Sublinear Random Access Generators for Preferential Attachment Graphs

GUY EVEN, Tel Aviv University
REUT LEVI, The Interdisciplinary Center Herzliya (IDC)
MOTI MEDINA, Faculty of Engineering, Bar-Ilan University, Ramat Gan, Israel
ADI ROSÉN*, CNRS and Université de Paris

We consider the problem of sampling from a distribution on graphs, specifically when the distribution is defined by an evolving graph model, and consider the time, space and randomness complexities of such samplers.

In the standard approach, the whole graph is chosen randomly according to the randomized evolving process, stored in full, and then queries on the sampled graph are answered by simply accessing the stored graph. This may require prohibitive amounts of time, space and random bits, especially when only a small number of queries are actually issued. Instead, we propose a setting where one generates parts of the sampled graph on-the-fly, in response to queries, and therefore requires amounts of time, space, and random bits which are a function of the actual number of queries. Yet, the responses to the queries correspond to a graph sampled from the distribution in question.

Within this framework we focus on two random graph models: the Barabási-Albert Preferential Attachment model (BA-graphs) (Science, 286(5439):509–512) (for the special case of out-degree 1) and the random recursive tree model (Theory of Probability and Mathematical Statistics, (51):1–28). We give on-the-fly generation algorithms for both models. With probability $1 - 1/\text{poly}(n)$, each and every query is answered in $\text{polylog}(n)$ time, and the increase in space and the number of random bits consumed by any single query are both $\text{polylog}(n)$, where n denotes the number of vertices in the graph.

Our work thus proposes a new approach for the access to huge graphs sampled from a given distribution, and our results show that, although the BA random graph model is defined by a sequential process, efficient random access to the graph's nodes is possible. In addition to the conceptual contribution, efficient on-the-fly generation of random graphs can serve as a tool for the efficient simulation of sublinear algorithms over large BA-graphs, and the efficient estimation of their performance on such graphs.

ACM Reference Format:
Guy Even, Reut Levi, Moti Medina, and Adi Rosén. 0. Sublinear Random Access Generators for Preferential Attachment Graphs. *ACM Trans. Algor.* 1, 1, Article 1 (0), 26 pages.

*Research supported in part by ANR project RDAM.

A preliminary version of this work appeared in the Proceedings of ICALP 2017 [11].
This research was supported by the Israel Science Foundation grant No. 1867/20.
This research was supported by the Israel Science Foundation Grant No. 867/19.
Authors' addresses: Guy Even, Tel Aviv University, guy@eng.tau.ac.il; Reut Levi, The Interdisciplinary Center Herzliya (IDC), reut.levi1@idc.ac.il; Moti Medina, Faculty of Engineering, Bar-Ilan University, Ramat Gan, Israel, moti.medina@biu.ac.il; Adi Rosén, CNRS and Université de Paris, adiro@irif.fr.

1549-6325/0/0-ART1 $15.00
https://doi.org/

1 INTRODUCTION

Consider a Markov process in which a sequence $\{S_t\}_t$ of states, $S_t \in \mathcal{S}$, evolves over time $t \geq 1$. Suppose there is a set $\mathcal{P}$ of predicates defined over the state space $\mathcal{S}$. Namely, for every predicate $P \in \mathcal{P}$ and state $S \in \mathcal{S}$, the value of $P(S)$ is well defined. A query is a pair (P, t) and the answer to the query is $P(S_t)$. In the general case, answering a query (P, t) requires letting the Markov process run for t steps until S_t is generated. In this paper we are interested in ways to reduce the dependency, on t, of the computation time, the memory space, and the number of used random bits, required to answer a query (P, t). We propose an approach to achieve that in the context of huge random graphs, samples according to some given distribution.

We focus on the case of generative models for random graphs, and in particular, on the Barabási-Albert Preferential Attachment model [3] with out-degree 1 (which we call BA-graphs), on the equivalent linear evolving copying model of Kumar et al. [18], and on the random recursive tree model [34]. The question we address is whether one can design a randomized *on-the-fly* graph generator that answers adjacency list queries of BA-graphs (or random recursive trees), without having to generate the complete graph. Such a generator outputs answers to adjacency list queries as if it first selected the whole graph at random (according the appropriate distribution) and then answered the queries based on the sampled graph.

We are interested in the following resources of a graph generator: (1) the number of random bits consumed per query, (2) the running time per query, and (3) the increase in memory space per query.

Our main result is a randomized on-the-fly graph generator for BA-graphs over n vertices that answers adjacency list queries. The generated graph is sampled according to the distribution defined for BA-graphs over n vertices, and the complexity upper bounds that we prove hold with probability $1-1/\mathrm{poly}(n)$. That is, with probability $1-1/\mathrm{poly}(n)$ each and every query is answered in $\mathrm{polylog}(n)$ time, and the increase in space, and the number of random bits consumed during that query are $\mathrm{polylog}(n)$. Our result refutes (definitely for $\mathrm{polylog}(n)$ queries) the recent statement of Kolda et al. [17] that: "The majority of graph models add edges one at a time in a way that each random edge influences the formation of future edges, making them inherently serial and therefore unscalable. The classic example is Preferential Attachment, but there are a variety of related models..."

We remark that the entropy of the edges in BA-graphs is $\Theta(\log n)$ per edge in the second half of the graph [33]. Hence it is not possible to consume a sublogarithmic number of random bits per query in the worst case if one wants to sample according to the BA-graph distribution. Similarly, to ensure consistency (i.e., answer the same query twice in the same way) one must use $\Omega(\log n)$ space per query.

From a conceptual point of view, the main ingredient of our result are techniques to "invert" the sequential process where each new vertex randomly selects its "parent" in the graph among the previous vertices. Instead, vertices randomly select their "children" among the "future" vertices, while maintaining the same probability distribution as if each child picked "in the future" its parent. We apply these techniques in the related model of random recursive trees [34] (also used within the evolving copying model [18]), and use them as a building block for our main result for BA-graphs. We next define the various random graph models we refer to.

1.1 Random Graph Models

Let $V_n \triangleq \{v_1, \ldots, v_n\}$. Let $G = (V_n, E)$ denote a directed graph on n nodes.[1] We refer to the endpoints of a directed edge (u, v) as the *head* v and the *tail* u. Let $deg(v_i, G)$ denote the *degree* of

[1]Preferential attachment graphs are usually presented as undirected graphs. For convenience of discussion we orient each edge from the high index vertex to the low index vertex, but the graphs we consider remain undirected graphs.

the vertex v_i in G (both incoming and outgoing edges). Similarly, let $deg_{in}(v_i, G)$ and $deg_{out}(v_i, G)$ denote the in-degree and out-degree, respectively, of the vertex v_i in G.

Preferential attachment [3]. We restrict our attention to the case in which each vertex is connected to the previous vertices by a single edge (i.e., $m = 1$ in the terminology of [3]). [2] We thus denote the random process that generates a graph over V_n according to the preferential attachment model by BA_n. The random process BA_n generates a sequence of n directed edges $E_n \triangleq \{e_1, \ldots, e_n\}$, where the tail of e_i is v_i, for every $i \in \{1, \ldots, n\}$. (We abuse notation and let $BA_n = (V_n, E_n)$ also denote the graph generated by the random process.) We refer to the head of e_i as the *parent* of v_i.

The process BA_n draws the edges sequentially starting with the self-loop $e_1 = (v_1, v_1)$. Suppose we have selected BA_{j-1}, namely, we have drawn the edges $e_1, \ldots, e_{j-1}$, for $j > 1$. The edge e_j is drawn such its head is node v_i with probability $\frac{deg(v_i, BA_{j-1})}{2(j-1)}$.

Note that the out-degree of every vertex in (the directed graph representation of) BA_n is exactly one, with only one self-loop in v_1. Hence BA_n (without the self-loop) is an in-tree rooted at v_1.

Evolving copying model [18]. Let Z_n denote the evolving copying model with out-degree $d = 1$ and copy factor $\alpha = 1/2$. As in the case of BA_n, the process Z_n selects the edges $E'_n = \{e'_1, \ldots, e'_n\}$ one-by-one starting with a self-loop $e'_1 = (v_1, v_1)$. Given the graph $Z_{n-1} = (V_n, E'_n)$, the next edge e'_n emanates from v_n. The head of edge e'_n is chosen as follows. Let $b_n \in \{0, 1\}$ be an unbiased random bit. Let $u(n) \in \{1, \ldots, n-1\}$ be a uniformly distributed random variable (the random variables $b_1, \ldots, b_n$ and $u(1), \ldots, u(n)$ are fully independent.) The head v_i of e'_n is determined as follows:

$$head(e'_n) \triangleq \begin{cases} u(n) & \text{if } b_n = 1 \\ head(e'_{u(n)}) & \text{if } b_n = 0 \end{cases}. \tag{1}$$

Random recursive tree model [34]. If we eliminate from the evolving copying model the bits b_i and define $head(e'_n) \triangleq u(n)$, we get a model where each new node n is connected to one of the previous nodes, chosen uniformly at random. This is the extensively studied (random) recursive tree model [34].

1.2 Results

Our main results are stated in the following theorems (which are informal versions of Theorem 19 and Lemma 16, respectively).

Theorem 1. *(Informal) There exists an algorithm that provides query access to the adjacency-lists of a graph G over n nodes, where n is a parameter and G is drawn according to the random process BA_n. With high probability, the complexities of executing each query are as follows.*

(1) The increase, during that query, of the space used by our algorithm is $O(\log^3 n)$.
(2) The number of random bits used during that query is $O(\log^5 n)$.
(3) The time complexity of that query is $O(\log^6 n)$.

Theorem 2. *(Informal) There exists an algorithm that provides query access to the adjacency-lists of a graph G over n nodes, where n is a parameter and G is drawn according to the random process Z_n. With high probability, the complexities of executing each query are as follows.*

(1) The increase, during that query, of the space used by our algorithm is $O(\log^2 n)$.
(2) The number of random bits used during that query is $O(\log^4 n)$.
(3) The time complexity of that query is $O(\log^5 n)$.

[2] As mentioned-above, while the process generates an undirected graph, for ease of discussion we consider each edge as directed from its higher-numbered adjacent node to its lower-numbered adjacent node.

1.3 Related work

A linear time randomized algorithm for efficiently generating BA-graphs is given in Batagelj and Brandes [4]. See also Kumar et al. [18] and Nobari et al. [27]. A parallel algorithm is given in Alam et al. [1]. See also Yoo and Henderson [35]. An external memory algorithm was presented by Meyer and Peneschuck [24].

Efficient generation of other graph models was also studied. Miller and Hagberg [25] introduced a randomized algorithm that generates a graph with a given sequence of expected degrees (also called the Chung and Lu model) with expected running time of $O(n + m)$, where n is the number of vertices and m is the number of edges of the generated graph. Additional efficient random graph generation algorithms for other graph models (e.g., Kronecker and the Stochastic block model) are provided in Ramani, Eikmeier, and Gleich [30].

Goldreich, Goldwasser and Nussboim initiate the study of the generation of huge random objects [15] while using a "small" amount of randomness. They provide an efficient stateless query access to an object modeled as a function, when the object has a predetermined property, for example graphs which are connected. They guarantee that these objects are indistinguishable from random objects that have the same property. This refers to the setting where the size of the object is exponential in the number of queries to the function modeling the object. In a followup paper by Bogdanov and Wee [6] stateful implementations of huge random objects were considered. They showed how to generate in an "on the fly" fashion a random Boolean function that supports XOR queries over sub-cubes of the function's domain hypercube. We note that our stateful generator provides access to graphs which are random BA-graphs and not just indistinguishable from random BA-graphs.

Mansour, Rubinstein, Vardi and Xie [22] consider local generation of bipartite graphs for local simulation of Balls into Bins online algorithms. They assume that the balls arrive one by one and that each ball picks d bins independently, and is then assigned to one of them. The local simulation of the algorithm locally generates a bipartite graph. Mansour et al. show that with high probability one needs to inspect only a small portion of the the bipartite graph in order to run the simulation and hence a random seed of logarithmic size is sufficient.

Our work has inspired subsequent work in the setting that we propose here, i.e., on-the-fly local generation of graphs according to a given distribution. In fact, subsequent to the initial publication of the present work [11], Biswas, Rubinfeld, and Yodpinyanee [5] have devised local graph generators for, most notably, the Erdös-Rényi model (with next-neighbor, and other, queries).

1.4 Applications

One reason for generating large BA-graphs is to simulate algorithms over them, or to experimentally estimate some of their properties (cf. [9]). Such algorithms often access only small portions of the graphs. In such instances, it is wasteful to generate the whole graph. An interesting example is sublinear approximation algorithms [26, 28, 29, 36] which probe a constant number of neighbors.[3] In addition, local computation algorithms probe a small number of neighbors to provide answers to optimization problems such as maximal independent sets and approximate maximum matchings [2, 12, 13, 19–23, 31, 32]. Support of adjacency list queries is especially useful for simulating (partial) DFS and BFS over graphs.

[3]Strictly speaking, sublinear approximation algorithms apply to constant degree graphs and BA-graphs are not constant degree. However, thanks to the power-law distribution of BA-graphs, one can "omit" high degree vertices and maintain the approximation. See also [31].

1.5 Techniques

The main difficulty in providing the on-the-fly generator is in "inverting" the random choices of the BA process. That is, we need to be able to randomly choose the next "child" of a given node x, although it will only "arrive in the future" and its choice of a parent in the BA-graph will depend on what will have happened until it arrives (i.e., on the node degrees in the BA-graph when that node arrives). One possibility to do so is to maintain, for any future node which does not yet have a parent, how many potential parents it still has, and then go sequentially over the future nodes and randomly decide if its parent will indeed be x. This is too costly because (1) we will need to go sequentially over the nodes, and (2) it may be too costly in computation time to calculate what is the probability that the parent of a node y that does not have yet a parent, will be node x (given the random choices already done in response to previous queries).

To overcome this difficulty we define for any node, even if it has already a parent, its probability to be a *candidate* to be a child of x. We show how these probabilities can be calculated efficiently given the previous choices taken in response to previous queries, and show how, based on these probabilities, we can define an efficient process to chose the next *candidate*. The candidate node may however already have a parent, and thus cannot be a child of x. If this is the case we repeat the process and choose another candidate, until we chose an eligible candidate which then is chosen to be the actual next child of x. We show that with high probability this process terminates quickly and finds an eligible candidate, so that with high probability we have an efficient process to find "into the future" the next child of x. This is done while sampling exactly according to the distribution defined by the BA-graphs process.

In addition to the above technique, which is arguably the crux of our result, we use a number of data structures, based on known constructions, to be able to run the on-the-fly generator with polylogarithmic time and space complexities. In the sequel we give, in addition to the formal definitions of the algorithms, some supplementary intuitive explanations into our techniques.

2 PRELIMINARIES

The *normalized degree distribution* of G is a vector $\Delta(G)$ with n coordinates, one for each vertex in G. The coordinate corresponding to v_i is defined by

$$\Delta(G)_i \triangleq \frac{deg(v_i, G)}{2 \cdot |E|} .$$

Note that $\sum_{i=1}^{n} \Delta(G)_i = 1$.

We also define the in-degree distribution $\Delta_{in}(G)$ by

$$\Delta_{in}(G)_i \triangleq \frac{deg_{in}(v_i, G)}{|E|} .$$

In the sequel, when we say that an event occurs *with high probability* (or *w.h.p*) we mean that it occurs with probability at least $1 - \frac{1}{n^c}$, for some constant $c > 0$. We state the complexities of our algorithm (and our subroutines) with the guarantee of high probability. Therefore, there is some negligible probability that our algorithm will require more resources [4].

For ease of presentation, define the algorithm making use of *arrays* of size n. However, in order to give the desired upper bounds on the space complexity, we implement these arrays by means of balanced search trees, where the keys are in $\{1, \ldots, n\}$. To access item i in the virtual array, key i is searched in the tree and the value in that node is returned; if the key is not found, then nil is

[4]We note that in any case the resources consumed by the algorithm can be bounded by the resources consumed by the sequential standard algorithm since one can easily abort and implement a sequential algorithm in the unlikely event that the on-the fly generator consumes too much resources.

returned. Thus, the space used by the virtual arrays is the number of keys stored, and the time complexity of our algorithms is multiplied by a factor of $O(\log n)$ compared to the time complexity that it would have with a standard random-access implementation of the arrays. When we state upper bounds on time, we take into account these $O(\log n)$ factors. As common, we analyze the space complexity in terms of words of size $O(\log n)$.

3 QUERIES

Consider an undirected graph $G = (V_n, E)$, where $V_n = \{v_1, \ldots, v_n\}$. Slightly abusing notation, we sometimes consider and denote node v_i as the integer number i and so we have a natural order on the nodes. The access to the graph is done by means of a user-query `BA-next-neighbor` : $\{1, \ldots, n\} \rightarrow \{1, \ldots, n+1\}$, which outputs entries of the adjacency list in increasing order (where $n+1$ denotes "no additional neighbor"). For example, consider the node 3 (namely the node that arrived third) whose parent is 1 and its children are 7, 10 and 50. When the user executes the query `BA-next-neighbor`(3) for the first time, 1 is returned. The next time the query `BA-next-neighbor`(3) is executed 7 is returned, and then 10 and 50. After that, `BA-next-neighbor`(3) always returns $n + 1$ since 3 does not have any additional neighbors. More formally, we number the queries according to the order they are issued, and call this number the *time* of the query. Let $q(t)$ be the node on which the query at time t was issued, i.e, at time t the query `BA-next-neighbor`$(q(t))$ is issued by the user. For each node $j \in V$ and any time t, let $last_t(j)$ be the largest numbered node which was previously returned as the value of `BA-next-neighbor`(j), or 0 if no such query was issued before time t. That is,

$$last_t(j) = \max\{0, \max_{t'<t}\{\texttt{BA-next-neighbor}(q(t'))|q(t') = j\}\} .$$

At time t the query `BA-next-neighbor`(j) returns $\arg\min_{i>last_t(j)}\{(i, j) \in E\}$, or $n + 1$ if no such i exists. When the implementation of the query has access to a data structure holding the whole of E, then the implementation of `BA-next-neighbor` is straightforward just by accessing this data structure. Figure 1 illustrates a "traditional" randomized graph generation algorithm that generates the whole graph, stores it, and then can answers queries by accessing the data structure that encodes the whole generated graph.

4 ON-THE-FLY GRAPH GENERATORS

An on-the-fly graph generator is an algorithm that gives access to a graph by means of the `BA-next-neighbor` query defined above, but itself does not have access to a data structure that encodes the whole graph. Instead, in response to the queries issued by the user, the generator modifies its internal data structure (a.k.a state), which is initially some empty (constant) state. The generator must ensure however that its answers are consistent with some graph G. An on-the-fly graph generator for a given distribution on a family of graphs (such as the family of Preferential Attachment graphs on n nodes) must in addition ensure that it samples the graphs according to the required distribution. That is, its answers to a sequence of queries must be *distributed identically* to those returned when a graph was first sampled (according to the desired distribution), stored, and then accessed (See Definition 20 and Theorem 21). Figure 2 illustrates an on-the-fly graph generation algorithm as the one we build in the present paper.

We now relate the various models defined in Section 1.1. The following relation is crucial for our implementation of the on-the-fly generator for BA-graphs.

Claim 3 ([1]). *The random graphs BA_n and Z_n are identically distributed.*

Proof. The proof is by induction on n. The basis ($n = 1$) is trivial. To prove the induction step, assume that BA_{n-1} and Z_{n-1} are identically distributed. We need to prove that the next edges e_n

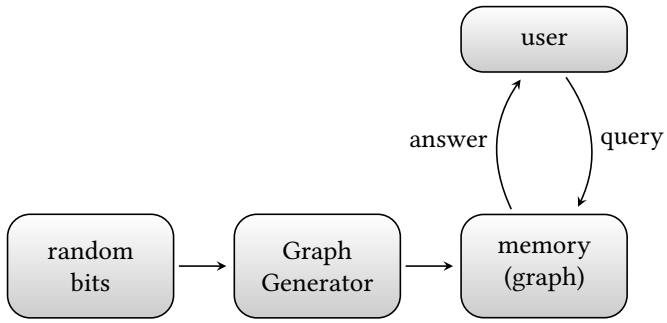

Fig. 1. A "traditional" sequential random graph generator

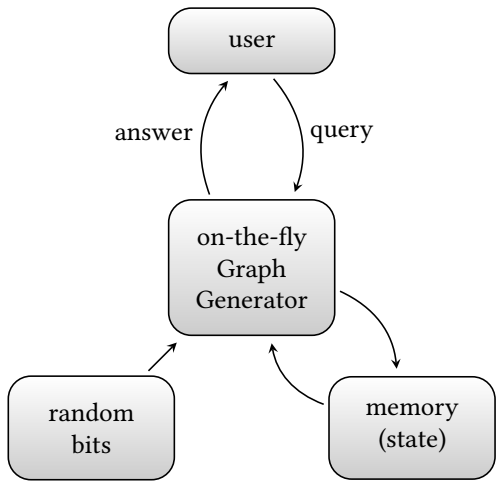

Fig. 2. An on-the-fly random graph generator

and e'_n in the two processes are also identically distributed, given a graph G as the realization of BA_{n-1} and Z_{n-1}, respectively.

The head of e_n is chosen according to the degree distribution $\Delta(BA_{n-1}) = \Delta(G)$. Since the out-degree of every vertex is one,

$$\frac{deg(v_i, BA_{n-1})}{2(n-1)} = \frac{1}{2} \cdot \left(\frac{1}{n-1} + \frac{deg_{in}(v_i, BA_{n-1})}{n-1} \right).$$

Thus, an equivalent way of choosing the head of e_n is as follows: (1) with probability $1/2$, choose a random vertex uniformly (this corresponds to the $\frac{1}{2} \cdot \frac{1}{n-1}$ term), and (2) with probability $1/2$ toss a $\Delta_{in}(BA_{n-1})$-dice (this corresponds to the $\frac{1}{2} \cdot \frac{deg_{in}(v_i, BA_{n-1})}{n-1}$ term).

Hence, case (1) above corresponds to the case when $b_n = 1$, in the process of Z_n. To complete the proof, we observe that, conditioned on the event that $b_n = 0$, the choice of the head of e'_n in Z_n can be defined as choosing according to the in-degree distribution of the nodes in $Z_{n-1} = G$: indeed, choosing according to the in-degree distribution $\Delta_{in}(G)$ is identical to choosing a uniformly distributed random edge in G and then taking its head. But, since the out-degrees of all the vertices in V_{n-1} are all the same (and equal one), this is equivalent to choosing a uniformly distributed random node in V_{n-1}. □

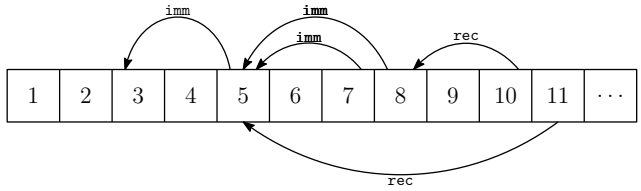

Fig. 3. **The pointers tree.** The parent of node 5 is node 3 and the children of node 5 are nodes 7, 8, and 10 (since 10 has a recursive pointer to node 8). Node 11 is a child of node 3 since it has a recursive pointer to node 5 (in particular, it is not a child of node 5).

We use the following claim in the sequel.

Claim 4 (cf. [14], Thm. 1 and Thm. 6.32 [10]). *Let T be a rooted directed tree on n nodes denoted $1, \ldots, n$, and where node 1 is the root of the tree. If the head of the edge emanating from node $j > 1$ is uniformly distributed among the nodes in $[1, j-1]$, then, with high probability, the following two properties hold:*

(1) The maximum in-degree of a node in the tree is $O(\log n)$.
(2) The height of the tree is $O(\log n)$.

Note that the claim still holds if we add to the tree a self loop on node 1.

5 THE POINTERS TREE

We now consider a graph inspired by the random recursive tree model [34] and the evolving copying model [18]. Each vertex i has a variable $u(i)$ that is uniformly distributed over $[1, i-1]$, and can be viewed as a directed edge (or pointer) from i to $u(i)$. We denote this random rooted directed in-tree (namely, a tree such that all its edges point towards the root) by UT. Let $u^{-1}(j)$ denote the set $\{i : u(i) = j\}$. We refer to the set $u^{-1}(i)$ as the *u-children* of i and to $u(i)$ as the *u-parent* of i.

In conjunction with each pointer, we keep a flag indicating whether this pointer is to be used as an `imm` (immediate) pointer, that is, whether it points to the *head*, or as a `rec` (recursive) pointer (as defined in Equation 1). We thus use the directed pointer tree to represent a graph in the evolving copying model (which is equivalent, when the flag of each pointer is equally distributed between `rec` and `imm`, to the BA model). See Figure 3 for an illustrative example.

In this section we consider the subtask of giving access to a random UT, together with the flags of each pointer. Ignoring the flags, this section thus gives an on-the-fly random access generator for the extensively studied model of random recursive trees (cf. [34]).

We define the following queries.

- $(i, flag) \leftarrow$ `parent`(j): i is the parent of j in the tree, and $flag$ is the associated flag.
- $i \leftarrow$ `next-child-flag`$(j, k, flag)$, where $k \geq j$: i is the least numbered node $i > k$ such that the parent of i is j and the flag of that pointer is of type $flag$. If no such node exists then i is $n + 1$.

Given a query `next-child-flag`$(j, k, \cdot)$, we assume that k is bounded above by the largest value returned by `next-child-flag`$(j, \cdot, \cdot)$ thus-far (and j if it is the first time `next-child-flag`$(j, \cdot, \cdot)$ is executed). Clearly, even under this assumption the neighbors of every node can be revealed one by one by repeated calls to `next-child-flag` (by setting k to be the returned value of the former execution of `next-child-flag`). In this case the output of the queries is the entries of the adjacency list, in increasing order (where the entries with the wrong flag are filtered).

The "ideal" way to implement `next-child-flag` is to go over all n nodes, and for each node j (1) uniformly at random choose its parent in $[1, j-1]$, (2) uniformly at random chose the associated flag in $\{\texttt{imm}, \texttt{rec}\}$. Then store the pointers and flags, and answer the queries by accessing this data structure.

In this section we give an *on-the-fly* generator that answers the above queries. As explained in more detail in the sequel, randomly selecting the parent of a node, to answer a `parent` query, is a rather easy task (even if the generator already has a state). The challenge is in randomly selecting the children of a node which corresponds to `next-child` queries. In this case we need to randomly select (according to the appropriate distribution) the first child of j between {j+1, …, n} and then the second child and so on. In what follows, we start with a naïve, non-efficient implementation that illustrates the task to be done. Then we give our efficient implementation.

5.1 Notations

We say that j is *exposed* if $u(j) \neq \textsf{nil}$ (initially all pointers $u(j)$ are set to nil). We denote the set of all exposed vertices by F. As a result of answering and processing `next-child-flag` and `parent` queries, the on-the-fly generator commits to various decisions (e.g., prefixes of adjacency lists). These commitments include edges but also non-edges (i.e., vertices that can no longer serve as $u(j)$ for a certain j). Therefore each query may change the state of the generator. Note that the answers of the generator to queries depend on its state, thus, the queries `parent` and `next-child-flag` are also a function of this state (which is not given as a parameter).

5.1.1 The front of a node. For every node $i \in \{1, \ldots, n-1\}$, the generator saves the *front* of i, which is roughly speaking, a value $k > i$ for which it holds that (1) $u(k) = i$; and (2) the generator decided for every node $j \in [i+1, k-1]$ whether $u(j) = i$ or not. Formally, at any given time, $\mathit{front}(i)$ is a pointer to a node in $[i+1, n+1]$ which has the following properties.

(1) Initially $\mathit{front}(i) = \textsf{nil}$. The first time $\mathit{front}(i) \neq \textsf{nil}$ is after the first `next-child-flag`$(i, \cdot, \cdot)$ query is issued.
(2) If $\mathit{front}(i) = \textsf{nil}$, then for any node in $j \in [i+1, n]$ for which $u(j) = \textsf{nil}$ it is possible that $u(j)$ will be set to i in the future.
(3) If $\mathit{front}(i) = k$ then
   (a) for any node in $j \in [i+1, k-1]$ for which $u(j) = \textsf{nil}$ it is *not* possible that $u(j)$ will be set to i in the future
   (b) for any node in $j \in [k+1, n]$ for which $u(j) = \textsf{nil}$ it is possible that $u(j)$ will be set to i in the future

5.1.2 The set of potential parents. We denote the set of vertices that can become u-parents of j at any given time t, by $\Phi(j)$ and their number by $\varphi(j)$ (for brevity, we omit t from the notation). The formal definition is as follows.

Definition 5. *At a given time t, and for any node j, let $\Phi(j)$ and $\varphi(j)$ be defined as follows:*

$$\Phi(j) \triangleq \{i \mid i < j \text{ and } (\mathit{front}(i) < j \text{ or } \mathit{front}(i) = \textsf{nil})\}, \text{ and } \varphi(j) = |\Phi(j)| \,.$$

We note that from technical reasons that will become clear below, according to the definition, $\Phi(j)$ is not necessarily empty even if $u(j)$ is already determined. Moreover, counter intuitively, it might be the case that $u(j) \notin \Phi(j)$.

5.2 A naïve implementation of `next-child`

We give a naïve implementation of a `next-child` query, with time complexity $O(n)$, with the purpose of illustrating the main properties of this query and in order to contrast it with the more

naïve-next-child

```
1:  procedure naïve-next-child(j, k)
2:      x ← k + 1.
3:      while x ≤ n do
4:          if u(x) = j then return (x)
5:          else
6:              if u(x) = nil then
7:                  Flip a random bit c(x) such that Pr[c(x) = 1] = 1/φ(x).
8:                  if c(x) = 1 then
9:                      return (x)
10:                 end if
11:             end if
12:         end if
13:         x ← x + 1
14:     end while
15:     return (n + 1)
16: end procedure
```

Fig. 4. pseudo code of naïve-next-child

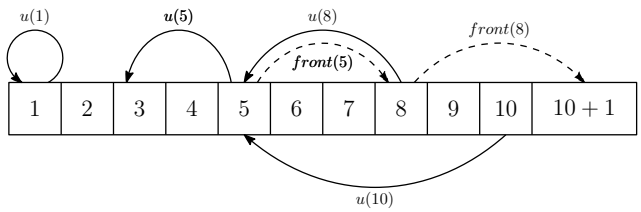

Fig. 5. **The state of the generator**. The state of the generator after performing a next-child(5) and a parent(10) queries. After the first query, which returns 8, the parent of node 5 is determined (in order to keep Invariant 6) and set to be node 3 and *front*(5) and the first child of node 5 are set to be 8. Consequently, in order to keep Invariant 7 the next child of node 8 is also discovered and is set to 10 + 1 (it has no children). After the second query, the parent of node 10 is set to be node 5. However, this does not change *front*(5) and so, potentially, node 9 could also pick node 5 as its parent (or any other node in $\{1, \ldots, 7\}$).

efficient implementation later. We do so in a simpler manner without looking into the "flag". The naïve implementation of next-child is listed in Figure 4. This implementation, and that of parent, share an array of pointers u, both updating it. A query next-child(i, k) is processed by scanning the vertices one-by-one starting from $k + 1$. If $u(x) = i$, then x is the next child. If $u(x)$ is nil, then a coin $c(x)$ is flipped and $u(x) = i$ is set when $c(x)$ comes out 1; the probability that $c(x)$ is 1 is $1/\varphi(x)$. If $c(x) = 0$, we proceed to the next vertex. The loop ends when some $c(x)$ is 1 or all vertices have been exhausted. In the latter case the query returns $n + 1$.

The correctness of naïve-next-child, i.e., the fact that the graph is generated according to the required probability distribution, is based on the observation that given that $u(x)$ has not been determined yet, all the vertices in $\Phi(x)$ are equally likely to serve as $u(x)$. Note that the description above does not explain how $\varphi(x)$ is computed, as explained next, this is one of the main challenges in designing our on-the-fly generator.

5.3 The challenge in obtaining an efficient implementation of next-child

We first shortly discuss the challenges on the way to an efficient implementation of next-child. Consider the simple special case where the only two queries issued are, for some j, a single parent(j) query followed by a single next-child(j) query (to simplify this discussion we assume that the the value of k is globally known). Consider the situation after the query parent(j). At this point, every node $x \in [j+1, n]$ may be a u-child of j (namely, $u(x)$ may be set to j). In particular, since $front(i)$ is initially nil for every i, it holds that $\varphi(x) = x - 1$ and $\mathbf{Pr}[u(x) = j] = 1/(x-1)$. Let P_x denote the probability that node x is the first child of j. Then $P_x = \frac{1}{x-1} \cdot \prod_{\ell=j+1}^{x-1}(1 - \frac{1}{\ell-1}) = \frac{j-1}{(x-1)(x-2)}$ and for P_{n+1} (i.e., j has no child) $P_{n+1} = \frac{j-1}{n-1}$. As explained in the sequel, each of the probabilities $P'_k = \sum_{x=j+1}^{k} P_x$ can be calculated in $O(1)$ time, therefore, this random choice can be done in $O(\log n)$ time by choosing uniformly at random a number in $[0, 1]$ and performing a binary search on $[j+1, n+1]$ to find which index it represents (see a more detailed and accurate statement of this procedure below). However, in general, at the time of a certain next-child query, limitations may exist, due to previous queries, on the possible consistent values of certain pointers $u(x)$. There are two types of limitations: (i) $u(x)$ might have been already determined, or (ii) $u(x)$ is still nil but the option of $u(x) = i$ has been excluded since $front(i) > x$. These limitations change the probabilities P_x and P'_x, rendering them more complicated and time-consuming to compute, thus rendering the above-defined process not efficient (i.e., not doable in $O(\log n)$ time). In the rest of this section we define and analyze a modified procedure that uses polylog(n) random bits, takes polylog(n) time, and increases the space (that is used to store the state of the generator) by polylog(n). This procedure will be at the heart of the efficient implementation of next-child.

5.4 The invariants regarding the state of the generator

In the implementation of the on-the-fly generator of the pointers tree we will maintain two invariants that are described below. We will later discuss the cost (in running time and space) of maintaining these invariants.

The purpose of these invariants is to obtain an efficient computation of the probabilities P_x and P'_x discussed in Subsection 5.3. Roughly speaking, if for every node $x \in \{1, \ldots, n-1\}$ it was the case that $\varphi(x+1) - \varphi(x) = 1$, then these probabilities were easy to calculate. However, since the commitments of the generator impose changes on φ, this property can not hold for all the nodes. The invariants ensure that the set of nodes for which this property does not hold are easy to identify and for which the u-parent is already determined.

Invariant 6. *For every node j, the first* next-child-flag($j, \cdot, \cdot$) *query is always preceded by a* parent(j) *query.*

We will use this invariant to infer that $front(j) \neq$ nil implies that $u(j) \neq$ nil. (recall that if a next-child-flag($j, \cdot, \cdot$) query was not issued thus far then $front(j) =$ nil). One can easily maintain this invariant by introducing a parent(j) query as the first step of the implementation of the next-child-flag($j, \cdot, \cdot$) query (for technical reasons we do that in a lower-level procedure next-child.)

Invariant 7. *For every vertex j, $front(j) \neq$ nil implies that $front(front(j)) \neq$ nil.*

The second invariant is maintained by issuing an "internal" next-child($front(j), front(j)$) query whenever $front(j)$ is updated. This is done recursively, the base of the recursion being node $n+1$. When analyzing the complexities of our algorithm we will take into account these recursive calls. Let $front^{-1}(j)$ denote the vertex i such that $front(i) = j$, if such a vertex i exists; (note that there can be at most one such node i, except for the case of $j = n+1$); otherwise $front^{-1}(j) =$ nil. We get that if $front^{-1}(j) \neq$ nil, then $u(j) \neq$ nil. See Figure 5 for an illustrative example.

We note that if at a given time we consider a node j such that $u(j) = \text{nil}$ (i.e., its parent in the pointers tree is not yet determined), then the set $\Phi(j)$ is the set of all the nodes that can still be the parent of node j in the pointers tree. As mentioned above, the set Φ is however defined also for nodes for which their parent is already determined.

We are now ready to define the set K which is, as we prove in the sequel, the set of nodes, i, for which $\varphi(i+1) = \varphi(i)$. Moreover, we prove that if $x \notin K$ then $\varphi(i+1) - \varphi(i) = 1$. It is also the case that for $x \in K$ it holds that $u(x) \neq \text{nil}$, as desired

Definition 8. *Let K denote the following set:*

$$K \triangleq \{i : \mathit{front}(i) \neq \text{nil} \text{ and } \mathit{front}^{-1}(i) = \text{nil}\} .$$

We shall prove the following lemma.

Lemma 9. *For any $x \in \{1, \ldots, n-1\}$:*

$$\varphi(x+1) - \varphi(x) = \begin{cases} 0 & \text{if } x \in K, \\ 1 & \text{if } x \notin K \end{cases} . \tag{2}$$

Lemma 9 follows directly from the following more general claim.

Claim 10. *For every $x \in \{1, \ldots, n-1\}$:*
(1) $\Phi(x) \subseteq \Phi(x+1) \subseteq \Phi(x) \cup \{x, \mathit{front}^{-1}(x)\}$.
(2) $\Phi(x+1) = \Phi(x)$ iff $x \in K$.
(3) $\varphi(x+1) - \varphi(x) \leq 1$.

Proof. We first observe that Item 1 follows directly from the definition of Φ and the properties of *front*. The see this, we first note that the fact that $\Phi(x) \subseteq \Phi(x+1)$ follows from that fact that for every i such that $\mathit{front}(i) < x$ it clearly holds that $\mathit{front}(i) < x+1$. On the other hand, there might be only a single node, i, such that $\mathit{front}(i) < x+1$ but $\mathit{front}(i) \geq x$. This is possible only when $\mathit{front}(i) = x$, which might be the case only for a single node, which is the u-parent of x. Finally, $x \in \Phi(x+1)$ if and only if $\mathit{front}(x) = \text{nil}$.

To prove Item 2, observe that by Item 1, $\Phi(x) = \Phi(x+1)$, iff $x \notin \Phi(x+1)$ and $\mathit{front}^{-1}(x) \notin \Phi(x+1)$. As stated above, $x \in \Phi(x+1)$ iff $\mathit{front}(x) = \text{nil}$. Similarly, $\mathit{front}^{-1}(x) \notin \Phi(x+1)$ iff $\mathit{front}^{-1}(x) = \text{nil}$.

Finally, to prove Item 3 we need to show that it is not possible for both x and $\mathit{front}^{-1}(x)$ to belong to $\Phi(x+1)$. Indeed, if $\mathit{front}^{-1}(x) \in \Phi(x+1)$, then there exists a vertex i such that $\mathit{front}(i) = x$. Invariant 7 implies that $\mathit{front}(x) = \mathit{front}(\mathit{front}(i)) \neq \text{nil}$. However, $x \in \Phi(x+1)$ implies $\mathit{front}(x) = \text{nil}$, a contradiction. □

5.5 Description of the efficient implementation

We are now ready to describe the implementation of `next-child-flag`$(j, k, flag)$ and `next-child`(j). The state of the generator is stored using the following data structures, of which the implementation of `next-child` (and of `parent`) makes use of.

- An array of length n, $u(j)$.
- An array of length n, $flag(j)$.
- An array of length n, $\mathit{front}(j)$ (We also maintain an array $\mathit{front}^{-1}(i)$ with the natural definition).
- An array of n balanced search trees, called `child`(j), each holding the set of nodes $i > j$ such that $u(i) = j$. The operations we use on the search trees are insert and successor, where `insert`(T, j) inserts the element j to the tree T and `successor`(T, j) returns the smallest elements in T which is larger than j. For technical reasons all trees `child`(j) are initiated with $n+1 \in$ `child`(j).

- A number of additional data structures that are implicit in the listing, described and analyzed in the sequel.

As seen in Figure 6, `next-child-flag`$(j, k, flag)$ is merely a loop of `next-child-from`(j, k), and `next-child-from`(j, k) is essentially a call to `next-child`(j). The "real work" is done in the implementation of `next-child`(j) that we describe now. Note that if j does not have children larger than k, then `next-child-from`(j, k) returns $n + 1$.

If $front(j) > k$ when `next-child-from`(j, k) is called, then the next child is already fixed and it is just extracted from the data structures.

Otherwise, an interval $I = [a, b]$ is defined, and it will contain the answer of `next-child`(j). Let $a = front(j) + 1$ if $front(j) \neq \text{nil}$; and $a = j + 1$, if $front(j) = \text{nil}$. Let $b = \min\{\{\ell > front(j) : u(\ell) = j\} \cup \{n + 1\}\}$ if $front(j) \neq \text{nil}$; and $b = \min\{\{\ell > j : u(\ell) = j\} \cup \{n + 1\}\}$, if $front(j) = \text{nil}$ Observe that no vertex $x \in F \cap [a, b)$ can satisfy $u(x) = j$. Hence, the answer is in $I \setminus (F \setminus \{b\})$.

The next child can be sampled according to the desired distribution in a straightforward way by going sequentially over the vertices in $I \setminus (F \setminus \{b\})$, and tossing for each vertex x a coin that has probability $1/\varphi(x)$ to be 1, until indeed one of those coins comes out 1, or all vertices are exhausted (in which case node b is taken as the next child). We denote by $D(x)$, $x \in I \setminus F$, the probability that x is chosen when the the above procedure is applied. This procedure, however, takes linear time.

In order to start building our efficient implementation for `next-child` we note that by the definition of K, $K \subseteq F$, and we consider a process where we toss $|[a, b) \setminus K|$ coins sequentially for the vertices in $[a, b) \setminus K$. The probability that the coin for $x \in [a, b) \setminus K$ is 1 is still $1/\varphi(x)$. We stop as soon as 1 is encountered or on b if all coins are 0. The vertex on which we stop, denote it x, is a *candidate next u-child*. If $x \in F \setminus K \setminus \{b\}$, then x cannot be a child of j (because it already has a u-parent) so we proceed by repeating the same process recursively, but with the interval $[x + 1, b]$ instead of the interval $[a, b]$. We denote by $D'(x)$, $x \in I \setminus F$, the probability that x is chosen when this procedure is applied.

5.5.1 Efficiently selecting a u-child candidate. We now build our efficient procedure that selects the candidate, without sequentially going over the nodes. To this end, observe that the sequence of probabilities of the coins tossed in the last-described process behaves "nicely". Namely, the probabilities $1/\varphi(x)$, for $x \in [a, b) \setminus K$, form the harmonic sequence starting from $1/\varphi(a)$ and ending in $1/(\varphi(a) + |[a, b) \setminus K| - 1)$. Indeed, Eq. (2) implies that if vertex i is the smallest vertex in $I \setminus K$, then $\varphi(i) = \varphi(a)$ and an increment between $\varphi(x)$ and $\varphi(x + 1)$ occurs if and only if $x \notin K$. Let $s = |[a, b) \setminus K|$ and let P_h, $0 \leq h \leq s$ be the probability that the node of rank h in $([a, b) \setminus K) \cup \{b\}$ is chosen as candidate in the sequential procedure defined above. Since $1/\varphi(x)$ forms the harmonic sequence for $x \in [a, b) \setminus K$, we can, given $\varphi(a)$, calculate in $O(1)$ time, for any $0 \leq i \leq s + 1$, the probability $P'_i = \sum_{q<i} P_q$ (i.e., the probability that a node of some rank q, $q < i$, is chosen). Indeed, for $i = 0$, $P_i = \frac{1}{\varphi(a)}$; for $0 < i < s$, $P_i = \frac{1}{\varphi(a)+i} \cdot \prod_{\ell=0}^{i-1} \left(1 - \frac{1}{\varphi(a)+\ell}\right) = \frac{\varphi(a)-1}{(\varphi(a)+i-1)(\varphi(a)+i)}$; and for $i = s$, $P_s = \prod_{\ell=0}^{s-1} \left(1 - \frac{1}{\varphi(a)+\ell}\right) = \frac{\varphi(a)-1}{\varphi(a)+s-1}$. Hence, for $0 \leq i \leq s$, $P'_i = 1 - \frac{\varphi(a)-1}{\varphi(a)+(i-1)}$, and for $i = s + 1$, $P'_{s+1} = 1$. This allows us to simulate one iteration (i.e., choosing the next *candidate next u-child*) by choosing uniformly at random a single number in $[0, 1]$, and then performing a binary search over 0 to s to decide what rank h this number "represents". After the rank $h \in [0, s]$ is selected, h is then mapped to the vertex of rank h in $([a, b) \setminus K) \cup \{b\}$, denote it x, and this is the *candidate next u-child*. As before, if $x \in F \setminus K \setminus \{b\}$, then x cannot be a child of j so we ignore it and proceed in the same way, this time with the interval $[x + 1, b]$. We denote by $\hat{D}(x)$, $x \in I \setminus F$ the probability that x is chosen when this third procedure is applied. See Figure 7 for a formal definition of this procedure and that of `next-child`.

Observe that this procedure takes $O(\log s)$ time (see Section 5.7 for a formal statement of the time and randomness complexities). We note that we cannot perform this selection procedure in the same time complexity for the set $[a, b) \setminus F$, because we do not have a way to calculate each and every probability P'_i, $i \in [a, b) \setminus F$, in $O(1)$ time, even if $\varphi(a)$ is given.

To conclude the description of the implementation of `next-child`, we give the following lemma which states that the probability distribution on the next child is the same for all three processes described above.

Lemma 11. *For all $x \in I \setminus F$, $\hat{D}(x) = D(x)$.*

Proof. To prove the claim we prove that $\hat{D}(x) = D'(x)$ and that $D'(x) = D(x)$.

To prove the latter, denote by $x_1 < x_2 < \ldots < x_k$ the nodes in the set $I \setminus F$, where $k = |I \setminus F|$, and let $p(x_j) = \frac{1}{\varphi(x_j)}$. For any $1 \leq j \leq k-1$ $D(x_j) = p(x_j) \cdot \Pi_{i=1}^{j-1}(1 - p(x_i))$, and for x_k (which is the node denoted b in the discussion above), $D(x_k) = 1 - \Pi_{i=1}^{k-1}(1 - p(x_i))$.

When we consider the sequential process where one tosses a coin sequentially for all nodes in $I \setminus K$ (and not only for the nodes in $I \setminus F$) we extend the definition of $D'(\cdot)$ to be defined also for nodes in $I \setminus K$. For a node $z \in (I \setminus K) \cap F$, $D'(z)$ is the probability that x is chosen as a *candidate next u-child*. Thus, if we denote by $y_1 < y_2 < \ldots < y_\ell$, $\ell = |I \setminus K|$, the nodes in $I \setminus K$ we have that $D'(y_j) = p(y_j) \cdot \Pi_{1 \leq i < j; y_j \in I \setminus F}(1 - p(y_i))$, and for y_ℓ (which is the node denoted b in the discussion above), $D(y_\ell) = 1 - \Pi_{1 \leq i < \ell; y_j \in I \setminus F}(1 - p(y_i))$. Thus, for any $x \in I \setminus F$, $D(x) = D'(x)$.

We now extend $\hat{D}(\cdot)$ to be defined for all nodes in $I \setminus K$. The assertion $\hat{D}(x) = D'(x)$, for any $x \in I \setminus K$, follows from the fact a number $M \in [0, 1]$ is selected uniformly at random and then the interval in which it lies is found. That is, i is selected if and only if $P'_i \leq M < P'_{i+1}$ which, by the definitions of P_i and P'_i, occurs with probability $P_i = D'(x_i)$. □

5.6 Implementation of parent

The implementation of `parent` is straightforward (see Figure 6). However, note that updating the various data structures, while implicit in the listing, is accounted for in the time analysis.

5.7 Analysis of the pointer tree generator

We first give the following claim that we later use a number of times.

Lemma 12. *With high probability, for each and every call to* `next-child`*, the size of the recursion tree of that call, for calls to* `next-child`*, is $O(\log n)$.*

Proof. Consider the recursive invocation tree that results from a call to `next-child`. Observe that (1) by the code of `next-child` this tree is in fact a path; and (2) this path corresponds to a path in the pointers tree, where each edge of this tree-path is "discovered" by the corresponding call to `next-child`. That is, the maximum size of a recursion tree of a call of `next-child` is bounded from above by the height of the pointers tree. By Claim 4, with high probability, this is $O(\log n)$. □

5.8 Efficient rolling of dice

In Algorithm 7 we implement a rolling of a dice whose time complexity is, with high probability, $O(\log n)$, as explained next. The commulative probabilities P'_y of each side of the dice are a function of ξ and y and are computable in $O(1)$ time. To determine the outcome of a roll of the dice, we pick $r = c \log n$ random unbiased bits, which are interpreted as a binary representation of a subinterval $[\ell_1, \ell_2]$ of $[0, 1]$ of length 2^{-r}. We say that the subinterval is good if it is contained in an interval, the endpoints of which are consecutive cumulative probabilities. Namely, $[\ell_1, \ell_2]$ is good if there exists

`next-child-flag`
Returns the least $i > k$, i is a u-child of j, i has flag "flag".
Assumes that $k \leq front(j)$

```
1: procedure next-child-flag(j, k, flag)
2:     x ← k
3:     repeat
4:         x ← next-child-from(j, x)
5:     until flag(x) = flag or x = n + 1
6:     return x
7: end procedure
```

`next-child-from`
Returns the least $i \geq k$, i is a u-child of j.
Assumes that $k \leq front(j)$.

```
1: procedure next-child-from(j, k)
2:     If (k ≥ n) return (n + 1)
3:     q ← successor(child(j), k)
4:     if q ≤ front(j) then
5:         return q
6:     else
7:         return next-child(j)
8:     end if
9: end procedure
```

`parent`
Returns the u-parent of j.

```
1: procedure parent(j)
2:     if u(j) = nil then
3:         u(j) ←_R [1, j − 1]
4:         flag(j) ←_R {imm, rec}
5:         insert(child(u(j)), j)
6:     end if
7:     return (u(j), flag(j))
8: end procedure
```

Fig. 6. Pseudo code of the pointers tree generator (part 1)

an i such that $[\ell_1, \ell_2) \subseteq [P'_i, P'_{i+1}]$. Note that in this case, we choose i as the outcome of the roll of the dice. The number of bad subintervals is bounded by t, which is at most n. Hence, the probability that the subinterval is bad is at most $n \cdot 2^{-r}$. Since $r = c \log n$, the probability of determining the side of the dice after r random bits is at least $1 - n^{-c+1}$. Hence, the time complexity in this case is dominated by the time complexity of the random search which is $O(\log n)$. On the other hand, if the interval we picked is bad (which happens with negligible probability), then we refine the size of the intervals so that all intervals are good (Line 10). In this case the time complexity is $O(\max\{t, \log n\})$.

We note that by a standard technique we could alternatively implement a Las-Vegas algorithm that rolls the dice. In this case the algorithm continues refining the intervals (by using additional

`next-child`
Returns the least $i > front(j)$ which is a u-child of j.

```
1:  procedure next-child(j)
2:      (p, t) ← parent(j)
3:      If (front(j) ≥ n) return (n + 1)
4:      a ← { front(j) + 1   if front(j) ≠ nil
            { j + 1          if front(j) = nil
5:      b ← { successor(child(j), front(j))   if front(j) ≠ nil
            { successor(child(j), j)          if front(j) = nil
6:      repeat
7:          s ← |[a, b) \ K|
8:          h ← toss(φ(a), s + 1)
9:          if h = s then
10:             return b
11:         else
12:             x ← the vertex of rank h in [a, b) \ K
13:             if u(x) = nil then
14:                 u(x) = j
15:                 flag(x) ←_R {imm, rec}
16:                 insert(child(j), x)
17:                 front(j) ← x
18:                 front^{-1}(x) ← j
19:                 if (front(x) = nil) next-child(x)
20:                 return (x)
21:             else /* i.e., if u(x) ≠ nil */
22:                 a ← x + 1
23:             end if
24:         end if
25:     until forever
26: end procedure
```

`toss`
Returns a random rank $0 \le y \le t-1$.

```
1:  procedure toss(ξ, t)
2:      α ← n^c (for some constant c > 1).
3:      Choose uniformly at random an integer M ∈ [0, α]
4:      H ← M · 1/α
5:      Using binary search on [0, t − 1] find 0 ≤ y ≤ t − 1 such that P'_y ≤ H < P'_{y+1}
6:                  (where, for 0 ≤ y ≤ t − 1, P'_y = 1 − (ξ^{-1})/(ξ^{+(y−1)}), and P'_t = 1)
7:      if (H + 1)1/α ≤ P'_{y+1} then
8:          return y
9:      else
10:         α ← α · Π_{y=0}^{t−1}(P'_{y+1} − P'_y)
11:         Choose uniformly at random an integer M ∈ [0, α]
12:         H ← M · 1/α
13:         Using binary search on [0, t − 1] find 0 ≤ y ≤ t − 1 such that P'_y ≤ H < P'_{y+1}
14:                 (where, for 0 ≤ y ≤ t − 1, P'_y = 1 − (ξ^{-1})/(ξ^{+(y−1)}), and P'_t = 1)
15:         return y
16:     end if
17: end procedure
```

Fig. 7. Pseudo code of the pointers tree generator (part 2)

random bits, one at a time) until the subinterval is good. Namely, the Las-Vegas algorithm, after each random bit, checks in time $O(\log n)$ if the subinterval is good by performing a binary search over the commulative probabilities and adds an additional bit only if the selected interval is bad.

5.8.1 Data structures and space complexity. The efficient implementation of `next-child` makes use of the following data structures.

- A number of arrays of length n, $u(j)$ and $flag(j)$, $front(j)$ and $front^{-1}(j)$, used to store various values for nodes j. Since we implement arrays by means of search trees, the space complexity of each array is $O(m)$, where m is the maximum number of distinct keys stored with a non-null value in that array, at any given time. The time complexity for each operation on this arrays is $O(\log m) = O(\log n)$ (since they are implemented as balanced binary search trees).
- For each node j, a balanced binary search tree called $\texttt{child}(j)$, where $\texttt{child}(j)$ includes all nodes i such that $u(i) = j$ (for technical reasons we define $\texttt{child}(j)$ to always include node $n + 1$.)[5] Observe that for each child i stored in one of these trees, $u(i)$ is already determined. Thus, the increase, during a given period, in the space used by the `child` trees is bounded from above by the the number of nodes i for which $u(i)$ got determined during that period. For the time complexity of the operations on these trees we use a coarse standard upper bound of $O(\log n)$ on each tree operation.[6]
  We store the roots of all non-empty trees $\texttt{child}(j)$ in an "array". Thus, using our implementation of arrays as balanced search trees, the space used by this "array" is $O(m)$ and the time to access the root of a certain $\texttt{child}(j)$ is $O(\log m) = O(\log n)$, where m is the number of non-empty trees $\texttt{child}(j)$ at a given time.

The listings of the implementations of the various procedures leave *implicit* the maintenance of two data structures, related to the set K and to the computation of $\varphi(\cdot)$:

- A data structure that allows one to retrieve the value of $\varphi(a)$ for a given node a. This data structure is implemented by retrieving the cardinality of the set of nodes that are not potential parent of a, i.e., $(a-1) - \varphi(a)$, for a given node a. The latter is equivalent to counting how many nodes $i < a$ have $front(i) \neq \text{nil}$ and $front(i) \geq a$. We use two balanced binary search trees (or order statistics trees) in a specific way and have that by standard implementations of balanced search trees the space complexity is $O(k)$ (and all operations are done in time $O(\log k) = O(\log n)$). Here k denotes the number of nodes i such that $front(i) \neq \text{nil}$. More details of the implementation of this data structure appear in the appendix (See Section A.1).
- A data structure that allows one to find the vertex of rank h in the ordered set $[a, n+1] \setminus K$. This data structure is implemented by a balanced binary search tree storing the nodes in K, augmented with the queries $rank_K(i)$ (as in an order-statistics tree [8, Sec. 14] [7]) as well as $rank_{\bar{K}}(i)$ and $select_{\bar{K}}(s)$, i.e., finding the element of rank s in the complement of K. To find the vertex of rank h in $[a, n+1] \setminus K$ we use the query $select_{\bar{K}}(rank_{\bar{K}}(a) + h)$. The space complexity of this data structure is $O(k)$, and all operations are done in time $O(\log k) = O(\log n)$ or $O(\log^2 k) = O(\log^2 n)$ (for the $select_{\bar{K}}(i)$ query). Here k denotes the number of nodes in K,

[5] So that we maintain low space complexity, for a given (j), $\texttt{child}(j)$ is initialized only at the first use of $\texttt{child}(j)$, at which time node $n + 1$ is inserted .

[6] In fact we can use the fact that with high probability $C_j = O(\log n)$ and get, with high probability, an upper bound of $O(\log C_j) = O(\log \log n)$ on the time complexity.

[7] An order-statistic tree is a data structure, that supports two operations beyond the classical balanced binary search tree operations (i.e., insertion, lookup, and deletion), as follows. Informally, given an index i the $select(i)$ operation returns the i-th element in the sorted list of the elements that are in the tree. On the other hand, the rank of an element x in the tree is its index in that sorted list.

which is upper bounded by the number of nodes i such that $front(i) \neq \text{nil}$. More details of the implementation of this data structure appear in the appendix (See Section A.2).

5.8.2 Time complexity.

Time complexity of `toss`(φ, s). The time complexity of this procedure is with high probability $O(\log n)$ (see Section 5.8)

Time complexity of "$x \leftarrow$ the vertex of rank h in $[a, n+1] \setminus K$". This operation is implemented using the data structure defined above, and takes $O(\log^2 n)$ time.

Time complexity of `parent`(j). As stated in Lemma 16, the time complexity of *parent* is $O(\log n)$.

Time complexity of `next-child`. First consider the time complexity consumed by a single invocation of `next-child` (i.e., without taking into account the time consumed by recursive calls of `next-child`):[8] The call to `parent` takes $O(\log n)$ time. Therefore, until the start of the repeat loop, the time is $O(\log n)$ (the time complexity of `successor` is $O(\log n)$). Now, the time complexity of a single iteration of the loop (without taking into account recursive calls to `next-child`) is $(O \log^2 n)$ because:

- Each access to an "array" takes $O(\log n)$ time.
- Calculating $\varphi(a)$ takes $O(\log n)$ time.
- The call to `toss` takes $O(\log n)$ time.
- Finding the vertex of rank h in $[a, n+1] \setminus K$ takes $O(\log^2 n)$ time.
- Each of the $O(1)$ updates of $front(\cdot)$ or $front^{-1}(\cdot)$ may change the set K, and therefore may take $O(\log n)$ time to update the data structure involving K.
- An update of any given `child`$(\cdot)$ binary search tree takes $O(\log n)$ time.

We now examine the number of iterations of the loop.

Claim 13. *With high probability, the number of iterations of the loop in a single invocation of* `next-child` *is* $O(\log n)$.

Proof. We consider a process where the iterations continue until the selected node is node b. A random variable, R, depicting this number dominates a random variable that depicts the actual number of iterations. For each iteration, an additional node is selected by `toss`. By Lemma 11 the probability that a node $j < b$ is selected by `toss` is $1/\varphi(j)$, and we have that $1/\varphi(j) \leq \frac{1}{j-1}$. Thus, $R = 1 + \sum_{j=a}^{b-1} X_j$, where X_j is 1 iff node j was selected, 0 otherwise. Since $\mu = \sum_{j=a}^{b-1} \frac{1}{\varphi(j)} \leq \log n$, using Chernoff bound[9] we have, for any constant $c > 6$, $P[R > c \cdot \log n] \leq 2^{-c \cdot \log n} = n^{-\Omega(1)}$. □

We thus have the following.

Lemma 14. *For any given invocation of* `next-child`*, with high probability, the time complexity is* $O(\log^3 n)$.

5.8.3 Randomness complexity. Randomness is used in our generator to randomly select the parent of the nodes (in `parent`) and to randomly select a next child for a node (in `toss`). We use the common convention that, for any given m, one can choose uniformly at random an integer in $[0, m-1]$ using $O(\log m)$ random bits and in $O(1)$ computation time. We give our algorithms and analyses based on this building block.

[8] We talk about an "invocation", rather than a "call", when we want to emphasize that we consider only the resources consumed by a single level of the recursion tree.

[9] cf. [16], Inequality (8).

In procedure parent we use $O(\log n)$ random bits whenever, for a given j, this procedure is called with parameter j for the first time.

In procedure toss the if condition holds with probability $1 - 1/n^{c-1}$ (where c is the constant used in that procedure). Therefore, given a call to toss, with probability $1 - 1/n^{c-1}$ this procedure uses $O(\log n)$ bits. By Claim 13, in each call to next-child the number of times that toss is called is, w.h.p., $O(\log n)$. We thus have the following.

Lemma 15. *During a given call to* next-child*, w.h.p., $O(\log^2 n)$ random bits are used.*

The following lemma states the time, space, and randomness complexities of the queries.

Lemma 16. *The complexities of* next-child-flag *and* parent *are as follows.*

- *Given a call to* parent *the following hold for this call:*
  (1) The increase, during the call, of the space used by our algorithm is $O(1)$.
  (2) The number of random bits used during that call is $O(\log n)$.
  (3) The time complexity of that call is $O(\log n)$.
- *Given an call to* next-child-flag*, with high probability, all of the following hold for this call:*
  (1) The increase, during that call, of the space used by our algorithm is $O(\log^2 n)$.
  (2) The number of random bits used during that call is $O(\log^4 n)$.
  (3) The time complexity of that call is $O(\log^5 n)$.

Proof. During a call to parent(j) the size of the used space increases when a pointer $u(j)$ becomes non-null or when additional values are stored in child($u(j)$). To select $u(j)$, $O(\log n)$ random bits are used, and $O(\log n)$ time is used to insert j in child($u(j)$) and to update the data structure for the set K (this is implicit in the listing).

For the analysis of next-child-flag, we first consider next-child. Observe that by Lemma 12, w.h.p., each and every root (non-recursive) call of next-child has a recursion tree of size $O(\log n)$. In each invocation of next-child, $O(1)$ variables *front*(j) and $u(j)$ may be updated. Therefore, w.h.p., for all root (non-recursive) calls to next-child it holds that the increase in space during this call is $O(\log n)$ (see Section 5.8.1). Using Lemmas 15 and 12 we have that, w.h.p., each root call of next-child uses $O(\log^3 n)$ random bits. Using Lemmas 14 and 12, we have that, w.h.p., the time complexity of each root call of next-child is $O(\log^4 n)$.

Because the flags of the pointers are uniformly distributed in {imm, rec}, each call to next-child-flag results, w.h.p., in $O(\log n)$ calls to next-child. The above complexities are thus multiplied by an $O(\log n)$ factor to get the (w.h.p.) complexities of next-child-flag. □

6 ON-THE-FLY GENERATOR FOR BA-GRAPHS

Our on-the-fly generator for BA-graphs (see definition in Section 1.1) is called O-t-F-BA, and simply calls BA-next-neighbor(v) for each query on node v. We present an implementation for the BA-next-neighbor query, and prove its correctness, as well as analyze its time, space, and randomness complexities. The on-the-fly BA generator maintains n standard heaps, one for each node. The heaps store nodes, where the order is the natural order of their serial numbers.[10] The heap of node j stores some of the nodes already known to be neighbors of j. In addition, the generator maintains for purely technical reasons an array of size n, $first\_query$, indicating if a BA-next-neighbor query has been issued for a given node. The implementation of the BA-next-neighbor query works as follows (see Figure 8).

[10]For simplicity of presentation we assume that the initialization of the heap occurs at the first insert, and make sure in our use of the heap that no extraction is performed before the first insert.

BA-next-neighbor
Returns the next neighbor of j in the BA-graph.

```
1:  procedure BA-next-neighbor(j)
2:      if first_query(j) = true then
3:  /* first query for j */
4:          first_query(j) ← false
5:          heap-insert(heap_j, n + 1)
6:          heap-insert(heap_j, next-child-flag(j, j, imm))
7:          return BA-parent(j)
8:      else
9:  /* all subsequent queries for j */
10:         r ← heap-extract-min(heap_j)
11:         if r = n + 1 then
12:             heap-insert(heap_j, n + 1)
13:             return n + 1
14:         else
15:             if flag(r) = imm then
16:                 heap-insert(heap_j, next-child-flag(j, r, imm))
17:                 heap-insert(heap_j, next-child-flag(r, r, rec))
18:             else
19:                 (q, flag) ← parent(r)
20:                 heap-insert(heap_j, next-child-flag(q, r, rec))
21:                 heap-insert(heap_j, next-child-flag(r, r, rec))
22:             end if
23:             return r
24:         end if
25:     end if
26: end procedure
```

BA-parent
Returns the parent of j in the BA-graph.

```
1:  procedure BA-parent(j)
2:      (i, flag) ← parent(j)
3:      if flag = imm then
4:          return i
5:      else
6:          return BA-parent(i)
7:      end if
8:  end procedure
```

Fig. 8. Pseudo code of the on-the-fly BA generator

- *For the first* BA-next-neighbor(j) *query*, for a given j, we proceed as follows. We find the parent of j in the constructed BA-graph, which is done by following, in the pointers tree, the pointers of the ancestors of j until we find an ancestor pointed to by an imm pointer (and not a rec pointer). See Figure 8. In addition, we initialize the process of finding neighbors of j to its right (i.e., with a bigger serial number) by inserting into the heap of j the "final node" $n + 1$ as well as the first child of v.

- For any *subsequent* BA-next-neighbor(j) *query* for node j we proceed as follows. Observe that any subsequent query is to return a *child* of j in the constructed BA-graph. The children of j in the BA-graph are those nodes x which have, in the pointers tree, a path of $u(\cdot)$ pointers starting at x and ending at j and with all pointers on that path, except the last one, being rec (the last one being imm). The query BA-next-neighbor(j) has, however, to report the children in increasing order of their index. To this end the heap of node j is used; it stores at any give time some of the children of j in the BA-graph, *not yet returned by a* BA-next-neighbor(j) *query*. We further have to update this heap so that BA-next-neighbor(j) will continue to return the next child according to the index order. To this end we proceed as follows. Whenever node, r is extracted from the heap, in order to be returned as the next child, we update the heap to include the following:
  - If r has an imm pointer to j, then we add to the heap (1) the next node, after r, with an imm pointer to j, and (2) the first node that has a rec pointer to r.
  - If r has a rec pointer to a node r', then we add to the heap (1) the first node, after r, with a rec pointer to r', and (2) the first node that has a rec pointer to r.

The proof of Lemma 17 below is based on the premise that the heap contains only children of v in the BA-graph, and that it always contains the child of v just after the one last returned.

Lemma 17. *The procedure* BA-next-neighbor *returns the next neighbor of* v.

Proof. Given a pointers tree we define the following notions:

- The set of nodes which have an imm pointer to a given node j. That is, for $1 \leq j \leq n$, $D(j) \triangleq \{i \mid u(i) = j, flag(i) = \texttt{imm}\}$.
- The set of nodes which have a rec pointer to a given node j. That is, for $1 \leq j \leq n$, $R(j) \triangleq \{i \mid u(i) = j, flag(i) = \texttt{rec}\}$.

Given a BA graph, for any node $1 \leq j \leq n$ and any prefix length $0 \leq \ell \leq n-1$, we denote by $N^{\ell}(j)$ the set of the first (according to the index number) ℓ neighbors of j in the BA graph.

In what follows we consider an arbitrary node j. We consider the actions of BA-next-neighbor (see Figure 8). Let $M^{\ell}(j)$ be the set of nodes returned by the first ℓ calls BA-next-neighbor(j). We first prove that the following invariant holds.
Just after call number $\ell \geq 1$ of BA-next-neighbor(v):

(1) The heap $heap_j$ contains only neighbors of j in the BA-graph.
(2) The heap $heap_j$ contains the minimum node in $D(j) \setminus M^{\ell}(j)$.
(3) Let q be the first neighbor of j in the BA graph. The heap $heap_j$ contains, for each node $i \in M^{\ell}(j) \setminus \{q\}$, the minimum node in $R(i) \setminus M^{\ell}(j)$.

We prove that the invariant holds by induction on ℓ. The induction basis, for call number $\ell = 1$, holds since (1) the first call to BA-next-neighbor(j) results in inserting into $heap_j$ the first node x which has an imm pointer to node j and (2) $heap_j$ was previously empty (see Figure 8). Thus all points of the invariant hold after call $\ell = 1$. For $\ell > 1$ assume that the induction hypothesis holds for $\ell - 1$ and let r be the node returned by the ℓ'th call to BA-next-neighbor(j). We claim that the invariant still holds after call ℓ by verifying each one of the two cases for the pointer of r and the insertions into the heap for each such case.
If r has an imm pointer, then the following nodes are inserted into $heap_j$: (1) The first node after r with an imm pointer to j. Since this is a neighbor of j in the BA graph Point 1 continues to hold. Since r, just extracted from the heap, was the minimum node in the heap, Point 2 continues to hold. (2) The first node after r which has a rec pointer to r. Since this is a neighbor of j in the BA graph Point 1 continues to hold; Point 3 continues to hold since nothing has changed for any other $i \neq r$,

$i \in M^\ell(j) \setminus \{q\}$, and for r the minium node in $R(i) \setminus M^\ell(j)$ is just inserted.
If r has a `rec` pointer, and let q be the parent of r in the pointers tree, then the following nodes are inserted into $heap_j$: (1) The first node after r which has a `rec` pointer to q; denote it x. Since x is a neighbor of j in the BA graph Point 1 continues to hold. Since r, just extracted from the heap, was the minimum node in the heap, x is the minimum node in $R(i) \setminus M^\ell(j)$ and Point 3 continues to hold (nothing changes for any $q' \neq q$, $q' \in M^\ell(j) \setminus \{q\}$). (2) The first node after r which has a `rec` pointer to r. The same arguments as those for the corresponding case when r has an `imm` pointer hold, and thus both Point 1 and Point 3 continue to hold.
This concludes the proof of the invariant.

We now use the above invariant in order to prove that, for any $\ell \geq 1$, $N^\ell(j) = M^\ell(j)$. We do this by induction on ℓ. For $\ell = 1$ the claim follows from the facts the first neighbor of node j is its parent in the BA graph and that the first call `BA-next-neighbor`(j) returns the value that `BA-parent`(j) returns. This proves the induction basis. We now prove the claim for $\ell > 1$ given the induction hypothesis for all $\ell' < \ell$. Let node x be the ℓ'th neighbor of j. We have two cases: (1) node x has an `imm` pointer to j; (2) node x has a `rec` pointer to another child of j in the BA graph (i.e., to another neighbor of j in the BA graph, which is not the first neighbor).
Case (1): By the induction hypothesis $N^{\ell-1}(j) = M^{\ell-1}(j)$, hence by Point 2 of the invariant x is in the heap $heap_j$ when the ℓ'th call occurs. Since any node returned by `BA-next-neighbor`(j) is no longer in $heap_j$, by Point 1 of the invariant, $heap_j$ does not contain any node smaller than x. Therefore the node returned by the ℓ'th call of `BA-next-neighbor`(j) is node x.
Case (2): Let node y be the parent of node x in the pointers tree, i.e., $u(x) = y$. Since y is a neighbor of j in the BA graph, and $y < x$, it follows that $y \in N^{\ell-1}(j)$, and by the induction hypothesis $y \in M^{\ell-1}(j)$. Moreover, any node $x' < x$ has $u(x') = y$, $flag(x') =$ `rec` if and only if it is a neighbor of j, hence any such node x' is in $N^{\ell-1}(j)$, and by the induction hypothesis also in $M^{\ell-1}(j)$. It follows from Point 3 of the invariant that x is in the heap $heap_j$ when the ℓ'th call occurs. Since any node returned by `BA-next-neighbor`(j) is no longer in $heap_j$, by Point 1 of the invariant, $heap_j$ does not contain any node smaller than i. Therefore the node returned by the ℓ'th call of `BA-next-neighbor`(j) is node i. This completes the proof of the lemma. □

Lemma 18. *For any given root (non-recursive) call of* `BA-parent`*, with high probability, that call takes* $O(\log^2 n)$ *time.*

Proof. Consider the execution of `BA-parent` as stated in Figure 8. Consider the recursive invocation tree that results from a call to `BA-parent`. Each path in this tree corresponds to a path in the pointers tree. Since, By Claim 4, w.h.p. the height of the pointers tree is bounded by $O(\log n)$, the lemma follows by Lemma 16. □

We can now conclude with the following theorem.

Theorem 19. *For any given call of* `BA-next-neighbor`*, with high probability, all of the following hold for that call:*

(1) The increase, during that call, of the space used by our algorithm is $O(\log^3 n)$.
(2) The number of random bits used during that call is $O(\log^5 n)$.
(3) The time complexity of that call is $O(\log^6 n)$.

Proof. Each call of `BA-next-neighbor` is executed, w.h.p., by $O(\log n)$ number of calls (and a constant number of calls in expectation) to `BA-parent` and `next-child-flag`, as well as $O(\log n)$ number of calls to `heap-insert` and `heap-extract-min`, and accesses to "arrays". The claim then follows from standard deterministic heap implementations (which uses $O(1)$ space per stored item and $O(\log n)$ time per query) and from Lemma 16. □

We now state the properties of our on-the-fly graph generator for BA-graphs.

Definition 20. *For a number of queries $T > 0$ and a sequence of* `BA-next-neighbor` *queries $Q = (q(1), \ldots, q(T))$, let $A(Q)$ be the sequence of answers returned by an algorithm A on Q. If A is randomized then $A(Q)$ is a probability distribution on sequences of answers.*

Let $\texttt{Opt-BA}_n$ be the (randomized) algorithm that first runs the Markov process to generate a graph G on n nodes according to the BA model, stores G, and then answers queries by accessing the stored G. Let $\texttt{O-t-F-BA}_n$ be the algorithm `O-t-F-BA` run with graph-size n. From the definition of the algorithm we have the following.

Theorem 21. *For any sequence of queries Q, $\texttt{Opt-BA}_n(Q) = \texttt{O-t-F-BA}_n(Q)$.*

We now conclude by stating the complexities of our on-the-fly BA generator.

Theorem 22. *For any $T > 0$ and any sequence of queries $Q = (q(1), \ldots, q(T))$, when using $\texttt{O-t-F-BA}_n$ it holds w.h.p. that, for all $1 \leq t \leq T$:*

(1) The increase in the used space, while processing query t, is $O(\log^3 n)$. Therefore, the total space that is used to store the state of the generator after T queries is $O(T \cdot \log^3 n)$.
(2) The number of random bits used while processing query t is $O(\log^5 n)$.
(3) The time complexity for processing query t is $O(\log^6 n)$.

Proof. A query `BA-next-neighbor`(v) at time t is a *trivial* if at some $t' < t$ a query `BA-next-neighbor`(v) returns $n + 1$. Observe that trivial queries take $O(\log n)$ *deterministic* time, do not use randomness, and do not increase the used space. Since there are less than n^2 *non-trivial* queries, the theorem follows from Theorem 19 and a union bound. □

We note that the various assertions in this paper of the form of "with high probability ... is $O(\log^c n)$" can also be stated in the form of "with probability $1 - \frac{1}{n^d}$ ... is $f(d) \cdot \log^c n$". Therefore, we can combine these various assertions, and together with the fact that the number of non-trivial queries is poly(n), we get the final result stated above.

7 A DIRECTION FOR EXTENDING TO GENERAL OUT-DEGREES

Given a random process that generates a Preferential-Attachment graph with out-degree 1, Bollobás and Riordan [7] consider the following generalization for defining a process that generates a Preferential-Attachment graph with general out-degree. A BA_n^m-graph, where BA_n^m denotes an n node Preferential-Attachment graph with out-degree m, is constructed from a BA_{nm}-graph by identifying each of the n nodes of the BA_n^m-graph with a block of m nodes in the BA_{nm}-graph [11]

Hence, a BA_n^m-graph can be generated on-the-fly by using an on-the-fly generator for a BA_{nm}-graph, in a black-box manner, by increasing the complexity by a factor of m (up to poly-logarithmic factors). We leave working out the details for this generalization to higher out-degrees, as well as for other variants of generalization, for further research.

8 CONCLUSIONS

We introduce an approach to probing and accessing a huge random graph, sampled from a given distribution, in a way that does not require to first sample the whole huge graph and then access it. The latter approach may require prohibitive amounts of time, space and randomness which could be avoided if only relatively small parts of the random graph are accessed. The feasibility

[11] Bollobás and Riordan define the process BA_n slightly different from the model we are using in this paper. According to their definition the head of the edge e_j is the node v_i, for $1 \leq i \leq j-1$, with probability $\frac{deg(v_i, BA_{j-1})}{2j-1}$ and is v_j with probability $\frac{1}{2j-1}$ (so there is some, vanishing, probability of forming self-loops).

of such savings may be especially challenging when the distribution at hand is usually defined by an evolving sequential process (over, e.g., nodes) such as in the Barabási-Albert Preferential Attachment model or the random recursive tree model.

We show how to achieve such savings for the Barabási-Albert graphs of out-degree 1, as well as for the evolving tree model, and give on-the-fly generation algorithms for both models such that with probability $1 - 1/\text{poly}(n)$, each and every query is answered in $\text{polylog}(n)$ time, and the increase in space and the number of random bits consumed by any single query are both $\text{polylog}(n)$, where n denotes the number of vertices in the graph.

Acknowledgments. We thank Yishay Mansour for raising the question of whether one can locally generate preferential attachment graphs, and Dimitri Achlioptas and Matya Katz for useful discussions. We further thank an anonymous ICALP reviewer for a comment that helped us simplify one of the data structure implementations.

REFERENCES

[1] Md. Maksudul Alam, Maleq Khan, and Madhav V. Marathe. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, pages 91:1–91:12, 2013.

[2] Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1132–1139, 2012.

[3] Albert-László Barabási and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[4] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[5] Amartya Shankha Biswas, Ronitt Rubinfeld, and Anak Yodpinyanee. Local-access generators for basic random graph models. *CoRR*, abs/1711.10692, 2017.

[6] Andrej Bogdanov and Hoeteck Wee. A stateful implementation of a random function supporting parity queries over hypercubes. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 298–309. Springer, 2004.

[7] Béla Bollobás and Oliver Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[9] Georgios Drakopoulos, Stavros Kontopoulos, Christos Markis, and Vasileios Megalooikonomou. Large graph models: A review. *CoRR*, abs/1601.06444, 2016.

[10] Michael Drmota. *Random Trees: An Interplay Between Combinatorics and Probability*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[11] Guy Even, Reut Levi, Moti Medina, and Adi Rosén. Sublinear random access generators for preferential attachment graphs. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 6:1–6:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[12] Guy Even, Moti Medina, and Dana Ron. Best of two local models: Local centralized and local distributed algorithms. *CoRR*, abs/1402.3796, 2014.

[13] Guy Even, Moti Medina, and Dana Ron. Deterministic stateless centralized local algorithms for bounded degree graphs. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 394–405, 2014.

[14] William Goh and Eric Schmutz. Limit distribution for the maximum degree of a random recursive tree. *Journal of computational and applied mathematics*, 142(1):61–82, 2002.

[15] Oded Goldreich, Shafi Goldwasser, and Asaf Nussboim. On the implementation of huge random objects. *SIAM J. Comput.*, 39(7):2761–2822, 2010.

[16] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Inf. Process. Lett.*, 33(6):305–308, 1990.

[17] Tamara G. Kolda, Ali Pinar, Todd Plantenga, and C. Seshadhri. A scalable generative graph model with community structure. *SIAM J. Scientific Computing*, 36(5), 2014.

[18] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. Random graph models for the web graph. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 57–65, 2000.

[19] Reut Levi, Guy Moshkovitz, Dana Ron, Ronitt Rubinfeld, and Asaf Shapira. Constructing near spanning trees with few local inspections. *CoRR*, abs/1502.00413, 2015.
[20] Reut Levi, Dana Ron, and Ronitt Rubinfeld. Local algorithms for sparse spanning graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, pages 826–842, 2014.
[21] Reut Levi, Ronitt Rubinfeld, and Anak Yodpinyanee. Brief announcement: Local computation algorithms for graphs of non-constant degrees. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 59–61, 2015.
[22] Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. Converting online algorithms to local computation algorithms. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, pages 653–664, 2012.
[23] Yishay Mansour and Shai Vardi. A local computation approximation scheme to maximum matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, pages 260–273, 2013.
[24] Ulrich Meyer and Manuel Penschuck. Generating massive scale-free networks under resource constraints. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 39–52, 2016.
[25] Joel C Miller and Aric Hagberg. Efficient generation of networks with given expected degrees. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 115–126. Springer, 2011.
[26] Huy N Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 327–336. IEEE, 2008.
[27] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th international conference on extending database technology*, pages 331–342. ACM, 2011.
[28] Krzysztof Onak. New sublinear methods in the struggle against classical problems. *Massachusetts Institute of Technology, PhD Thesis*, September 2010.
[29] Krzysztof Onak, Dana Ron, Michal Rosen, and Ronitt Rubinfeld. A near-optimal sublinear-time algorithm for approximating the minimum vertex cover size. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1123–1131, 2012.
[30] Arjun S Ramani, Nicole Eikmeier, and David F Gleich. Coin-flipping, ball-dropping, and grass-hopping for generating random graphs from matrices of edge probabilities. *SIAM Review*, 61(3):549–595, 2019.
[31] Omer Reingold and Shai Vardi. New techniques and tighter bounds for local computation algorithms. *J. Comput. Syst. Sci.*, 82(7):1180–1200, 2016.
[32] Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 223–238, 2011.
[33] Martin Sauerhoff. On the entropy of models for the web graph. Manuscript.
[34] Robert T Smythe and Hosam M Mahmoud. A survey of recursive trees. *Theory of Probability and Mathematical Statistics*, (51):1–28, 1995.
[35] Andy Yoo and Keith W. Henderson. Parallel generation of massive scale-free graphs. *CoRR*, abs/1003.3684, 2010.
[36] Yuichi Yoshida, Masaki Yamamoto, and Hiro Ito. Improved constant-time approximation algorithms for maximum matchings and other optimization problems. *SIAM J. Comput.*, 41(4):1074–1093, 2012.

A IMPLEMENTATIONS OF DATA STRUCTURES

A.1 Data Structure for $\varphi(\cdot)$

We use two balanced binary search trees (or order statistic trees). One, called `left`, stores all vertices i such that $front(i) \neq \text{nil}$. The other, called `right`, stores (the multi-set) $\{front(i) \mid front(i) \neq \text{nil}\}$. To determine $\varphi(a)$ we find, using tree `right`, how many nodes i have $front(i) > a-1$ (and $front(i) \neq \text{nil}$). Let this number be R. Using tree `left` we find how many nodes $i < a$ have $front(i) \neq \text{nil}$. Let this number be L. Then $\varphi(a) = R - L$.

By standard implementations of balanced search trees the space complexity is $O(k)$ and all operations are done in time $O(\log k) = O(\log n)$) Here k denotes the number of nodes i such that $front(i) \neq \text{nil}$.

A.2 Data Structure to find the node of rank h in $[a, n+1] \setminus K$

We start with a number of definitions useful for specifying the data structure and its operations.

For a node $j \in \{1, \ldots, n+1\}$ and a subset of nodes $Q \subseteq \{1, \ldots, n+1\}$, define $Q(j)$ as follows:

$$Q(j) \triangleq \begin{cases} j & \text{if } j \in Q \text{ or } j = 1 \\ \max_{j' \in Q}\{j' | j' < j\} & \text{otherwise} \end{cases} .$$

Note that for technical reasons for $j = 1$ we define $Q(j) = 1$ whether or not $j \in Q$.

For a node $j \in \{1, \ldots, n+1\}$ and a subset of nodes $Q \subseteq \{1, \ldots, n+1\}$, define $rank_Q(j)$ as follows:

$$rank_Q(j) \triangleq |\{i \mid i < Q(j); i \in Q\}| .$$

We note that using these definitions we have that, for any $j \in \{1, \ldots, n+1\}$, the number of items $i < j$ in $\bar{Q}$, where $\bar{Q} = \{1, \ldots, n+1\} \setminus Q$, is $(j-1) - rank_Q(j)$.

The $insert_Q$, $delete_Q$ and $rank_Q$ operations are implemented as in a standard order-statistics tree based on a balanced binary search tree. The operation $rank_{\bar{Q}}$ is implemented using the $rank_Q$ and then performing the calculation above. To implement $select_{\bar{Q}}(s)$ we proceed as follows. We traverse the search tree with the value s, and in each node of the tree that contains the vertex j we compare s with $(j-1) - rank_Q(j)$. Thus, we can find the maximum $j \in Q$ such that $rank_{\bar{Q}}(j) \leq s$. Denote this node j'. We then return the node $j' + \left[s + 1 - ((j'-1) - rank_Q(j'))\right]$.

The time complexities of $insert_Q$, $delete_Q$ and $rank_Q$ and $rank_{\bar{Q}}$ are therefore $O(\log n)$ based on standard order statistics trees. The time complexity of $select_{\bar{Q}}$ is $O(\log^2 n)$: for each node along the search path of length $O(\log n)$ we need to use the query $rank_Q$.