Sublinear Random Access Generators for Preferential Attachment Graphs

Guy Even[1], Reut Levi[2], Moti Medina[3], and Adi Rosén[*4]

1 **Tel Aviv University, Tel Aviv, Israel**
guy@eng.tau.ac.il
2 **MPI for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
reuti.levi@gmail.com
3 **MPI for Informatics, Saarland Informatics Campus, Saarbrücken, Germany**
moti.medina@gmail.com
4 **CNRS and Université Paris Diderot, Paris, France**
adiro@liafa.univ-paris-diderot.fr

Abstract

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 to generate the graph on-the-fly, in response to queries, and therefore to require amounts of time, space, and random bits which are a function of the actual number of queries.

We focus on two random graph models: the Barabási-Albert Preferential Attachment model (BA-graphs) [3] and the random recursive tree model [24]. 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 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.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity

Keywords and phrases local computation algorithms, preferential attachment graphs, random recursive trees, sublinear algorithms

Digital Object Identifier 10.4230/LIPIcs.ICALP.2017.6

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

* Research supported in part by ANR projet RDAM.

44th International Colloquium on Automata, Languages, and Programming (ICALP 2017).
Editors: Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl;
Article No. 6; pp. 6:1–6:15


Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(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 time, the memory space, and the number of used random bits, required to answer a query (P, t).

We focus on the case of generative models for random graphs, and in particular, on the Barabási-Albert Preferential Attachment model [3] (which we call BA-graphs), on the equivalent linear evolving copying model of Kumar et al. [10], and on the random recursive tree model [24]. 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 samples 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 $1 - 1/\text{poly}(n)$. That is, 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 during that query are $\text{polylog}(n)$. Our result refutes (definitely for $\text{polylog}(n)$ queries) the recent statement of Kolda et al. [9] 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 [23]. 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 insure 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 [24] (also used within the evolving copying model [10]), and use them as a building block for our main result for BA-graphs.

Due to space limitations, some of the proofs are omitted from this extended abstract.

Related work. A linear time randomized algorithm for efficiently generating BA-graphs is given in Betagelj and Brandes [4]. See also Kumar et al. [10] and Nobari et al. [18]. A parallel algorithm is given in Alam et al. [1]. See also Yoo and Henderson [25]. An external memory algorithm was presented by Meyer and Peneschuck [16]. Generating huge random objects while using "small" amounts of randomness was studied by Goldreich, Goldwasser and Nussboim [8]. Mansour et al. [14] consider local generation of bipartite graphs in the context of local simulation of Balls into Bins online algorithms.

Applications. One reason for generating large BA-graphs is to simulate algorithms over them. 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 [20, 26, 17, 19] which probe a constant number of neighbors. 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 [6, 7, 21, 22, 2, 14, 15, 11, 12, 13]. Support of adjacency list queries is especially useful for simulating (partial) DFS and BFS over graphs.

2 Preliminaries

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 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.

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.

For ease of presentation, we extensively use in the algorithms arrays of size n. However, in order to keep the space complexity low, we implement these arrays by means of balanced search trees, with keys in $[1, n]$. Thus, the space used by the "arrays" is the number of keys stored. The time complexities that we give are therefore to be multiplied by a factor of $O(\log n)$.

3 Queries and On-the-Fly Generators

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` : $[1, n] \to [1, n + 1]$, where $n + 1$ denotes "no additional neighbor". 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 $v \in V$ 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, $last_t(v) = \min\{0, \min_{t' < t}\{\texttt{BA-next-neighbor}(q_{t'}) | q_{t'} = v\}$. At time t the query `BA-next-neighbor`(v) 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.

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

[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.

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 16 and Theorem 17).

4 Random Graph Models

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]). 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 $i \in [1, n]$. (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 $\frac{\deg(v_i, G)}{2(j-1)}$.

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 [10]. 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 $u(n) \in [1, n-1]$ be a uniformly distributed random variable (the random variables $b_1, \ldots, b_n$ and $u(1), \ldots, u(n)$ are all pairwise independent.) The head v_i of e'_n is determined as follows: $head(e'_n) \triangleq u(n)$, if $b_n = 1$; and $head(e'_n) \triangleq head(e_{u(n)})$, if $b_n = 0$.

Random recursive tree model [24]. If we eliminate from the evolving copying model the bits b_i and the "copying effect" 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 [24].

We now relate the various models. Proof omitted from this extended abstract.

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

We use the following claim in the sequel.

▶ **Claim 2** (cf. [5], Thm. 6.12 and Thm. 6.32). *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 $O(\log n)$.*

5 The Pointers Tree

We now consider a graph inspired by the the random recursive tree model [24] and the evolving copying model [10]. 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 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 a `dir` (direct) pointer or as a `rec` (recursive) pointer. 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 equality distributed between `rec` and `dir`, to the BA model).

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. [24]). 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-tp`$(j, k, type)$, 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 "type". If no such node exists then i is $n + 1$.

The "ideal" way to implement this task 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 $\{$`dir`, `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. We start with some notations. We say that j is *exposed* if $u(j) \neq$ nil (initially all pointers $u(j)$ are set to nil). We denote the set of all exposed vertices by F. We say that j is *directly* exposed if $u(j)$ was set during an invocation of `next-child-tp`$(i, \cdot, \cdot)$. We say that j is *indirectly* exposed if $u(j)$ was determined during an invocation of `parent`(j). As a result of answering and processing `next-child-tp` 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). For a node i, $front(i)$ denotes the largest value (node) $k \in [1, n+1]$ such that k was returned by a `next-child-tp`$(i, \cdot, \cdot)$ query, and nil if no such returned value exists. Observe that $front(i) = k$ implies that (1) $u(k) = i$; and (2) we know already for each node $j \in [j+1, k-1]$ if $u(j) = i$ or not. We denote - roughly speaking - the set of vertices that cannot serve as u-parents of j by $not\text{-}u\text{-}parent\text{-}candidate(j)$, the nodes that can still be u-parents of j by $\Phi(j)$, and their number by $\varphi(j) = |\Phi(j)|$. The formal definitions are:

$$
\begin{aligned}
not\text{-}u\text{-}parent\text{-}candidate(j) &\triangleq \{i \in [1, j-1] : front(i) \geq j\} \ , \\
\Phi(j) &\triangleq [1, j-1] \setminus not\text{-}u\text{-}parent\text{-}candidate(j) \ , \\
\varphi(j) &\triangleq |\Phi(j)| \ .
\end{aligned}
$$

5.1 An efficient implementation of `next-child`

We first shortly discuss the challenges on the way to an efficient implementation of `next-child`. Observe that before the first `next-child`(j) query, for a given j, is issued, the probability for any $x > j$ to be a u-child of j is $1/\varphi(x)$, because all nodes $x' < x$ can still be the u-parent of x. But once `next-child`$(\cdot)$ queries are issued, this may no longer be the case. For example, if $x > front(j')$, then, even if the u-parent of x is not yet determined, j' is no longer an option to be the u-parent of x. This renders the calculation of $\mathbf{Pr}[u(x) = j]$ more complicated and more computation-time consuming, which renders the process of selecting the next child of a node j non-efficient. In the rest of this section we show how to overcome these

difficulties and give a procedure that selects the next child, with the appropriate probability distribution, using polylog(n) random bits and in polylog(n) time, and while increasing the space by polylog(n). This procedure will be at the heart of our efficient implementation of `next-child`.

The efficient implementation of `next-child` (and of `parent`) makes use of the following data structures.

- An array of length n, $u(j)$
- An array of length n, *type*(j)
- An array of length n, *front*(j) (We also maintain an array $front^{-1}(i)$ with the natural definition.)
- An array of n balanced search trees, called `child`(j), each holding a set of nodes $i > j$ such that $u(i) = j$ (not necessarily all such nodes). For technical reasons we initiate all trees `child`(j) with $n + 1 \in$ `child`(j).
- A number of additional data structures that are implicit in the listing, described and analyzed in the sequel.

In the implementation we maintain the following two invariant.

▶ **Invariant 3.** *For every node j, the first* ***next-child-tp***$(j, \cdot, \cdot)$ *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. One can easily maintain this invariant by introducing a `parent`(j) query as the first step of the implementation of the `next-child-tp`$(j, \cdot, \cdot)$ query (for technical reasons we do that in a lower-level procedure `next-child`.)

▶ **Invariant 4.** *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 $n+1$. Let $front^{-1}(j)$ denote the vertex i such that *front*$(i) = j$, if such a vertex i exists; otherwise $front^{-1}(j) =$ nil. We get that if $front^{-1}(j) \neq$ nil, then $u(j) \neq$ nil.

▶ **Definition 5.** At a given time t, and for any node j, let $\Phi(j)$ and $\phi(j)$ be defined as follows: $\Phi(j) \triangleq \{i \mid i < j$ `and` (*front*$(i) < j$ `or` *front*$(i) =$ nil$)\}$, and $\phi(j) = |\Phi(j)|$.

The following lemma gives properties of the series $\{\Phi(x)\}_x$. Proof omitted.

▶ **Lemma 6.** *For every $x \in [1, n-1]$:*

1. $\Phi(x) \subseteq \Phi(x+1) \subseteq \Phi(x) \cup \{x, front^{-1}(x)\}$.
2. $\Phi(x+1) = \Phi(x)$ *iff* $x \in K$.
3. $\varphi(x+1) - \varphi(x) \leq 1$.

We now describe the implementation of `next-child-tp`$(j, k, type)$ and `next-child`(j). `next-child-tp`$(j, k, type)$ is a loop of `next-child-from`(j, k) until the right type is found, and `next-child-from`(j, k) is essentially a call to `next-child`(j) (see Figure 1). 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*$(j) \neq$ nil; otherwise $a = j + 1$. Let b denote the smallest, larger than *front*(j), indirectly exposed child of j if one exists (i.e., if *front*$(j) \neq$ nil then $b = \min\{\ell > front(j) \mid u(\ell) = j\}$); if no such b

exists then $b = n+1$. By the definition of K, $K \subseteq F$, and 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). However, this procedure takes linear time. We denote by $D(x)$, $x \in I \setminus F$, the probability that x is chosen according to the above procedure. In order to start building our efficient implementation for `next-child` we consider the same process, with the same probabilities $1/\varphi(x)$, but this time for $[a,b) \setminus K$, rather than $[a,b) \setminus F$. The vertex on which we stop, denote is x, is a *candidate next u-child*. If $x \in F \setminus K$, then x cannot be a child of j so we proceed in the same way, but with the interval $[x+1, b]$.

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, Lemma 6 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 $s = |I \setminus K|$ and let P_q, $0 \leq q \leq s-1$ be the probability that the node of rank q in $I \setminus K$ is chosen as candidate in the sequential procedure defined above. Since $\varphi(x)$ form the harmonic sequence for $x \in [a,b) \setminus K$, we can calculate in $O(1)$ time, for any $0 \leq i \leq s-1$, the probability $P'_i = \sum_{q<i} P_q$ (i.e., a node of some rank q, $q < i$, is chosen). Indeed, for $i = 0$, $P_i = \frac{1}{\varphi(a)}$; for $0 < i < s-1$, $P_i = \frac{1}{\varphi(a)+i} \cdot \prod_{\ell=0}^{i-1} \left(1 - \frac{1}{\varphi(a)+\ell}\right) = \frac{\varphi(a)-1}{(\varphi(a)+i-1)(\varphi(a)+i)}$; and for $i = s-1$, $P_{s-1} = \prod_{\ell=0}^{s-2} \left(1 - \frac{1}{\varphi(a)+\ell}\right) = \frac{\varphi(a)-1}{\varphi(a)+s-2}$. Hence, for $0 \leq i \leq s-1$, $P'_i = 1 - \frac{\varphi(a)-1}{\varphi(a)+(i-1)}$, and for $i = s$, $P'_s = 1$. 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 $s-1$ to decide what rank h this number "represents". After we randomly chose a rank $h \in [0, s-1]$, h is then mapped to the vertex of rank h in $I \setminus K$, denote it x, and this is the *candidate next u-child*. As before, if $x \in (F \setminus K)$, then x cannot be a child of j so we ignore it and proceed in the same way, this time with the interval $[x+1, b]$. See the pseudo code of of `next-child` and `toss` (Figure 1). We denote by $\hat{D}(x)$, $x \in I \setminus F$ the probability that x is chosen according to this third procedure. See Figure 1 for a formal definition of this procedure.

Observe that this procedure takes $O(\log s)$ time (see Section 5.2 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.

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. The (technical) proof is omitted.

▶ **Lemma 7.** *For all $x \in I \setminus F$, $\hat{D}(x) = D(x)$.*

The implementation of `parent` is straightforward (see Figure 1). However, note that updating the various data structures, while implicit in the listing, is accounted for in the time analysis.

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

```
1: procedure NEXT-CHILD-FROM(j, k)
2:     If (k ≥ n) return (n + 1)
3:     q ← succ(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
```

```
1: procedure PARENT(j)
2:     if u(j) = nil then
3:         u(j) ←_R [1, j − 1]
4:         type(j) ←_R {dir, rec}
5:     end if
6:     return (u(j), type(j))
7: end procedure
```

```
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 ← { succ(child(j), front(j))   if front(j) ≠ nil
           { n + 1                      if front(j) = nil
6:     repeat
7:         s ← |[a, b] \ K|
8:         h ← toss(φ(a), s)
9:         x ← the vertex of rank h in [a, b] \ K
10:        if x = b then
11:            return b
12:        else
13:            if u(x) = nil then
14:                u(x) = j
15:                type(x) ←_R {dir, rec}
16:                front(j) ← x
17:                front^{-1}(x) ← j
18:                if (front(x) = nil) next-child(x)
19:                return (x)
20:            else
21:                if u(x) = j then
22:                    front(j) ← x
23:                    front^{-1}(x) ← j
24:                    if (front(x) = nil) next-child(x)
25:                    return(x)
26:                else
27:                    a ← x + 1
28:                end if
29:            end if
30:        end if
31:    until forever
32: end procedure
```

```
1: procedure TOSS(φ, s)
2:     α ← n^c (for some constant c > 1).
3:     Choose uniformly at random an integer H ∈ [0, α]
4:     M ← H · 1/α
5:     Using binary search on [0, s − 1] find y such that P'_y ≤ M < P'_{y+1}
6:                 (where, for 0 ≤ y ≤ s − 1, P'_y = 1 − (φ−1)/(φ+(y−1)), and P'_s = 1)
7:     if (H + 1) 1/α ≤ Pr_{y+1} then
8:         return y
9:     else
10:        α ← α · Π_{y=0}^{s−1}(P'_{y+1} − P'_y)
11:        Choose uniformly at random an integer H ∈ [0, α]
12:        M ← H · 1/α
13:        Using binary search on [0, s − 1] find y such that P'_y ≤ H < P'_{y+1}
14:                (where, for 0 ≤ y ≤ s − 1, P'_y = 1 − (φ−1)/(φ+(y−1)), and P'_s = 1)
15:        return y
16:    end if
17: end procedure
```

Figure 1 Pseudo code of the pointers tree generator.

5.2 Analysis of the pointer tree generator

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

▶ **Lemma 8.** *With high probability, for each and every invocation of* `next-child`*, the size of the recursion tree of that invocation 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 an invocation tree of a call of `next-child` is bounded from above by the height of the pointers tree. By Claim 2, with high probability, this is $O(\log n)$. ◀

5.2.1 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 $type(j)$, $front(j)$ 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 is $O(\log m) = O(\log n)$.
- For each node j, a balanced binary search tree called $\texttt{child}(j)$, where $\texttt{child}(j)$ stores (some of the known) children of node j. (for technical reasons we define $\texttt{child}(j)$ to always include node $n+1$.) 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 $O(\log n)$.

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 vertex a. This data structure is implemented by retrieving the cardinality of *not-u-parent-candidate*(a) for a given node a. The latter is equivalent to counting how many nodes $i < a$ have $front(i) \neq \mathsf{nil}$ 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 $front(i) \neq \mathsf{nil}$. The details of the implementation are omitted from this extended abstract.
- 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) 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, which is upper bounded by the number of nodes i such that $front(i) \neq \mathsf{nil}$. The details of the implementation are omitted from this extended abstract.

5.2.2 Time complexity

Time complexity of `toss`(φ, s). The time complexity of this procedure is $O(1)$ regardless of whether or not the `if` condition holds or not.

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). Examining the listing (Figure 1), one observes that the number of operations is constant. However, though implicit in the listing, one should take into account the update of the data structures `child`(j) as well as the data structure that stores the set K, each taking $O(\log n)$ time.

Time complexity of `next-child`. First consider the time complexity of a single invocation of `next-child`, involving the update of the various data structures: 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 `succ` 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:

- The call to `toss` takes O(1) 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.
- Each update of a pointer $u(\cdot)$ results also in an (implicit) update in a certain `child` search tree, taking $O(\log n)$ time.

We now examine the number of iterations of the loop.

▶ **Claim 9.** *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 7 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 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 10.** *For any given invocation of `next-child`, with high probability, the time complexity is $O(\log^3 n)$.*

5.2.3 Randomness complexity

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 an invocation of `toss`, with probability $1 - 1/n^{c-1}$ this procedure uses $O(\log n)$ bits. By Claim 9, in each invocation of `next-child` the number of times that `toss` is called is, w.h.p., $O(\log n)$. We thus have the following.

▶ **Lemma 11.** *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 12.** *The complexities of* `next-child-tp` *and* `parent` *are as follows.*

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

Proof.

`parent`. During an invocation of `parent`(j) the size of the used space increases when a pointer $u(j)$ becomes non-nul 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 consumed to insert j in `child`$(u(j))$ and to update the data structure for the set K (this is implicit in the listing).

`next-child-tp`. We first consider `next-child`. Observe that by Lemma 8, w.h.p., each and every root (non-recursive) invocation 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 invocation is $O(\log n)$ (see Section 5.2.1). Using Lemmas 11 and 8 we have that, w.h.p., each root invocation of `next-child` uses $O(\log^3 n)$ random bits. Using Lemmas 10 and 8, we have that, w.h.p., the time complexity of each root invocation of `next-child` is $O(\log^4 n)$.

Because the types of the pointers are uniformly distributed in {`dir`, `rec`}, each call to `next-child-tp` 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-tp`. ◀

6 On-the-fly Generator for BA-Graphs

Our on-the-fly generator for BA-graphs 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. The heap of node j stores some of the nodes already known to be neighbors of j.

- *For the first* `BA-next-neighbor`(v) *query*, for a given v, we proceed as follows. We find the parent of v in the BA-graph, which is done by following, in the pointers tree, the pointers of the ancestors of v until we find an ancestor pointed to by a `dir` pointer (and not a `rec` pointer). See Figure 2. In addition, we initialize the process of finding neighbors of v to its right (i.e., with a bigger serial number) by inserting into the heap of v the "final node" $n+1$ as well as the first child of v.
- Observe that any *subsequent* `BA-next-neighbor`(v) *query* is to return a *child* of v in the BA-graph. The children x of v in the BA-graph have, in the pointers tree, a path of $u(\cdot)$ pointers starting at x and ending at v with all pointers, except the last one, being `rec`

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

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

Figure 2 Pseudo code of the on-the-fly BA generator.

(the last being `dir`). The query has to report the children in increasing index number. To this end the `heap` of v is used; it stores some of the children of v, *not yet returned by a* `BA-next-neighbor`(v) *query*. This heap is also updated so that `BA-next-neighbor`(v) will continue to return the next child according to the index order. To do so, whenever a node, r, is extracted from the heap, the heap is updated to include the following:

- If r has a `dir` pointer to v, then we add to the heap (1) the next, after r, node with a `dir` pointer to v, 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 the first, after r, node with a `rec` pointer to r'.

The proof of the next lemma, by induction on the number of queries, is omitted.

▶ **Lemma 13.** *The procedure `BA-next-neighbor` returns the next neighbor of v.*

Since the flags in the pointers tree are uniformly distributed, and by Lemma 12, we have:

▶ **Lemma 14.** *For any given root (non-recursive) invocation of `BA-parent`, with high probability, that invocation takes $O(\log^2 n)$ time.*

The next theorem follows from the code, standard heap implementation, and Lemma 12.

▶ **Theorem 15.** *For any given invocation of `BA-next-neighbor`, with high probability, all of the following hold for that invocation:*

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

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

▶ **Definition 16.** 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.

▶ **Theorem 17.** *For any sequence of queries Q, $\texttt{Opt-BA}_n(Q) = \texttt{O-t-F-BA}_n(Q)$.*

▶ **Theorem 18.** *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 $O(\log^2 n)$.*
2. *The number of random bits used while processing query t is $O(\log^4 n)$.*
3. *The time complexity for processing query t is $O(\log^5 n)$.*

Proof. A query $\texttt{BA-next-neighbor}(v)$ at time t is a *trivial* if at some $t' < t$ a query $\texttt{BA-next-neighbor}(v)$ 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 15 and a union bound. ◀

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. doi:10.1145/2503210.2503291.

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. URL: http://dl.acm.org/citation.cfm?id=2095205.

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

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

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

6 Guy Even, Moti Medina, and Dana Ron. Best of two local models: Local centralized and local distributed algorithms. *CoRR*, abs/1402.3796, 2014. URL: http://arxiv.org/abs/1402.3796.

7 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. doi:10.1007/978-3-662-44777-2_33.

8 Oded Goldreich, Shafi Goldwasser, and Asaf Nussboim. On the implementation of huge random objects. *SIAM J. Comput.*, 39(7):2761–2822, 2010. doi:10.1137/080722771.

9 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. doi:10.1137/130914218.

10 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. doi:10.1109/SFCS.2000.892065.

11 Reut Levi, Guy Moshkovitz, Dana Ron, Ronitt Rubinfeld, and Asaf Shapira. Constructing near spanning trees with few local inspections. *CoRR*, abs/1502.00413, 2015. URL: http://arxiv.org/abs/1502.00413.

12 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. doi:10.4230/LIPIcs.APPROX-RANDOM.2014.826.

13 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. doi:10.1145/2755573.2755615.

14 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. doi:10.1007/978-3-642-31594-7_55.

15 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. doi:10.1007/978-3-642-40328-6_19.

16 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. doi:10.1137/1.9781611974317.4.

17 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.

18 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.

19 Krzysztof Onak. New sublinear methods in the struggle against classical problems. *Massachusetts Institute of Technology, PhD Thesis*, September 2010.

20 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. URL: http://dl.acm.org/citation.cfm?id=2095204.

21 Omer Reingold and Shai Vardi. New techniques and tighter bounds for local computation algorithms. *J. Comput. Syst. Sci.*, 82(7):1180–1200, 2016. doi:10.1016/j.jcss.2016.05.007.

22 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. URL: http://conference.itcs.tsinghua.edu.cn/ICS2011/content/papers/36.html.

23 Martin Sauerhoff. On the entropy of models for the web graph. Manuscript. URL: http://ls2-www.cs.uni-dortmund.de/~sauerhof/papers/ent.pdf.

24 Robert T. Smythe and Hosam M. Mahmoud. A survey of recursive trees. *Theory of Probability and Mathematical Statistics*, (51):1–28, 1995.

25 Andy Yoo and Keith W. Henderson. Parallel generation of massive scale-free graphs. *CoRR*, abs/1003.3684, 2010. URL: http://arxiv.org/abs/1003.3684.

26 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. doi:10.1137/110828691.