Generating Practical Random Hyperbolic Graphs in Near-Linear Time and with Sub-Linear Memory*

Manuel Penschuck

Goethe University, Frankfurt, Germany
mpenschuck@ae.cs.uni-frankfurt.de

Abstract

Random graph models, originally conceived to study the structure of networks and the emergence of their properties [8], have become an indispensable tool for experimental algorithmics. Amongst them, hyperbolic random graphs form a well-accepted family, yielding realistic complex networks while being both mathematically and algorithmically tractable. We introduce two generators MemGen and HyperGen for the $G_{\alpha,C}(n)$-model, which distributes n random points within a hyperbolic plane and produces $m = n\bar{d}/2$ undirected edges for all point pairs close by; the expected average degree $\bar{d}$ and exponent $2\alpha{+}1$ of the power-law degree distribution are controlled by $\alpha{>}1/2$ and C. Both algorithms emit a stream of edges which they do not have to store. MemGen keeps $\mathcal{O}(n)$ items in internal memory and has a time complexity of $\mathcal{O}(n \log\log n + m)$, which is optimal for networks with an average degree of $\bar{d} = \Omega(\log\log n)$. For realistic values of $\bar{d} = o(n/\log^{1/\alpha}(n))$, HyperGen reduces the memory footprint to $\mathcal{O}([n^{1-\alpha}\bar{d}^{\alpha} + \log n]\log n)$.

In an experimental evaluation, we compare HyperGen with four generators among which it is consistently the fastest. For small $\bar{d} = 10$ we measure a speed-up of 4.0 compared to the fastest publicly available generator increasing to 29.6 for $\bar{d} = 1000$. On commodity hardware, HyperGen produces $3.7{\cdot}10^8$ edges per second for graphs with $10^6 \leq m \leq 10^{12}$ and $\alpha{=}1$, utilising less than 600 MB of RAM. We demonstrate nearly linear scalability on an Intel Xeon Phi.

1998 ACM Subject Classification G.2.2 [Graph Theory] Graph Algorithms

Keywords and phrases Random hyperbolic graph generator, streaming algorithm

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.26

1 Introduction

Even though most practical algorithms aim for a good performance on real-world data, artificial benchmarks are crucial for their development. Suited real-world datasets are typically scarce, do not scale, may exhibit noise or have uncontrollable properties. Tunable synthetic instances based on random models alleviate these issues. They are indispensable for systematic experiments allowing to quantify an algorithm's performance as a function of controllable parameters. Selecting the right model depends on the use case:

Many real-world networks (e.g., communication or social networks) exhibit basic features, such as a small diameter, a power-law degree distribution, and a non-vanishing local cluster coefficient [2, 3, 21, 23]. Amongst suited models, geometric random networks seem most natural. They explain the high local clustering of social networks[1] by embedding the nodes into a geometric space. Then the distance between any two nodes determines the probability of an edge between them. While Euclidean space is appropriate for spatial networks (e.g., [13]),

* Partially supported by the DFG grant ME 2088/3-2.
[1] I.e., a high triangle count expressing the intuition that two friends of a person are likely to acquaint too.

it distorts complex networks, such as the internet graph, for which hyperbolic embedding (cf. Section 1.3) performs well [7, 24].

The task of actually generating instances of hyperbolic random graphs has been approached recently yielding generators that are either fast in practice [27] or optimal in theory [9]. We target the generation of large instances whose set of nodes[2] does not fit into memory. Space requirements are crucial especially in the context of co-processors with small dedicated memory. Another application of such a generator is the experimental evaluation of streaming [11, 20] or external memory algorithms [1, 22]. Since our algorithm is typically faster than the time it takes to write data to disk, one can connect it to the algorithm under testing without a round-trip to secondary storage. In such a case, the generator should leave the majority of memory to the main application in order to allow fast context switches.

1.1 Our contribution

We introduce two related generators MemGen (Section 2) and HyperGen (Section 3) for the $G_{\alpha,C}(n)$-model (Section 1.3) for large instances with n nodes and $m = \bar{d}n/2$ edges where $\bar{d}$ is the expected average degree. Both generators target a streaming setting and are compatible with the external memory model for practical instances. MemGen requires $\mathcal{O}(n)$ internal memory and has a time complexity of $\mathcal{O}(n \log\log n + m)$, which is optimal for networks with an average degree of $\bar{d}{=}\Omega(\log\log n)$. For realistic values of $\bar{d}{=}o(n/\log^{1/\alpha}(n))$, HyperGen reduces the memory footprint to $\mathcal{O}([n^{1-\alpha}\bar{d}^{\alpha}{+}\log n]\log n)$, where $\alpha{>}1/2$ controls the exponent $2\alpha{+}1$ of the result's power-law degree distribution. In an experimental evaluation (Section 5), HyperGen consistently the previously fastest generators we are aware of.

In the quest for a smaller memory footprint, we increase the data locality leading to an easily parallelisable algorithm. While we only explore shared memory parallelism, HyperGen works in a distributed setting with constant communication.

1.2 Notation

Define $[k] := \{1, \dots, k\}$ for $k \in \mathbb{N}_{>0}$. A graph $G = (V, E)$ has $n = |V|$ sequentially numbered nodes $V = \{v_i\}_{i\in[n]}$ and $m = |E|$ edges. Unless stated differently, graphs are undirected, unweighted, and have an average degree of $\bar{d} = 2m/n$. Let $N_G(v) \subseteq V$ be the neighbourhood of node v in graph G, i.e. the set of adjacent nodes.

We mainly consider points (r, θ) in polar coordinates where r is the radius (i.e. distance from the origin) and θ is the polar angle or azimuth. A point with radius r_1 is said to be *above* a point with radius r_2 if $r_1 > r_2$ and *below* if $r_1 < r_2$. Let $B_y(z)$ the ball of radius y and centre at radius z, i.e. the set of all points with distance at most y from point[3] z [12]. We apply standard set operations to balls where $\setminus$, $\cup$, and $\cap$ denote set differences, union and intersection accordingly.

Let $\mu(X)$ denote the probability mass of X. We denote a Binomial distribution over n items with probability p as $\mathcal{B}(n, p)$. *Also refer to Appendix A for a summary of definitions.*

1.3 The hyperbolic random graph model $G_{\alpha,C}(n)$

We consider the well-accepted $G_{\alpha,C}(n)$ model [12]. It follows the initial zero-temperature model of Krioukov et al. [16], but removes a redundant curvature parameter by setting $\zeta{=}1$.

[2] Implementations typically use at least 80 byte/node (cf. Section 5.2).
[3] We omit the azimuth of z as it is irrelevant in our analysis due to polar symmetry.

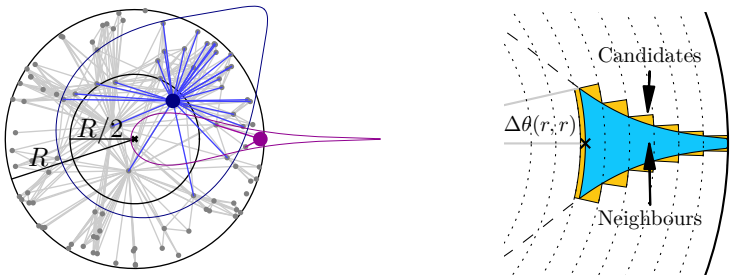

Figure 1 Left: The $G_{\alpha,C}(n)$ model with n=150, α=1, C=−2. The area enclosed by each coloured lobe corresponds to all points in distance at most R around its highlighted centre. **Right:** Band model introduced by NkGen (not to scale). The partial blue lobe indicates the area in which candidates can be found. The step-wise overestimation for candidate selection is shown in yellow.

▶ **Definition 1** (Gugelmann et al. [12]). Let $\alpha > 1/2$, $n \in \mathbb{N}_{>0}$, and $C > -2\log n$. The random graph $G_{\alpha,C}(n) = (V, E)$ has the following properties (cf. Figure 1):

- Each node $v_i \in V = \{v_1, \dots, v_n\}$ is modelled by a random point $p_i = (r_i, \theta_i)$ in the hyperbolic plane.[4] Its angular coordinate θ_i is drawn uniformly from $[0, 2\pi]$ while its radius $0 \le r_i < R$ with $R := 2\log n + C$ is governed by the density function

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

- The distance $d(p_i, p_j)$ between the two points p_i and p_j is given by

$$\cosh(d(p_i, p_j)) = \cosh(r_i)\cosh(r_j) - \sinh(r_i)\sinh(r_j)\cos(\theta_i - \theta_j). \tag{2}$$

  Two nodes $v_i, v_j \in V$ are adjacent iff they have a small distance $d(p_i, p_j) < R$ and $i \neq j$.

Intuitively, the smaller α the more likely are points with small radial components, which are expected to have a high number of neighbours. The parameter hence controls the skewness of the resulting power-law degree distribution with an exponent of $\gamma = 2\alpha + 1 > 2$ [16]. We assume $\alpha = \mathcal{O}(1)$ since real networks typically exhibit $2 \le \gamma \le 4$ (e.g., [10, 17]). Further, while with high probability there exists a giant component of linear size for $\alpha < 1$, networks with $\alpha > 1$ have components of sub-linear size [6]. The parameter C controls the average degree $\bar{d}$ of the graph which is governed as follows: [12]

$$\mathbb{E}\left[\,\bar{d}\,\right] = \frac{2}{\pi}\left(\frac{\alpha}{\alpha - 1/2}\right)^2 e^{-C/2}(1 + o(1)) \tag{3}$$

1.4 Hyperbolic graph generators

A naïve generator for hyperbolic graphs checks all $\binom{n}{2}$ pairwise distances and emits an edge for each pair of points close enough. On the one hand, such an approach can be implemented with constant memory overhead based on a pseudo-random hash function mapping node ids to coordinates. On the other, it incurs a sequential runtime of $\Theta(n^2)$ and is hence prohibitively expensive for large n. Similarly, while it can be fully parallelised yielding a

[4] We treat a node v_i and its corresponding point p_i as equivalent and use the terms interchangeably. Similarly, the symbols r_i and θ_i always refer to the radius and azimuth of point p_i.

$\mathcal{O}(1)$ time computation on a EREW-PRAM [14] with $p = \Theta(n^2)$ processors, such a solution requires $\Theta(n^2)$ work and is infeasibly inefficient.

All sub-quadratic algorithms we are aware of rather rely on a two-step approach: For each node $v \in V$, the generators firstly identify a set of candidates $C(v) \subseteq V$ by some geometric means (see below). Edges are then generated by computing only the distances between v and $C(v)$. In order to avoid false negatives, all neighbours $N(v)$ have to be a subset of $C(v)$. Techniques include:

- Looz et al. [26] project all points into the Poincaré disk model which allows neighbourhood queries based on Euclidean disks. Candidates are selected using a polar quad-tree. The authors bound the generator's runtime to $\mathcal{O}((n^{3/2} + m) \log n)$ with high probability.
- Later, Looz et al. improve the runtime significantly by dropping the angular separation of the quad-tree [27]. As sketched in Figure 1, their generator (which is the basis of our work and to which we refer as NKGEN[5]) decomposes the hyperbolic plane into $k{=}\Theta(\log n)$ bands, each covering the radial range $[b_j, b_{j+1})$ where $b_j = (1 - \beta^{j-1})R/(1 - \beta^k)$ for $j \in [k{+}1]$ and a tuning parameter $\beta{\approx}0.9$. In a preprocessing step, the points are randomly scattered over the plane by inserting each point (r, θ) into the appropriate band j, where $b_j \leq r < b_{j+1}$. The points are then sorted by their angular coordinates independently for each band.

  In order to query the neighbour candidates of a point $p{=}(r, \theta)$ stored in band i, the algorithm iterates over all bands $i \leq j \leq k$. For each band j, it computes the angular range $A_j = [\theta - \Delta\theta(r, b_j), \theta + \Delta\theta(r, b_j)]$ where the maximal angular distance $\Delta\theta(r, b_j)$ between p and any hypothetical point in band j is given by

$$\Delta\theta(r, b) := \begin{cases} \pi & \text{if } r + b < R \\ \operatorname{acos}\left[(\cosh(r)\cosh(b) - \cosh(R))/(\sinh(r)\sinh(b))\right] & \text{otherwise} \end{cases}. \quad (4)$$

  The points within A_j constitute all candidates from band j. Since points are sorted by their angular coordinates, the bounds of A_j can be identified using two binary searches in time $\mathcal{O}(\log n)$. The authors experimentally find a runtime of $\mathcal{O}(n \log n + m)$.
- Bringmann et al. [9] propose the *Geometric Inhomogeneous Random Graph* model (GIRG) and show that $G_{\alpha,C}(n)$ is a special case of GIRG which can be generated with their sampling algorithm in expected time $\mathcal{O}(n + m)$. Their sampling method for hyperbolic graphs is similar to the quad-tree approach in the sense that it partitions the space uniformly along the angular axis and exponentially in the radial direction. The resulting cells roughly correspond to leaves in a quad-tree. However, the algorithm does not execute fine-grained neighbourhood queries for each node; it rather tests all point pairs of two related cells in a pessimistic and data-oblivious fashion. Despite its expected linear runtime, the algorithm seems to suffer from high constants (cf. Section 5). Bläsius et al. provide an implementation[6] to which we refer as GIRGGEN [5].
- Very recently and independently from this work S. Lamm proposed a communication-agnostic distributed generator RHGEN with a partitioning scheme similar to [9], although with different radial limits [18]. Each band is split into disjoint buckets of equal angular size. Their number is chosen such that each cell is expected to contain k points, where $k \approx 4$ is a tuning parameter. RHGEN allows all processing units to compute the points within any bucket independently, eliminating the need of communication. The author

[5] A reference implementation is included in NetworKit [25], `https://networkit.iti.kit.edu/`.
[6] `https://bitbucket.org/HaiZhung/hyperbolic-embedder/overview`

Algorithm 1: MemGen

```
   Input  : Number of nodes n, Radius of bounding circle R, Density α, Spacing β
 1 Δθ(a,b) := π if a+b < R else acos[(cosh(a)cosh(b) − cosh(R))/(sinh(a)sinh(b))];
 2 noBands ← max(2, ⌈βR⌉);
 3 limits ← [0, R/2, c+R/2, 2c+R/2, ..., R−c, R] with c = R/2/(noBands − 1);
 4 for i ∈ [1, ..., n] do
 5     r ← random radius from [0, R) with density ρ(r) = α sinh(αr)/(cosh(αR) − 1);
 6     b ← search band s.t. limits[b] ≤ r < limits[b + 1];
 7     θ ← next non-decreasing uniformly random polar angle;
       // In case θ + 2Δθ(r,r) > 2π special treatment is necessary -
          cf. text
 8     bands [b].addPoint(Point(i, (r, (θ+Δθ(r,r)) mod 2π)));
 9     b ← max(2, b);
10     bands[b].addRequest(Request(i, [r, r+2Δθ(r, max(r, limits[b+1]))], (r, θ+Δθ(r,r))))
   // Main Phase: Generation of Edges
11 foreach u, v ∈ bands[1].points with u < v do
12     emit edge {u.id, v.id};
13 reqsToAbove ← [];
14 for b ∈ [2, ..., noBands] do
15     sort bands [b].points by angle;
16     reqsFromBelow ← sorted(reqsToAbove);
17     initialise empty reqsToAbove, candidates;
18     foreach pt ∈ bands[b].points do
19         remove all requests from candidates ending before pt.θ;
20         foreach req ∈ (bands[b].reqs ∪ reqsFromBelow) with req.rangeBegin ≥ pt.θ do
21             insert req into candidates if not existing;
22             insert req into reqsToAbove with updated range;
23         foreach req ∈ candidates do
24             if (req.r, req.id) ≤_lexico (pt.r, pt.id) ∧ dist(pt, req) ≤ R then
25                 emit edge {pt.id, req.id};
```

shows an expected sequential runtime of $\mathcal{O}(n+m)$, bounds the generation time of the distributed grid structure to $\mathcal{O}(P \log n + n/P)$, where P is the number of processors, and empirically finds a time-complexity of $\mathcal{O}(\frac{n+m}{P} + P \log n)$ for the parallel algorithm.

2 MemGen: a fast algorithm with linear memory usage

To simplify the description of HyperGen and present its main design, we start with a sequential version MemGen (cf. Alg. 1) requiring $\mathcal{O}(n)$ memory. Most arguments regarding the runtime of this algorithm will later translate into the space complexity bound of HyperGen.

Geometrically, MemGen employs a band partitioning similar to the one introduced by NkGen and illustrated in Figure 1. However, we alter their contents and access patterns, and use different radial band limits: all bands except the lowest one have a constant height $x = R/2k$, where $k+1$ is the number of bands and $x = \Theta(1)$ a tuning parameter

(typically $x \in [1, 2]$). Band $1 \leq i \leq k{+}1$ covers a radial range of $[l_i, l_{i+1})$ with $l_1 = 0$ and $l_i = [1 + (i{-}2)/k] \cdot R/2$ for some $k{=}\Theta(R)$. It is not necessary to further divide the lowest band since all points with radius $r \leq R/2$ are forming a clique (cf. Figure 1) and can be handled without vicinity tests.

Band b stores all points contained. For each point p within b or below it, the band additionally maintains a so-called *request* $\texttt{req}_b(p)$, storing the coordinates of p itself as well as the angular range in which neighbours of p can lie in band b. Such requests effectively reduce random accesses during the candidate selection and carry pre-computed values repeatedly required for the distance calculations (cf. Section 4).

In fact, the algorithm chooses a request-centric view and randomly draws the beginnings of each request range, computes its radius-dependent length, and then places a point at its centre.[7] We draw the polar components as sorted random numbers using the online technique detailed in [4] requiring constant time per element. The generation process may yield requests with a range $[a, b]$ with $b > 2\pi$. To take the azimuthal 2π-period of the hyperbolic disk into account, we split such queries into two separate ranges $[0, b{-}2\pi]$ and $[a, 2\pi]$ respectively and mark the latter as a copy. Analogously, points with $\theta > 2\pi$ are remapped to $\theta - 2\pi$. After the generation phase, the points are sorted by their polar coordinate.

In the main phase, we iterate over the bands starting from the centre for which we simply emit the clique of all nodes contained. For all higher bands, we scan through the points and requests in lock-step and keep a separate list of candidates $C(\cdot)$. Since both streams are sorted, we can efficiently update $C(v)$ when moving from one point to the next.

Each time we reach a new unmarked request $\texttt{req}_j(p)$, we propagate it to the next higher band $j{+}1$ by adding $\texttt{req}_{j+1}(p)$ to the appropriate insertion buffer. Here, it may be again necessary to split a request due to the 2π-periodicity. Further observe that the range of a request may shrink during the propagation. As a consequence, the insertion buffer has to be sorted when switching to band $j{+}1$ (cf. Section 2.2) before it can be merged with the requests generated in the preprocessing phase. In a last step, we compute the distance between a point and all candidates in order to emit the edges.

The linear time generators we are aware of use discrete buckets along the angular axis to avoid sorting [9, 18]. However, preliminary experiments with MemGen suggested that a more involved candidate selection process is faster in practice (especially in the context of vectorisation) and does incur only small theoretical penalties (cf. Theorem 7). Thus, we maintain a data structure which keeps active candidates in a continuous array to facilitate vectorisation efficiently (cf. Section 4). The array has an arbitrary order allowing to implement deletions as moves of the array's back. The data structure is further augmented with a search tree to find the position of a candidate using its point id as key. We also keep a priority queue with range-ends to quickly find and remove obsolete candidates.

2.1 Candidate selection is at worst a constant approximation

In this section, we establish all necessary facts to show that the candidate selection incurs a non-substantial overhead. In Lemma 2, we will see that most points issue only a constant number of requests.

Subsequently, we derive a high-probability bound on the number of candidates processed for any node in two steps: Observe that a node has to process all requests from nodes below. Lemma 3 bounds their number in terms of n and average degree $\bar{d}$. Further, Lemma 4 states

[7] While this is an arbitrary choice for MemGen, it will become a crucial ingredient for HyperGen.

that MemGen overestimates the probability mass during candidate selection by at most a constant factor. Therefore, the bound on the number of neighbours from below carries over to the number of candidates processed.

▶ **Lemma 2.** *The expected number of bands* $\mathbb{E}[B_i]$ *a random node* v_i *sends requests to is* $\mathbb{E}[B_i] = 1 + \frac{1-e^{-\alpha R/2}}{e^{\alpha/2k}-1} = \mathcal{O}(1)$ *where* $k{+}1 = \Theta(R)$ *is the number of bands used by* MemGen.

Proof. Each point with radius r sends requests to its own band j with $b_j \leq r < b_{j+1}$ as well as to all above. Consequently, the probability of a random point p_i contributing to band j is governed by the mass function $\mu(B_{b_{j+1}}(0))$ as given by Eq. (21). Using indicator variables for the reception of a request by band j, we obtain the claimed expectation value:

$$\mathbb{E}[B_i] = \sum_{j=0}^{k} \mu(B_{b_{j+1}}(0)) = \sum_{j=0}^{k} e^{\alpha[\frac{R}{2}(1+j/k)-R]} = e^{-\alpha R/2} \sum_{j=0}^{k} \left(e^{\alpha \frac{R}{2k}}\right)^j = 1 + \frac{1-e^{-\alpha R/2}}{e^{\alpha/2k}-1}. \quad ◀$$

▶ **Lemma 3.** *Let* N_j *be the number of neighbours the point* $p_j{=}(r_j, \theta_j)$ *has from below, i.e. neighbours with smaller radius. With high probability, there exist* $\mathcal{O}(n/\log^2 n)$ *points with* $N_j = \mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}\log(n))$ *while the remainder of points with* $r_j > R/2$ *has* $N_j = \mathcal{O}(n^{1-2\alpha}\log^{2\alpha}(n)\bar{d}^{2\alpha})$ *neighbours.*

Proof. Let $X_1, \dots, X_n$ be indicator variables with $X_i{=}1$ if p and p_i are adjacent. Due to radial symmetry we directly obtain the expectation value of X_i conditioned on the radius p_i:

$$\mathbb{E}[X_i \mid r_i{=}\,x] = \mathbb{P}[X_i{=}1 \mid r_i{=}\,x] = \begin{cases} 1 & \text{if } x < R - r \\ \Delta\theta(x,r)/\pi & \text{otherwise} \end{cases} \tag{5}$$

We remove the conditional using the Law of Total Expectation and equations (20) and (21):

$$\mathbb{E}[X_i] = \int_0^{R-r} \rho(x)\mathrm{d}x \quad + \quad \frac{1}{\pi}\int_{R-r}^{r} \rho(x)\Delta\theta(x,R)\mathrm{d}x \tag{6}$$

$$= \left[e^{-\alpha r} - e^{-\alpha R}\right](1+o(1)) + \frac{1}{\pi}\frac{\alpha}{\alpha-\frac{1}{2}}e^{-\alpha r}\left[e^{(\alpha-\frac{1}{2})(2r-R)} - 1\right](1 \pm \mathcal{O}(e^{-r})) \tag{7}$$

Fix the radius $r_T = R - \frac{2}{\alpha}\log\log n$ with $R/2 < r_T$ (wlog) and consider three cases for r:

- We ignore all points $r \leq R/2$ as they belong to the central clique and are irrelevant here.
- Observe that with high probability there exist $\mathcal{O}(n/\log^2(n))$ points below r_T. Exploiting the monotonicity of Eq. 7 in r, we maximise it by setting $r = R/2$, which cancels out the second term. Linearity of the expectation value, substitution of $R = 2\log(n) + C$, and Eq. (3) yield $\mathbb{E}[\sum_i X_i] = \mathcal{O}\left[n\left(\bar{d}/n\right)^{\alpha}\right]$. Then, Chernoff's bound gives $\sum_i X_i = \mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}\log(n))$ with high probability.
- For all points above r_T, set $r = r_T$ yielding $\sum_i X_i = \mathcal{O}(n^{1-2\alpha}\log^{2\alpha}(n)\bar{d}^{2\alpha})$ with high probability analogously. ◀

▶ **Lemma 4.** *Consider a query point with radius* r *and a band with boundaries* $[a, b)$. MemGen*'s candidate selection overestimates the probability mass of the actual query range by a factor of* $OE(b{-}a, \alpha)$ *where* $OE(x, \alpha) := \frac{\alpha-1/2}{\alpha}\frac{1-e^{\alpha x}}{1-e^{(\alpha-1/2)x}}$.

Proof. If $r < R{-}b$, the requesting point covers the band completely which renders the candidate selection process optimal. We now consider $r \geq R{-}a$ and omit the fringe case of $R{-}b < r < R{-}a$ which follows analogously (and by continuity between the two other cases).

Then, the probability mass μ_Q of the intersection of the actual query circle $B_R(r)$ with the band $B_b(0) \setminus B_a(0)$ is given by

$$\mu_Q := \mu\left[(B_b(0) \setminus B_a(0)) \cap B_R(r)\right] \tag{8}$$

$$= \mu\big[\big(B_R(0) \cap B_R(r)\big) \setminus B_a(0)\big] - \mu\big[\big(B_R(0) \cap B_R(r)\big) \setminus B_b(0)\big] \tag{9}$$

$$\overset{(22)}{=} \frac{2\alpha e^{-\frac{r}{2}}}{\pi(\alpha-\frac{1}{2})}\left[\left(1+\frac{\alpha-\frac{1}{2}}{\alpha+\frac{1}{2}}e^{-2\alpha b}\right)e^{(\alpha-\frac{1}{2})(b-R)} + \left(1+\frac{\alpha-\frac{1}{2}}{\alpha+\frac{1}{2}}e^{-2\alpha a}\right)e^{(\alpha-\frac{1}{2})(a-R)}\right](1+\epsilon) \tag{10}$$

$$= \frac{2}{\pi}e^{-\frac{r}{2}-(\alpha-\frac{1}{2})R}\left[\frac{\alpha}{\alpha-\frac{1}{2}}\left(e^{(\alpha-\frac{1}{2})b} - e^{(\alpha-\frac{1}{2})a}\right) + \mathcal{O}\left(e^{-(\alpha-\frac{1}{2})a}\right)\right], \tag{11}$$

where ϵ substitutes the error term expanded in the last line (cf. Figure 1, blue cover of a band).

MemGen overestimates the actual query range at the border and covers the mass μ_H (see Figure 1, yellow cover of a band):

$$\mu_H := \frac{1}{\pi}\Delta\theta(r,a)\int_a^b \rho(y)dy = \frac{1}{\pi}\cdot 2e^{\frac{R-a-r}{2}}(1+\mathcal{O}(e^{R-a-r}))\cdot\frac{\cosh(\alpha b)-\cosh(\alpha a)}{\cosh(\alpha R)-1} \tag{12}$$

$$= \frac{2}{\pi}e^{\frac{R-a-r}{2}}(1+\mathcal{O}(e^{R-a-r}))\cdot\left[e^{\alpha(b-R)} - e^{\alpha(a-R)}\right](1\pm o(1)) \tag{13}$$

$$= \frac{2}{\pi}e^{-\frac{r}{2}-(\alpha-\frac{1}{2})R}\left[e^{\alpha b-a/2} - e^{(\alpha-\frac{1}{2})a}\right]\cdot\left(1\pm\mathcal{O}\left(e^{(1-\alpha)(R-a)-r}\right)\right) \tag{14}$$

The claim follows by the division of both mass functions μ_H/μ_Q. ◀

▶ **Corollary 5.** *Given a constant band height, i.e. $b-a=\mathcal{O}(1)$, Lemma 4 implies a constant overestimation for any $\alpha>1/2$. In case of $b-a=1$, we have $OE(1,\alpha)\leq\sqrt{e}\approx 1.64\ \ \forall\alpha>1/2$.*

2.2 Nearly sorted points/request allow for faster sorting

MemGen's scheme to update the candidate list requires the input streams of requests and points to be increasing in their angular coordinate. Since we are not aware of a technique that directly yields both in an ordered fashion, we have to sort them. Using naïve methods this would amount to $\mathcal{O}(n\log(n))$ time (cf. Lemma 2). Since the number $m=n\bar{d}/2$ of edges generated constitutes a lower bound on the time complexity of any generator, this approach is optimal for $\bar{d}=\Omega(\log n)$.

Observe, however, that the points are calculated based on ordered requests and are therefore already nearly sorted. Similarly, requests have to be sorted after being propagated from ordered streams. In both cases, and with high probability, the change of rank of each item is bounded to some $\Delta=o(n)$.[8] Such a Δ-ordered sequence can be sorted in time $\mathcal{O}(n\log\Delta)$, e.g. using a sliding window coupled with a priority queue of size Δ.

The following Lemma gives a rough bound on the time complexity which suffices to show that MemGen is optimal for $\bar{d}=\Omega(\log\log(n))$ with high probability:

▶ **Lemma 6.** *Sorting all points initially and requests after their propagation requires $\mathcal{O}\left(n\min[\log(\bar{d}\log n),\log n]\right)$ time.*

[8] Split requests and remapped points are sorted separately and merged in linear time.

Proof. It suffices to bound the claim for requests since every point contributes at least one request and has a shorter lifetime. As stated in the introduction of the Lemma, we can rely on classical sorting in time $\mathcal{O}(n \log n)$ for the case of $\bar{d} = \Omega(\log n)$. Thus assume $\bar{d} = o(\log n)$.

The proof consists then of two steps: We pick a radius r_T, s.t. with high probability there are only $\mathcal{O}(n/\log^2(n))$ points below r_T. Since each point issues at most $\mathcal{O}(\log n)$ requests, we can classically sort their $\mathcal{O}(n/\log(n))$ tokens in time $\mathcal{O}(n)$. For the remaining points, we bound the number of overlapping requests from above and thereby also the maximal change in rank that can occur during sorting.

The number n_T of points below radius r_T is governed by the Binomial distribution $\mathcal{B}(n, B_{r_T}(0))$ with $B_{r_T}(0) = 1/\log^2(n)$. Solving for r_T yields $r_T = R - \frac{2}{\alpha}\log\log n$ and hence $n_T = \mathcal{O}[nB_{r_T}(0)]$ with high probability.

We now tend to the requests above r_T and exploit the two following facts:

- The number of bands above r_T is constant since $r_T/R \to 1$ as $n \to \infty$.
- During sorting only those requests that overlap can change their relative position. Therefore, we fix $\theta \in [0, 2\pi)$ and let n_θ be the number of requests that include θ.

To maximise n_θ, assume without loss of generality that all remaining requests lie at radius r_T. Then, n_θ is binomially distributed around its mean $n\mu$ with[9]

$$n\mu = n\frac{\Delta\theta(r_T, r_T)}{\pi} = 2ne^{-\frac{R}{2}+\frac{2}{\alpha}\log\log n} = \mathcal{O}\left(\bar{d}\log^{\frac{2}{\alpha}}(n)\right). \tag{15}$$

With high probability only $\mathcal{O}\left(\bar{d}\log^{\frac{2}{\alpha}}(n)\right)$ requests overlap due to Chernoff's inequality. We thus can sort them in time $\mathcal{O}(n\log(\bar{d}\log n))$. ◀

▶ **Theorem 7.** MemGen *requires* $\mathcal{O}(n)$ *memory and has a runtime of* $\mathcal{O}(n\log\log n + m)$ *with high probability.*

Proof. The space complexity directly follows from Alg. 1: each of the n points is stored in exactly one band, yields at most two requests, and requires $\mathcal{O}(1)$ space in the candidate list. During the main phase, there further exists only one insertions buffer at a time to which a point may contribute $\mathcal{O}(1)$ items. △

We bound MemGen's time complexity by considering each component individually:

- The preprocessing (until line 10) requires $\mathcal{O}(1)$ time per point making it non-substantial.
- Handling of cliques is trivially bounded by $\mathcal{O}(m)$ since every iteration emits an edge.
- The sorting steps (lines 15 and 16) require $\mathcal{O}(n\log(\bar{d}\log n)) = \mathcal{O}(n\log\bar{d} + n\log\log n)$ time in total with high probability according to Lemma 6.[10]
- By applying Lemma 3 and Cor. 5, the candidate selection requires $\mathcal{O}(n\log\bar{d})$ time with high probability.
- All distance calculations require in total $\mathcal{O}(m)$ time since Cor. 5 bounds the fraction of computations that do not yield an edge to $\mathcal{O}(1)$. ◀

3 HyperGen: reducing MemGen's memory footprint

In the analysis of MemGen, we repeatedly exploited the facts that requests are generated in increasing angular order and the majority affects only a small fraction of the hyperbolic

[9] It can be improved to $\mathcal{O}(\bar{d}\log\log n)$ by replacing the assumption that all requests lay at r_T with an appropriate integral; we omit this non-substantial calculation in favour of simplicity.

[10] We consider only the first min-term: In case the second term becomes smaller, the theorem's claim is dominated by the $\mathcal{O}(m)$ where $m = n\bar{d}/2$.

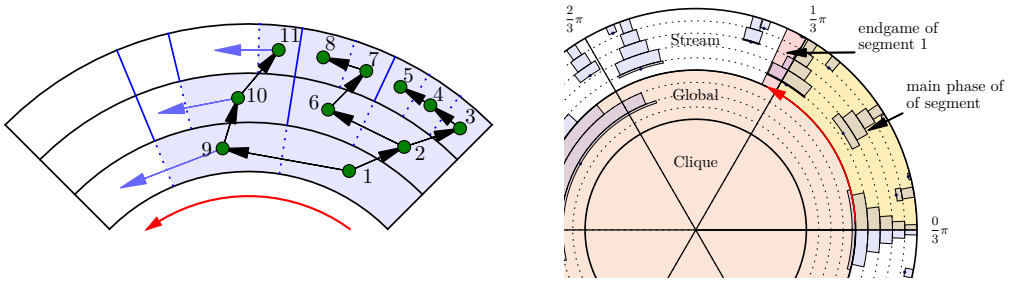

Figure 2 Left: HyperGen streams through each band consuming batches whose size is limited by two factors: either due to a polar limit imposed by the underlying band (solid blue line) or due to the limited number of requests a batch is allowed to have (dotted blue line). We traverse the indicated tree in depth-first order. **Right**: The hyperbolic plane is partitioned along the polar axis into p segments of equal size. Radially, there are two groups: the lower *global bands* which are preprocessed and kept in memory, and the upper *streaming bands*. In the main phase, each execution thread streams through its segment towards increasing polar angles (red arrow). Requests overlapping into the next segment are then completed in the endgame.

plane. This is also the foundation of HyperGen, which strives to additionally reduce the memory requirements of the generator. In order to do so, we do not draw all points globally and insert them into their bands, but rather reverse the scheme.

HyperGen first computes how many points go into each band. It is then able to draw points for each band independently. Due to the radial distribution function $\rho(r)$, band i with boundaries $[l_i, l_{i+1})$ carries a probability mass of $\mu_i = \mu[B_{l_{i+1}}(0) \setminus B_{l_i}(0)]$. Consequently, the numbers $N = (n_1, \ldots, n_k)$ of points per band with $n = \sum_i n_i$ are governed by a multinomial distribution with μ_i as event probabilities. We sample N and build for each band i a stream $S_i(n_i, s_i)$ that outputs exactly n_i requests with monotonously increasing angles as detailed in Section 2. Storing the seed value s_i used to initialise the underlying pseudo-random number generator enables HyperGen to replay the stream from the beginning.

Analogous to MemGen, each band maintains such a request stream S_i, the current candidates, and a small list of recently produced points. The generator starts with the innermost band $i = 1$ (cf. optimisation in Section 3.1) and draws a batch of at most c requests from its stream S_i, computes the positions of their corresponding points, and finally sorts the latter by their angle. Let θ_L be the beginning of the last request generated (θ_L=2π if the batch is empty). We merge the newly generated points with those remaining from the band's last batch, update the set of candidates, and match points against them as described for MemGen. Edges produced are pushed into the output stream.

Before we continue in the current band i, we first process all higher bands, hence limiting the amount of requests in memory. HyperGen propagates the recently generated requests to the band i+1. Observe that the request of a point (r, θ) is always centred around θ but its range shrinks as it is moved to higher bands. As a direct consequence, the higher range is completely enclosed by the lower one and no future request produced for band i will ever start before θ_L. Therefore, we recurse to band i+1 but limit processing there to points with $\theta < \theta_L$. In effect, HyperGen performs a depth-first traversal of the recursion tree illustrated in Figure 2 in which every node corresponds to a batch.

Due to the processing limit imposed on higher bands, we make sure they have the same information they would receive in MemGen. One subtle difference, however, concerns the fact that MemGen splits requests and remaps points overlapping the 2π threshold to take their angular periodicity into account. This is not possible in HyperGen since overlaps in outer bands are only detectable quite late in a run.

We resolve this issue by ignoring it at first, i.e. we perform the main computation phase exactly as described above. If there are still pending candidates or points after its completion, we restart the request streams to handle the so-called *endgame*. During endgame, HyperGen executes the same algorithm as before but only emits edges for pairs in which either the point or the request originate from the main phase. Therefore, it can be stopped as soon as all such *old* points and candidates have been processed. A single rewind suffices and thus does not affect the asymptotic runtime since a request has a length of at most 2π rad and a point can only be moved π rad in forward direction.

▶ **Theorem 8.** *For $c{=}\mathcal{O}(1)$ HyperGen requires $\mathcal{O}([n^{1-\alpha}\bar{d}^{\alpha}+\log n]\log n)$ memory with high probability, where $\bar{d}$ is the expected average degree and n the number of nodes.*

Proof. Each of the $k=\Theta(\log n)$ bands requires auxiliary data structures of constant size. Regarding the data contained, it again suffices to show the result for requests (cf. proof of Lemma 6). The number of points N_C with radius below $R/2$ is governed by a binomial distribution $\mathcal{B}(n,\mu(B_{R/2}(0)))$. Thus, with high probability $N_C=\mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}+\log n)$ where the second term ensures concentration for small $(\bar{d}/n)^{\alpha}$. Each such point contributes requests to $k=\mathcal{O}(\log n)$ bands; multiplication yields the claim.

According to Lemma 3 and Cor. 5 and for any fixed θ, there are with high probability $\mathcal{O}(n^{1-\alpha}\bar{d}^{\alpha}\log n)$ points with radius $r\geq R/2$ that have at least one request including θ. By Lemma 2, they contribute to $\mathcal{O}(1)$ bands on average and thus are covered by the claim. ◀

▶ **Corollary 9.** *In the external memory model with $M=\Omega([1+n^{1-\alpha}\bar{d}^{\alpha}]\log n)$, HyperGen only triggers I/Os to write out the resulting m edges in $\mathcal{O}(\mathit{scan}(m))$ I/Os.*

3.1 Accelerating the Endgame

A runtime/memory trade-off can be implemented to improve the runtime (especially in the context of the parallel variant). Rather than starting the streaming approach introduced above, we compute all bands with radii at most r_G and store them as in MemGen in the so-called *global phase*. This allows us to propagate split requests to the streaming bands which in turn allows us to stop the endgame earlier.

Observe that a request of a point (r_G,θ) has a length of at most $2\Delta\theta(r_G,r_G)$. To restrict the endgame to a fraction $1/f$ of the hyperbolic plane, we solve $2\Delta\theta(r_G,r_G)=2\pi/f$ for r_G. The number $n_G(f)$ of points generated in the global phase, which have to be kept in internal memory, is thus binomially distributed around the mean of

$$\mathbb{E}\left[n_G(f)\right]=n\mu(B_{r_G}(0))=n\left(\frac{\bar{d}f}{2n}\right)^{\alpha}\left(\frac{\alpha-\frac{1}{2}}{\alpha}\right)^{2\alpha}=\mathcal{O}\left(n^{1-\alpha}\bar{d}^{\alpha}f^{\alpha}\right). \tag{16}$$

3.2 Parallelism

Similarly to NkGen, HyperGen can easily be parallelised by decomposing the hyperbolic plane into p segments of equal size along the polar axis. As shown in Figure 2, we use a global phase with $f\geq p$ to handle the n_G requests spanning more than one segment. We enqueue a copy of each such request into all segments it affects. For realistic settings, it suffices to execute this phase sequentially; however, parallelism can be applied as in NkGen's implementation. The number of points in each segment $(\nu_1,\ldots,\nu_p)$ with $n-n_G=\sum_i\nu_i$ is then sampled from a multinomial distribution in which each event is equally likely. Based on this distribution, each band continues independently as described in the original formulation

of HyperGen. In the endgame, each segment retrieves the seed values of its successor's pseudo-random number generators and replays its streams.

In a distributed scenario the seed values can be computed using a pseudo-random hash function mapping the segment id to a pseudo-random seed value. Further, the initial distribution as well as the fast global phase can be computed repeatedly by each compute node, yielding constant communication.

4 Implementation

The prototypical implementation is available at `https://github.com/manpen/hypergen/`.

4.1 Adjacency tests without trigonometric functions

In a preliminary study we found that NkGen's runtime is dominated by trigonometric computations during the calculation of distances between points and their neighbour candidates. We approach this issue by introducing a new pre-computing scheme inspired by the usage of the Poincaré disk model in [26]. We project the random points into the unit disk causing additional work per point but simplifying all further distance computation. Thus, the speed-up increases with the average degree.

Our implementation applies the transform only to the distance calculations and does not change the candidate selection process. Let $p{=}(r_p, \theta_p)$ and $q{=}(r_q, \theta_q)$ be two points in the hyperbolic space and $p' = (\mathrm{cdm}(r_p), \theta_p)$ and $q' = (\mathrm{cdm}(r_q), \theta_q)$ their counterparts in the Poincaré disk model, where $\mathrm{cdm}(r) := [(1-r^2)/(1+r^2)]^{1/2}$. Then p and q are adjacent if

$$R > d(p', q') = \operatorname{acosh}\left(1 + 2\frac{||q-p||^2}{(1-||p||^2)(1-||q||^2)}\right) \tag{17}$$

$$\Leftrightarrow \quad \frac{\cosh(R)-1}{2} > \frac{||q-p||^2}{(1-||p||^2)(1-||q||^2)} = \frac{(x_{p'}-x_{q'})^2 + (y_{p'}-y_{q'})^2}{(1-r_{q'}^2)(1-r_{q'}^2)} \tag{18}$$

$$= \left((x_{p'}-x_{q'})^2 + (y_{p'}-y_{q'})^2\right) \cdot \gamma(r_{p'}) \cdot \gamma(r_{q'}), \tag{19}$$

where $x_{p'}{=}r_{p'}\sin(\theta_p)$ and $y_{p'}{=}r_{p'}\cos(\theta_p)$ are the Cartesian coordinates of point p' (analogously for q'). We reduce a distance computation to three additions and four multiplications by pre-computing $x_{p'}, y_{p'}$ and $\gamma(r_{p'}) := 1/(1-r_{p'}^2)$ for each point. The resulting expression can be vectorised effectively and even allows to partially fuse operations (cf. FMA instructions).

Our implementation uses explicit vectorisation[11] only during the distance computation. For graphs with small average degree, a speed-up may be possible by vectorising per-point computations such as the random number generation and geometric transformations.

4.2 Optimising NkGen for streaming

In addition to the default implementation of NkGen, we study a variant NkGenOpt to which we apply the following optimisations:[12]

[11] Based on libVC – SIMD Vector Classes for C++ [15], `https://github.com/VcDevel/Vc`.

[12] NkGen originally generates an adjacency-list-like internal-memory data-structure using the NetworKIT's GraphBuilder module. This limits the graph sizes and explains NkGen optimisation for smaller graphs. Further, the removal of the GraphBuilder in this work shifts the implementation's balance and leads to the large optimisation potential demonstrated. Porting the optimisations back to NetworKIT showed insignificant changes for typical instances which could be likely solved with an optimised GraphBuilder.

- It avoids recalculations similar to Section 4.1, but does not rely on the Poincaré transform. In NkGen's case all additional data has to be kept in memory amounting to roughly 32 bytes per points. We expect that this increase is only significant for very sparse graphs as NetworKit keeps the whole adjacency list in RAM.
- The number of binary searches as well as their range[13] is reduced. Further, the amount of data copied is significantly decreased also resulting in less (de-)allocation operations. This optimisation roughly compensates the increased footprint due to the pre-computations.
- We removed several checks which are not required for the restricted case of $G_{C,\alpha}(n)$.

HyperGen and NkGenOpt are verified against NkGen over a wide range of parameters. Here, we observed only acceptable numerical discrepancies for large graphs affecting less than one edge out of 10^5, caused by the different implementations of the distance computation.

5 Experimental evaluation

In this section, we compare six configurations: HyperGen on CPU / Xeon Phi (cf. Section 3), NkGen [27], NkGenOpt (cf. Section 4.2), RHGen [18], and GirgGen [5]. They are implemented in C++ and built as release versions using the same compiler. As an exception, HyperGen has to use a hardware-specific compiler, links against Intel's TBB malloc_proxy, but otherwise has the same code basis as the CPU version.

To fully exploit HyperGen's on-the-fly edge generation, none of the implementations writes the edge list into memory. We rather simulate a very simplistic streaming algorithm which consumes the edge stream and computes a fingerprint by summing all node indices contained.[14] This choice enforces that the generators have to compute and forward every edge but does not impose memory restrictions. With the exception of GirgGen, all generators support parallelism and are configured to use all available hardware threads. RHGen employs a multi-process design using MPI allowing several compute nodes, while HyperGen, NkGen and NkGenOpt use lightweight threads based on OpenMP.

The runtime benchmarks were conducted on one of the following systems:

- Indicated by (Phi): Intel Xeon Phi 5120D (60 cores, 240 threads, 1.05GHz), 8 GB GDDR5 RAM Linux 2.6.38, ICC 17.0.0, Intel TBB malloc_proxy
- Otherwise: Intel Xeon CPU E5-2630 v3 (8 cores, 16 threads, 2.40GHz) with AVX2/SSE4.2 support for 4-way double-precision vectorisation, 64 GB 2133 MHz RAM, Linux 4.8.1, GCC 6.2.1, VC (8. Dec. 2016), MPICH 3.2-7

The number of repetitions per data point (with different random seeds) is denoted by S. All plots show the median of repeated measurements and errorbars corresponding to the unbiased estimation of the standard deviation. Due to its large runtime GirgGen typically only includes one measurement per data point.

5.1 Runtime

We study the generators' runtimes for a wide range of graph sizes. For each run, we fix the number of nodes $10^5 \leq n \leq 10^9$ as well as the average target degree $\bar{d} \in \{10, 1000\}$, which we consider as lower and upper limits of realistic inputs [3, 19, 21, 23]. In order

[13] By replacing $\Delta\theta(r, b_i)$ by $\Delta\theta(r, r)$ when searching candidates for point (r, θ) in band i with $b_i \leq r < b_{i+1}$.
[14] We removed the appropriate memory allocations and accesses from NkGen, RHGen, and GirgGen, and added the streaming simulation. The patches are included in our repository.

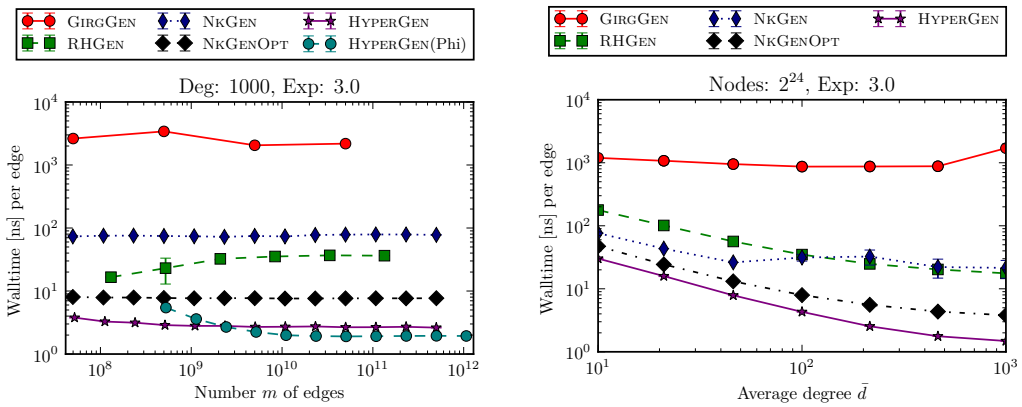

Figure 3 Runtime/edge generated for α=1 (power-law exp. γ=3) as a function of n and $\bar{d}$. S=5.

to achieve compatible results, all implementations use values of R derived with NkGen's `getTargetRadius`-method. In case of HyperGen, we use two segments per thread to balance load for large average degrees. For RHGen we chose an expected bucket size of four which resulted in the best performance in preliminary tests.

As shown in Figures 3 and 6 (Appendix) and Table 1 (Appendix), HyperGen is consistently the fastest generator, followed by NkGenOpt which outperforms NkGen. GirgGen is always the slowest. If we assume perfect parallelisability and divide GirgGen's walltime by the number of cores, it is on par with NkGen for small degrees but remains up to one order of magnitude slower for $\bar{d} = 1000$. For $\bar{d} = 10$ NkGen outperforms RHGen, while for $\bar{d} = 1000$ and $\alpha = 1$ the opposite is true.

All generators but HyperGen (Phi) exhibit an almost constant computation time per edge for large n. The improvements of HyperGen (Phi) towards larger n can be attributed to the very high number of threads ($p = 240$) which incur more overhead compared to runs performed on a CPU. This overhead is amortised only for high values of n.

Based on Figure 6 (Appendix), we measure a speed-up of 4.0 for $\bar{d}$=10 and 29.6 for $\bar{d}$=1000 when comparing HyperGen to NkGen for $n \geq 10^8$ and $\alpha = 1$. Similar results for smaller n are included in Table 1 (Appendix). On (Phi), HyperGen is 2.3 times faster ($\bar{d}$=10) compared to the execution on the more modern CPU-based reference system. The speed-up reduces to 1.2 for $\bar{d}$=1000 which seems to be caused by a smaller cache per thread.

When using HyperGen to test a multi-pass streaming algorithm, it is virtually always faster to repeatedly regenerate the graph than to buffer it in external memory.

5.2 Memory consumption

The memory consumption is measured for the same parameter settings as above. We consider the maximal resident set size (i.e. the peak allocation of the generator) as reported by the operating system. While all implementations seem to have potential for further savings, Figures 4 and 7 (Appendix) show a clear trend: With the exception of HyperGen, all generators seem to converge to a linear growth for large n requiring ≈ 80 byte per node. RHGen exhibits higher constants which may be partially caused by overheads due to its MPI architecture spawning independent processes rather than lightweight threads and preventing cheap shared-memory utilisation.

Consistent with our analysis, HyperGen exhibits a sub-linear footprint rendering it orders of magnitude cheaper for large n. As the number n of nodes increases (and hence R

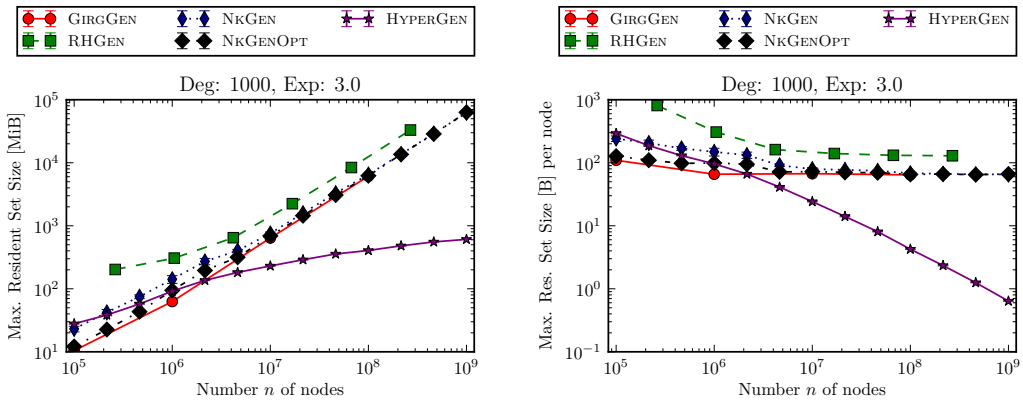

Figure 4 Maximal memory allocated during execution as measured by `time` for $\alpha = 1$.

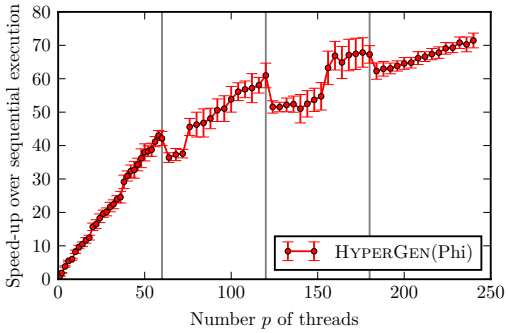

Figure 5 Strong scaling of HyperGen on (Phi) for a graph with n=10^8 and $\bar{d} = 10$. $S = 8$. Each vertical division marks a new level of HyperThreading.

for fixed $\bar{d}$), more points lie in the outer bands. Thus, a smaller fraction of points has to be handled (and stored) during the global phase. For the same reason, the memory footprint decreases with increasing α. To support Theorem 8 and the analysis in Section 3.1, we carried out additional runs up to n=10^{11} whose memory footprint is well within the noise observed for n=10^8. We do not include measurements for (Phi) since the memory allocation scheme adopted for the high number of threads does not yield meaningful set sizes.

5.3 Scalability

We measure HyperGen's scalability using strong scaling experiments on (Phi). This processor features 60 physical cores each offering four virtual threads (HyperThreading). While fixing the graph instance to n=10^8 and $\bar{d} = 10$, we record the runtime for an increasing number p of threads. As illustrated in Figure 5, the implementation exhibits a nearly linear speed-up of 43.0 ± 1.5 when utilising $p = 58$ threads. Surpassing this point, the computational power provided by the hardware does not scale linearly any more. Thus, the additional speed-up is less pronounced peaking at 71.4 ± 6 for $p = 240$.

Acknowledgments. The author thanks Ulrich Meyer, Kamil René König, Moritz von Looz and Alexander Schickedanz for valuable discussions and suggestions, Sebastian Lamm for providing the code and support for RHGen, Ivan Kisel and Egor Ovcharenko for their help

with the Xeon Phi, as well as the anonymous reviewers for their insightful comments and recommendations.

References

1 Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM, 31(9)*, pages 1116–1127, 1988.

2 Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *CoRR*, cond-mat/0106096, 2001. doi:10.1103/RevModPhys.74.47.

3 Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *Web Science 2012, WebSci'12, Evanston, IL, USA – June 22-24, 2012*, pages 33–42, 2012. doi:10.1145/2380718.2380723.

4 Jon Louis Bentley and James B. Saxe. Generating sorted lists of random numbers. *ACM Trans. Math. Softw.*, 6(3):359–364, 1980. doi:10.1145/355900.355907.

5 Thomas Bläsius, Tobias Friedrich, Anton Krohmer, and Sören Laue. Efficient embedding of scale-free graphs in the hyperbolic plane. In *24th Annual European Symposium on Algorithms, ESA 2016, Aarhus, Denmark*, 2016. doi:10.4230/LIPIcs.ESA.2016.16.

6 Michel Bode, Nikolaos Fountoulakis, and Tobias Müller. *On the giant component of random hyperbolic graphs*, pages 425–429. Scuola Normale Superiore, Pisa, 2013. doi:10.1007/978-88-7642-475-5_68.

7 Marián Boguñá, Fragkiskos Papadopoulos, and Dmitri Krioukov. Sustaining the internet with hyperbolic mapping. *Nature Communications*, Sep 2010. doi:10.1038/ncomms1063.

8 Béla Bollobás. *Random Graphs*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2 edition, 2001. doi:10.1017/CBO9780511814068.

9 Karl Bringmann, Ralph Keusch, and Johannes Lengler. Geometric inhomogeneous random graphs. *CoRR*, abs/1511.00576, 2015. URL: http://arxiv.org/abs/1511.00576.

10 Sergei N Dorogovtsev and José FF Mendes. *Evolution of networks: From biological nets to the Internet and WWW*. OUP Oxford, 2013.

11 Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi, editors. *Data Stream Management – Processing High-Speed Data Streams*. Data-Centric Systems and Applications. Springer, 2016. doi:10.1007/978-3-540-28608-0.

12 Luca Gugelmann, Konstantinos Panagiotou, and Ueli Peter. Random hyperbolic graphs: Degree sequence and clustering – (extended abstract). In *Automata, Languages, and Programming – 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, pages 573–585, 2012. doi:10.1007/978-3-642-31585-5_51.

13 Piyush Gupta and P. R. Kumar. The capacity of wireless networks. *IEEE Trans. Information Theory*, 46(2):388–404, 2000. doi:10.1109/18.825799.

14 Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

15 Matthias Kretz. *Extending C++ for explicit data-parallel programming via SIMD vector types*. PhD thesis, Goethe University Frankfurt am Main, 2015.

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

17 Ravi Kumar, Jasmine Novak, and Andrew Tomkins. Structure and evolution of online social networks. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, pages 611–617, 2006. doi:10.1145/1150402.1150476.

18 Sebastian Lamm. Communication efficient algorithms for generating massive networks. Master's thesis, Karlsruhe Institute of Technology, 2017. doi:10.5445/IR/1000068617.

19 Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

20 Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014. doi:10.1145/2627692.2627694.

21 Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. Graph structure in the web – revisited: a trick of the heavy tail. In *23rd International World Wide Web Conference, WWW'14, Seoul, Republic of Korea, April 7-11, 2014, Companion Volume*, pages 427–432, 2014. doi:10.1145/2567948.2576928.

22 Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

23 Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy J. Lin. Information network or social network?: the structure of the twitter follow graph. In *23rd International World Wide Web Conference, Seoul, Republic of Korea*, 2014. doi:10.1145/2567948.2576939.

24 Yuval Shavitt and Tomer Tankel. Hyperbolic embedding of internet graph for distance estimation and overlay construction. *IEEE/ACM Trans. Netw.*, 16(1):25–36, 2008. doi:10.1145/1373452.1373455.

25 Christian Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. doi:10.1017/nws.2016.20.

26 Moritz von Looz, Henning Meyerhenke, and Roman Prutkin. Generating random hyperbolic graphs in subquadratic time. In *Algorithms and Computation – 26th International Symposium, ISAAC 2015, Nagoya, Japan, December 9-11, 2015, Proceedings*, pages 467–478, 2015. doi:10.1007/978-3-662-48971-0_40.

27 Moritz von Looz, Mustafa Safa Özdayi, Sören Laue, and Henning Meyerhenke. Generating massive complex networks with hyperbolic geometry faster in practice. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–6, 2016. doi:10.1109/HPEC.2016.7761644.

A Definitions, useful identities and approximations

A.1 Hyperbolic functions

$$\sinh(x) := \frac{1}{2}\left(e^x - e^{-x}\right) \qquad \operatorname{asinh}(y) = x \;\Rightarrow\; \sinh(x) = y$$
$$\cosh(x) := \frac{1}{2}\left(e^x + e^{-x}\right) \qquad \operatorname{acosh}(y) = x \;\Rightarrow\; \cosh(x) = y$$

A.2 Geometry related definitions

$$\rho(r) := \alpha\frac{\sinh(\alpha r)}{\cosh(\alpha R)} \qquad \text{radial density, cf. Eq 1}$$
$$\mu(B_r(0)) := \int_0^r \rho(x)\mathrm{d}x = \frac{\cosh(\alpha x)-1}{\cosh(\alpha R)} \qquad \text{radial cdf}$$

$$\Delta\theta(r,b) := \begin{cases} \pi & \text{if } r+b < R \\ \operatorname{acos}\left[\frac{\cosh(r)\cosh(b)-\cosh(R)}{\sinh(r)\sinh(b)}\right] & \text{otherwise} \end{cases} \qquad \text{cf. Eq. 4}$$

A.3 Approximations

Gugelmann et al. derived the following approximations[15] [12]:

$$\Delta\theta(r,b) = \begin{cases} \pi & \text{if } r+b < R \\ 2e^{\frac{R-r-y}{2}}(1+\Theta(e^{R-r-y})) & \text{if } r+b \geq R \end{cases} \tag{20}$$

$$\mu(B_r(0)) = \int_0^r \rho(x)\mathrm{d}x = \frac{\cosh(\alpha r)}{\cosh(\alpha R)-1} = e^{\alpha(r-R)}(1+o(1)) \tag{21}$$

$$\mu\left[(B_R(r)\cap B_R(0))\backslash B_x(0)\right] = \frac{2}{\pi}\frac{\alpha e^{-r/2}}{\alpha-\frac{1}{2}} \cdot \begin{cases} 1\pm\mathcal{O}(e^{-(\alpha-\frac{1}{2})r}+e^{-r}) & \text{if } x < R-r \\ \left[1-(1+\frac{\alpha-\frac{1}{2}}{\alpha+\frac{1}{2}}e^{-2\alpha x})e^{-(\alpha-\frac{1}{2})(R-x)}\right](1\pm\mathcal{O}(e^{-r}+e^{-r-(\alpha-\frac{3}{2})(R-x)})) & \text{if } x \geq R-r \end{cases} \tag{22}$$

[15] We drop the $(1+\mathcal{O}(\cdot))$ error terms in our calculations without further notice if they are either irrelevant or dominated by other simplifications made

B Additional experimental results

Table 1 Comparison of generators for $n = 2^{26}$, $\alpha \in \{0.55, 1\}$, and $\bar{d} \in \{10, 1000\}$. *Comp* refers to the number of distance computations between two points. It does not include node pairs that could be ruled out earlier (e.g., by comparing indices or radii). For HyperGen the value is higher due to vectorisation which often prevents such early discarding. *RSS* is the maximal resident set size (i.e. peak memory allocation) as reported by the operating system. In case of RHGen it is the sum of RSS of all MPI processes yielding a higher overhead. GirgGen is a purely sequential implementation and includes fewer data points due to the high runtime. We report the standard deviation of the S measurements as uncertainty and apply statistical error propagation.

† Experiment was cancelled after a runtime of 10^5 s.

n=2^{26}, $\bar{d}$=10, α=0.55, R=39.2			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	6	10.3 ± 0.4	7.5 ± 0.2	0.0 ± 0.0	11.8 ± 0.1	2.1 ± 0.1	34.0 ± 1.4	1	1	1
NkGen	6	9.8 ± 0.7	5.6 ± 0.3	4.6 ± 0.3	57.1 ± 3.0	1.7 ± 0.2	173 ± 22	0.8 ± 0.1	290 ± 22	4.8 ± 0.3
NkGenOpt	6	9.6 ± 0.4	5.3 ± 0.2	4.1 ± 0.0	36.0 ± 0.3	1.7 ± 0.1	111.7 ± 6.1	0.7 ± 0.0	263.5 ± 3.2	3.1 ± 0.0
RHGen	4	8.3 ± 0.1	7.9 ± 0.0	6.7 ± 0.6	110.0 ± 1.0	2.8 ± 0.0	395.8 ± 6.7	1.1 ± 0.0	428 ± 39	9.3 ± 0.1
GirgGen	3	10.0 ± 0.0	18.1 ± 0.0	3.9 ± 0.0	884.3 ± 0.8	5.4 ± 0.0	2635.5 ± 2.8	2.4 ± 0.1	248.1 ± 2.2	75.0 ± 0.5

n=2^{26}, $\bar{d}$=10, α=1.00, R=33.3			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	5	9.7 ± 0.0	7.0 ± 0.0	0.0 ± 0.0	12.9 ± 0.1	2.1 ± 0.0	39.6 ± 0.3	1	1	1
NkGen	5	10.0 ± 0.0	5.5 ± 0.0	4.1 ± 0.0	54.9 ± 0.5	1.6 ± 0.0	163.5 ± 1.6	0.8 ± 0.0	602 ± 13	4.3 ± 0.1
NkGenOpt	5	10.0 ± 0.0	5.2 ± 0.0	4.1 ± 0.0	34.4 ± 0.2	1.6 ± 0.0	102.3 ± 0.8	0.8 ± 0.0	596 ± 12	2.7 ± 0.0
RHGen	5	10.0 ± 0.0	8.1 ± 0.0	7.5 ± 0.5	120.5 ± 0.4	2.4 ± 0.0	359.2 ± 1.2	1.2 ± 0.0	1100 ± 99	9.4 ± 0.1
GirgGen	3	10.0 ± 0.0	16.6 ± 0.0	3.9 ± 0.0	819.8 ± 7.1	5.0 ± 0.0	2443 ± 21	2.4 ± 0.0	566 ± 11	63.6 ± 1.0

n=2^{26}, $\bar{d}$=1000, α=0.55, R=29.5			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	5	1052.2 ± 1.6	622.8 ± 1.2	0.2 ± 0.0	86.9 ± 0.4	1.8 ± 0.0	2.5 ± 0.0	1	1	1
NkGen	5	994.4 ± 3.3	456.2 ± 1.3	6.4 ± 0.3	955.5 ± 4.9	1.4 ± 0.0	28.6 ± 0.2	0.7 ± 0.0	27.2 ± 1.3	11.0 ± 0.1
NkGenOpt	5	991 ± 19	441.6 ± 7.8	4.2 ± 0.0	299.5 ± 5.1	1.3 ± 0.0	9.0 ± 0.3	0.7 ± 0.0	17.6 ± 0.3	3.4 ± 0.1
RHGen	5	889.1 ± 2.2	426.0 ± 1.1	23.9 ± 2.4	2205 ± 64	1.4 ± 0.0	73.9 ± 2.3	0.7 ± 0.0	101 ± 11	25.4 ± 0.9
GirgGen	1	1000.0	1160.6	3.8	55756.0	3.5	1661.6	1.9 ± 0.0	16.2 ± 0.1	641.5 ± 2.8

n=2^{26}, $\bar{d}$=1000, α=1.00, R=24.1			in total			per edge		relative to HyperGen		
Algo	S	Degree	Comp. [10^8]	RSS [GB]	Time [s]	Comp.	Time [ns]	Comp.	RSS	Time
HyperGen	5	1015.8 ± 1.3	616.2 ± 0.7	0.1 ± 0.0	84.3 ± 0.5	1.8 ± 0.0	2.5 ± 0.0	1	1	1
NkGen	5	999.9 ± 0.6	443.3 ± 0.3	4.4 ± 0.1	1878 ± 468	1.3 ± 0.0	56 ± 14	0.7 ± 0.0	43.7 ± 2.4	22.3 ± 5.7
NkGenOpt	5	999.7 ± 0.4	428.5 ± 0.2	4.2 ± 0.0	261.1 ± 6.7	1.3 ± 0.0	7.8 ± 0.2	0.7 ± 0.0	41.6 ± 1.1	3.1 ± 0.1
RHGen	5	999.1 ± 0.0	410.5 ± 0.0	8.1 ± 0.2	1234.8 ± 5.8	1.2 ± 0.0	36.8 ± 0.2	0.7 ± 0.0	79.8 ± 3.7	14.6 ± 0.2
GirgGen†	1				$\geq 10^5$					≥ 1150

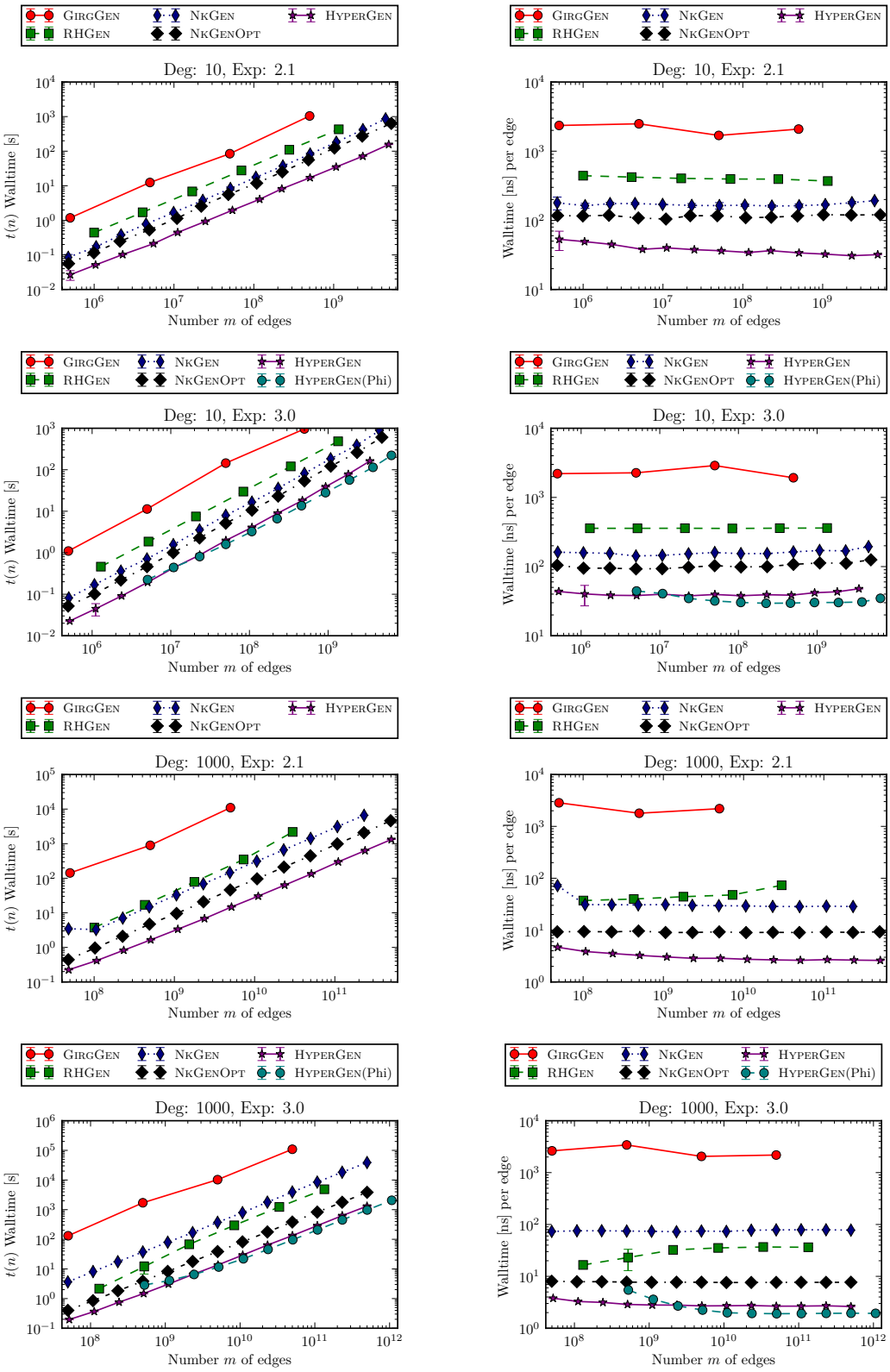

Figure 6 Runtime of generators as function of the number n of nodes.

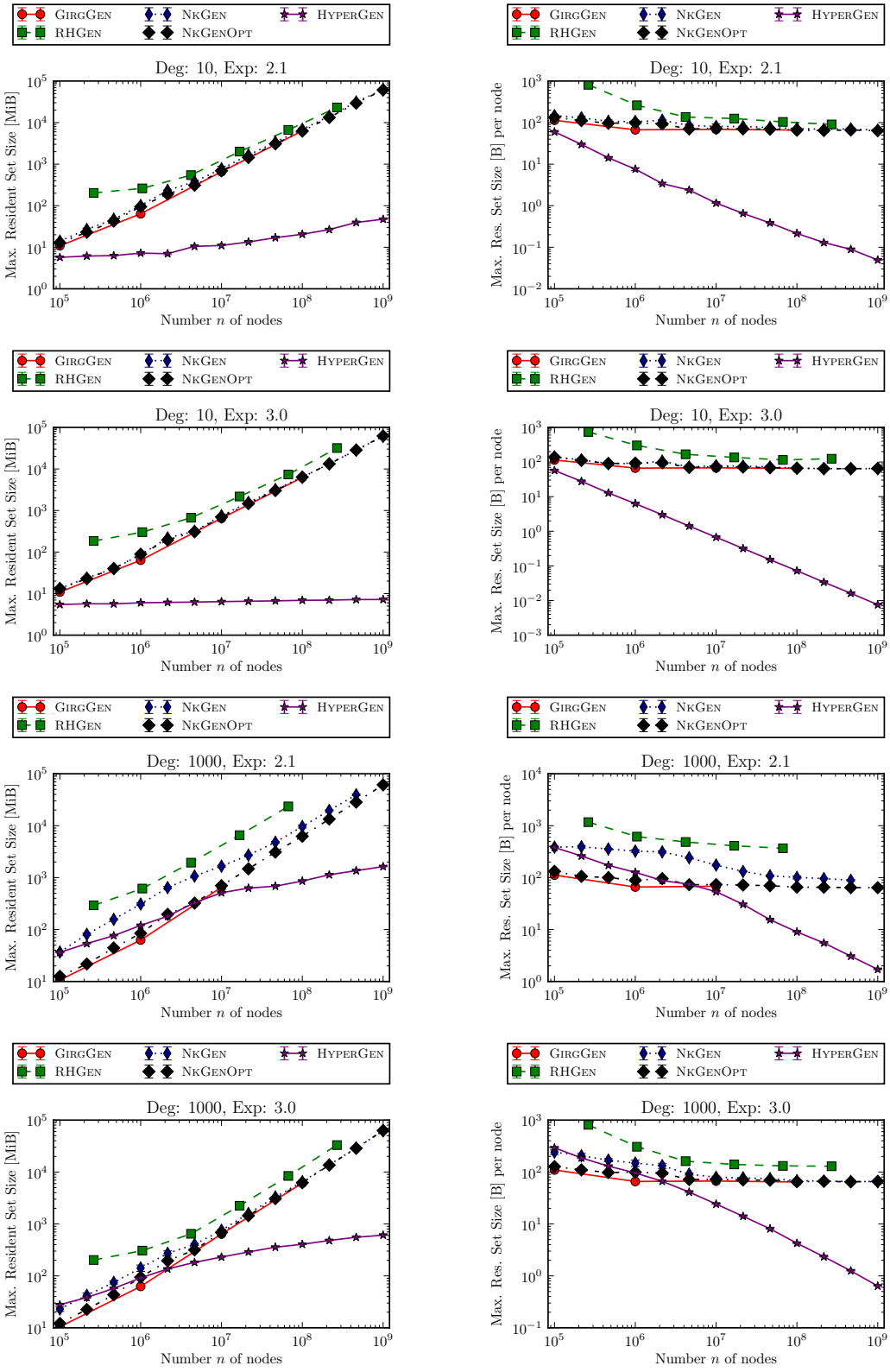

Figure 7 Max. memory allocation of generators as function of the number n of nodes.