Optimal Uniform Shortest Path Sampling

Simon Dreyer, Antoine Genitrini, Mehdi Naima

Optimal Uniform Shortest Path Sampling

Simon Dreyer, Antoine Genitrini, and Mehdi Naima

Sorbonne Université – LIP6 – CNRS UMR 7606
{Firstname.Lastname}@lip6.fr

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 a new variant of the Alias method is optimal in terms of worst-case running time and number of random bits needed, 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.

Keywords: Graph Algorithm · Random Generation · Uniform Sampling · Shortest Paths · Alias Sampling

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 [4] employ this method to evaluate network damage, while [7] analyzes shortest path sampling as an approximation of traceroute paths. The practice of approximating graphs through shortest path sampling has been explored in [14,17,1,20], where the authors investigate the biases introduced by exploring a graph via random shortest paths. In [11], 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 [24] 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 [7,4,18,24,15,6], 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 [14,5,23,9]. 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.

Main contributions : We propose several algorithms for the uniform shortest path sampling and analyse their theoretical complexities as well as their behaviors in practice. We also propose a new variant of the Alias sampling (method to sample from a discrete probability distribution) that uses only integer arithmetic avoiding bias due to floating arithmetic approximations. This variant is then used in an unranking scheme to ensure random bits optimality as well as optimal running time.

The paper is organized as follows. In Section 2 after presenting our formalism we give the problem statement and review the state of the art. Then, in Section 3 we present 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. In Section 5 we present an unranking algorithm that uses the integer variant of the Alias method. Finally, in Section 6, we assess the performance of our algorithms using both real-world and synthetic graphs. 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 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, $\mathbf{W}_{s\bullet} = \cup_{t\in V}\mathbf{W}_{st}$ 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: 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 $W \notin \mathbf{W}_{st}, \mathbb{P}(W) = 0$.

The simplest approach addressing this problem, 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 [7,4,18,24,15,6]. 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 Fig. 4 in Appendix A.1. Therefore, we aim to design polynomial-time algorithms that still ensure uniform sampling.

To overcome this issue, one idea used in works such as [14,5,23,9] 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. 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 see Appendix A.1 for a counterexample.

3 A random walk approach for uniform generation of shortest paths

In this section, source s and target t nodes are fixed and we design algorithms to tackle the source-target uniform shortest path problem. An overview of the complexity results of our approaches is presented in Table 1.

Algorithm	Bit opt.	Preprocessing	Space	Query	Section
BRW-linear	no	$O(m+n)$	$O(m+n)$	$O(n)$	4.1
BRW-Ordered	no	$O(m+n\log n)$	$O(m+n)$	$O(n)$	4.2
BRW-Binary	no	$O(m+n)$	$O(m+n)$	$O(\ell\log(\frac{n}{\ell}))$	4.3
BRW-Alias	no	$O(m+n)$	$O(m+n)$	$O(\ell)$	4.4
Unrank-Alias	yes	$O(m+n)$	$O(m+n)$	$O(\ell)$	5

Table 1: Overview of the discussed random sampling methods: running times correspond to the worst-case scenario given a graph G with n nodes and m edges. The value ℓ is the length of the sampled shortest path and Bit opt. corresponds to whether the algorithm is optimal in terms of random bits usage.

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 step 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 [19] as opposed to single-shot. There are many problems on shortest paths that operate with this two stages procedure. We refer to [21] for a review of these methods.

Definition 1 (Predecessor Set [3]). *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:*

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

Said differently, the set $pre_s(v)$ contains the nodes w such that the edge $w \to v$ is the last one of a shortest path from s to v.

Definition 2 (Successor Graph). *Let s be a node. The successor graph related to s 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 are the nodes accessible from s.

Said differently E_s is the subgraph of G containing all the edges belonging to a shortest path starting 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 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 BFS that keeps all the optimal predecessors of a node. Fig. 1 introduces a graph-example and its associated successor graph G_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 1, once the preprocessing stage has been computed. During the

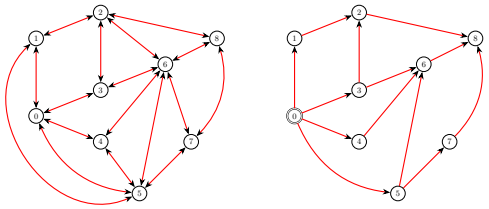


Fig. 1: (left) A graph and (right) the associated successor graph for $s = 0$.

latter preprocessing step, a *weight distribution* for edges of G_s is computed. We will discuss the weight distribution and the function `rand_pred` in Section 4. The second stage computes a *random walk* that starts from t and goes backwards until reaching s. Different random walks could be considered on $G_s = (V_s, E_s)$ by using different *weight functions* $\mathscr{W}$ that assign weights to the edges E_s. Then, the random walk standing on node $v \in V_s$ calculates a predecessor w given by the list $\mathbf{N}_v^- = [w_0, \ldots, w_k]$ according to the weight function $\mathscr{W}$.

Proposition 1 (Generic random walk complexity). *Any implementation of the preprocessing stage and Algorithm 1 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.*

Algorithm 1 Generic Random Walk

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, 𝒲)
    path = [ ], v = t
    while v ≠ s do
        path = [v] + path
        v = rand_pred(N_v^-, 𝒲)   ▷ choosing a random predecessor according to 𝒲
    end while
    return [s] + path
end function
```

The most straightforward random walk that we consider is the *Unbiased Random Walk* where the *weight function* $\mathscr{W}$ assigns weight 1 to all the edges in the successor graph. The random walk is thus unbiased. We can then show that the probability of sampling a shortest path $W \in \mathbf{W}_{st}$ is: $\mathbb{P}(W) = \prod_{v \in W \setminus \{s\}} \frac{1}{|\mathbf{N}_v^-|}$, which is not the uniform probability in general. Such a random walk is called *isotropic* in the literature (cf. e.g. [10]). The unbiased random walk (*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).

3.2 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:

$$\forall (u, v) \in E_s, \mathscr{W}(u \to v) = \sigma_s(u). \tag{1}$$

Proposition 2 (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` (inside the preprocessing stage) and `rand_pred` (in Algorithm 1) are implemented in different ways depending on the amount of preprocessing done 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 alternative 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 appear in 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:

$$\sigma_s(s) = 1 \text{ and } \forall v \neq s,\ \sigma_s(v) = \sum_{w \in pre_s(v)} \sigma_s(w). \tag{2}$$

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

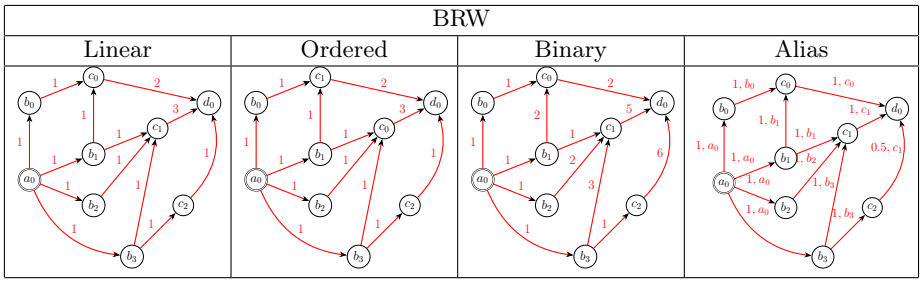


Fig. 2: Successor graph from $s = 0$ for the left graph of Fig. 1 with the different weight functions $\mathscr{W}$.

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 (2). An illustration of the successor graph with weights is given in second from left graph of Fig. 2. The *query step* then defines a predecessor by sampling a value $r \in [\![\sigma_s(v)]\!]$(where $[\![n]\!] = \{1, \ldots, n-1\}$) and iterating through each predecessor of v to determine where r falls within the predecessor values. To run this procedure we implement the function `rand_pred`$(\mathbf{N}_v^-, \mathscr{W})$ that returns a predecessor w of v according the number of shortest paths to each $w \in \mathbf{N}_v^-$ with `rand_pred_it` that is given in Appendix A.2.

Proposition 3 (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 (2) 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.

Proposition 4 (linear Query). *The worst-case time complexity of the generation phase is $O(n)$ and 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 $\mathbf{N}^-$. 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 Fig. 2, 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 $i \in [\![\mathbf{N}_v^-]\!]$. This classical ordering is called *boustrophedonic order* int the context of recursive generation [8].

Proposition 5 (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 unchanged.

Proposition 6 (Ordered Preprocessing and Query). *The worst-case time of preprocessing is $O(m + n \log n)$ and the worst-case time complexity of the generation phase is $O(n)$ and this bound is tight.*

4.3 Binary Implementation

The procedure `rand_pred` used in Algorithm 1 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 obviously improves the theoretical bounds of the queries afterward.

In function `rand_pred_it` (Appendix A.2), 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 Fig 2. 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 binary search on $\mathbf{N}_v^-$ to find the index i such that

$$\mathscr{W}(w_{i-1} \to v) \leq r < \mathscr{W}(w_i \to v). \tag{3}$$

The predecessor of v we look for is then w_i. Thus, we use Algorithm 4 (cf. Appendix A.2) to implement `rand_pred` and modify the weights $\mathscr{W}$ computed during the preprocessing.

Proposition 7 (Binary Preprocessing). *The preprocessing time complexity 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. □

Proposition 8 (Binary Query). *Let W be the sampled shortest path, let $\ell = |W|$ then the generator's worst-case time complexity is $O\left(\ell \log\left(\frac{n}{\ell}\right)\right)$.*

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. This distribution is fixed, and not dynamical. We can use this constraint to design a constant-time random generator that selects a predecessor by applying the classical *Alias method* defined by Walker [22].

Let us recall the fundations of the method. 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

$$\forall w_j \in \mathbf{N}_v^-,\ t_j + \sum_{\substack{i \in [\![|\mathbf{N}_v^-|]\!] \\ al_i = w_j}} (1 - t_i) = \frac{\sigma_s(w_j)}{\sigma_s(v)} \cdot |\mathbf{N}_v^-|. \tag{4}$$

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. We then use the alias version `rand_pred_al` given in Algorithm 6 (Appendix A.2) to implement `rand_pred` of Algorithm 1. As an example, in the rightmost graph of Fig. 2, 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.

Proposition 9. *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). Let $w_j \in \mathbf{N}_v^-$. Denote by X the uniformly drawn index in $[\![|\mathbf{N}_v^-|]\!]$. The probability law gives: $\mathbb{P}(w_j) = \frac{1}{|\mathbf{N}_v^-|}(\mathbb{P}(w_j|X=j) + \sum_{i\neq j}\mathbb{P}(w_j|X=i))$. 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 10 (Alias Preprocessing). *The preprocessing stage 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) 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 11 (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 1. *The alias implementation is an optimal implementation in terms of the asymptotic worst case complexity for both preprocessing and sampling.*

Proof. It is a consequence of the Proposition 1 with the results from Propositions 10 and 11. □

5 Optimal random bit complexity and unranking

The Alias method is a powerful technique to obtain an efficient generation algorithm. However the main drawback concerning the method is that it relies on floating arithmetic which raises a number of questions related to the rounding approximations, the accuracy of the generation, the bias induced and the quantity of required random bits. We propose in the next section what seems a new approach by adapting the Alias method in the context of arbitrary-size integer arithmetic. We thus rely on the exact uniform distribution (that is central for us) and furthermore we are able to prove the optimality of our method for the number of random bits that are necessary during the sampling. As far as we know, we never encountered this adaptation in the literature.

5.1 A new Alias method with integers

We focus on a set $\{v_0, v_1, \ldots, v_{n-1}\}$ of objets, each v_i has a positive integer weight w_i. We aim at sampling each v_i with probability w_i/W where $W = \sum_{j=0}^{n-1} w_j$.

We first run a preprocessing step building two data structures of cumulated space complexity that is linear in n and that will allow during the random sampling step to get a random object with time complexity $O(1)$. Our approach is directly related to the Euclidean division of W by n, the number of objects. Let us introduce the quotient q and the remainder r satisfying

$$W = q \cdot n + r \quad \text{with} \quad 0 \leq r < n. \tag{5}$$

We then dispatch the total weight W among two tables: a first one of size n corresponding to the Alias table T. Cells of T are indexed between 0 to $n-1$ and each one contains a triplet. And a second table R of size r containing the remaining weight of the distribution. Let us devise how to distribute the objects inside the two tables. We first define two stacks S_0 and S_1 containing respectively the objects whose weight is larger than or equal to q and those whose weight is (strictly) smaller than q. While T is not full, we iteratively consider

- the first elements of each stack (if both exist), x_0 and x_1 with respective weight $y_0 \geq q$ and $y_1 < q$. We then distribute the weight y_0 as $q - y_1$ and $y_0 - (q - y_1)$. We take the first empty cell of the table T, and assign to it the triplet (y_1, x_1, x_0). There are two possibilities for the remaining part of x_0 with remaining weight $y_0 - (q - y_1)$. Either the later is still greater than or equal to q, and we keep it in S_0 with its remaining weight $y_0 - (q - y_1)$, or the remaining weight $y_0 - (q - y_1)$ is smaller than q and we move the object x_0 with its remaining weight $y_0 - (q - y_1)$ in the second stack S_1.
- the second stack S_1 is empty, and x_0 with weight y_0 is the first element from S_0. Then, we fill the next cell from T with the triplet $(q, x_0, \emptyset)$, and we deal with the remaining weight $y_0 - q$ for x_0 as in the previous case.

When the Alias table T is full, we fill the table R with the remaining weight of the distribution, whose value is r, in fact we put in each cell one of the remaining object (with weight 1). Eventually if an object has a remaining weight greater than 1 it will appear several times in R. The key idea proving the correctness of the building of T and R is based on the fact that S_0 is never empty (until T is full). This is due to the fact that whenever S_1 is not empty we remove one of its element at each loop.

We are now ready to state the sampling step algorithm satisfying the claimed constant time complexity.

Proposition 12. *In order to sample an object according to the weight distribution, one samples an integer $i \in [\![W]\!]$ uniformly at random, and then computes its Euclidean division by q: $i = q_i \cdot q + r_i$. If $q_i < n$, then one extracts the triplet (x, v, w) in $T[q_i]$ and returns the object v if $r_i < x$, otherwise, one returns the object w. If $q_i = n$, then one returns the object $R[r_i]$. The whole process is done in $O(1)$ operations.*

Proof details are provided in Appendix A.3.

5.2 Unranking algorithms

The problem of unranking objects emerges as one of the most fundamental challenges in combinatorial generation, as seen in Kreher and Stinson's book [12]. Usually, the approach for constructing structures is based on a recursive decomposition [16]. The schema involves leveraging this decomposition to build a larger object from smaller ones. The term *unranking* comes from the fact that we totally order the objects under consideration, thus each one gets a rank between 0 and the number of objects minus 1. Then we build the object of rank r from scratch. Our three algorithms BRW-linear, Ordered and Binary can easily be adapted to the context of unranking, without modifying their time-complexity. However, some care is needed to keep the time complexity of Alias method especially during the update of the rank after each predecessor determination. In our Alias method with integers, we derive in a way an unranking algorithm based on a multiset of objects where each object appears a number of times equal to its weight and the total order is related to the constructions of both tables T and R.

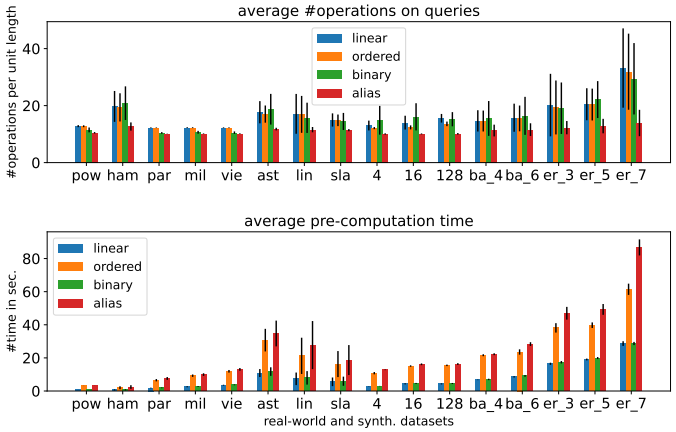
We are now ready to derive an unranking algorithm to reconstruct the shortest path of a given rank, as follows: the preprocessing stage consists of computing the predecessor graph G_s and $\forall v \in V, \sigma_s(t)$ using Equation (2) and for each list $\mathbf{N}_v^-$ it computes augmented versions of the tables T and R as described in Section 5.1 keeping additionally partial sums (see Appendix A.3). The preprocessing still necessitates $O(n+m)$ operations. The query stage does the following: (1) Sample uniformly at random an integer $\rho \in [\![\sigma_s(t)]\!]$. (2) Alias method with integers is used to choose the predecessor x of t and update the rank as $\tilde{\rho}$ which requires $O(1)$ time. (3) Iterate the process with the new rank $\tilde{\rho}$ and new node x until reaching s. The query stage then requires $O(\ell)$ operations. We only use a single call for random bits in the rank sampling ($\rho \in [\![\sigma_s(t)]\!]$) which is clearly optimal. Since we need at least to distinguish between all shortest paths. The process is detailed in Appendix A.3. From the preceding, it follows that:

Theorem 2. *The unranking algorithm is an optimal implementation in terms of the asymptotic worst case complexity for both the preprocessing and the sampling stages and is also optimal in terms of random bits complexity.*

6 Experimental evaluation and application

In our experiments, we compare *BRW* and its various implementations to assess their performance on both real-world and synthetic datasets. 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 and Barabási-Albert random graphs, and 2-d grids.

Fig. 3 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



data	dir.	#nodes	#edges	ref.
power_grid	u	4.94K	6.59K	[13]
hamster_full	u	2.43K	16.6K	[13]
paris	d	9.52K	18.3K	[2]
milan	d	12.9K	25.3K	[2]
vienna	d	16.1K	35.7K	[2]
astro_ph	u	18.8K	198K	[13]
linux_mail	d	26.9K	237K	[13]
slashdot	d	51.1K	130K	[13]
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

Fig. 3: (Up Left): Query complexity on the different graphs. (Down Left): Preprocessing complexity on the different graphs. (Right): Real-world and synthetic datasets. More details about the experiments are presented in Appendix A.4.

pair of values with each edge. Although the theoretical worst-case running times are similar for all variants, 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.

References

1. Achlioptas, D., Clauset, A., Kempe, D., Moore, C.: On the bias of traceroute sampling: or, power-law degree distributions in regular graphs. Journal of the ACM **56**(4), 1–28 (2009)
2. Boeing, G.: Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. Computers, environment and urban systems **65**, 126–139 (2017)
3. Brandes, U.: A faster algorithm for betweenness centrality. Journal of mathematical sociology **25**(2), 163–177 (2001)
4. Ciulla, F., Perra, N., Baronchelli, A., Vespignani, A.: Damage detection via shortest-path network sampling. Physical review E **89**(5), 052816 (2014)
5. Clauset, A., Moore, C.: Traceroute sampling makes random graphs appear to have power law degree distributions. arXiv preprint cond-mat/0312674 (2003)

6. Crespelle, C., Tarissan, F.: Evaluation of a new method for measuring the internet degree distribution: Simulation results. Computer Communications **34**(5), 635–648 (2011)
7. Dall'Asta, L., Alvarez-Hamelin, I., Barrat, A., Vázquez, A., Vespignani, A.: Exploring networks with traceroute-like probes: Theory and simulations. Theoretical Computer Science **355**(1), 6–24 (2006)
8. Flajolet, P., Zimmermann, P., Van Cutsem, B.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science **132**(1-2), 1–35 (1994)
9. Flaxman, A.D., Vera, J.: Bias reduction in traceroute sampling–towards a more accurate map of the internet. In: International Workshop on Algorithms and Models for the Web-Graph. pp. 1–15. Springer (2007)
10. Genitrini, A., Pépin, M., Peschanski, F.: 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)
11. Guillaume, J.L., Latapy, M.: 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. vol. 2, pp. 1084–1094. IEEE (2005)
12. Kreher, D.L., Stinson, D.R.: Combinatorial Algorithms: generation, enumeration, and search. CRC Press (1999)
13. Kunegis, J.: Konect: the Koblenz network collection. In: Proceedings of the 22nd international conference on world wide web. pp. 1343–1350 (2013)
14. Lakhina, A., Byers, J.W., Crovella, M., Xie, P.: Sampling biases in ip topology measurements. In: IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies. vol. 1, pp. 332–341. IEEE (2003)
15. Leguay, J., Latapy, M., Friedman, T., Salamatian, K.: Describing and simulating internet routes. Computer Networks **51**(8), 2067–2085 (2007)
16. Nijenhuis, A., Wilf, H.S.: Combinatorial algorithms. Computer science and applied mathematics, Academic Press, New York, NY (1975)
17. Petermann, T., De Los Rios, P.: Exploration of scale-free networks: Do we measure the real exponents? The European Physical Journal B **38**, 201–204 (2004)
18. Pósfai, M., Fekete, A., Vattay, G.: Shortest-path sampling of dense homogeneous networks. Europhysics Letters **89**(1), 18007 (2010)
19. Preparata, F.P., Shamos, M.I.: Computational geometry: an introduction. Springer Science & Business Media (2012)
20. Rezvanian, A., Meybodi, M.R.: Sampling social networks using shortest paths. Physica A: Statistical Mechanics and its Applications **424**, 254–268 (2015)
21. Sommer, C.: Shortest-path queries in static networks. ACM Computing Surveys (CSUR) **46**(4), 1–31 (2014)
22. Walker, A.J.: New fast method for generating discrete random numbers with arbitrary frequency distributions. Electronics Letters **8**(10), 127–128 (1974)
23. Wang, H., Van Mieghem, P.: Sampling networks by the union of m shortest path trees. Computer Networks **54**(6), 1042–1053 (2010)
24. Zhang, G.Q., Zhou, S., Wang, D., Yan, G., Zhang, G.Q.: Enhancing network transmission capacity by efficiently allocating node capability. Physica A: Statistical Mechanics and its Applications **390**(2), 387–391 (2011)

A Appendix

A.1 Appendix related to Section 2

Fig. 4 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.

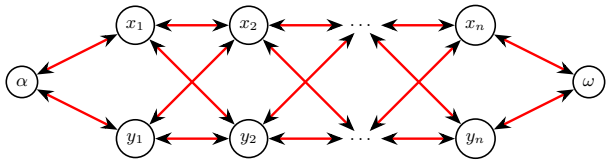


Fig. 4: Graph with $2n+2$ nodes and 2^n shortest paths between α and ω.

In the following we show that the probability of each shortest path between α and ω of the graph G_k see Fig. 5 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.

We denote $a, b_1, \ldots, b_k, c_1, \ldots, c_k, d, e, f$ the random weights on the edges as in Fig. 5. The 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 $U(0,1)$.

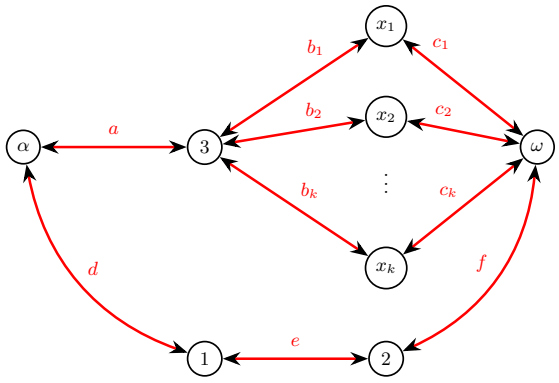


Fig. 5: The family of graphs G_k with the random variables defining the weights of the edges.

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

$$\mathbb{P}(W) = \frac{1}{24} + \int_0^2 (1-F_Y(t))^k \int_0^1 f_Z(t+u)du\,dt. \tag{6}$$

For $k=2$ the Equation 6 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 6 gives $\mathbb{P}(W_0) \overset{k\to+\infty}{\longrightarrow} \frac{1}{24}$.

A.2 Appendix related to Section 3

Algorithm 2 Preprocessing step

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)
    𝒲 = compute_weights(G_s)
    return (G_s, 𝒲)
end function
```

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 5). 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

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
```

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 6). 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 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 $\mathscr{W}$

function RAND_PRED_BIN($\mathbf{N}_v^-, \mathscr{W}, \sigma_s$)
  $r = \text{uniform}([\![\sigma_s[v]]\!])$
  $i =$ `bin_search_index`$(\mathbf{N}_v^-, \mathscr{W}, r)$
  return $w = \mathbf{N}_v^-[i]$
end function

Algorithm 5 Binary implementation of `rand_pred`

function BIN_SEARCH_INDEX($\mathbf{N}_v^-, \mathscr{W}, r$)
  $i = -1$, $j = |\mathbf{N}_v^-| - 1$
  while $j - i > 1$ **do**
    $x = \lfloor \frac{i+j}{2} \rfloor$, $w = \mathbf{N}_v^-[x]$
    if $\mathscr{W}(w \to v) > r$ **then**
      $j = x$
    else
      $i = x$
    end if
  end while
  return j
end function

Here we present the proof of Proposition 8.

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 (3). 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 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 $n_0 + \cdots + n_{\ell-1} \leq n$. □

Algorithm 6 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
```

A.3 Appendix related to Section 5

We first exhibit an example for the Alias method with integers. To provide an interesting example, we modify a little our main example as:

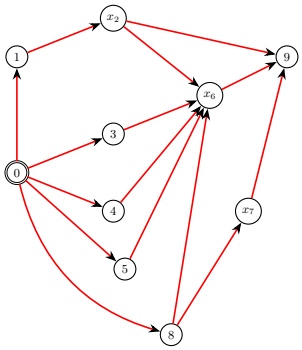


Fig. 6: Example used for our Alias method.

In our new example (Fig. 6) we get:

$$\sigma_0(9) = 7 \qquad \begin{aligned} \sigma_0(x_2) &= 1 \\ \sigma_0(x_6) &= 5 \\ \sigma_0(x_7) &= 1. \end{aligned}$$

Let us exhibit both tables T and R by our Alias algorithm to reach the last node 9 starting either with x_2, x_6 or x_7 with respective weights $1, 5$ and 1. The Euclidean division is $7 = 2 \times 3 + 1$, thus we have

$$T = [(1, x_2, x_6), (1, x_7, x_6), (2, x_6, \emptyset)] \quad \text{and} \quad R = [x_6].$$

A call to Alias with integer 0 gives x_2. A call with integer 2 gives x_7 and all other possibilities $1, 3, 4, 5, 6$ give x_6, respectively from $T[0]$ then $T[1]$, then two times from $T[2]$ and finally from $R[0]$.

Now we are ready to prove Proposition 12 that gives the time-complexity for our Alias method with integers.

Proof (Proposition 12). We use the notations introduced in Equation (5)) and for each stack S_i $(i = 0, 1)$ we define its weight $\|S_i\|$ to be the cumulated weight of the elements it contains. Furthermore if a cell from T is not empty then it contains a weight q.

The algorithm is based on the following loop invariant:
At each iteration i, we remove a weight q from the total weight $\|S_0\| + \|S_1\|$ that is added to the total weight contained in T.

The algorithm terminates after n iterations and finally the remaining weight in $\|S_0\| + \|S_1\|$ is r. Then, R is filled. Finally, the sampling an integer $i \in [\![W]\!]$ and returning the result allows to sample an object according to the distribution of w_j for $j \in [\![n]\!]$.

□

In order to get the unranking algorithm to calculate the shortest path of rank $r \in [\![\sigma_s(t)]\!]$ we must adapt a little our Alias method with integers to keep track of a little more of information (partial sums of appearances of elements). We show how this works on the previous example as it contains most ideas of the proof. The complete proof is left out to avoid a lengthy presentation. We have the following new tables

$$\tilde{T} = [(1, (x_2, 0), (x_6, 0)), (1, (x_7, 0), (x_6, 1)), (2, (x_6, 2), \emptyset)] \quad \text{and} \quad \tilde{R} = [(x_6, 4)].$$

Let us focus on $\tilde{T}[1]$ containing $(1, (x_7, 0), (x_6, 1))$. The third element from the triplet is $(x_6, 1)$ meaning that the object under consideration is x_6 and furthermore, before in the table $\tilde{T}$, i.e. in $\tilde{T}[0]$ we have already a weight equal to 1 for the object x_6 that has been dispatched.

Suppose now, we unrank the shortest path 5 in $\sigma_0(9)$. The Alias method with tables $\tilde{T}$ and $\tilde{R}$ has been preprocessed. The quotient $q = 2$ is the weight of each cell of $\tilde{T}$, we thus are interested in the cell $\lfloor 5/2 \rfloor = 2$ containing $(2, (x_6, 2), \emptyset)$. The object we get is x_6 because its value $5 \mod 2 = 1$ satisfies $1 < 2$. Thus the predecessor of 9 we are interested in is x_6. We must then update the rank value. Inside the cell we desire the value $5 \mod 2 = 1$ but for going on in the process we must recall that x_6 had already been dispatched before in $\tilde{T}$, with weight 2, corresponding to the second coordinate of $(x_6, 2)$. Finally the new rank to continue the process is $1 + 2$, corresponding to the element of rank 3 in $\sigma_0(x_6)$.

A.4 Appendix related to Section 6

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.

The Fig. 3 is built using the following information.

- (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.