w \in \mathbf{N}_v^-$. Then, for every $w \in V_s$ in decreasing order of $\sigma_s(w)$, and for every successor node $v \in \mathbf{N}_w^+$, we write w in $\mathbf{N}_v^-$. After this operation, all the $\mathbf{N}_v^-$ are rewritten in decreasing order of $\sigma_s(w)$. The operation of rewriting costs $O(m)$ as $\sum_{v\in V} |\mathbf{N}_v^+| \leq m$. Thus, the overall preprocessing stage cost becomes $O(m + n \log n)$. □

Binary search based on a weight function.

Algorithm 6 Binary implementation of `rand_pred`

```
function BIN_SEARCH_INDEX(N_v^-, 𝒲, r)
    i = −1, j = |N_v^-| − 1
    while j − i > 1 do
        x = ⌊(i+j)/2⌋, w = N_v^-[x]
        if 𝒲(w → v) > r then
            j = x
        else
            i = x
        end if
    end while
    return j
end function
```

We present the simplest counter-example graph G where the average distance d_G and the average shortest path length ℓ_G are distinct : the 4-cycle graph C_4.

Proof. [of Lemma 5.1] Let $S_d = \sum_{s\in V} \sum_{t\in V} d(s,t)$ and $S_\ell = \sum_{s\in V} \sum_{t\in V} \sigma_s(t) d(s,t)$. Let s be a node of C_4.

$$\sum_{t\in V} d(s,t) = 0 + 2 \times 1 + 2 = 4$$

Then by symmetry

$$d_{C_4} = \frac{S_d}{n^2} = \frac{4 \times \sum_{t\in V} d(s,t)}{4^2} = \frac{4}{4} = 1$$

Moreover as there are two shortest paths between antipodal nodes, we have $\sigma_{s\bullet} = 5$ shortest paths starting from s and

$$\sum_{t\in V} \sigma_s(t) d(s,t) = 1 \times 0 + 1 \times 1 + 2 \times 2 + 1 \times 1 = 6$$

Then by symmetry

$$\ell_{C_4} = \frac{S_\ell}{4 \times \sigma_{s\bullet}} = \frac{4 \times \sum_{t\in V} \sigma_s(t) d(s,t)}{4 \times 5} = \frac{6}{5} = 1.2$$

□

Proof. [of Lemma 5.2] Let L be the random variable indicating the length of the path W sampled with $\mathcal{A}$. Then the expected value of L is

$$\begin{aligned}\mathbb{E}(L) &= \sum_{W\in\mathbf{W}} |W|.\mathbb{P}(W) = \sum_{s,t\in V} d(s,t) \sum_{W\in\mathbf{W}_{st}} \mathbb{P}(W) \\ &= \sum_{s,t\in V} d(s,t)\mathbb{P}(s,t) = \frac{1}{n^2} \sum_{s,t\in V} d(s,t) = d_G\end{aligned}$$

□